Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Science
Department Computer Science
Web & Information Systems Lab

Making existing educational games adaptive using AOP

Graduation thesis submitted in partial fulfillment of the requirements for the degree of Master in Computer Science

Ben Corne

Promoter: Prof. Dr. Olga De Troyer

JUNE 2013

Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-Ingenieurswetenschappen
Departement Computerwetenschappen
Web & Information Systems Lab

Making existing educational games adaptive using AOP

Proefschrift ingediend met het oog op het behalen
van de titel Master in de Ingenieurswetenschappen: Computerwetenschappen

Ben Corne

Promotor: Prof. Dr. Olga De Troyer

JUNI 2013

Abstract

In this dissertation, we present a generic methodology, GAAOP, for adding adaptivity to educational games. A detailed overview of the multi-disciplinary domain of adaptive educational games is given, in order that readers with background in either the Computer Science, Cognitive Psychology, or Engineering can understand the framework. This includes definitions and interpretations of adaptivity for the domain of Cognitive Psychology and for the domain of virtual reality, patterns and architectures used in the domain of game development, and Aspect-Oriented programming techniques that target the issues related to programming cross-cutting concerns.

The methodology was created via an inductive approach, where a general method was extracted from the manual implementation of adaptivity to an example educational game, TuxMath, in the context of a pilot study for the CAdE project, a multi-disciplinary research project on cognitive adaptivity for educational games. Where the first implementation required the programmer to introduce adaptive behaviour in between the original game code, GAAOP provides a framework of aspects that allows the programmer to separate the adaptivity concern from the rest of the game. As a proof of concept, this general pattern was applied to a simple game that can serve as a test case for implementations in different languages and aspect systems.

The resulting framework is generic enough to be applied to any game, and can be extended with more specific reusable behaviour to lower the implementation effort, and by extend also the development cost.

Compared to other solutions, GAAOP stands out in separating the adaptivity concern from the original game code. The focus of other solutions lies more with interpretations of adaptivity, indicating that our work is complementary with existing work.

Keywords: Cognitive Psychology, adaptivity, educational games, Aspect-Oriented programming

Samenvatting

In deze thesis presenteren we een generieke methodologie, GAAOP, die gebruikt kan worden om bestaande educatieve video games adaptief te maken. We geven een gedetailleerd overzicht van het multidisciplinaire domein van adaptieve educatieve video games, opdat lezers met achtergrond in de Computerwetenschappen, of in de Cognitieve Psychologie, of in de Ingenieurswetenschappen het werk zouden begrijpen. We geven definities en interpretaties van adaptiviteit in het domein van Cognitive Psychologie en Virtual Reality, patronen en architecturen gebruikt in het domein van game ontwikkeling, en aspect-gerichte programmeer technieken die een oplossing bieden voor problemen omrent het programmeren van cross-cutting concerns.

De methodologie is inductie ontwikkeld, waarbij we een algemene methode ontwikkeld hebben gebaseerd op de ervaring van het manueel implementeren van adaptiviteit in een voorbeeld educatief spel, TuxMath, in de context van een initiële studie in het kader van het CAdE project. CAdE is een multidisciplinair onderzoeksproject omtrent cognitieve adaptiviteit voor educatieve videogames. Voor de implementatie van adaptiviteit voor TuxMath moesten we, als programmeur, adaptieve code manueel invoegen in de code die instaat voor de rest van het spel. Dit staat in contrast met GAAOP, waar er een framework wordt aangeboden dat programmeurs toelaat om de code die instaat voor de adaptiviteit te scheiden van de spel-code. Om aan te tonen dat het framework en de methodologie aan de doelstellingen voldoet, hebben we de generieke methode toegepast op een zelfgeschreven simplistisch spel dat kan dienen als een test-case voor framework implementaties voor verschillende talen en aspect systemen.

Het resulterende framework is generiek genoeg om toegepast te worden op elk spel. Om de implementatiekost te verminderen, kunnen we het framework uitbreiden met interfaces die specifieker zijn voor het spel of spel-genre waar we adaptiviteit aan willen toevoegen.

In vergelijking met andere oplossingen, onderscheidt GAAOP zich op het vlak van het scheiden van de adaptiviteit's code van de rest van de spel code. Bij andere oplossingen ligt de nadruk van het onderzoek voornamelijk bij de verschillende interpretaties van adaptiviteit, wat aangeeft dat de ideeën uit GAAOP complementair zijn met bestaand werk.

Sleutelwoorden: Cognitieve Psychologie, adaptiviteit, educatieve videogames, Aspect-Oriented programmeren

Acknowledgemends

I would like to express my sincere gratitude towards the people who supported me in the years that lead to this research and in writing this dissertation.

First, and foremost, I would like to thank Prof. Dr. Olga De Troyer for promoting my research and for continuously trying to keep me motivated to finish what Ive started. Besides countless proof-reads with content suggestions, I would have not been where I am today without the frequent meetings where I not only got feedback but also pep-talk to continue to work and mental support to help me with tackling my lack of concentration.

I would also like to thank Koen Homblé, Georgios Patsis and Frederik Van Broeckhoven for their availability and expertise before and during the pilot study that was conducted in the Sint-Anna school in Duisburg. Thanks also goes out to the school board of Sint-Anna for allowing us to conduct it there.

In her quest to keep me focused and motivated, Prof. Dr. Olga De Troyer allowed me to work in an office together with Pejman Sajjadi, who I would like to thank for sharing his expertise in the domain of educational games and adaptivity, as well as for frequent but well-needed distractions.

A special thanks also goes out to all my friends, the people of Infogroep, Swamp and BierKultuur, who shaped my interests to what they are today, and who also provided me with distractions during the year that made sure I stayed in touch with the university.

A very big thank you also goes out to my family; Luc, Colette and Ian who each supported me in their own way and allowed and motivated me to continue to work during the past year.

Finally, I am very grateful to have found an amazing girlfriend, Jill Dehasque, with whom I shared many laughs and tears in the past two years. In her own way, she was always understanding and supporting my study progress. I hope that one day I can return the favour.

Our greatest weakness lies in giving up. The most certain way to succeed is to always try one more time - Thomas A. Edison

Ben Corne
Londerzeel, 2013

Contents

Abstract I

Abstract (dutch) II

Acknowledgemends III

Contents IV

List of Figures VIII

List of Tables IX

Listings IX

Glossary X

1 Introduction 1
- 1.1 Context . . . 1
  - 1.1.1 Educational Games . . . 1
  - 1.1.2 Aspect-Oriented Programming . . . 2
  - 1.1.3 Web Games . . . 3
- 1.2 Research Question . . . 3
  - 1.2.1 Sub questions . . . 4
- 1.3 Research Approach . . . 5
  - 1.3.1 Explore an example game . . . 5
  - 1.3.2 Explore the game's source code . . . 5
  - 1.3.3 Map adaptive points to the source . . . 5
  - 1.3.4 Implement adaptive interventions and assessment with AOP 6
  - 1.3.5 Make the method generic . . . 6
  - 1.3.6 Define design limitations . . . 6
  - 1.3.7 Verify generality . . . 6
  - 1.3.8 Develop software support for applying the method . . . 7
- 1.4 Thesis Structure . . . 7

2 Background 8
2.1 Educational Games . . . 8
2.1.1 TuxMath . . . 9
2.1.2 Sumon . . . 9
2.1.3 Monkey Tales . . . 9
2.2 Adaptivity . . . 12
2.2.1 User model . . . 12
2.2.2 Adaptive interventions . . . 14
2.2.3 VR and adaptivity . . . 15
2.3 Game Architecture . . . 19
2.3.1 Runtime game engine architecture . . . 20
2.3.2 Game Engine Genres . . . 33
2.3.3 CAAT . . . 34
2.4 Aspect-Oriented Programming . . . 38
2.4.1 Example . . . 38
2.4.2 Concepts . . . 39
2.4.3 Implementations . . . 40
2.5 Background Summary . . . 48

3 Pilot Study 49
3.1 Study Summary . . . 49
3.2 Study Setup . . . 50
3.3 Adaptive Implementation . . . 52
3.3.1 Game conceptual structure . . . 53
3.3.2 Game engine structure . . . 54
3.3.3 Adding adaptivity . . . 56
3.4 Pilot Study Summary . . . 59

4 Proposed Method 60
4.1 Methodology . . . 60
4.1.1 Step 1 : Finding the base for adaptation . . . 61
4.1.2 Step 2 : Building the user model . . . 63
4.1.3 Step 3 : Defining the adaptations . . . 65
4.1.4 Step 4 : Linking the definitions of adaptivity to the game code . . . 67
4.1.5 Step 5 : Adding adaptivity to the game . . . 68
4.2 Summary . . . 73
4.3 Proof of concept . . . 74
4.3.1 Output comparison . . . 75
4.3.2 Implementation effort . . . 75
4.4 Proposed method summary . . . 77

5 Limitations 80
5.1 Game architectural limitations . . . 80

5.1.1 Object-oriented design style dependency 80
5.1.2 AOP dependency 81
5.1.3 Educational games domain limitation 81
5.1.4 Support for specific game genres 82
5.2 Adaptations 82
5.2.1 Adaptation thresholds 83
5.2.2 Applying settings that change the game-didactic ratio . . . 83
5.2.3 Applying settings to component code 83
5.2.4 User model DSL 84
5.2.5 Inter-aspect dependencies 84
5.2.6 Generic aspect interface 84
5.3 Limitations Summary 85

6 Related Work **86**
6.1 ELEKTRA 86
6.2 80Days 88
6.3 ALIGN 89
6.4 Related Work Summary 90

7 Future Work **91**
7.1 Aspect extensions 91
7.1.1 AdaptiveEngine aspect extensions 91
7.1.2 Configuration aspect extensions 92
7.1.3 Assessment aspect extensions 92
7.1.4 Adaptation aspect extensions 93
7.1.5 ApplySettings aspect extensions 93
7.2 Tool Support 93
7.3 Genericity Verification 93
7.4 Future Work Summary 94

8 Conclusions **95**

Bibliography **97**

Appendices **101**

A MathGame and GAAOP in Java **102**
A.1 be.bennit.mg 102
A.2 be.bennit.gaaop 107
A.3 be.bennit.mg.gaaop 112

B MathGame and GAAOP in JavaScript **119**
B.1 MathGame code 119
B.2 GAAOP framework 124
B.3 Adaptive MathGame 129

C Generic aspect interface in Java 133

List of Figures

2.1 TuxMath gameplay screenshot 9
2.2 Sumon gameplay screenshot 10
2.3 Monkey Tales gameplay screenshot 11
2.4 Marking a VR object (source: [8]) 17
2.5 Game engine reusability scale (source: [12]) 20
2.6 Runtime game engine architecture (source: [12]) 35
2.7 Gameplay foundations layer components (source: [12]) 36
2.8 Hypothetical game object model for PacMan (source: [12]) 36
2.9 Hypothetical game object class hierarchies (source: [12]) 36
2.10 A finite state machine for representing the behaviour of a light switch (source: `http://ai-junkie.com/architecture/state_driven/tut_state1_files/image002.jpg`) . 37
2.11 Join points of AspectJ (source: [20]) 41
2.12 AOP interface provided by dojox/lang/aspect. 44
2.13 Composition pattern for reusable trace aspect and specifying a possible binding (based on images from [6]) 45
2.14 Output from binding Trace composition pattern (source: [6]) . . . 46

3.1 The Self-Assessment Manikin (SAM) used to rate the affective dimensions of valence (top row), arousal (middle row), and dominance (bottom row) (source: [1]) 52
3.2 Overall structure of TuxMath. 53

4.1 The facets of adaptivity in hypermedia. Source: [2] page 3. 62
4.2 Generic user model for the proposed methodology 64
4.3 User model for adaptivity in modified version of TuxMath. 65
4.4 Model of the adaptations for the modified version of TuxMath. . . 66
4.5 Abstract aspects used for adding adaptivity to an educational game 70
4.6 Extract of the output produced by adaptive MathGame. 75
4.7 Extract of the output procuded by normal MathGame. 76
4.8 Design for MathGame 77
4.9 Design for the concrete aspects that add adaptivity to MathGame . 79

6.1 Screenshots of the ELEKTRA game (source: [19]) 87
6.2 Summarization of the ELEKTRA framework (source: [19]) 87
6.3 Conceptual separation of generic, abstract elements and game specific elements when using ALIGN (source: [33]) 89

List of Tables

2.1 Object-centric vs property-centric data view 23
2.2 State transition table for a guard. 27
2.3 Mapping AOP elements to composition pattern elements. 47

Listings

2.1 FSM for guard NPC in C++. 28
2.2 Pseudo-code for embedded FSM state transition rules. 29
2.3 Simple game loop for Pong. 31
3.1 Game loop for TuxMath 55
3.2 Assessment and adaptation interface 56
3.3 Updating the difficulty based on correctness and average height. . 58

Glossary

GAAOP : The methodology and framework for adding adaptivity to games by using AOP techniques.

AOP / Aspect-Oriented Programming : Programming paradigm that extends the OO-paradigm to capitalize on separation of concerns.

OO / Object-Oriented Programming : Programming paradigm that represents concepts as object with data fields and relations to other objects.

CAdE project : “Towards Cognitive Adaptive Edu-Game“, research project supported by the Vrije Universiteit Brussel (VUB).

Game flow : Intrinsic motivation, created by games, to continue playing.

One-size-fits-all : A teaching approach for groups of people, without taking their personal experience, skill or abilities into account.

Adaptive hypermedia : Adapt the system towards the needs of the user, based on a model of the goals, preferences and knowledge of each individual user.

User model : Represents the state of the user (user data), individual traits (usage data, such as learning style) and his context (user environment).

Learning style : The way learners perceive, process, store and recall attempts of learning.

Adaptive interventions : Adaptations to a system that tackle cognitive- and motivation-related issues.

VR environment : Virtual Reality environment, created by a 3D world.

Adaptation types : Adaptive interventions that target a single element from the VR environment.

Adaptation strategies : Complex adaptive interventions that target multiple elements from the VR environment simultaneously.

Game engine : This provides the runtime environment for running games. It provides the programmer with tools and interfaces to modify the same runtime environment.

FSM : Finite state machine, concept from the Artificial Intelligence domain, also used in games to represent the state of the game and model the behaviour of non-player characters.

Game loop : The core of the game engine or any game. It starts the interactive process that represents a game simulation.

CAAT : Canvas Advanced Animation Toolkit, a game engine for creating scene-based games in JavaScript.

Aspect : An aspect encapsulates a concern or subproblem that is composed with the other concerns to form a solution to the entire problem.

Join point : A specific place in the runtime control flow of a program.

Pointcut : Describes a set of join points, using the pointcut language that was specified by the AOP system.

Advice : The behaviour that is (partially) responsible for solving the concern encapsulated by an aspect.

Weaver : The technology that is responsible for injecting the aspect's advices into the global problem solution.

AspectJ/AspectC++/Dojo AOP : Implementations of aspect systems with respectfully Java, C++ and JavaScript as their component language.

Composition pattern : The composition pattern comes from the subject-oriented design model and allows software designers to compose entities from different concerns in one design via a set of UML extensions.

SAM : Self-Assessment Manikin, used to rate the valence, arousal and dominance of a subject.

JSON : JavaScript Object Notation, syntax to describe objects in JavaScript.

LoC : Lines of Code, the amount of lines that was needed to implement some functionality.

Metaprogramming : Programs that use and possibly modify data and behaviour that runs the program (metadata).

Chapter 1

Introduction

In the first chapter we describe the context of the thesis and the research question that is answered. We also give a summary of approach followed to answer the research question and sketch the structure of the rest of the thesis.

1.1 Context

This thesis is part of a joint project between three faculties at the VUBrussels. COPS research group from the faculty of Psychology and Educational Sciences, DSSP (ETRO) research group from the faculty of Engineering Sciences and WISE research group from the faculty of Sciences.

The name of the project is CaDE or Towards Cognitive Adaptive Edu-Game. The first goal of the project is to investigate the cognitive processing involved in educational video games and it's impact on learning. Finding out what aspects may influence cognitive processing and the impact on learning. The second goal is to investigate how we can adapt these cognitive processes.

The thesis is mainly situated in the domain of educational game development, Aspect-Oriented programming and web applications (more in particular web games). We briefly discuss each of these domains in the following subsections.

1.1.1 Educational Games

Educational games are games that could leverage the level of concentration and motivation (or flow[35]) created by games to achieve higher learning rates than traditional educational approaches. Games that challenge and reward learners could effectively produce higher motivation.

Digital games have the potential to take the skills and learning capabilities of the learner at hand into account, by reacting to them and adapting the game based on reactions and behaviour of the player. If these issues are taken into account, the education process will be more personalized. We call this personalized adaptive e-learning. Studies show that personalized learning trajectories result in more efficient learning curves[22, 45, 47].

Game development is expensive because of its complexity and time consuming (especially if realistic and fancy graphics are used). More complexity and development time is added when you go from a 2D world to a 3D world, as this requires 3D modelers. Adding a didactic goal to the game in order to provide education, drives the cost up even higher since ideally this aspect would be designed by teachers or at least with input of educational people. Many educational games exist already, ranging from very small and simple (e.g shooting) games towards more complex adventure games (e.g TuxMath[1], Sumon[2], Monkey Tales[3]), but as far as we are aware of they do not attempt to personalize the game for each individual learner. However, in the context of increased attention for a more individually oriented education, there is a market in making these existing game more reactive towards the learners preferences and skills, i.e. turn them into personalized adaptive educational games.

Concrete, we want to modify existing educational games so that they keep motivating the learner while playing order to keep his attention level high and and by making sure the challenge is appropriate for the learner's skill level.

1.1.2 Aspect-Oriented Programming

Aspect-Oriented Programming or AOP aims at bundling solutions to subproblems that would otherwise have to be implemented scattered out through an entire system. We call these subproblems aspects. Some canonical examples of such subproblems or aspects are monitoring and logging functionality[24].

Adapting educational games based on game and user feedback can be seen as a monitoring aspect where we react in case a threshold for a given feedback computed value is exceeded.

Typically introducing this kind of functionality in object oriented programming languages would require us to put code throughout the entire project. We call such a subproblem a crosscutting concern since it cuts through the entire project. However, using the Aspect-Oriented programming paradigm, all code related to this concern can be concentrated in one place.

[1] http://tux4kids.alioth.debian.org/tuxmath/

[2] http://www.ludei.com/sumon

[3] http://www.monkeytalesgames.com/

Having a clear distinction of all concerns increases readability of the code and thus on the long term also maintainability is improved. A drawback of aspects are that applying them relies on naming conventions and API stability.

1.1.3 Web Games

The evolution of the web allows us to offer applications through web browsers. With the growing interest for graphic toolkit support in browsers (CSS3, HTML5, WebGL,) the possibility comes to offer graphically more complex applications, such as games.

Graphic toolkits in a way offer basic primitives to make use of the graphics processor in computers. Libraries and frameworks extend these primitives and aim at facilitating coding certain types of graphic applications. For example the CAAT-framework[4] aims at supporting scene-based web games with animations.

Offering (educational) games through web browsers has many advantages. A non-comprehensive list of these advantages: a continually up-to-date service, ease of deployment and distribution, easier access to a large pool of collective data from both external sources and individual users[31].

In practice this could be useful to allow children to continue learning from any place. This would allow games to be integrated with the surroundings, provided there is Internet access. E.g. a game with the educational goal of teaching children about animals could be integrated with the setting of an animal farm, by having them walk around the farm with a tablet to answer little quizzes about the spotted animals.

1.2 Research Question

If we assume the learning rates are positively influenced by an adaptive learning environment, then we can formulate our research question as follow: **How can we add adaptivity to existing educational games in a generic way and with a minimal effort?**

Based on this research question, we can formulate a number of sub questions that can also be considered as objectives for the thesis. The sub questions are as follows:

[4] http://hyperandroid.github.com/CAAT/

1.2.1 Sub questions

What types of information can be obtained from a player while he is playing? The answer to this question is needed to be able to know what type of information need to be monitored during a game in order to be able to make the game adaptive to the reactions and behaviours of the player.

How can we facilitate gathering this information from a player? Not only do we need to know what needs to be monitored, we also should try to find out how we can do this. Some aspects can be easily monitored (e.g. reaction time), but others will be much more difficult (e.g. attention). In the context of this thesis we will not deal with different possible techniques to measure different user aspects, but rather we investigate how we can facilitate this information gathering.

How do AOP-techniques facilitate applying the method? Aspect-Oriented Programming (AOP) techniques allow for separation of cross-cutting concerns. The adaptation technique needed to come to personalized learning adaptive games could be considered as such a concern.Therefore we want to investigate if and how AOP can be used to add adaptivity to existing educational games.

Does the game's architectural design have an impact on the proposed method? Are there different architectures applicable to educational games and how applicable is our method for them? Different games may have a different architectural design. We need to investigate how this might have an influence on the method that we will propose. Also utilizing a framework often implies a shift in control of the control-flow from the programmer to the framework. In what way do we have to investigate if this influences the use of our method to implement adaptivity?

How does the proposed method map to the client-server architecture? Games in general use a version of the client-server architecture when they offer multi-player functionality. If educational games use this architecture, is our method still applicable to the game or how could we extend it if it is not?

What are the implications of implementing client-server on top of the web technology stack? As we are considering web games, it is necessary to take the client-server architecture of the web into account, or at least investigate the impact of this architecture on the proposed method.

What is the performance impact from using our method? Games should have a good performance, i.e. they should react fast. Therefore it is important to investigate if the application of the proposed method to modify the game will have an impact on the performance of the result game.

How can we automate implementing adaptivity using our method? The proposed method will provide a procedure on how to turn an existing game into an adaptive one. Automating this process as much as possible would provide great added value for educational game developers.

1.3 Research Approach

Our research approach to answer the formulated research question and sub questions, is an inductive approach. We will start with an existing educational game and develop a method to turn it into an adaptive game. Next we will generalize it. So we will actually build our method by example. We can split this process up in a number of steps.

1.3.1 Explore an example game

We look at the possibilities of an existing educational game. This is best done by playing the game so we get a good idea of where the difficulties (for the player) lie and where task repetition is found which could make the game boring over time. In this way, we also get an idea of the places where the game could assess the learner's skill level and motivation.

1.3.2 Explore the game's source code

We need to get a rough idea of the structure of the educational game. We do this by looking at the structure of the source code of our example game.

1.3.3 Map adaptive points to the source

Having identified game components as difficulties, task repetition and skill level assessment, we can look for the code pieces that are responsible for each of these points. Now that we know where in the code we can assess skill level and motivation, we can implement a method to track skill level throughout the game. We can insert assessment code wherever we can learn about the learners skill.

Next we can implement adaptive interventions by inserting code that takes into account the learner's current level of skill and motivation.

1.3.4 Implement adaptive interventions and assessment with AOP

Modifying code of an existing game is not always that easy to debug and to keep an overview of your modifications. Therefore we resort to Aspect-Oriented programming and try to implement the same functionality added in the previous step as aspects. In this case we have to write the join points on which aspects are applied by writing the mapping defined in the previous step to join point descriptors as defined by the Aspect language of our choice.

1.3.5 Make the method generic

This is the inductive step of the approach. To come to a method that is applicable to more than one game, we need to answer the following question : How did the game's design choices influenced how we implemented skill assessment and adaptive interventions?. We abstract our implementation up to the level where the design choices do not dictate our implementation anymore.

1.3.6 Define design limitations

Of course, some limitation on game design has to be considered, else we should abstract to the point where we define a game. These limitations will be discussed in a later chapter.

1.3.7 Verify generality

To test the generality of the proposed approach, we will take a second game whose design is within the earlier mentioned game design limitations and yet as far away from the original game as possible. Since it is hard to define as far away from the original, ideally, we would want to apply our method on as many different games as possible within the time limit for producing this thesis.

We make try to make this second game adaptive as well by applying our method. If the method is not applicable, we have to make the method more abstract / general. If this would make our method not feasible anymore, we have to limit our game design further.

1.3.8 Develop software support for applying the method

Having verified the generality of our method, we can develop software that aids us in turning an existing game into an adaptive one, e.g. a tool that generates a code skeleton in a given programming language for a specific type of adaptation and for a specific game.

1.4 Thesis Structure

In this **first chapter** we explain the context of the thesis, the research question of the thesis, the research approach followed and we sketch the structure of the thesis.

In the **second chapter** we provide background information related to terms and technologies used in this thesis. We explain in detail the following topics: educational games, adaptivity in educational games, game development in general, the CAAT-framework, AOP, AspectC++, AspectJS and designing reusable aspects.

In the **third chapter** we describe the first case study conducted on the game TuxMath. Besides the made adaptations, we also describe the deployment of the game in a school.

In the **fourth chapter** we describe GAAOP (Game Adaptivity using AOP), the method we propose to answer our research question. We also describe our second case study where we apply GAAOP to the game Sumon.

In **chapter five** we elaborate on the limitations of our research. This goes from the design of the targeted game to the performance impact induced by our method.

In **chapter six** we compare our findings with related work.

In **chapter seven** we describe future work both on GAAOP as well as on the topic in general.

In **chapter eight** we conclude by summarizing how and where each research question was answered. We also summarize our contributions, the limitations and the strong points of GAAOP.

Chapter 2

Background

In this chapter we introduce the concepts of educational games and game flow. We also describe three educational games of different genres, that are implemented using different technologies. Because of the differences, these games could be used to to verify the genericity of our methodology.

2.1 Educational Games

Educational games could create a flow needed to keep motivation and attention for a learner high. This is typically achieved by finding the correct mix of game and didactic elements.

Many such educational games exist already. Often they determine the ratio of game and didactic elements statically. We will list some games that do this and which are relevant to this thesis because they fit the requirements and boundaries of GAAOP, our method to implement adaptivity using AOP.

The most obvious requirement for modifying a game is that we need to be able to modify the source code of that game. The second (high-level) requirement is that the target educational game should contain situations or scenes where we can identify either didactic or game elements or both, in order to be able to modify the ratio of presence of these elements.

More low-level requirements or boundaries will be discussed in chapter 5 on page 80 about limitations of GAAOP.

2.1.1 TuxMath

TuxMath[44] is an open source educational game written in C with the SDL[25] multimedia libraries. TuxMath is aimed at children from ages 8 to 12. The goal of the game is to defend your penguin's igloos by solving sums and multiplications that are crashing down as comets towards the igloos as seen in figure 2.1 . The game rewards the learner by adding points to the total score after each wave of comets. The better igloos are defended, the more points are added. Difficulty is chosen by the learner. Difficulty settings are configurable via configuration files that define speed, range, and amount of comets and the mathematical operations that are used.

Figure 2.1: TuxMath gameplay screenshot

2.1.2 Sumon

Sumon[28] is an open source educational web game written in JavaScript with the CAAT framework (see section Game Development). Sumon combines eye-hand coordination with the sum operation. Learners are given a number and a field of numbers between 1 and 9. They have to sum up to the given number by selecting numbers in the field as seen in 2.2 . The more numbers used in the sum and the more distance between selected numbers on the field, the more points are rewarded. The learner loses if the time runs out before he forms the given number. The difficulty rises linear in function of the time.

2.1.3 Monkey Tales

Monkey Tales[41] is a 3D adventure game framework written in C++. There are a number of games making use of the framework, each targeting an age category going from 7 up to 11 year old children. Monkey Tales was developed by a european schoolbook publisher Die Keure and game developer Larian Studios. the games are based on years of research and was developed with the active participation of teachers, schools, universities and educational method-makers.

The learner is asked to solve mazes that lead from one room to the next as seen in figure 2.3 . This insight training is interspersed with smaller mathematical arcade

Figure 2.2: Sumon gameplay screenshot

games where learning is made more explicit. Learners are motivated to do the arcade games by rewarding them with monkeys that are put in the learner's zoo once an arcade game is completed. The arcade games are comparable to the two previously discussed games.

Figure 2.3: Monkey Tales gameplay screenshot

2.2 Adaptivity

Brusilovsky describes adaptivity as an alternative to the traditional one-size-fits-all approach[3]. This approach is applied in many domains.

For example a museum usually offers the same guided tour and the same narration to all visitors, although the user could have very different goals and background knowledge. It would be more effective to adapt the guided tour and the narration to each individual visitor or to a group of visitors.

Also in traditional education systems the one-size-fits-all approach is adopted. All students in a class have one teacher, and all have the same learning trajectory, regardless of their learning capabilities and personal skill-set and preferences. It is generally accepted, that personalized learning approaches would be much more effective.

More general we could say that while user population is relatively diverse, traditional software systems will present the same content and possibilities to all users. Systems that adapt to the user got attention with the advent of the web. These systems are known under the name of adaptive hypermedia.

Brusilovsky defines adaptive hypermedia as follows:

“Adaptive hypermedia systems build a model of the goals, preferences and knowledge of each individual user and use this model throughout the interaction with the user, in order to adapt to the needs of that user.”

This definition is also applicable to adaptive educational games. In the first chapter we mentioned the concepts of motivation, attention level, and competence. They can be compared to goals, preferences and knowledge. These three properties can be described in the the user model. The user model contains all information about the user relevant for adapting the system to the user’s needs.

Brusilovsky extended the definition of adaptive hypermedia to adapt not only to the user data but also to the usage data and to the user’s environment.

Typically, in a game environment we want to assess these properties in a non-invasive manner where possible, to avoid distracting the user from the goal or breaking the game flow.

2.2.1 User model

Information in the user model can be divided into the information about the user himself and the information on how the user is using the system (or data provided by the system). A careful analysis of what information will be maintained in the

user model is needed, as this will be the information to which the system will adapt. It has no point to maintain or collect information that will not be used.

User Data

This data can be retrieved by simple questionnaires or from previously entered interactions with the system. This can be done for the user goals, background knowledge, experience, and preferences. There are also other ways to collect this information, using some implicit acquisition technique. For instance by tracking and analyzing browsing behaviour, searching behaviour, or analyzing or detecting patterns in web server logs. An overview of such techniques can be found in [17] and [11].

Usage Data

Usage data can be seen as user properties (individual traits of users such as personality, cognitive factors and learning styles) that don't change or only change over a long period of time. Furthermore usage data is distinguished from user data if it has to be extracted using specially designed psychological tests rather than simple interviews.

Learning styles Earlier we mentioned learning styles as an individual trait of a user. Literature has a number of definitions for a learning style. One that is intuitive for non-experts in the domain of cognitive psychology is:

The way in which learners perceive, process, store and recall attempts of learning[14].

In our own words we would say that a learning style is the learner's best way of acquiring new knowledge. In practice, a learner always has a combined learning style with one dominant learning style[9].

A categorization of learning styles is made in [15] stating the name, the learner's categorization and how to assess the style. The categorization is left for the interested reader and can be found in [15], pages 3-4.

User Environment

The user environment came into play with the rise of mobile web-based systems. The location of where systems were used (museum, library, bookstore,) was no longer static, but changes depending from where the user accesses the system. Like the animal farm example explained in chapter I, section I, where children would walk around an animal farm with a tablet and get small quizzes about spotted

animals. We could use the user's direction of sight and movement to adapt the content displayed on the tablet.

2.2.2 Adaptive interventions

When developing an adaptive system, it is also necessary to determine how the system will adapt to the user. In the context of educational games, providing the correct ratio of game and didactic elements is only one type of adaptivity. In [40] we find a categorization of adaptive interventions as they call it. All adaptations are either cognitive or motivational. We summarize their categorization as we will compare this in later chapters to adaptations made by GAAOP.

Cognitive interventions

Cognitive interventions strive to enhance cognitive abilities and support the learner based on his/her user model. A number of subtypes of this type of intervention are distinguished. However, the line between these subtypes can be blurry.

Meta-cognitive interventions - This type of interventions attempts to provoke the learner's reflection about his/her own abilities, thinking process and confidence. E.g. ask the question "Does this solution make sense?".

Competence activation interventions - When a learner is stuck in a certain task because results led to the assumption that the learner possesses the necessary skills, the temporary inactive skills could be reactivated. E.g. hint the learner "we have come across this issue already before".

Competence acquisition interventions - When the system concludes that the learner lacks certain skills, the system could provide the necessary information.

Problem solving support - This intervention consists of providing support in an ongoing problem solving process via hints and indications that bring the learner closer to the solution.

Dissolving interventions - This intervention consists of providing the solution to a task which the learner did not complete within reasonable number of actions / time. This assures the game flow continuity.

Progress feedback - This intervention provides the learner with information about learning progress of the game (through NPC or scoring mechanisms). This fosters monitoring and reflection on the learner's own performance.

Cognitive assessment interventions - If the non-invasive assessment of skills led to unclear or ambiguous results after a number of actions, the clarity could be improved with explicit questions or problems.

Motivational Interventions

Motivational interventions strive to enhance and retain the learner's motivation and engagement on a high level.

Praising interventions - This consists of congratulating the learner in case of success. E.g. in Monkey Tales, upon successfully completing a mini-game, the learner is awarded with a monkey to put in his petting zoo.

Encouraging interventions - This intervention could be applied in case of failure. To promote further attempts to find the solution, the learner is encouraged to try again.

Attributional interventions - These interventions go one step further than an encouraging interventions. The goal is to foster self-worth in case of lacking confidence or dysfunctional attributional styles. A motivational training is conducted where we provide feedback for success (e.g effort and ability) and failure (e.g lack of effort and bad luck).

Incitation interventions - The purpose of this intervention is to foster motivation by announcing pleasing outcomes as rewards.

Affective interventions - This type of interventions fosters emotional-affective aspects of the game and social interaction with other game characters.

Attention-catchers - If the system detects a decreasing attention through interpretation of the learner's actions, we could introduce unexpected changes or incidents to increase variability and keep the game interesting.

Motivational assessment interventions - Similar to the cognitive assessment interventions, we could have an explicit questioning (e.g. a dialogue with an NPC) to make the learner's motivational state conclusive.

2.2.3 VR and adaptivity

As three-dimensional (3D) games are gaining in popularity, we could leverage 3D to increase motivation in educational games. This is done by games such as Monkey Tales[41] and 80Days[30]. The virtual reality (VR) environment created by a 3D world has some drawbacks compared to 2D worlds. Navigation in 3D worlds is more complex and requires learners to grow accustomed to new navigational controls. The richness added by 3D representation might overwhelm the learner by distracting him or even making him lost or in the virtual reality environment. We can see that we could benefit from dynamically adapting the environment to counter the added complexity of VR.

Olga De Troyer et al.[8] propose, as they call it, a set of adaptation types and adaptation strategies based on the different components that make up a virtual re-

ality environment. They focus on desktop VR, which they define as a 3D computer representation of space, displayed on a screen, in which users can move their viewpoints freely in real time and perform several actions. We summarize the components making up a virtual reality environment used in the proposed adaptations:

The scene - The 3D space in which objects are located. It contains lights, viewpoints and cameras. Sometimes there are properties that apply to all objects within a scene like gravity, causing all objects to fall down at a given speed until they hit the ground.

Objects - Objects have a visual representation within the scene. This representation is often in 3D but sometimes also 2D occur in the 3D space. Properties like color, texture, size, position and orientation are associated to the objects and reflected in the visual representation.

User Avatar - User avatars are the user(s) in the 3D world. They can be represented explicitly (by an object) or implicitly in which case the camera's viewpoint is used to show the current user's position.

Behaviours - Behaviours may reflect real life behaviours on the objects. For example objects can move, rotate, transform, ...

User interaction - Users can trigger an object's behaviour (e.g. by clicking it with a mouse).

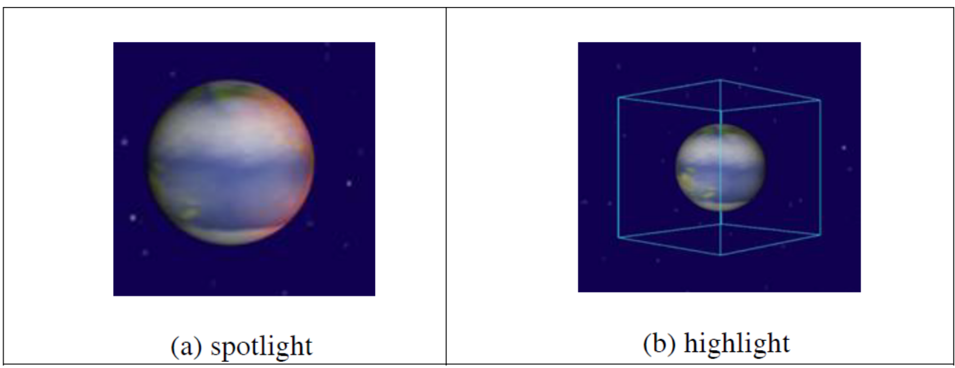
An example virtual reality learning environment is a virtual solar system. The learner navigates through our solar system (scene) to learn about planets, moons and stars (objects) in our galaxy. Every planet has an associated text which can be shown as an annotation in the VR world. The surface of a planet is visualized by a texture. The learner flies through the solar system by using the camera's viewpoint as the current user's position (user avatar). The planets rotate around the sun (behaviour). The learner can also click on planets to select it and learn more about it (user interaction) by seeing it's annotations or moons.

Adaptation Types

Earlier we mentioned adaptation types and adaptation strategies. Adaptation types are adaptations to a single VR component. We describe adaptation types for each type of component.

Object adaptations Objects in a 3D world have a visual representation (texture, color, shape,). We can adapt these visual properties to draw attention to or withdraw attention from the objects. In our example we mark objects that we haven't yet learned about and hide those that we have seen and learned about. Figure 2.4

shows two methods for marking objects in a VR world to increase visibility and thus drawing attention to it. (a) marks an object with a spotlight, giving it more visibility, while (b) marks an object by highlighting it. This can be seen as an attention-catcher mentioned in the previous section.



(a) spotlight (b) highlight

Figure 2.4: Marking a VR object (source: [8])

Behaviour adaptations Objects that have behaviour are active in a VR world (e.g. planets rotating around the sun). Having all behaviours active at all times might confuse or distract the learner. We can adapt these behaviours by enabling, disabling or modifying them.

User interaction adaptations In a VR world there are multiple ways to interact with an object. In first-person shooter games we can interact with other object by shooting them. In the solar system example we can interact with planets, moons and stars by passing by them or clicking them. Disabling an interaction with an object could be used to foster the learner's focus on the objects he is allowed to interact with. In some form this is also an attention-catcher. Enabling an interaction with an object could be used to reward the learner after having studied an object. This is a praising intervention.

User avatar adaptations Avatars are used to represent the user's point of view in the VR world, either implicit or explicit. Explicit representations can take the form of a person but can be anything that comes to mind. Avatars also have behaviours that allow them to navigate in the VR world. We can think of behaviours such as walking, flying or jumping. Both the avatars representation and behaviours could be adapted to the learner's preferences. E.g. we could make the avatar fly slower or make it look more like the learner.

Adaptation Strategies

Adaptation strategies are more high-level adaptations that involve more than one VR component. As we are interested in VR learning environments we focus on strategies with learning purposes. We can categorize these strategies in five groups.

Navigational - Strategies that relate to how or where we navigate are navigational adaptation strategies. We could restrict the learner to only being able to navigate to a list of objects or guide the learner through the VR learning environment via a tour guide. We could also restrict behaviours and interactions while we are navigating. This fosters the learners attention on navigation. Another possibility is to provide the learner with tips or hints while navigating by marking nearby objects.

Object groups - Strategies that relate to the visual representation of a set of objects. We could hide a group of objects and start showing them gradually as the learner obtains more knowledge. If hiding objects is not desirable (e.g. because we want to show connections between objects), we could mark the objects that require the learner's attention.

Conditional display - Strategies that relate when to show or hide objects. We could limit the amount of times objects are shown to the learner or only show objects after a condition is satisfied. The latter could be used to keep the VR learning environment interesting to avoid the learner from being bored. This is an encouraging motivational intervention.

Conditional behaviour - As with displaying objects we could limit the amount of times a behaviour is triggered to avoid the user from wasting time on the same object after he already learned about it. We could also state when a behaviour should be executed (e.g. start rotating moons around a planet after the planet has been studied). Finally we could also adjust the speed of a behaviour to allow the learner to better observe what is happening.

Conditional interaction - Again as with displaying objects we can limit the amount of interactions possible and conditionally state when an interaction can occur.

2.3 Game Architecture

While there are many different computer games each with their own concepts, design and implementation, many games exhibit the same coarse-grained architectural patterns. Jason Gregory describes these patterns in his book Game Engine Architecture[12]. Most two- and three-dimensional video games are examples of what computer scientists would call *soft real-time interactive agent-based computer simulations*:

Simulations - Games model a subset of the real world (or an imaginary world). A model is an approximation and a simplification of reality. As such a model is a simulation of the modeled world.

Agent-based - In the simulation, a number of distinct entities can interact, entities such as vehicles, fireballs, characters, . Because of the agent-based nature of many games, the object-oriented programming model is often a good choice as implementation language.

Interactive - Games respond to user input by modifying the model.

Real-time - All video games are temporal simulations, the game's state changes over time. To make the simulation more realistic, games introduce deadlines for some components (e.g. show 24 frames per second to provide the illusion of motion, make characters recompute their next action every second to make them look smart). Unlike in some systems (e.g. power plant cooling), missing deadlines in games is not catastrophical, hence games are soft real-time systems.

Early generation games, such as the arcade hall game *PacMan*, were designed and implemented for one specific type of hardware and worked only on that hardware. In the mid 1990's the term game engine made its entry, originating from first-person shooters such as *Doom*. Doom was architected with a reasonably well-defined separation between the core software components (audio system, 3D rendering, collision detection, ...) and game assets (artwork, object models, game worlds, rules, spoken text, ...). The value of this separation became clear when developers began to create new games by taking old games and creating new art, world layouts, character and object models and different game rules with minimal changes to the core software components. The better the separation and the more customizable the engine, the more possibility there is to reuse the engine for different games. Jason Gregory defines the term game engine as software that is extensible and can be used as the foundation for many different games without major modification.

Figure 2.5 shows game engines from different games and how reusable they are (or were) to create new games. Engines on the right side of the reusability scale would be capable of creating virtually any game content imaginable. It is safe to

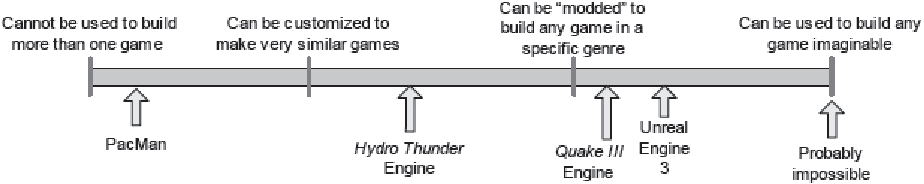


Figure 2.5: Game engine reusability scale (source: [12])

say that the more general purpose a game engine, the less optimal it is for running a particular game on a particular platform. While increasing hardware and algorithm performance softens the differences between graphic engines for different genre of games, the trade-off between generality and optimality still exists.

2.3.1 Runtime game engine architecture

A game engine generally consists of a tool suite (to create and manage content) and a runtime environment (for actually running the game). Figure 2.6 shows the minimal components that are present in the runtime environment of a typical 3D game engine. 2D game engines are simpler but still many components overlap with ideas used in 3D engines. Game engines are usually built in layers where the higher layers depend on the lower layers, but not visa versa, since that would introduce a cyclic dependency, leading to undesired coupling and all of it's disadvantages. In the next subsections we explain the components relevant to this thesis, i.e. those that are used to make a game adaptive. A more complete overview can be found in [12].

Third-Party SDKs and Middleware layer

Most engines leverage one or more third-party software development kits (SDKs) and middleware that provide abstraction for concepts needed in these engines, such as:

- Data structures and algorithms to manipulate them, e.g. C++'s standard template library.
- Hardware interface for rendering graphics, e.g. OpenGL for portable 3D graphics rendering.
- Collision detection and physics, e.g. Nvidia's PhysX.
- Character animation, e.g. Granny (3D model and animation exporters).

- Artificial Intelligence, e.g. Kynapse provides the AI building blocks such as pathfinding, and walking graph structures for non-human characters.

Simple DirectMedia Layer (SDL) is described as a cross-platform multimedia API[25]. It provides interface for among others video, events, threads, timers and audio officially only for the Win32 and Linux platform, however there are also unofficial ports to different platforms. We could say that SDL is somewhere in between a third-party SDK and the platform independence layer in the overview of a game engine's architecture (figure 2.6).

Front end layer

Both 2D and 3D game engines often provide a 2D graphics overlay for various purposes, including:

- A heads-up display (HUD), e.g. to show the available actions or the player's inventory
- In-game menu's, e.g. to select the desired level or difficulty.
- Full-Motion video (FMV), e.g. to play cutscenes that tell the storyline.

Collision and Physics layer

Collision detection is important for every 3D game. Without it, it would be hard to impossible to know when to interact with other objects in the world and objects' representation could overlap from time to time. Physics or rigid-body dynamics make dynamic simulations more realistic by taking the kinematics of and the forces that work on rigid bodies into account. In most recent games, third-party SDK's are used rather than writing a custom collision / physics engine.

Gameplay foundations layer

The term gameplay refers to everything related to how the virtual world and it's objects behave, including the abilities of player characters (known as *player mechanics*), non-player characters (NPCs) and other objects in the virtual world. Gameplay is typically implemented in either the same programming language in which the rest of the engine is written, or in a scripting language defined and interpreted by the game engine, or as a combination of both. If present, the gameplay foundations layer provides abstraction from the low-level game-engine details, upon which game-specific logic can be implemented easier. If a line can be drawn between where the game starts and where the game engine ends the gameplay foundations layer would be right under or right above it depending how game-genre

specific the layer is implemented. The components of the layer are shown in figure 2.7 . We discuss the components below.

Game World and Game Object Model The virtual *game world* consists of static and dynamic elements. The collection of object types that make up the game world is called the game object model. These include player characters, NPCs, weapons, projectiles, terrain, lights, static background geometry etc. There are three aspects to the game object model, the abstract game object mode, the software object model and the runtime game object model.

The *abstract game object model*, often modelled in an object-oriented style, is the model advertised to game designers via the world editor (either a scripting language or a graphical creation tool). It defines the various types of dynamic elements that can exist in the game, how they behave and what kinds of attributes they have.

The *software object model* refers to the set of language features, policies and conventions that are used to implement a piece of object-oriented software and is intimately tied to the game world model. It answers questions such as the style in which the game engine is designed, which programming language will be used, how objects will be referenced and identified (pointers, name, smart pointers, handles). Often, but not always, game engines use an Object-Oriented language to translate better from the abstract game object model..

The *runtime game object model* provides a faithful reproduction of object types, attributes and behaviours advertised by the tools made available to game developers. The runtime model is an implementation of the abstract game object model. Although designs vary widely, most game engines follow one of two basic architectural styles: the object-centric or property-centric style.

Object-centric In an object-centric architecture, each abstract game object is represented at runtime by a single class instance or a small collection of interconnected class instances, encapsulating its attributes and behaviour. In this style there are once again many variations in design. A commonly used pattern is that of a monolithic class hierarchy, where every class eventually inherits from the same class. A hypothetical design for the PacMan[1] game object model is given in figure 2.8 . In this picture we see that every object inherits from GameObject via one or more is-a relationships (inheritance), depending on which behaviour the game object needs.

A problem with such monolithic class hierarchy is the so-called *bubble-up effect*. In the initial design, the root classes are usually very simple, exposing only a minimal feature set. But as more game objects are added that require the same behaviour,

[1] http://en.wikipedia.org/wiki/Pac-man

Object-centric	Property-centric
Object 1	**Position**
Position=(0,3,15)	Object 1=(0,3,15)
Orientation=(0,43,0)	Object 2=(-12,0,8)
Object 2	**Orientation**
Position=(-12,0,8)	Object 1=(0,43,0)
Health=15	Object 3=(0,-87,10)
Object 3	**Health**
Orientation(0,-87,10)	Object 2=15

Table 2.1: Object-centric vs property-centric data view

this behaviour bubbles up in the hierarchy, even though not all game objects require the behaviour. The Actor root-class of Unreal Tournament 2004's game engine is an example of the bubble-up effect. It contains data and behaviour for accessing all the low-level systems such as network replication, audio effects, physics, animations, etc. Concrete actors do not always need all this data and behaviour. For example an invisible zone that detects if people walk in do not need to be rendered.

Figure 2.9 shows two hypothetical game object models. Model (a) uses only inheritance to specify an objects behaviour, while model (b) favors composition (has-a relationship) over inheritance (is-a relationship). Model (b) provides a solution to the bubble-up problem by isolating the various features of a GameObject into independent classes that are instantiated in concrete GameObjects where needed.

Another solution is using multiple inheritance and mixins to specify exactly which behaviour is needed, but on their turn these come with more problems that are out of the scope of this discussion[2]

Property-centric In the object-centric style, the programmer thinks in terms of objects that contain attributes and behaviour. In the property-centric style, the programmer instead thinks in terms of attributes. Table 2.1 shows a comparison of how data is associated to objects in the object-centric view and how it is associated in the property-centric view. In the latter, we define all properties that a game object might have. Then for each property, we store associations between game objects that have the property and the values of that property. Much like a column in a table in a relational database.

Where the behaviour is defined, that uses the attribute values specified in the properties-table, is another design choice that each game engine has to make. There

[2] The interested reader can find the problems related to multiple inheritance and game design in [12], pages 720 to 722.

are two generally used approaches.

In the first we implement property data and behaviour together in a property cclass. This design is very close to pure component models, a special case of the object-centric style. Pure component models remove all the functionality from the root GameObject and move them into separate classes. The GameObject class is possibly removed and the component classes are linked only indirectly via a reference or identifier for the game object.

The second approach is to implement the behaviour via scripts and store the data in a database-like structure. The scripts respond to events that are triggered when certain changes in the data are observed by the engine. Events and scripts are discussed further in the following sections.

Event System Events in games can be seen as anything of interest that happens during gameplay. E.g. an explosion going off, a player getting hit or a power-up item that is picked up. Games generally need a way to notify interested game objects of the event occurrence (*event listeners*) and provide the means for those objects to respond to the event in various ways (*event handlers*).

Events (or as called in some engines *messages* or *commands*) are comprised out of a type and the arguments. The type tells the game objects what event occurred (e.g. explosion) and the arguments provide more information about this specific occurrence (e.g. location, radius, . . .). Encapsulating events provides some benefits over the alternative solution, where game engines do not explicitly use events and event handlers but place function calls where the event occurred.[3]

Distinguishing between different types of events can be achieved in many ways. In one approach event types are mapped to integers (with a language construct like *enum* in C++), which can be used to compare event types very fast. C++'s enum require the event types to be centralized and known in advance. Adding new event types requires you to recompile the game engine. For this reason, some engines choose for a different approach, which is allows data-driven approach to add, edit and delete events. This is achieved by representing event types by strings (or for performance reasons their hashed values are sometimes used). To avoid typos in string names some engines provide an event-type database and tools to manage it, ensuring no two events overlap and providing checks to see if an event type string is valid.

To pass event arguments around some common patterns are used. A first pattern derives a new type of event class for each type of event, holding the data that is associated to this event. A second commonly used pattern is to have generic event classes that have different data type members. Concrete events use type that

[3]The benefits from encapsulating events are out of the scope of this paper. The book Design Patterns: Elements of Reusable Object-Oriented Software calls this the *Command Design Pattern.*

adheres closest to the slots they require. This is used for game engines that target hardware with memory restrictions since it allows the technique of *memory pooling* instead of allocating fresh memory each time a new event is created. A third pattern tackles the issue of order dependency both previous patterns suffer from, meaning that both sender and receiver have to know the exact order of the arguments. The problem is solved by presenting the arguments as key-value pairs.

It is reasonable to say that most game objects do not need to respond to every type of event. Most game objects are only interested in a small set of event types. Some game engines targeted at small and simplistic games use the naive solution and broadcast all events to all game objects. When the number of event types and number of game objects is scaled up, for more complex games, this approach is not feasible. Such game engines provide game objects with the means to register interest for a given type of event and broadcast events only to the game objects that are interested in them. An extension to handling events immediately after being sent is to keep track of an event queue and allow them to be handled at an arbitrary future time.

Scripting System In game engines scripting is used a mean to customize and control the behaviour of the game engine. It enables a data-driven approach towards game development. Scripting languages in games are high-level programming languages that are easier to use than the underlying language the engine implemented in and that provides access to most of the commonly used features of the engine. Often they can be used by programmers and non-programmers alike to develop a new game or customize an existing game. There are two general uses for scripting languages in game engines. The first is for data-definition, where (declarative) scripts are used to describe data that forms the content of the game. The second form, and the form relevant to this thesis, is runtime scripting. Runtime scripts are used to implement gameplay features by extending the engine's game object model and other engine systems.

Game engines have to make a choice what scripting language(s) to use. Some engines define their own languages (such as Quake's QuakeC and UnrealEngine's UnrealScript) that trades an increased maintenance cost and training cost for more flexibility and customizability for script programmers. Other games and game engines use existing scripting languages such as Python, Lua and Pawn. Whatever the choice, the scripting language is connected to the engine in some way. The architecture of this connection determines the role the scripting language has within a game engine. We list the most commonly used architectures below.

Scripted callbacks In this architecture, the largest part of the the engine's functionality is implemented hard-coded in the native programming language, but certain bits are customizable by placing function hooks or callbacks. These are func-

tions written in either the native language or a script language, supplied by script programmers, that are called by the engine. For example, when updating the game objects in the game loop, an optional callback can be called to customize the update of a specific object.

Scripted event handlers Scripted event handlers are a special case of hook functions. In the earlier mentioned event system, event handlers can be registered by interested game objects to react to an occurrence of a relevant event. For example, a function can be supplied to customize the behaviour that is called each time an object is created. This callback can be used to count the amount of NPCs currently in-game or to prompt the player with a warning that a certain type of NPC entered the arena.

Extending game object types or defining new ones Some scripting languages allow the script programmer to extend or define new game object types, possibly by inheriting behaviour and state from more generic classes written in the native language. This architecture allows the programmer to customize the game a lot more in terms of which game objects that can be represented by the engine.

In a property-centric game object model, it only makes sense to allow new properties to be created. In this model, behaviour can be customized through event handlers.

Script-driven engine systems One step further than extending game object types would be by defining all game object types through scripts. In this case, the only link with the game engine is when a script calls the native engine code of lower-level engine components (such as the renderer or the physics engine). Other engine systems could also be written using scripts, or we could say that the engine is driven by scripts.

Script-driven game Some game engines flip the relationship of scripting language and native language. These engines use script code to run the entire game and use components, that require high-performance, as a library inside scripts. Panda3D is an example of such an engine, where programmers can choose to program the games entirely in the interpreted scripting language Python and the native engine, implemented in C++, is called by script code.

High-Level Game Flow While the game object model only allows us to define which objects exist in the game world and how objects behave individually, it says nothing about the player's objectives and what happens if objectives are completed or when the player fails at them. In game engines this is often implemented by

Current State	Condition	State Transition
RunAway	Safe	Patrol
Attack	Weaker	RunAway
Patrol	Threatened AND Stronger	Attack
Patrol	Threatened AND Weaker	RunAway

Table 2.2: State transition table for a guard.

some form of a finite state machine. Finite state machines or FSMs are often used in the domain of artificial intelligence but also found their way into other domains such as game development. In the book Programming Game AI by Example[4] a descriptive definition of FSMs is given:

A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.

A FSM decomposes an object's behaviour into smaller, more manageable, chunks or states. A very simple finite state machine would be that of a light switch which can be on or off at any given time. Input to both states comes under the form of the switch being flipped resulting in a state transition to the other state. The out state also outputs light while the FSM is in that state. This FSM is shown in figure 2.10 . There are many ways to implement such FSMs.

A naive approach would be to use a series of if-then statements whose bodies define the behaviour associated to the state that led to the if-branch. A slightly tidier, and more often used approach is using a switch statement over the state type. Example 2.1 shows an implementation of a FSM that could be used to capture the AI behaviour of an NPC character.

Another mechanism to organize states and transitions is a state transition table. This is a table of conditions and states those conditions lead to. The table is queried at regular intervals (e.g. by the game loop) to make the necessary transitions if conditions are met thanks to stimuli from the game environment. Table 2.2 shows an example of such a state transition table.

A third mechanism is to embed the rules for state transitions within the states themselves. An agent or procedure keeps track of the current state and at periodic intervals (e.g. game loop) tells that state to execute its state transition rules. The agent or procedure also provides an interface to the state transition rules to change the current state. Example 2.2 shows such a mechanism in pseudocode.This pattern is know as the *State Design Pattern*.

```cpp
enum StateType { Runaway, Patrol, Attack }
void Agent::UpdateState(StateType CurrentState) {
  switch(CurrentState) {
    case state_RunAway:
      EvadeEnemy();
      if(Safe())
        ChangeState(state_Patrol);
      break;
      case state_Patrol;
      FollowPatrolPath();
      if(Threatened())
        if(StrongerThanEnemy())
          ChangeState(state_Attack);
        else
          ChangeState(state_RunAway);
    case state_Attack:
      if(WeakerThanEnemy())
        ChangeState(state_RunAway);
      else
        BashEnemyOverHead();
      break;
  }
}
```

Listing 2.1: FSM for guard NPC in C++.

```
1  class State { void Execute (Troll troll); }
2
3  class Troll {
4    State currentState;
5
6    void Update() {
7      currentState.execute(this);
8    }
9
10   void changeState(State newState) {
11     currentState = newState;
12   }
13 }
14
15 class State_RunAway < State {
16   void execute(Troll troll) {
17     if(troll->isSafe())
18       troll->ChangeState(new State_Sleep());
19   }
20 }
```

Listing 2.2: Pseudo-code for embedded FSM state transition rules.

Artificial Intelligence Foundations A final component of the game foundations layer is the Artificial Intelligence Foundations layer. If present, basic elements that are often present include path finding (e.g. A*-algorithm), perception systems (e.g. line of sight), and some form of memory. Some systems (e.g. Kynapse) provide a more sophisticated interface but these systems have not been around very long so high-level patterns amongst them have not been excessively studied. On a lower level though, FSMs are also often used in the context of Game AI. The mechanisms explained before are also used in game AI FSMs.

Game Loop

As seen in the previous sections, a game is composed of many interacting subsystems. Most of these systems require periodic servicing while the game is running. This means they require some behaviour to be periodically run. The frequency or rate at which these subsystems have to run varies. For example, the animation system has to run at 30 Hz or 60 Hz (cycles per second), to provide fluent-looking animations, synchronized with the rendering loop. AI systems often needn't be serviced at such high rates but might be ok to run only once every two seconds. There are many ways to coordinate synchronization and consistent periodic updating of subsystems, but at the core they all boil down to one or more simple loops. This loop is called the *game loop* or the main thread of control. We describe three architectural styles for these loops that are relevant to this thesis.

Simple Game Loop In it's most simplistic form, a game loop consists of running all the subsystems subsequently. In example 2.3 , a gameloop is implemented in pseudocode that runs the game Pong[4]. Pong is a well-known video arcade game where a ball bounces back and forth between two paddles that can move. A player controls one of the paddles and the other paddle is controlled either by another player or by the computer.

The pong example starts by initializing all the subsystems via a call to `initGame`. Then an infinite loop is started (that continues until control flow is interrupted). In each loop iteration we poll for new input from the player and process it via a call to `readHumanInterfaceDevices`. Next we check if the player wants to exit the game and interrupt the control flow of the game loop if this is the case. If the input indicated to move a paddle, that paddle is moved. In each iteration the ball also advances further in the way it was advancing. If the ball collides with a paddle it changes sides. If the ball collides with a side, this is registered and later checked to give the player from the opposite side a score increase. After all these player mechanics are processed, the playfield is rendered and shown to the player(s). We can summarize the simple game loop structure as follows:

[4] http://en.wikipedia.org/wiki/Pong

```
void main() {
  initGame();
  while(true) // game loop
  {
    readHumanInterfaceDevices();
    if(quitButtonPressed())
    {
      break; // exit the game loop
    }

    movePaddles();

    moveBall();

    collideAndBounceBall();
    if(ballImpactedSide(LEFT_PLAYER)) {
      incrementScore(RIGHT_PLAYER);
      resetBall();
    }
    else if(ballImpactedSide(RIGHT_PLAYER)) {
      incrementScore(LEFT_PLAYER);
      resetBall();
    }

    renderPlayfield();
  }
}
```

Listing 2.3: Simple game loop for Pong.

1. Initialize subsystems
2. Start the game loop
    (a) Poll for input
    (b) Process input
    (c) Advance game simulation time
    (d) Apply game rules
    (e) Render the game

Components like AI, sound and network replication can be added in this game loop without any structural change. However, for more complex games that have servicing frequency requirements for some subsystems this structure is too simplistic.

Callback-Driven Frameworks Most game engine subsystems and third-party game middleware packages are usable as libraries. A library is a set of functionality that can be called as the programmer sees fit. In contrast, some game middleware and game engines are structured as frameworks. A framework is a partially-constructed application that, when used, requires the programmer to implement the missing details to create the complete application. When using frameworks, the programmer has little or no control over the overall control flow of the application. Framework game engines manage the overall control flow by providing a main game loop that is largely empty. The game programmer customizes the game loop by providing callback functions that are called by the framework when the framework sees it best fit. Ogre3D is an example of such a framework[36]. Amongst other function hooks, it provides hooks for when a game frame starts and when it ends.

Event-Based Updating In section 2.3.1 about event systems, we introduced a system that allowed game objects to register their interest in occurrences of a specific type of events. Some engines leverage the same system to implement periodic servicing of some or all of their subsystems, if the event system allows events to be posted in the future (events do not have to be handled immediately, but can be queued for later delivery). The engine can queue events that indicate which subsystem should be serviced at 1 / *rate* seconds in the future (e.g. queue an *AnimationService* event 1/30 seconds in the future).

2.3.2 Game Engine Genres

Game engines are typically designed for building a specific genre of games, although sometimes engines are used to create games of other genres (e.g. UnrealEngine was designed for creating FPS shooters, while it has also successfully been used for third-person platform games). This design per genre reflects the different requirements for each game genre. Due to technological advancements, differences between genres that arose because of optimization concerns are disappearing, allowing game engines that were initially not built with those requirements in mind to create games of the genre that used to have those requirements. As the optimization requirements fade, games start to incorporate minigames of a different genre in their game. We discuss the engine differences for genres relevant to this thesis below.

Arcade games

Arcade games constitute a broad game genre that comes from the video hall coin-operated arcade machines. Compared to modern games, these games have less emphasis on graphical requirements. An appealing factor to them is the use of real-life attributes such as guns, joysticks and steering wheels to immerse the player more in the game[5]. Arcade game engines put more emphasis on human interface devices, level difficulty, level transition and scoring mechanisms.

Third-Person games

In third-person games, the player follows an avatar in a 3D world over the character's shoulder (unlike first-person shooters where the player is immersed in the character). Compared to first-person shooters, third-person game developer emphasis lies more on the main character's abilities and animations, though some engine requirements are shared:

- Efficient rendering of large 3D virtual worlds
- A Responsive camera control and aiming mechanism
- High-fidelity animations of the player and usable items or character gear.
- A forgiving character motion and collision model
- Small-scale online multiplayer capabilities (matchmaking, network replication)

[5] http://en.wikipedia.org/wiki/Arcade_game#Technology

Platform games

Platformer is the term applied to games where the player's character jumps from platform to platform. Technologies provided by engines that target this game genre include:

- Moving platforms, ladders, ropes and other locomotion modes (moving from one place to another)
- Puzzle-like environmental elements
- A third-person follow camera for graphically more advanced platformers.
- A complex camera collision system that ensures the view never collides with background geometry.

Some well known platformers from the 2D era are *Donkey Kong* and *Super Mario Brothers*. The 3D era includes platformers like *Crash Bandicoot* and *Rayman*. Third-person platformer game engines also inhibit the requirements posed for third-person games.

2.3.3 CAAT

CAAT stands for Canvas Advanced Animation Toolkit. It is a game engine for creating scene-based games in the JavaScript programming language. Like Ogre3D it is a callback-driven framework that allows game developers to create games that go from one scene to another. The engine features, amongst others, the following technologies[43]:

- Actors, for representing game objects that can send and receive events.
- Scene transitions.
- Various data structures (containers).
- Browser independence by allowing different rendering technologies.
- Management of game object lifecycle.
- Timers and time management
- Matrix transformations
- Game loop hooks

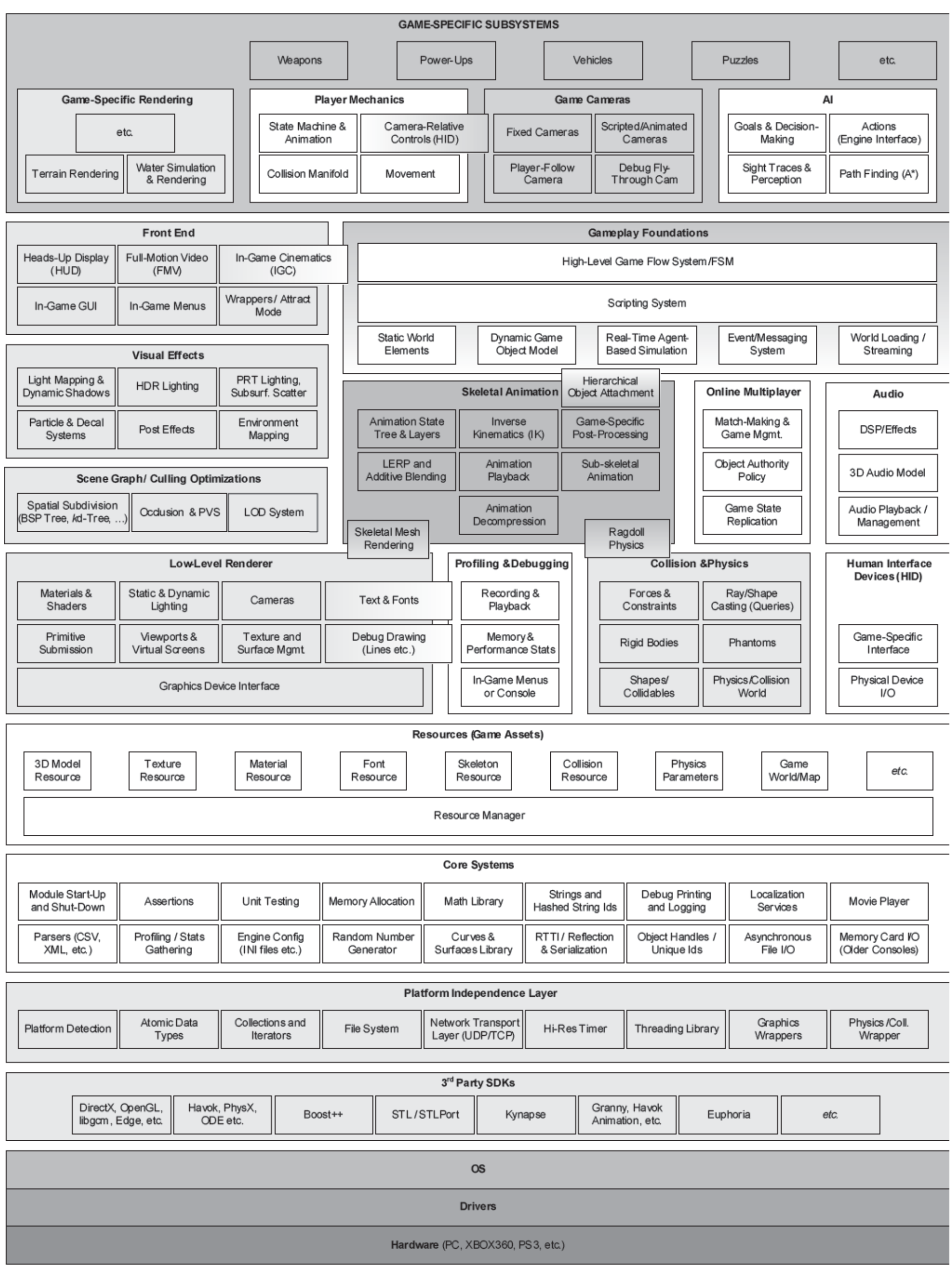


Figure 2.6: Runtime game engine architecture (source: [12])

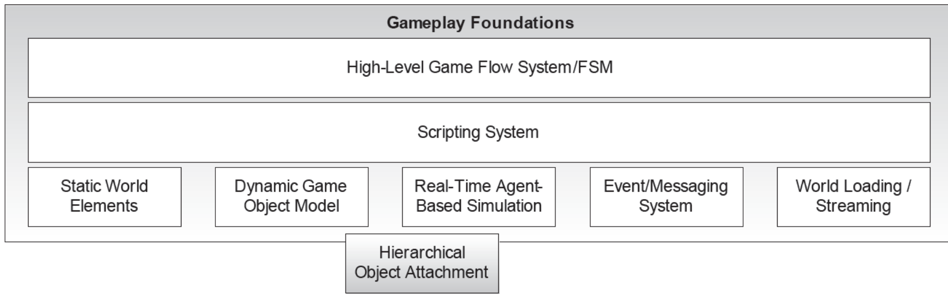


Figure 2.7: Gameplay foundations layer components (source: [12])

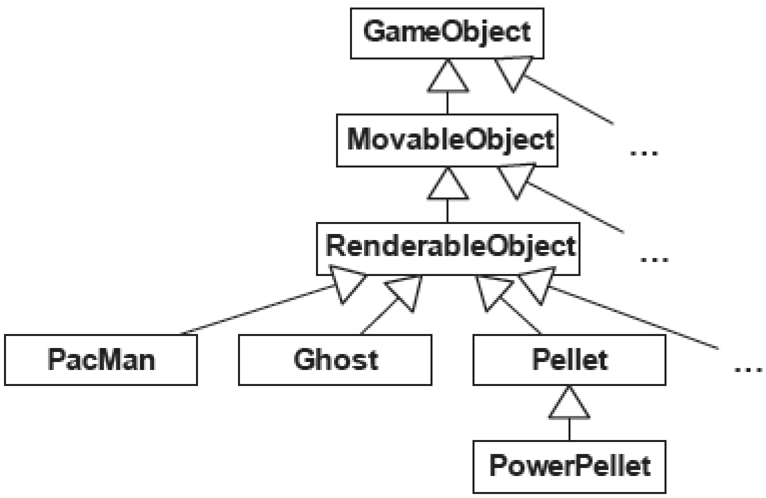


Figure 2.8: Hypothetical game object model for PacMan (source: [12])

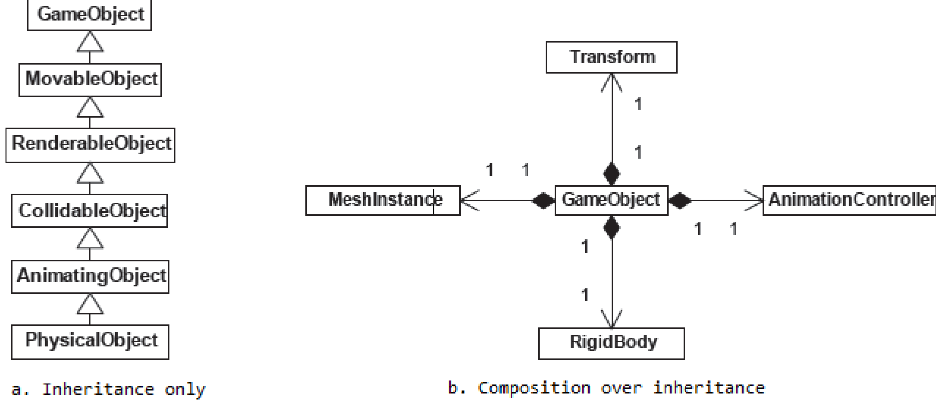


Figure 2.9: Hypothetical game object class hierarchies (source: [12])

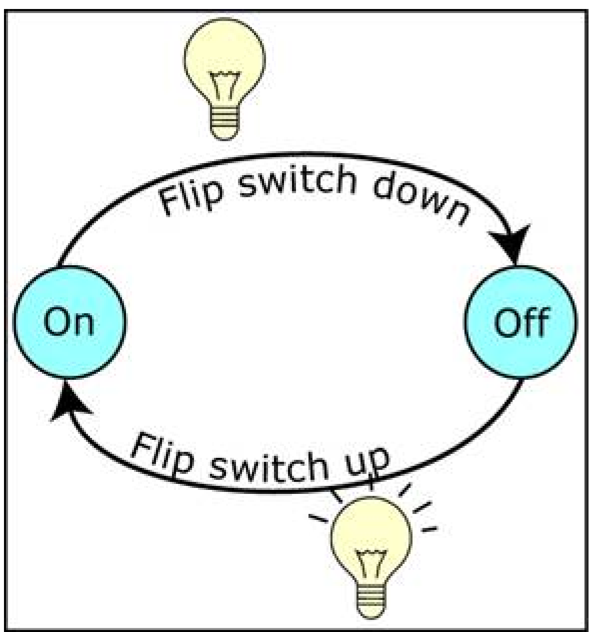


Figure 2.10: A finite state machine for representing the behaviour of a light switch (source: http://ai-junkie.com/architecture/state_driven/tut_state1_files/image002.jpg)

2.4 Aspect-Oriented Programming

There is an important relationship between a programming language and the software design processes. They strive to support each other. Designing a system is achieved by breaking it down in smaller and smaller units (functionality, features, requirements, . . .). The better the abstraction and composition mechanisms provided by a programming language support the type of units broken down by the design process, the better the language and the design process work together. Due to this relationship, design methods often evolve together with a programming paradigm or a specific programming language.

Many programming languages, including Object Oriented, procedural and functional languages, have in common that their abstraction and composition mechanisms come down to generalizing procedures, hence Kiczales calls these languages Generalizing Procedures-languages or GP-languages[21]. The design methods that focus on these type of languages tend to break down complex systems or processes into smaller, more simple units of function, or so called functional decomposition[32]. The GP-language provides abstraction to encapsulate each unit in a procedure, object, class, ...

Some system requirements will lead to system units that, using GP-languages, cannot be encapsulated in a single unit and will be scattered throughout the system in multiple encapsulated function units. These requirements are called crosscutting concerns. It is explained in [21] at the hand of an image processing application.

2.4.1 Example

The goal is to have a system that allows to perform operations on or applying filters to images (cut the edges, transform all specified color values to another value,). The image processing application should be written using clean modular filters. This means we can easily add new filters by composing existing ones. The filters should be optimized both performance- as memory usage-wise. A more complete description of the system requirements can be found in [29].

Filters are functions that take one or more input images and generate an output image. Primitive filters such as bitwise *and- or- not*-filters take two images and render and return an image with the results from pointwise applying the respective logical operator. More complex filters such as *top-edge* remove a row from the bottom of the image and a colored row to the top. These filters are created by combining and chaining primitive filters.

The filters can be cleanly encapsulated in functions, objects or classes. However, the primitive filters all produce intermediate results that are only used to pass on

to the next filter. This is not very memory efficient and should be handled as per requirement.

The case study suggests and implements three solutions:

- A naïve OO-implementation that ignores the performance requirements.
- A more complex OO-implementation where performance requirements are considered.
- An implementation that uses Aspect-Oriented Programming (AOP) to implement the performance requirements.

The proposed filter implementations use a loop construct that reads the image and generate and return an output image. The complex OO-implementation handles the memory concern by manually put filters together and specifying when to reuse an intermediary image to save memory. This results in an application where the code for managing memory is scattered or tangled[46] throughout all the filters. This tangling makes the filters less readable and thus less maintainable.

The AOP-based solution starts from the clear and readable nave implementation. The loop constructs are recognized by a self-designed language, called the aspect language, that reasons about the OO-code. The aspect language extracts the composition of the filters as a graph and fuses the loops, of filters that allow reuse of intermediary images, to a single loop. This step is called aspect weaving. The resulting program meets both requirements of clean modular filters as well as memory-efficient filters. Reusing intermediary images is tackled by a generic solution that will also work when new filters are composed and added.

In a way we could say that this solution doesn't require someone that implements new filters to be aware of how memory is optimized and someone that implements optimizations is not required to be aware of what specific filters do.

2.4.2 Concepts

In the previous example we saw how we come to a modular solution of implementing an extensible, memory efficient image filter system. In [21] Kiczales describes this process in a more formal and generic applicable way.

When aiming at an implementation using a GP-based language, designing that system results in two type of system properties:

A component , which can be cleanly encapsulated in a generalized procedure. These tend to be units of the system's functional decomposition.

An aspect , which can not be cleanly encapsulated in a generalized procedure. These tend to be properties that affect the semantics of multiple components in a systematic way.

Having determined the type of each system property, the next step is to choose a **component language**, in which the components are implemented. This can be an existing GP-language (e.g. Java, C++, ...) or a domain specific one (as is the case in the example). For the aspects we also define or choose an **aspect language** that supports the programmer best in expressing where each aspect affects which component.

To combine programs written in the aspect and component languages to one functional system, we use an **aspect weaver**. An aspect weaver works in three phases:

- **Phase 1**: Recognize the points in the component program where aspects should act, specified by the aspect program. In later work, Kiczales calls these the **join points**[20].
- **Phase 2**: Apply the aspect code by merging it with the specified parts of the component code. In [20] the code that is to be merged is called **advice**.
- **Phase 3**: Take the merged aspect and component code and generate valid code that can be used by a compiler or an interpreter.

Weavers are not *smart compilers*, they do not reason about where a system can be optimized, made more secure or provide logging information. Instead, aspect programmers are provided with the tools (abstractions provided by the aspect language) to express an implementation strategy at a better abstraction level. Aspect programmers are not addressing implementation details of components and the weaver takes over their job to work directly with tangled code.

The core of any aspect-oriented language is the join point model. It defines how the execution of aspect and non-aspect programs can be combined and coordinated.

2.4.3 Implementations

In [20] Kiczales et. al. presents AspectJ, an aspect language extension to the OO-language Java. AspectJ is designed to be generic enough to support modular implementation of a set of crosscutting concerns. The design of AspectJ forms the basis for many component language specific aspect languages. To introduce AOP in the existing user community of Java, AspectJ main design requirement was compatibility:

- All Java programs must be legal AspectJ programs.
- Existing tools (IDE's, documentation, unit-test frameworks,) should be still be usable.
- All legal AspectJ programs are able to run on standard JVM's[6]

[6] Java Virtual Machine, the bytecode interpreter for compiled java source code.

- The provided extensions feel natural to Java programmers.

In the join point model of AspectJ, join points can be seen as nodes in an object's runtime call graph. E.g. when an object's method is called or when a field is accessed. The edges of this graph are the control flow relations between nodes. Computation passes twice in every node: once in (when called) and once out (when returning). Figure 2.11 shows an overview of the join points offered by AspectJ. This overview covers the join points possible in most GP-languages targeted by other aspect languages.

kind of join point	*Points in the program execution at which...*
method calls constructor calls*	a method is called (or a constructor of a class is called). Call join points are in the calling object, or from no object if the call is from a static method.
method call receptions constructor call receptions	an object receives a method or constructor call. Reception join points are before method or constructor dispatch, i.e. they happen inside the called object, at a point in the control flow after control has been transferred to the called object, but before any particular method/constructor has been called.
method executions* constructor executions*	an individual method or constructor is invoked.
field gets	a field of an object, classs or interface is read.
field sets	a field of an object or class is set.
exception handler executions*	an exception handler is invoked.

Figure 2.11: Join points of AspectJ (source: [20])

Pointcuts, a new term introduced by AspectJ, is a collection of join points, that optionally exposes values in the execution context of those join points. Join points can be combined with logical operators (and, or, not). Primitive pointcut designators provide the means to identify join points. We will explain the designators used in this thesis when they occur since not all are present in any aspect language and their semantics can differ.

Advice is a mechanism used to declare that certain code (aspect implementation) should execute at a given join point (or multiple join points described by a pointcut). There are three basic types of advice for join points from GP-languages:

- **Before**: the aspect code should be executed before execution reaches the join point.
- **After**: the aspect code should be executed when execution returns from the join point.

- **Around**: the aspect code that should be executed instead of the join point it operates on.

In most aspect languages, these three types of giving advice will be available. AspectJ provides two more, namely after **returning** and **after throwing**, which is syntactic sugar for code that can be written using the three basic advices.

AspectJ also provides aspects as a modular unit of crosscutting implementation, syntactically comparable to Java's classes. The semantics of these aspect units are out of the scope of this thesis as they are strongly related to the component language. The order in which aspects are applied in large systems that implement multiple crosscutting concerns and on a lower level the order in which advices are executed is specified by the aspect language's weaver.

AspectC/C++

AspectC/C++[37] is an aspect language extension to C++ and C. It was developed to bring AOP to C and C++ to allow programs running on embedded systems and high performance applications to benefit from the same advantages concerning program modularity brought to Java by the AspectJ language extension. The language design and semantics are presented in [39]. While AspectJ and AspectC/C++ are conceptually very similar, some small adaptations were made based on C/C++ conventions or language peculiarities. AspectC/C++ extends AspectJ idea of compatibility to support the systems targeted by C/C++ programs, namely high-performance and embedded systems[38]:

- There should be no computational and resource overhead to an AOP application compared to it's tangled plain C/C++ version.
- Like C++, AspectC/C++ should attempt to do as much as possible at compile time.
- AspectC/C++ should not rely on any runtime infrastructure provided by C++ to ensure compatibility on embedded systems where often these features are not available.

At the time of writing this thesis, AspectC/C++ statically transforms AspectC/C++ source code to C++ source code, regardless if the component language is C or C++. Like AspectJ, AspectC/C++ has aspects and advices, where aspects are translated into C++ classes and advices are method bodies of the class generated from the aspect the advice was in.

Weaving happens compile-time since all join points are computed at compile time for performance reasons. This implies that deploying and undeploying aspects (as if they were never there) is not possible with AspectC/C++. However the behaviour can be mimicked by using runtime checks that conditionally execute the aspect

body, producing a small computational overhead and code tangling throughout the advices of an aspect.

Dojo AOP

Dojo[10] is a Javascript library that aims at facilitating programming the web. It's core contains the most general helper functions used in the more complex components of the library. Dojox/lang/aspect[26] is such a component that brings AOP to Javascript. Javascript is a dynamically typed, prototypical language and also fits the properties of Kiczales' description for GP-languages.

Javascript provides the programmer with many meta-programming tools allowing the user to easily add advice before, after or around method calls without the need for a framework. As made clear by [27], this would result in implementing every advice as functions, increasing the computational overload with more function calls than if we'd place the advice body inline in the component code. The order in which these advice functions are called would also be defined by the runtime moment where the advice is added to a Javascript method. This makes it harder to predict the control flow of an AOP program.

Dojo solves these problems by offering the programmer more expressiveness to describe join points and allowing the programmer to manage active aspects in a consistent way. Figure 2.12 summarizes the interface for AOP provided by dojo. We see that an aspect in dojo is a collection of advices (in contrast to AspectJ where aspects also consist of pointcuts and where the aspects themselves declare where they are applied). An aspect's advices are applied to a set of join points described matches of a regular expression over method names.

Dojo takes advantage of the dynamic component language and does runtime weaving to allow enabling and disabling aspects at runtime. Because Javascript uses the prototype paradigm, applying aspects to *all* methods that match a descriptor of *all* object instances at runtime would be a huge performance hit. Therefore, in contrast to AspectJ, dojo only applies aspects locally to the specified objects.

The core of dojo's aspects only allows us to describe join points based on which of an object's methods match a regular expression. In extension an array of regular expressions can also be given, where the join points are the set of methods returned by matching each expression. AspectJ offers pointcut modifiers that restrict join points with dynamic context, such as a method call stack (e.g. `cflow(call(void Figure.move()))` restricts join points to be inside the control flow of a call to `Figure.move`). Dojo offers this functionality under the form of runtime checks. As seen in figure 2.12 , we can obtain the method call stack and the join point the advice is being applied on at runtime.

```
MyObject = {
    // Component code
};
MyAspect = {
    before: function(){ ... },
    around: function(){ ... },
    afterReturning: function(ret){ ... },
    afterThrowing: function(excp){ ... },
    after: function(){ ... },
    destroy: function(){ ... }
};
var methodDescriptor = /set*/; // regexp or string
var handle = aop.advise(MyObject, methodDescriptor, MyAspect);
aop.unadvice(handle);

// Runtime information
var context = aop.getContext();
// This object defines an AOP context for advices
context = {
    instance:  ..., // the instance we operate on, the same as "this"
    joinPoint: ..., // the joinpoint object (see below)
    depth:     ...  // current depth of the context stack
};
// This object defines a joinpoint's attributes
joinpoint = {
    target:     ..., // the original method
    targetName: ...  // name of the method
};
// This function returns an array of all previous context objects
var ctxStack = aop.getContextStack();
```

Figure 2.12: AOP interface provided by dojox/lang/aspect.

Design of crosscutting concerns

Software design reaps many benefits, such as early assessment of technical feasibility, correctness and completeness of requirements, management of complexity, enhanced comprehension, greater possibilities for reuse and improved evolvability. In practice, designs tend to bundle too many pieces together causing entities designed for a particular system to be too specialised to be of general use. One possible cause for this is the presence of crosscutting concerns, resulting in a design where the responsibilities associated to crosscutting concerns are scattered throughout multiple encapsulation entities. The core responsibilities of these encapsulation entities are tangled up with those from the crosscutting concerns. Scattering and tangling reduce the expected benefits from a design in terms of comprehensibility, traceability, evolvability and reusability, similar to what scattering and tangling does to code.

To maintain a better separation of concerns on the design level and allow the design to evolve with the code, [5] introduces the subject-oriented design model. This model is based on a more flexible decomposition and composition of separate design models as independent views called design subjects (the `<<subject>>` stereotype in UML[7]). Design subjects can be composed with entities from other design subjects to specify crosscutting behaviour. By parameterizing design subjects and then using the mechanism, provided by the model, to bind those parameters to other design entities, we can specify the composition of crosscutting behaviour in a reusable way, regardless of the base design and or environment. This specification is called a composition pattern.

In [6] they illustrate what can be expressed with composition patterns at the hand of examples of crosscutting concerns that occur in many systems. For example an aspect to handle the concern of tracing method calls can be reused by all any class in an object oriented language. In order to design this system using traditional object-oriented design models, we would need to add tracing functionality to each class that we want traced, also when a new class is created. That's where the composition pattern comes in.

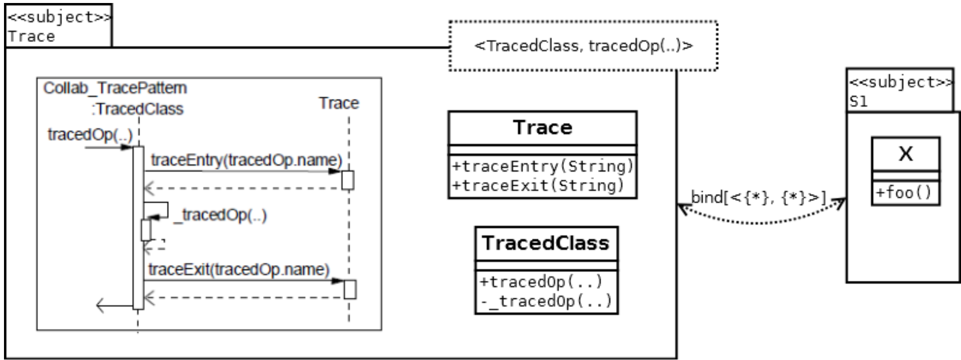


Figure 2.13: Composition pattern for reusable trace aspect and specifying a possible binding (based on images from [6])

Figure 2.13 shows the composition pattern for the tracing concern. The Trace design subject is parameterized with the class whose methods are to be traced, and the methods that need to be traced. The subject consists of two classes: the class that traces and the class that is to be traced. The composition of how these two classes collaborate is shown by the sequence diagram. To use the reusable design for a specific system, we need to use the bind relationship (provided by UML) to indicate which classes and operations are to be traced, $\{*\}$ is a wildcard, resulting in tracing all operations from all classes in the design subject.

The output generated by applying this bind relationship is shown in figure 2.14

[7]Stereotypes are a way to extend UML with building blocks with properties specific to a certain domain.

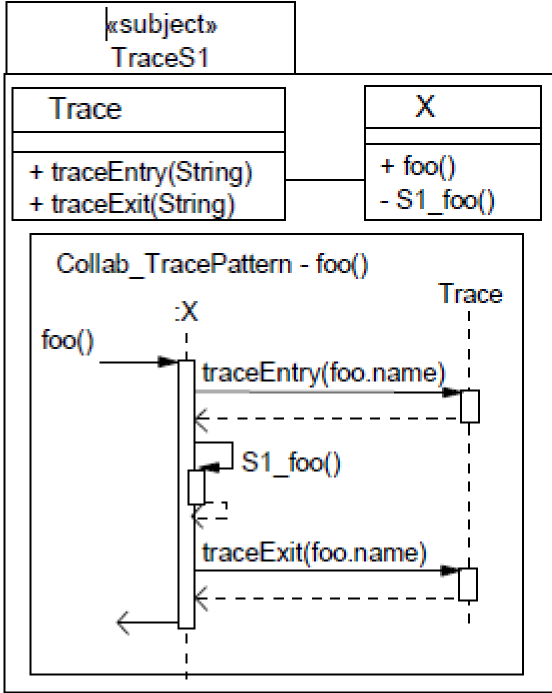


Figure 2.14: Output from binding Trace composition pattern (source: [6])

. This is comparable what the the tangled design would have looked like. Just as with aspects, the designer did not have to work with the tangled design directly. All tracing functionality is now cleanly separated from the concerns expressed in design subject S1: the person that designed the Trace design subject doesn't have to know the design details of S1 and visa versa.

At a conceptual level, composition pattern design and aspect-oriented programming have similar goals. Composition patterns allow us to separate and design reusable cross-cutting behaviour (such as the tracing problem) and aspect-oriented programming provides the means to maintain this separation in the coding phase. Table 2.3 summarizes the mapping of Aspect-Oriented concepts to composition pattern elements. Note that in the original mapping in [6], introduction was also explicitly mentioned. We did not include this since introduction does not belong to the core of all aspect languages.

AOP	**Composition patterns**
An aspect is a type that encapsulates cross-cutting behaviour.	Design subjects model one concern.
A pointcut describes the points where cross-cutting behaviour is required during program execution.	Operation template parameters may be defined and referenced within interaction specifications. A template may be replaced by actual operations multiple times.
A piece of advice is code that executes at a pointcut in a given order (before, around, after).	In the sequence diagram, cross-cutting behaviour may be specified to execute when a template operation is called.

Table 2.3: Mapping AOP elements to composition pattern elements.

2.5 Background Summary

In this chapter, we first discussed three existing educational games, that could serve well as test cases or study subjects for making them adaptive.

In the next section, we elaborated on the research that has been done concerning game adaptivity and how it can be interpreted. Most research focussed around a user model that tracks the skill, motivation and other properties of the user that is playing a game. Based on the state of the user model, adaptations can be made that personalize the game towards the player at hand. A categorization of game adaptivity was given by specifying a range of cognitive and motivational adaptive interventions.

Research on adaptivity in virtual reality (VR), i.e. a 3D world, was also considered as many recent high-quality games leverage 3D to emerge the player further in the game, resulting in higher motivation than with traditional games. To understand what adaptivity means in VR, it is important to know what components a 3D world consists of. With this knowledge, adaptation types were introduced as descriptions of adaptations to a single VR component. Adaptation strategies were introduced as more complex adaptations that span multiple VR components.

The third section focussed on the architectural resemblances and differences between game types and game genres, in order to find a generic base for developing a framework to add adaptivity to any type or genre of educational game. The differences result in game-specific design choices such as the game object model, the structure of the game loop, use of frameworks, or using scripting systems or finite state machines to implement the storyline. This knowledge could be used for creating more game-specific extensions to the generic adaptivity framework.

Finally, the fourth section explained the basic concepts of- and ideas behind Aspect-Oriented programming (AOP). The motivation for resorting to AOP can be found in a goal that every programmer strives for: code readability and maintainability. Using traditional programming languages (general purpose languages), there are projects in which some concerns will be inherent cross-cutting (a part of) the projects code, resulting in concerns for which the code is scattered throughout other concerns of the project, making it less readable and less maintainable. AOP brings a solution for this problem, by providing the programmer with tools to describe where in the code (join points) some behaviour has to be applied (advices), centralized in an external component (aspect). An aspect weaver is used to interpret these join points and apply the advices to the original component code resulting in a program that has the same behaviour as if the cross-cutting behaviour were put in the component code manually. Three AOP implementations (AspectJ, AspectC++, dojox.lang.aop) were also discussed to illustrate different interpretations of these concepts. Finally a design technique for aspects is explained.

Chapter 3

Pilot Study

In cooperation with all the parties involved in the CAdE project, an educational game has been modified to serve as a tool to investigate the benefits that come from using educational games as a means towards learning. As a pilot study, the game was deployed for two weeks in the elementary school Sint-Anna in Tervuren. The study was managed by three people: Koen Homblé (experimental design), Yorgos Pastis (recording sessions) and Ben Corne (software adaptations). In this chapter we describe the research protocol as well as the interesting parts of the implementation. An extended description of the research protocol can be found in [13].

3.1 Study Summary

The primary goal of the study was to investigate whether playing an educational game can lead to improvements of scholastic skills (e.g. mathematics). A secondary goal, more interesting to this thesis, is whether the introduction of adaptivity in task difficulty could enhance learning. To be able to measure the improvement and compare adaptive task difficulty to other difficulties, we developed a modified version of the existing arithmetical educational game TuxMath, described in 2.1.1 on page 9. The goal of the game is to solve mathematical problems, represented by falling meteors, before they hit the ground and damage nearby igloos.

The game was stripped of elements that could influence the observed results while retaining the most basic game elements to ensure game flow described in 2.1 on page 8. Non-basic elements that were removed include bonus comets that fly horizontally and game-over when all igloos have been destroyed.

Only three task difficulty settings were kept for the player to choose from. The game task difficulty could either remain at the same level, gradually increase as

the game progressed, or change adaptively based on user performance. Effectiveness of playing the game under different conditions was measured by comparing performance on a standardized pre- and posttest.

The goals and objectives of this study were the following:

- Measurement of feasibility using an educational game as a didactic tool. Find out if playing an educational video game can lead to enhancement of scholastic skills.
- Compare the feasibility of learning under different game difficulty settings. More specific, does a difficulty setting that adapts to the user's skill enhance learning in comparison to traditional settings such as static or increasing difficulty.
- Comparison of appreciation of different difficulty adjustments: which adjustment is preferred by the users?
- Exploration of player data collection in educational games. How can we use data gathered during the game (e.g. reaction times, accuracy, emotional state) as input for adapting the game's difficulty?
- Obtaining both video recording and emotional state data to train the facial action unit detection software by relating facial expressions to an emotional state. Detecting emotions in real-time can be advantageous for more accurate game difficulty adaptivity.
- General test of our design and procedure before implementing the experiment on a larger scale.

3.2 Study Setup

A two-week training program with the game was conducted to improve the automaticity of the basic arithmetic skills of third-grade elementary school pupils. For 10 consecutive school days (interspersed by two weekends), 20 pupils each played the game for 10 minutes at the start of the school day. Each pupil was assigned a game task difficulty for the program. For conducting the experiment 10 computers were made available and one computer was equipped with a video recording setup that captured the pupil's facial expressions and keyboard strokes.

In order to measure the effect of the training program, both before and after the program the Tempotoets Rekenautomatismen[34] test was administered to the pupils. The goal of this test is to determine the fluency and accuracy of pupils when solving basic combinations of mental arithmetics. The pupils receive five columns of 30 arithmetical exercises of which they have to complete as many as they can within 1

minute (per column). Each column gives one type of exercises: additions, subtractions, splicing (e.g. 8 = 4 + ?), multiplication and division. Higher scores on the posttest as compared to the pretest can indicate a beneficial effect on performance in educational tasks by using the educational games[1].

While TuxMath was modified in multiple ways, the core idea of the game was kept. As in the original game, the player have to defend four igloos. Comets, with a mathematical exercise on top, enter the screen above one of the four igloos and start falling down towards the igloo. The player should attempt to solve the exercise and type the answer before the comet hits the iglo underneath as otherwise an animation is shown of the igloo being destroyed. Unlike the original game, the modified game never ends itself before all waves and comets have been beaten, not even if all four igloos are destroyed.

The game difficulty settings delivered with the game were modified as well to provide the players with the three difficulty settings mentioned in the summary (static, increasing and adaptive). In the *static* settings, the game difficulty was always the same: each wave consisted of ten comets, falling down at the same speed[2]. The exercises were randomly generated with the following restrictions:

- Terms and solutions of additions, subtractions and splicing exercises don't exceed 20.
- Factors and solutions of multiplications and divisions can not exceed 100.

When the player entered a correct solution to any of the visible comets, a laser beam is shot at the lowest comet with a corresponding solution (in the original game all comets with this solution were shot at the same time). The maximum number of comets on the screen at the same time is limited to 10.

In the *increasing* difficulty setting, the number of comets starts with two and increases by two each wave. The speed of the comets also increases by 5% each wave. There was no maximum to the number of simultaneously visible comets. The terms, factors and results of each computation had the same limits as with the static settings.

Finally, in the *adaptive* difficulty setting, each wave consisted of 10 comets and the speed was adjusted based on user performance. If the accuracy was greater than 90% and if, on average, exercises were solved before reaching the middle of the screen, the speed would go up by 10% in the following wave. If either accuracy was below 70% or exercises were not solved before reaching the middle

[1] As there was no control group, other possible explanations such as a test-retest effect or progress in education cannot be excluded.

[2] The game was programmed to scale the speed according to the resolution of the window or the screen incase the game is played in fullscreen mode. Since not all computers made available to the program have the same resolution, the speed in terms of pixels / second was different for different resolutions.

of the screen, speed would go down by 10%. Otherwise the difficulty remained unchanged.

In each game mode, once every two waves a digital version[3] of the SAM scale or Self-Assessment-Manikin scale was administered to the players. The scale is used to measure the level of valence, arousal and dominance the player is experiencing.The test presents the test subject with a series of cartoons for each property. The left-hand-side of the series represents a low value of that property while the right-hand-side represents a high value. Figure 3.1 shows the series associated with the three properties. The results from this test can be used to train the facial action unit to translate facial expressions to emotions more accurately.

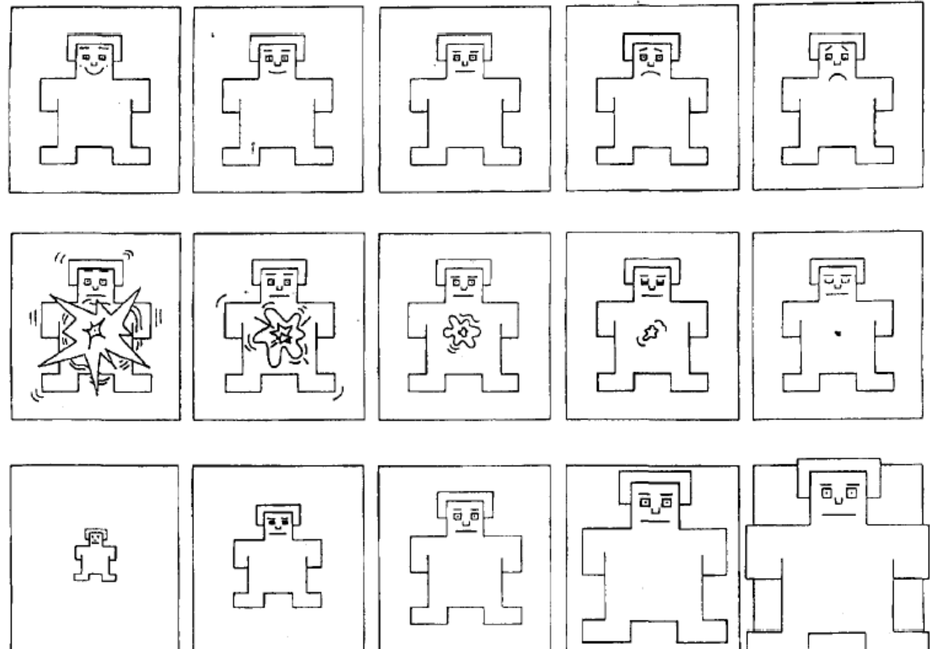


Figure 3.1: The Self-Assessment Manikin (SAM) used to rate the affective dimensions of valence (top row), arousal (middle row), and dominance (bottom row) (source: [1])

3.3 Adaptive Implementation

The part of this study that is of interest to this thesis, is how the adaptive mode was added to the existing educational game TuxMath. The approach used for this

[3] http://irtel.uni-mannheim.de/pxlab/demos/index_SAM.html

game provided the basis for GAAOP, the methodology of how to add adaptivity to existing educational games.

3.3.1 Game conceptual structure

To know how to add adaptivity to TuxMath, the game was first explored by playing it, providing insight in the structure of the gameplay and game concept. The overall structure of TuxMath is depicted by figure 3.2 . The game consists of individual games that are specified by a settings file. Each individual game consists of a number of waves and a set of mathematical exercises of the types that are specified in the settings. The game also specifies the change in wave properties as the game progresses. A wave consists of a set of exercises from the exercises created for the surrounding game. The exercises start at the top of the screen and move down the screen at a speed specified by the wave. The wave also specifies the maximum amount of exercises that can be simultaneously present on the screen. The original game defines more settings for each subsystem but the effects associated to these settings were removed in the modified game.

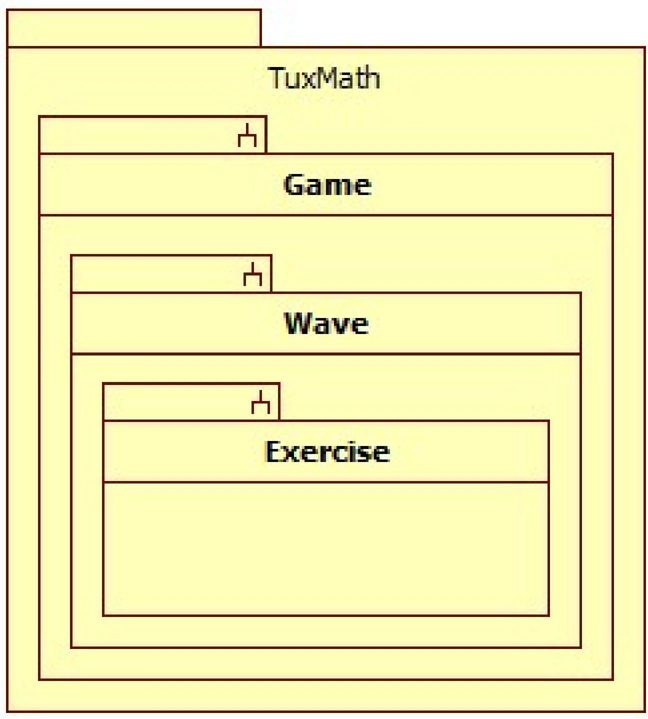


Figure 3.2: Overall structure of TuxMath.

With this structure in mind, the next step is to decide the level on which game difficulty adaptations will be made and on which level skill assessment will be performed. For the pilot study, adaptations to the exercise falling speed were made on the wave level (i.e. once per wave). Skill was assessed for each exercise. The skill assessment consisted of the accuracy of the answers and time needed to solve each individual exercise in terms of exercise height (how many pixels the exercise meteor dropped before being solved).

3.3.2 Game engine structure

The implementation of TuxMath does not rely on a framework nor does it extend an existing game engine. Some tasks such as building menus and menu transition have been factored out in the t4k_common library. Although TuxMath uses no previously existing game engine, many of the architectural patterns described in section 2.3 can be found in the custom engine. With the creation of a second game TuxType[4], some behaviour common to both games has been factored out to the t4k_common library[5]. This includes menu transitions, resource allocation and initialization and more. For audio, text and graphics, SDL (Simple DirectMedia Layer, see 2.3.1 page 21) libraries are used. The implementation parts interesting to this thesis are grouped in the tuxmath code itself.

The game loop's architecture resembles to the architecture of the simple game loop given in 2.3.1 on page 30. The looping part of code is shown in code example 3.1 , with some parts stripped out, irrelevant to this thesis.

The code example shows that every event or action has his own handler function. For example, to handle the event where a player gives an answer, `game_handle_answer()` is called each game loop iteration. The functions indicated in bold in the example are the ones of interest for adding adaptivity to TuxMath. The behaviour of the functions executes in a specific tier of the game's structure shown in figure 3.2 . We discuss each function.

- `game_initialize()` : This function initializes the questions and playfield for a game. This occurs in the game tier. After a call to this method, the comets start falling.
- `game_handle_answer()` : Every individual answer is processed by this function. Answers are given in the exercise tier. A correct answer leads to an animation where the associated meteor is destroyed by a laser.
- `game_handle_comets()` : This function advances the comets in each iteration of the game loop. The method handles the wave tier. The function

[4] http://tux4kids.alioth.debian.org/tuxtype/
[5] http://anonscm.debian.org/gitweb/?p=tux4kids/t4kcommon.git

```
1  int game(void) {
2    game_initialize();
3    /* --- MAIN GAME LOOP: --- */
4    do
5    {
6      /* reset or increment various things with each
          loop: */
7      frame++;
8      old_tux_img = tux_img;
9      tux_pressing = 0;
10     int i;
11     for(i=0;i<MAX_LASER;i++)
12     {
13       if (laser[i].alive > 0)
14         laser[i].alive--;
15     }
16
17     // 1. Check for user input
18     game_handle_user_events();
19     // 2. Update state of various game elements
20     game_handle_demo();
21     game_handle_answer();
22     game_countdown();
23     game_handle_tux();
24     game_handle_comets();
25     game_handle_powerup();
26     game_handle_cities();
27     game_handle_penguins();
28     game_handle_steam();
29     game_handle_extra_life();
30     // 3. Redraw:
31     game_draw();
32     // 4. Figure out if we should leave loop:
33     game_status = check_exit_conditions();
34   }
35   while(GAME_IN_PROGRESS == game_status);
36   /* END OF MAIN GAME LOOP! */
37   game_handle_game_over(game_status);
38   game_cleanup();
39 }
```

Listing 3.1: Game loop for TuxMath

```
1  // the wave's statistics
2  typedef struct {
3    int      questions;  // how many questions were asked
4    int      correct;    // how many questions were
         correctly answered
5    int      shots;      // how many times was shot
6    double   height;     // the wave's average height
7  } Adaptive_wave;
8
9  void AD_Init(game_option_type * options,SDL_Surface *
       screen);
10 void AD_Cleanup(void);
11 void AD_NewWave(void);
12 void AD_Crash(double height);
13 void AD_Hit(double height);
14 void AD_Shoot(void);
15 void AD_Miss(void);
16 void AD_UpdateSettings(void);
```

Listing 3.2: Assessment and adaptation interface

is responsible for the behaviour associated to meteors crashing onto an igloo and for advancing the wave if no meteors are left.

- `game_cleanup()` : Back in the game tier, this function is the counterpart for `game_initialize()`. It cleans up the resources that were allocated during the game.

3.3.3 Adding adaptivity

For this initial study, we decided to adapt the difficulty on the level of waves (wave tier and below). This means that during each wave we want to assess the player's skill level and before a new wave, adapt the difficulty accordingly. Code example 3.2 shows the interface for both assessment and adaptation.

The interface shows us that each wave, we keep track of the number of questions that were posed, the amount of questions answered correctly, the total shots that were fired (or answers that were tried) and the average height of answering questions. We will call this part the **player profile** as it specifies properties of the player. Note that besides game performance data, such as the ones presented in the code example, also data obtained by observing the player can be present in the profile. For example the emotional state (bored, excited, intimidated, . . .) or concentration level (deducted from results returned by a brain wave sensor such as those created

by NeuroSky[6].

A next part of adaptivity is **skill assessment**. When player performance are measured, it is assessed and the player profile is modified accordingly. For the initial study, we assessed only when meteors crashed (`AD_Crash`), when answers were fired (`AD_Shoot`), if the fired answer was correct (`AD_Hit`) and when it was wrong (`AD_Miss`).

Actually adapting the difficulty happens in `AD_UpdateSettings`. This method boils down to verifying if the values from the player profile cross predefined thresholds and act if they are. The essence is shown in code example 3.3 . Here we check the average answering height against a threshold and the accuracy of the player's answers. We call the combination of checking and updating the game adaptations, they are encapsulated in the adaptivity engine, which is also responsible for (re)initializing and cleaning up the resources used by the component itself. The other parts are also responsible for doing this for themselves but in the TuxMath implementation, initialization and cleanup behaviour from all parts were merged in three methods `AD_Init`, `AD_Cleanup` and `AD_NewWave`.

In order to add the adaptive functionality to the game, we need to know where to call the necessary functions and we need to pass the correct contextual data (e.g. answering height) to these functions.

It is not hard to see that `game_initialize()` and `game_cleanup()` are good places to initialize (`AD_init`) and cleanup (`AD_Cleanup`) the resources needed by the adaptive engine, skill assessment unit and the player profile. `AD_NewWave` is responsible for cleanup and preparation of the three components in between waves. The point in the control flow where waves are switched is found in the `game_handle_comets()` function.

For assessing the player's skill, we limited ourselves only to assess only the accuracy and speed of answering questions (expressed respectfully in percentages and number of pixels the comet had to travel until being destroyed by a correct answer). Correct and incorrect answers can be recorded in the control flow of `game_handle_answer()`. Unsolved exercises (crashing comets) can be recorded in the control flow of `game_handle_comets()`.

Finally updating (adapting) the game difficulty is done in between waves based on the performance from last wave (`AD_UpdateSettings`). As mentioned earlier, advancing to a next wave is done by `game_handle_comets()`.

In the implementation for the initial study, we inserted function calls on the previously discussed places. Sometimes we had to make data explicitly available so it could be passed as an argument to the adaptivity interface functions. Only the call to `AD_UpdateSettings` was guarded by a check to ensure the player is playing the game in adaptive mode. Other adaptivity calls were not provided the

[6] http://www.neurosky.com

```
if(height > h_threshold) {
  //  => SLOWER, LESS SIMULTANIOUS COMETS
  fprintf(stderr,
      "[AD]␣Decreasing␣skill:␣height(%f/%f),␣correctness(%f/%f)\n",
      height,h_threshold,
      correctness,c_threshold);

  Opts_SetStartingComets(starting_comets - Opts_ADMeteorDecrease());
  Opts_SetSpeed(Opts_Speed() / Opts_ADSpeedupDecrease());
}

else if(correctness > c_threshold) {
  //  => FASTER, MORE SIMULTANIOUS COMETS
  fprintf(stderr,
      "[AD]␣Increasing␣skill:␣height(%f/%f),␣correctness(%f/%f)\n",
      height,h_threshold,
      correctness,c_threshold);

  Opts_SetStartingComets(starting_comets + Opts_ADMeteorIncrease());
  Opts_SetSpeed(Opts_Speed() * Opts_ADSpeedupIncrease());
}
```

Listing 3.3: Updating the difficulty based on correctness and average height.

same guards because the data gathered by the skill assessment unit is also used by a GameLog component that is responsible for persistently storing performance reports of the current player. More information on the implementation can be found in the internship report[7].

3.4 Pilot Study Summary

This chapter presented a pilot study that was conducted as part of the CAdE project. The main goal of the study was to investigate the benefits of playing adaptive educational games on scholastic skills. The study used a modified version of the mathematical game TuxMath in which different versions of adaptivity were used to compare them. A pre-test and post-test was also administered to measure the improvement of the related math skills. The comparison was realised in three difficulty settings for the modified version of TuxMath: static difficulty, linear increasing difficulty, and difficulty that is continuously adapted towards the user's performance. In this initial study, the game was only adapted towards the player's performance in terms of correctness and swiftness. The game itself was modified to record more data, such as exact timings of answers and queries of the emotional state of the player, to allow more complex adaptations (e.g. adaptations modifying the game based on the real-time determined emotional state of the player).

The interesting part for this thesis of this pilot study was the approach that was taken to implement the adaptations. The approach can be summarized in three steps: adaptivity engine initialization of an adaptivity engine, constructing a player profile (or user model) and updating it with data from assessment of the player's skill, and adapting the game's difficulty based on the skill stored in the player profile. The actual implementation was done by manually inserting function calls to adaptivity-related functions in the regular game code.

Chapter 4

Proposed Method

In this chapter we define a methodology that could serve as a guideline for adding adaptivity to existing educational games, and which is based on the work Brusilovsky did on adaptive hypermedia[2] and on the process of implementing adaptivity in TuxMath as described in 3. Certain tasks in the methodology can be automated and this behaviour can be extracted into a library for reuse across multiple games. Besides the methodology, this chapter also describes the design and implementation of the reusable behaviour and finally a proof of concept is given by applying the methodology and using the library on a small textual game.

4.1 Methodology

As explained in 2, section 2.3, the architecture of games and their game engines differs a lot and each game engine comes with design choices that allows a certain type of games to be created with it. Since educational games can be any type of game where a didactic element is present (e.g. practicing solving mathematical exercises in the arcade game TuxMath), finding a single solution to add adaptivity to every possible educational game of every possible game type is hard, not to say impossible. Instead, we look for a generic description of how to tackle the problem of adding adaptivity to a game, providing the programmer with guidelines and tools that help him or her in doing so.

The methodology can be framed by the work Brusilovsky did on adaptive hypermedia[2]. Figure 4.1 summarizes the facets of adaptivity in hypermedia, with How? - conceptual level being the central question that relates to all the other facets. We could see the methodology described in this chapter as an answer to this question.

- Where? indicates the domain the adaptivity is applied to, in our case this is the domain of educational games.

- Why? explains the goals of adaptivity in our domain, that is to provide more efficient learning than with the traditional one-size-fits-all approach.
- To what? relates to the elements that are used to decide how to adapt the material more suited to the user's needs and preferences. These elements are user data, usage data and environment data as described in section 2.2.1 on page 12 .
- What? describes the type of adaptation that is performed. In his paper, Brusilovsky describes two distinct areas of adaptation: content level adaptation (adaptive presentation) and link level adaptation (adaptive navigation support). Adapting games to ensure game flow can fall in both categories, but the examples used in this thesis are mainly focussed on content level adaptation.

  Brusilovsky created a taxonomy to classify the concrete adaptation types (e.g. link hiding, text modification, sorting, ...). However in the domain of educational games, the adaptation types can not be as concrete as they are for the domain of educational hypermedia. Educational games allow more conceptual adaptation types such as difficulty adjusting and altering the relation between game and didactic elements, where the interpretation of these adaptation types can be very different between two games.
- How? - techniques, implementation level relates to the technologies used to create or provide adaptive systems (e.g. course authoring tools, component-based frameworks, user model servers). In our methodology, this would be the reusable aspect library that serves as a framework to add adaptivity to existing educational games.

As explained earlier, the methodology will consist of guidelines and tools for the programmer. The guidelines are related to five tasks. The first task is to search for what we define to be adaptivity for the target game. The second task is to inspect the game code and create a possible user model consisting of user data, usage data and environmental data. The third task is specifying the conditions for a given adaptation to occur. These conditions are checked against the user model from the previous task. The fourth task is to link our definition of adaptivity for the target game to the code that implements the game. The fifth and final task is to use the tools (in the form of an aspect library) to add the adaptation to the original game. We describe each task in detail.

4.1.1 Step 1 : Finding the base for adaptation

To make a game adaptive, one first needs to define what kind of adaptivity is required or suitable for this game. This is best done by actually playing the game and exploring it. It should give a rough idea of the learning challenges, the difficulties

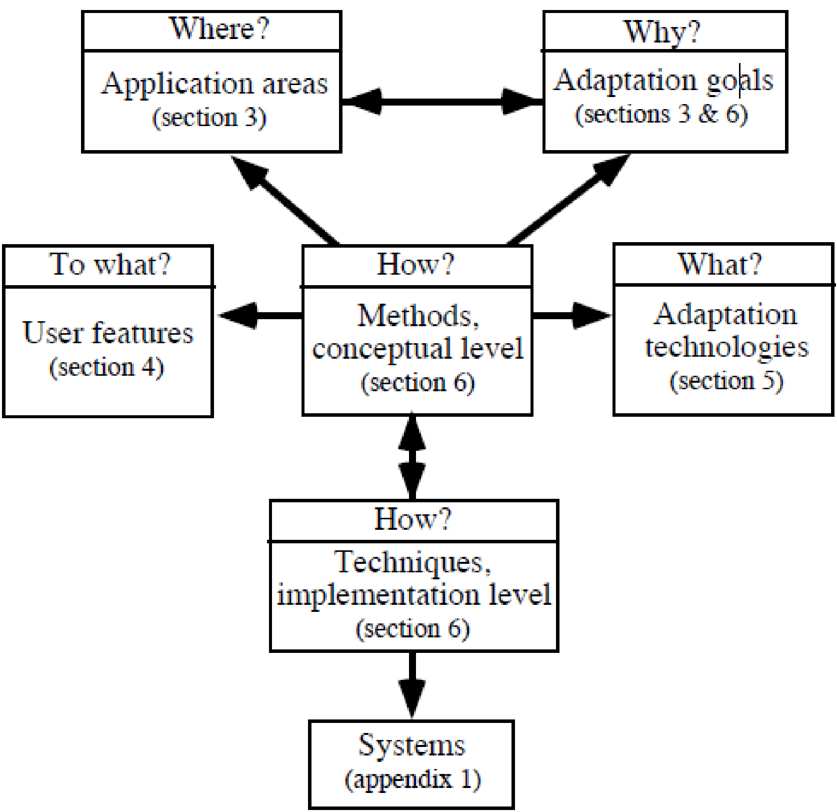


Figure 4.1: The facets of adaptivity in hypermedia. Source: [2] page 3.

for the learner, the rewards for performing well, and how the game can be altered to make it more challenging or more interesting to play.

We give an example based on the exploration phase performed on TuxMath for the study described in chapter 3. Note that this is not a full description for the original TuxMath game, but for the modified version that contains less game elements that could influence the results of the study.

- What are the learning challenges?
  - Solving mathematical problems with sums, subtractions, multiplications, and divisions.
- Where do the difficulties lie?
  - The meteor falling speed determines the amount of time there is to solve a question.
  - An exercise consists of operands, operators and a solution. The range of the operands and the type of the operator determines the difficulty

of the problem.

- What are the rewards?
  - Solving individual problems results in a point reward.
  - A negative reward is given each time an exercise is not answered in time under the form of an igloo melting and making the residing penguin to leave the game.
  - Completing a round of waves of exercises results in an increase of score, based on the amount of exercises that were answered in time.
- How can the game be altered to make it more challenging? Children may become demotivated if they don't manage to solve the question in time. Others may loose interest if the exercises are too easy. As such, the following actions are possible to make the game more challenging:
  - Increasing / Decreasing the falling speed.
  - Increasing / Decreasing the range of the operands.
  - Increasing / Decreasing the occurrence of certain type of operators.
  - Increasing / Decreasing the maximum number of exercises shown to the user at any given moment in time.
- How can the game be altered to make it more interesting to play?
  - Increasing / Decreasing the amount of awarded points for solving exercises.

Another benefit of the exploration phase is that, by playing it, we should be able to sketch the game structure. Based on the structure, one can decide on what level(s) adaptations will be applied.

For example, in TuxMath, the player starts by selecting a game mode. Each game mode is divided in rounds. The rounds consist of waves and each wave has multiple questions. For the study we decided to apply adaptation on the level of waves.

4.1.2 Step 2 : Building the user model

We use the user model as the basis for adaptation. The user model is an aggregation of user data, usage data and environmental data as described in chapter 2 section 2.2.1. We specify the individual elements that we want to track or assess. A generic user model is shown in figure 4.2 . The aggregated data is gathered in multi-type data association structure, e.g. a JSON object[1].

[1] http://www.json.org/

Based on the aggregated user data or properties, a number of features can be computed for the adaptation decision process, i.e. the values that are used to determine if and how the game should be adapted. The features are nothing more than a function (with return type in statically typed languages) which takes the user model as an argument to compute its value.

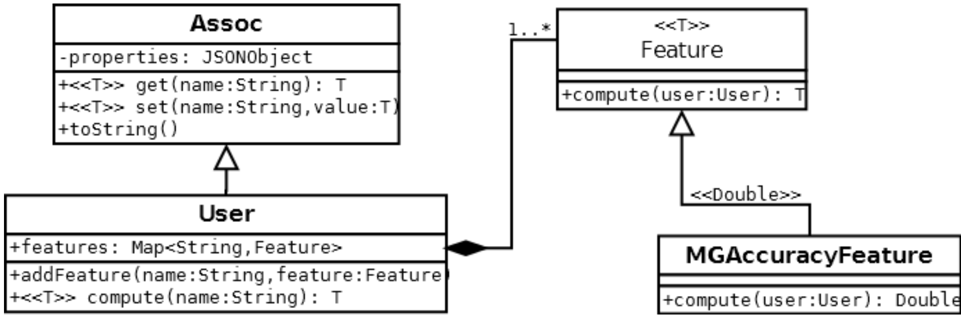


Figure 4.2: Generic user model for the proposed methodology

As we can see in figure 4.2 , the user model provides getters and setters for the properties as well as a method to add and compute features, given the associated property name or feature name. Note that adding features could be done automatically by the use of reflection, lowering the burden of the programmer, but introducing a dependency on the availability and functionality of reflection tools. In Java, if we implement Feature as an interface, it requires non-standard reflection techniques that involve configuration files and file scanning.

Defining the user features

In the domain of educational games, there are a number of user features that can be collected for virtually all games. These include learning style, level of concentration, and emotional state. However, sometimes the cost of recording and tracking changes in data that is used to compute these features can be too high for the allowed resources the game has. In this sentence resources is a very broad term, going from processor time to available funds to buy hardware or external devices, and even to the game-flow cost induced by invasive questionnaires. For this reason, the user model should be adjusted to fit the available resources.

A feature that can be imagined to be important in many (if not all) educational games is performance. This feature finds its counterpart in traditional educational approaches as for instance the grades that are assigned to pupils for a period in time based on tests taken during this period. The versatile nature of games makes it impossible to give a single definition of what performance is for any educational game. For each game this differs based on what the educational challenges are and where the difficulties lie. Just as with grades in traditional educational approaches,

there is (usually) a direct relation between difficulty and the value of grades.

In TuxMath we can find such a performance feature as the relation between the amount of exercises that were correctly solved, the amount of answers that were given and the amount of exercises queried to the player. A second possible performance feature is the more simplistic relation between correctly solved exercises and amount of queried exercises (e.g. If the performance is 70%, 7 out of 10 answers were correctly). A third performance feature in TuxMath is the average time taken to answer questions, describing a relationship between the time that was needed for each exercise and the total amount of queried exercises.

```
User
-questions_queried: int
-answer_attempts: int
-correct_answers: int
-heights: int[]
+getAverageHeight()
+getCorrectness()
```

Figure 4.3: User model for adaptivity in modified version of TuxMath.

Figure 4.3 shows the concrete user model used as a base for adaptivity in the version of TuxMath that was created for the study. The model includes two performance features: the average height and correctness of answers. These features are a relation between usage data such as the amount of queried questions and the amount of questions that were answered correctly. Using the user model, we can adapt the game's difficulty to the user's performance.

4.1.3 Step 3 : Defining the adaptations

In it's most simple form, adaptations verify that certain features or properties meet some criteria, and if this is the case, the game's default behaviour is modified to be more adapted towards the user's skill or interests. For more programming flexibility (explained below), this process is split up in modifying settings and actually applying these settings to the game. For each game adaptation that is desired, one should define the following:

- **Adaptation settings**: A description of the values that indicate how the game should be adapted, such as game speed (e.g. increase by 20%), operands range (e.g. between 0 and 100) or learning style (e.g. auditory)

- **Adaptation conditions**: The criteria, based on user features, for adaptations to occur. E.g. is the accuracy above 70%?.
- **Adaptation settings adaptations**: Behaviour that makes modifications to the adaptation settings, triggered when the associated adaptation condition becomes true. E.g. increase the maximum amount of questions asked by 2.
- **Adaptation settings application**: The behaviour that adapts the game, based on the settings. E.g. change the game's actual settings to use the maximum amount of questions specified in the adaptation settings.

The principle is that when the value of a feature meets a condition, the associated adaptation is triggered and applied to the game settings (adaptation settings application). The adaptation settings are to be considered during the game, resulting in a change in either the relationship between didactic and game elements, or in the level of challenge imposed on the player (adaptation settings application).

The only guidance towards defining the settings and adaptations that are applied to them, currently provided by the method, is to use the answers to the question on how the game can be made more challenging and more interesting (step one). The method provides no tool support to ease this process. This lack of support is further discussed in chapter 5. Possible extensions to the method regarding settings adaptations are given in chapter 7.

In the case of numeric features such as the performance features from the TuxMath user model example in the previous step, conditions manifest themselves as thresholds that describe the maximum or minimum values. When a threshold is exceeded, an adaptation is made that makes the game more or less challenging. The interaction between adaptation conditions (in the form of thresholds) and settings adaptations is shown in code example 3.3 on page 58.

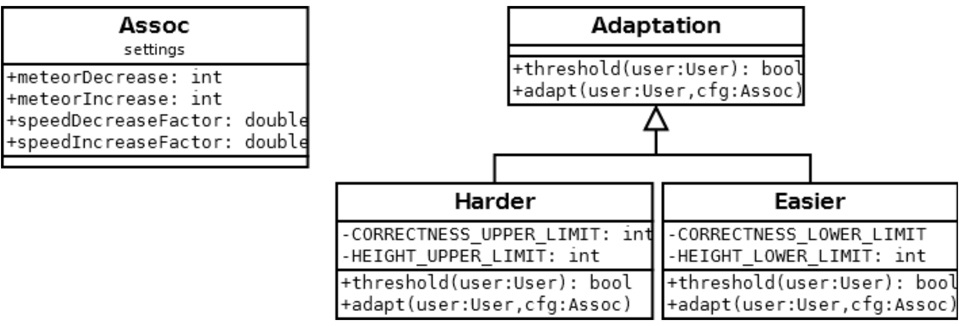


Figure 4.4: Model of the adaptations for the modified version of TuxMath.

The design for the adaptations for the modified version of TuxMath is shown in figure 4.4 . A concrete Adaptation associates an adaptation condition to a settings adaptation. The settings are an association between names and values, represented by class members in the figure.

Decoupling adapting the adaptation settings from applying them is in the spirit of Aspect-Oriented programming, to provide more flexibility and better reuse to the programmer. For example, one could assess properties from which one could determine the player's learning style and save them as a setting. One only needs to compute and adapt the setting once to change all of the game's affected default behaviour that might be spread throughout the code. This allows the programmer to focus on changing the behaviour of one concept in the game at a time without having to take into account the performance impact of recomputing the setting, since this is cached by the adaptation settings.

4.1.4 Step 4 : Linking the definitions of adaptivity to the game code

The next phase in the methodology is to define a relationship between the base for adaptation and the code that implements the game. A good place to start is by sketching the high-level game engine architecture that is used and associate it to parts of the code. At least the following elements should be found:

- Game initialization: Initialization of resources that are used by the game engine.
- Game loop: Where the game engine starts.
- The structure that was observed during the exploration phase: The code responsible for every subtree of the observed structure.
- Game cleanup: Cleanup of resources that are no longer needed by the game engine.

With this high-level architecture, we can inspect the code further and link it with the more concrete parts of our adaptation base. For each game level on which adaptations will be applied, we should find or detect a number of points in the code. Concrete examples of these points are given in chapter 3 section 3.3.2, but below we describe a more general pattern that can be used to find these sets of points.

Configuration points

The first set describes points where the user model and the game settings are initialized or reconfigured (reset). The initialization point is in most cases the same as the game initialization. Reinitialization points are points where one wants to (partially) reset user properties or settings. They depend on the game structural level on which one wants to apply adaptations.

Resetting user properties and settings is sometimes needed to avoid the user getting stuck on an unsuited game setting due to legacy data. E.g. the user is bad at

multiplications, resulting in a low accuracy performance feature, resulting in it's turn in easier exercises with smaller operands. But when he wants to change the type of exercises back to additions, he gets addition exercises with small operands as well, because the operand range was not reset.

Assessment points

The second set describes points where data can be collected that can be used to keep the user model up to date. E.g. places where the game gives feedback to user input. Besides the feedback given by the game, we could also assess and record properties such as the emotional state and attention level of the user (e.g. by using special external devices).

Performance is to be taken into consideration for deciding the time and frequency of assessing such external properties. For game feedback, the code can be optimized so that it has a minimal performance impact by using the values computed by the game rather than recalculating them explicitly.

Adaptation settings adaptation points

The third set describes points where adaptation conditions should be checked. Incase of a positive check, the appropriated adaptation settings update will also occur here. In most cases this set will either consist of points where assessment of data used by the adaptation conditions is performed (the second set of points) or points where the settings are applied (the third set of points).

For performance reasons, the programmer could consider less frequently occurring points or choose points where the resources allow us to compute features (as this is possibly a computationally intensive task).

Adaptation settings application points

The fourth set describes points where the game is made more or less challenging or where the relationship between game and didactic elements can be altered, based on the game settings. Points for the first type of adaptations can be for example places where game settings such as game speed or exercise difficulty are applied or updated.

4.1.5 Step 5 : Adding adaptivity to the game

To know what still needs to be done to actually make the game adaptive we summarize the information gathered so far:

- Answers to questions that describe what adaptation is for the target game.
- A user model that describes what features can or should be tracked for each user.
- The adaptations defined by the condition for the adaptation to be applied and the behaviour that changes the adaptation settings when this condition is met.
- A model for the settings that are applied to the game to modify the difficulty or game-didactic relation.
- A model for the settings that are adapted and applied in the game.
- The points in the code where the game starts, where it ends and where the game loop begins.
- The points in the code where the user model and settings are initialized and reconfigured.
- The points in the code where data that is to be used to update the user model, can be tracked.
- The points in the code where settings can be adapted.
- The points in the code where settings can be applied.

The process of adding adaptivity to the game is split up in providing concrete implementations for 5 abstract aspects. Each concrete implementation uses pieces of the information listed above.The design of each aspects is shown in figure 4.5 . We can see they all inherit from the abstract aspect `AdaptivityBaseAspect`, which provides an interface for accessing the user model and the game settings. Debugging functionality is not part of the methodology but is also provided by `AdaptivityBaseAspect`.

As mentioned in step 3 (4.1.3), dividing the main task in different aspects promotes reuse, reduces code entanglement and increases readability. Besides the indirect link to the user model and adaptation settings via the `AdaptivityBaseAspect`, there are no direct dependencies between the aspects. Below we explain the use of each aspect and how it fits in the methodology.

AdaptiveEngine aspect

Exactly one concrete implementation of the `AdaptiveEngine` aspect should be given. This aspect instance will hold the user model and game settings. It will initialize them and allows the programmer to provide custom initialization code such as opening database connections, requesting file handlers or authenticating

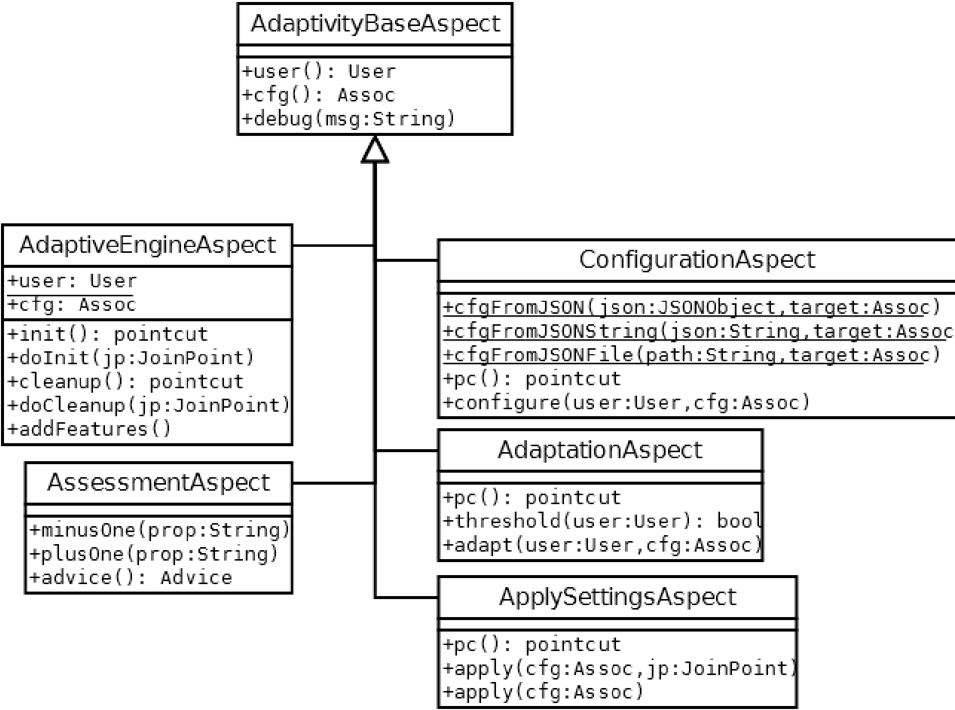


Figure 4.5: Abstract aspects used for adding adaptivity to an educational game

with services that are used for assessments or adaptations. Custom cleanup code can also be provided if necessary.

The programmer is required to define the points where the game starts and where it ends as a pointcut in the aspect language of choice. For flexibility, if made available by the aspect weaver, the concrete join point is passed to both initialization and cleanup methods if join point context is required, though this will rarely be the case for initialization/cleanup as this behaviour is related to resources created for and used by the adaptation process rather than resources for the game.

As said earlier, to avoid a dependency between the aspect language and available reflection tools, the method requires the programmer to manually specify the features that can be computed for the user model via the `addFeatures()` method.

Configuration aspect

The configuration aspect should be implemented for each of the points where either the user profile or game settings should be (re)initialized. The abstract Configuration aspect requires the user to implement the pointcut that describes such a point and provide the behaviour that is related to this point as a method

`configure(User u,Assoc cfg)`. This behaviour is applied as a *before advice* right before each join point described by the pointcut.

The reason to pass both user and settings configuration is aimed at more concrete configuration aspects such as load and save user, which will typically load and save both user data as well as settings data. An alternative approach would be to provide an interface for user configuration aspects and one for settings configuration aspects.

The configuration aspect itself can be extended with some utility functions to batch-initialize user properties or settings. As an example, the default configuration aspect provides utility methods to initialize Assoc objects from a JSONObject, a JSON-valid string or from the path to a file that represents a valid JSONObject. Performance should be taken into consideration when configuring many settings at once or for user models with many properties. More extensions are described in section 7.1.2 on page 92 .

Assessment aspect

The programmer should give concrete `Assessment` aspect implementations for each assessment point defined earlier. This implementation contains any possible type of advice that is applied at a pointcut that describes the given assessment point. The only core tool support for performing assessments is property manipulation, e.g. add (`plusOne()`) or remove (`minusOne()`) one from the value of a property. The property manipulations aim at facilitating interaction with the user model. In statically typed languages this avoids type casting.

Concrete implementations can however extend the core functionality with features targeted at specific cases or patterns of assessment. We could say that the assessment aspect forms the base for more concrete reusable assessment aspects. Possible extensions are described in section 7.1.3 on page 92 .

Providing an interface with methods that can optionally be overridden such as the interface provided by the `AdaptiveEngine` aspect would either introduce less expressivity for the programmer to inspect and manipulate context around the assessment join points or if the same level of expressivity is to be kept, it would introduce issues that relate to the component and aspect language that is used. This is further discussed in section 5.2.6 on page 84 .

Adaptation aspect

`Adaptation` aspects should be implemented for each adaptation (an adaptation condition and behaviour that adapts the adaptation settings) that we described earlier. This is done by providing the adaptation condition as the boolean method

`threshold(User u)`, the adaptation settings adaptation as the method `adapt(User u, Assoc cfg)` and the pointcut that describes the points where the adaptation condition should be checked. If the condition returns a positive truth value (true), the adaptation behaviour is applied as *before advice* on the join points that are described by the given pointcut.

As said in 4.1.4, the available (computation) time at the join points described by the pointcut is to be taken into consideration. The programmer could wrap the adaptation condition with code that only checks the condition if there is enough computation time available at that moment (e.g. more time can be dedicated to computing features when the game shows loading screen than in between meteors spawning in TuxMath). If there is not enough computation time, the threshold function immediately returns a negative truth value (false).

Note that we could argue to provide a more generic adaptation aspect that allows different types of advices and an easier interface to access contextual information on the concrete join point. However, our design hints that adaptations should not use contextual data but only use the features from the user model that depend on the game's contextual information that was extracted by concrete assessment aspects. If we want to extend the base functionality of the adaptation aspect with support for checking the available time, contextual information can become relevant.

ApplySettings aspect

Finally a concrete implementation for the `ApplySettings` aspect should be given for each of the settings application points. The basic interface requires the programmer to provide a pointcut that describes such a point and the behaviour that applies the (adapted) game settings from the AdaptiveEngine aspect as a method `apply(Assoc cfg,)`. The behaviour is applied as a *before advice* to the join points described by the given pointcut.

Unlike with the `Adaptation` aspect, the programmer could benefit from more expressiveness and flexibility on where and how the settings are applied, as the programmer might want to access the target or the arguments of a method call to modify them based on the settings. For this reason, the interface of `ApplySettings` offers a second method `apply(Assoc cfg, JoinPoint jp)` that gives the programmer full access to the context of the join point to which the before advice was applied. Since we restrict the user only to provide before advice (unlike with the Assessment aspect, where the programmer could also need around advice), the problems related to type safety caused by around advice are not present, as before advice does not require a return type.

Although this interface already provides a lot of flexibility to the programmer, for some component languages, the programmer will have to resort to the metaprogramming facilities to modify restricted or hidden fields and variables that are used

by the game to determine the difficulty, appearance or the relation between didactic and game elements. Possible extensions to the basic `ApplySettings` aspect target reflection tools, as stated in section 7.1.5 on page 93 .

As for the performance impact there is no general rule since the performance impact can vary wildly depending on what impact the settings have on the game. For example, a game setting that determines to use visual, textual or auditive learning techniques (based on the learning style feature) has a much larger impact than modifying the range of operands of a mathematical exercise.

4.2 Summary

We summarize the 5 steps of the methodology to make an existing game adaptive using AOP techniques and principles.

Step 1: Answer questions that describe what, where and how the game will be adapted, based on one or more exploratory game sessions.

Step 2: Based on these answers, design a user model that can hold the data that will be used to provide more complex features about the user. Define the features that will be used to decide when adaptations will be made.

Step 3: Define the settings that will modify the game when applied (adaptation settings). Define the conditions on which to decide to make changes to the settings, based on the user features (adaptation conditions). Define the behaviour that updates the settings if an adaptation condition becomes true (settings adaptation). Finally, define the behaviour that applies the settings to the game (settings application).

Step 4: Link the adaptations to the code that implements the game. This consists of defining 5 sets of points in the code:

- **Initialization/Cleanup**: points in the game code where the game starts and ends
- **Configuration**: points in the game code where the adaptation settings and user model are initialized or reconfigured.
- **Assessment**: points in the game code where data, that is used to update the properties of the user model, can be collected.
- **Settings adaptation**: points in the game code where the adaptation conditions can be checked and if a positive reply is given, the adaptation settings are adapted with the settings adaptation behaviour.
- **Settings application**: points in the game code where the game could be altered according to the adaptation settings.

Step 5: Add the code to invoke the adaptations. For each set of the 5 sets of points from step 4, an abstract aspect is provided by the methodology. One must provide a concrete implementation of an aspect interface for each point in a set. The aspects are enumerated with respect to the enumeration order from step 4.

- Exactly one **AdaptiveEngine aspect** implementation takes care of the initialization and cleanup for all the other adaptivity components (aspects, user model, adaptation settings, external resources).
- Implementations for the **Configuration aspect** take care of (re)initialization of (parts of) the user model and adaptation settings.
- Implementations for the **Assessment aspect** collect the user data at the assessment points and record them in the user model.
- Implementations for the **Adaptation aspect** update the adaptation settings based on adaptation conditions on the user features.
- Implementations for the **ApplySettings aspect** make the actual adaptations to the default game's behaviour, based on the adaptation settings.

4.3 Proof of concept

To illustrate the use of the methodology, we created a small textual game that presents the important facets and shows the basic guidelines and functionality, provided by the methodology. The game is basically a quiz where all questions are mathematical computations, much like TuxMath except it is entirely textual for easier testing and less cluttered code. The game code is written in Java (component language) and uses AspectJ to weave adaptation related aspects in the game code. The interested reader can find the documented source code in appendix A or online as a github repository[2].

A second goal of this very simplistic game is providing a simple test case for concrete implementations of the aspect models described by the method, aimed at different component and aspect languages. The component code tries not to depend on any language or operating-system features (often the case with graphical toolkits) in order to allow easier translation to other component languages. The aspect implementations can be translated fairly straightforward to other aspect languages and weavers as long as the AOP basics described in section 2.4.2 on page 39 are present. Translation to aspect languages with dynamic aspect weavers (such as Dojo AOP, see 2.4.3) will differ slightly in how and when the aspects are applied. To support these claims, a second implementation for the same game is also given with JavaScript as component language and Dojo AOP as the aspect system (see

[2]https://github.com/Bennit/GenericAdaptivityJ

appendix B or github repository[3]). A third implementation with C++ as component language and AspectC/C++ as the aspect system exists, but at the time of writing, the implementation is not functional, nor complete.

4.3.1 Output comparison

Together with the implementation, a test script is also provided which plays the game for three rounds. The first round the game starts off at the default difficulty settings which entails that the operands of each computation go from 0 to 20. The script answers every question correctly causing the game difficulty to increase for the second round. The operands are now in the 0 to 2000 range. In round 2 the script doesn't give any right answer so the difficulty goes down again in round 3, with operands in the 0 to 200 range. Figure 4.6 shows an extract of the output produced by the game. Note that a large factor was chosen by which the operands change to make the changing difficulty more explicit. Figure 4.7 shows the output of running the game with the same script without weaving the concrete adaptation aspects in the code. We clearly see a difference in round 2 where the operands are proportionally bigger in the adaptive game.

Welcome to MathGame!		
Let's start counting.	-------------------------	-------------------------
Round 1	Round 2	Round 3
Question 0: 15 * 1 = 15	Question 0: 860 - 89 = 7712	Question 0: 18 + 16 = 34
Correct!	Wrong, the correct answer was 771	Correct!
Question 1: 19 * 14 = 266	Question 1: 1092 * 1336 = 14589122	Question 1: 1 - 17 = -16
Correct!	Wrong, the correct answer was 1458912	Correct!
Question 2: 19 + 4 = 23	Question 2: 317 * 873 = 2767412	Question 2: 1 - 4 = -3
Correct!	Wrong, the correct answer was 276741	Correct!
Question 3: 12 + 3 = 15	Question 3: 1129 - 1832 = -7032	Question 3: 1 - 16 = -15
Correct!	Wrong, the correct answer was -703	Correct!
Question 4: 0 + 8 = 8	Question 4: 1957 * 1104 = 21605282	Question 4: 1 - 4 = -3
Correct!	Wrong, the correct answer was 2160528	Correct!
Question 5: 9 * 4 = 36	Question 5: 323 - 1169 = -8462	Question 5: 16 - 14 = 2
Correct!	Wrong, the correct answer was -846	Correct!
Question 6: 7 - 5 = 2	Question 6: 1935 - 1243 = 6922	Question 6: 1 - 15 = -14
Correct!	Wrong, the correct answer was 692	Correct!
Question 7: 11 * 15 = 165	Question 7: 479 + 1935 = 24142	Question 7: 3 + 0 = 3
Correct!	Wrong, the correct answer was 2414	Correct!
Question 8: 2 + 2 = 4	Question 8: 704 + 1246 = 19502	Question 8: 15 * 18 = 270
Correct!	Wrong, the correct answer was 1950	Correct!
Question 9: 12 - 5 = 7	Question 9: 1094 - 1413 = -3192	Question 9: 16 * 14 = 224
Correct!	Wrong, the correct answer was -319	Correct!
Round grade: 10/10	Round grade: 0/10	Round grade: 10/10

Figure 4.6: Extract of the output produced by adaptive MathGame.

4.3.2 Implementation effort

The implementation of MathGame is straightforward, using the design shown in picture 4.8 . The `Game` class starts the gameloop and continues to query the user

[3] https://github.com/Bennit/GenericAdaptivityJS

```
Welcome to MathGame!
Let's start counting.
Round 1
Question 0: 19 + 11 = 30
Correct!
Question 1: 10 * 17 = 170
Correct!
Question 2: 8 - 7 = 1
Correct!
Question 3: 9 - 13 = -4
Correct!
Question 4: 9 - 16 = -7
Correct!
Question 5: 6 + 13 = 19
Correct!
Question 6: 6 - 0 = 6
Correct!
Question 7: 2 - 6 = -4
Correct!
Question 8: 9 * 0 = 0
Correct!
Question 9: 0 - 16 = -16
Correct!
Round grade: 10/10
```

```
--------------------------
Round 2
Question 0: 17 + 7 = 242
Wrong, the correct answer was 24
Question 1: 19 * 10 = 1902
Wrong, the correct answer was 190
Question 2: 7 + 12 = 192
Wrong, the correct answer was 19
Question 3: 14 * 6 = 842
Wrong, the correct answer was 84
Question 4: 10 * 13 = 1302
Wrong, the correct answer was 130
Question 5: 10 + 17 = 272
Wrong, the correct answer was 27
Question 6: 14 - 2 = 122
Wrong, the correct answer was 12
Question 7: 0 + 9 = 92
Wrong, the correct answer was 9
Question 8: 15 * 7 = 1052
Wrong, the correct answer was 105
Question 9: 0 + 10 = 102
Wrong, the correct answer was 10
Round grade: 0/10
```

```
--------------------------
Round 3
Question 0: 4 * 0 = 0
Correct!
Question 1: 13 * 3 = 39
Correct!
Question 2: 18 - 19 = -1
Correct!
Question 3: 0 - 14 = -14
Correct!
Question 4: 3 * 1 = 3
Correct!
Question 5: 17 + 11 = 28
Correct!
Question 6: 6 * 18 = 108
Correct!
Question 7: 10 - 1 = 9
Correct!
Question 8: 16 + 15 = 31
Correct!
Question 9: 7 + 16 = 23
Correct!
Round grade: 10/10
```

Figure 4.7: Extract of the output procuded by normal MathGame.

with rounds of questions until the user wants to stop. Questions are composed of two operands and an operator. Operators can be applied to a left-hand operand and a right-hand operand. Input and output are abstracted, to keep the implementation simple and platform independent. In the Java implementation, input and output is done via the console, while the Javascript implementation handles input and output through DOM events and updates. The JavaScript MathGame implementation counts 95 lines of JavaScript code and 15 lines of HTML code. The Java version counts 150 lines of code (LoC).

To implement adaptivity, we created a detailed design that describes the concrete implementations of the aspect interfaces provided by GAAOP, as seen in figure 4.9 . The design counts 1 concrete class and 8 concrete aspects (in the JavaScript implementation, the feature class and subclass were replaced by a simple function).

On average, the concrete implementations of GAAOP Java aspects count 20 LoC, and 178 LoC in total. The concrete JavaScript GAAOP aspects count 13 LoC on average, and 118 LoC in total. To add the concrete aspects to the Java version, no additional code was needed. In the JavaScript version however, `gaaop.js` takes care of practical, implementation-specific issues such as adding the aspects at runtime. To handle inclusion and initialization of the JavaScript GAAOP framework, 20 lines are inserted manually in `game.html`. These could be reduced to 2 lines of code but this was not done to avoid browser restriction issues.

The time that was needed to implement and test adaptivity using our framework was around 3 hours of programming in both Java and Javascript implementations (not counting the time that was needed to create the actual game, or the

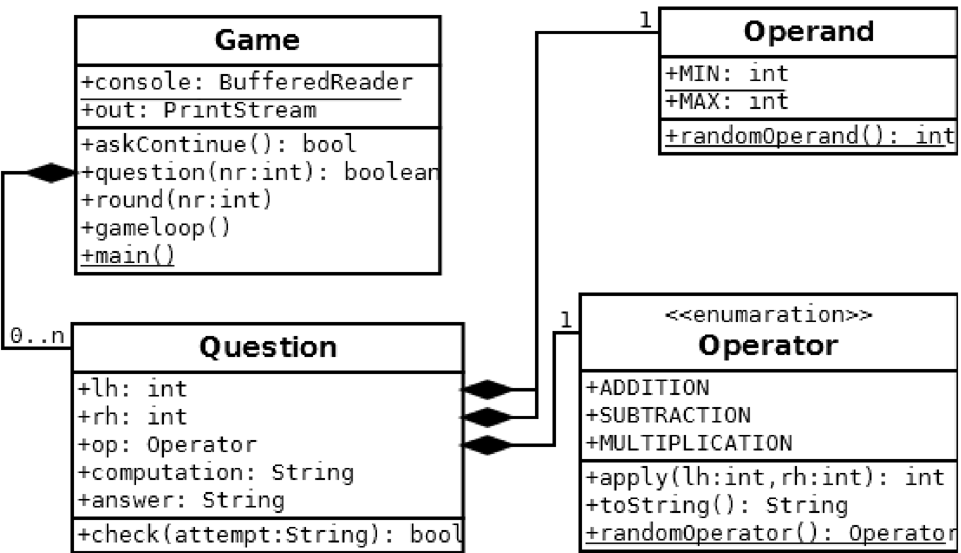


Figure 4.8: Design for MathGame

GAAOP framework). Most encountered issues when creating the concrete aspect implementations, were related to the proper use of the aspect system at hand. Though, the time spent on a first experimental and non-functional implementation in C++/AspectC++, should be taken into consideration.

4.4 Proposed method summary

In this chapter we described a methodology to add adaptivity to an existing educational game using a framework consisting of abstract aspects. The methodology guides the programmer in five steps to implement adaptivity for the target game:

- Step 1: Define what kind of adaptivity is required for the target game.
- Step 2: Design a user model that can store and compute the required user features.
- Step 3: Define conditions on these features for adaptations to occur, describe the modifications to the adaptivity settings and the effects of applying the adaptivity settings to the game.
- Step 4: Find in the game code the code needed for initialization, configuration, assessment, settings adaptation, and settings application.
- Step 5: Provide concrete implementations for the abstract aspects: `AdaptiveEngine`, `Configuration`, `Assessment`, `Adaptation`, and `ApplySettings`.

To support the claim that this methodology is applicable for games that are written in programming languages where an aspect system is available, a proof of concept is given in the form of a concrete implementation for the same game (MathGame), written in two different languages (Java and JavaScript), using two different AOP systems (AspectJ and dojox.lang.aspect). A test-case is provided in the form of a scripted control flow, and the output of a version of the game that is not adaptive is compared to the output of an adaptive version, demonstrating that the game's difficulty has been adapted using our methodology and framework. The implementation effort is described by giving the design, time and lines of code needed for implementing the concrete adaptivity aspects.

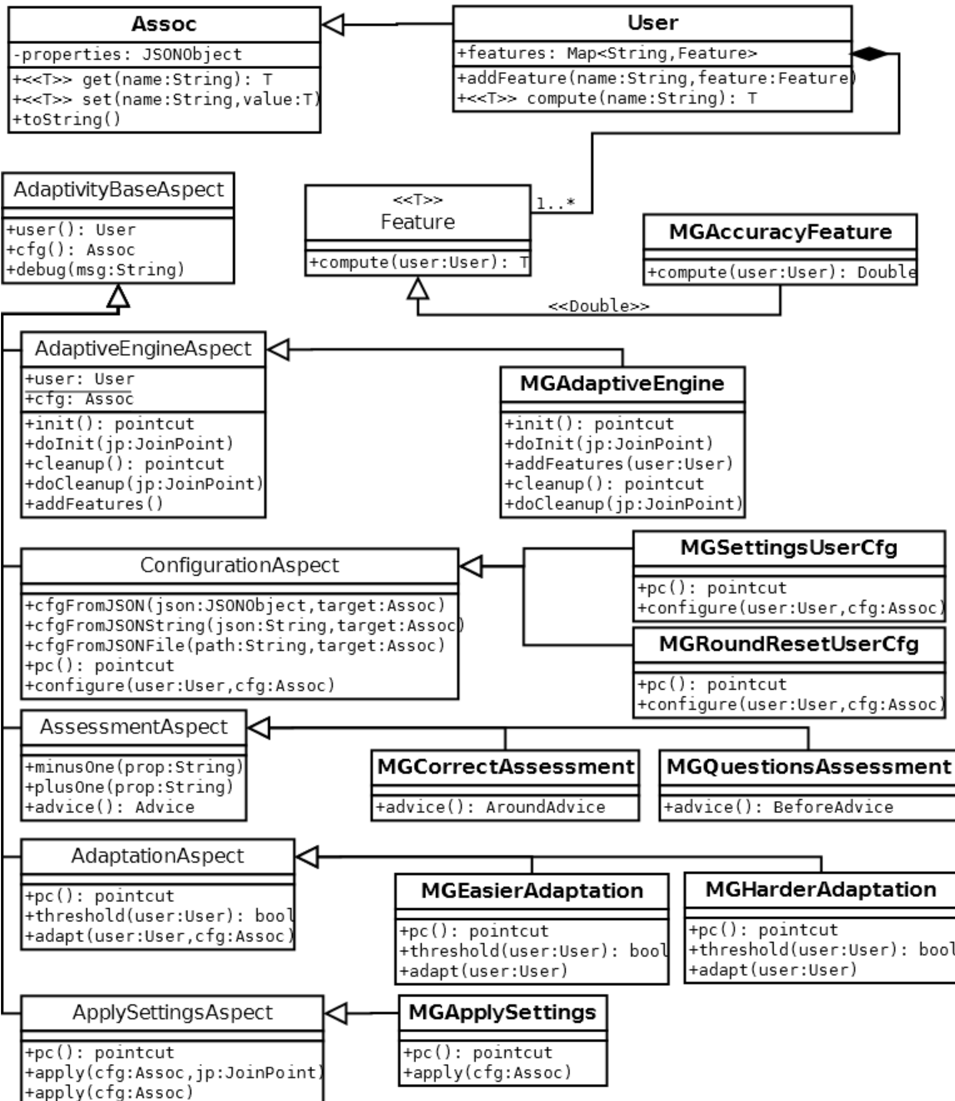


Figure 4.9: Design for the concrete aspects that add adaptivity to MathGame

Chapter 5

Limitations

In this chapter we elaborate on fundamental limitations to solving the problem of making existing games adaptive as well as on limitations that are entailed by the proposed methodology and how these could be solved. The limitations are divided into two groups. The first group of limitations focuses on the game architecture of the existing target educational games. The second group contains limitations that are more specific to the type of adaptations that can be used.

5.1 Game architectural limitations

In section 2.3 on page 19 we claimed that many games contain the same coarse-grained architectural patterns. The methodology seeks a balance between the type of supported games and the amount of support that is given to the programmer. If we want to make the methodology applicable to games that are not based on the described traditional patterns, less support and guidance in terms of implementation can be provided to the programmer. The methodology seeks a balance between both sides by introducing dependencies and restrictions on game-architecture related concepts that are explained below.

5.1.1 Object-oriented design style dependency

The software object model, as described in section 2.3.1 on page 22, specifies the language features that are needed to implement the game world model. Often an object-oriented (OO) language is chosen to keep the implementation of the game world model close to the design. If a different language paradigm (such as declarative or functional) is used, implementing the design of the game world model is less straightforward as it was modelled towards an object-oriented implementation. Although our methodology targets all educational games, following

the same argumentation that most games are implemented using an OO-language, the OO-design style was chosen to model the library that is provided to adaptivity-programmers.

5.1.2 AOP dependency

GAAOP depends on AOP techniques to add adaptivity. We defend this dependency with the main motivation to use Aspect-Oriented programming: separate the adaptivity concern from the core of the game. The benefits of using AOP are described in section 2.4 on page 38 . The use of AOP for implementing game features such as adaptations is supported by Uir Kulesza et. al[23]. In this work, the extensibility of existing frameworks is improved by AOP, based on their claim that many frameworks have a crosscutting nature. As a case study, features such as drawing loading screens and drawing alternative images, were implemented as aspects to a mobile game engine. These features have comparable crosscutting behaviour to our adaptation features.

The AOP dependency entails the requirement of an aspect system that targets the game's implementation language (or component language), supporting the basic AOP concepts (pointcuts, before, after and around advice, see fullrefaopconcepts). While there exist one or more aspect systems for many traditional languages[1] that are used in games, some games use languages that do not have such an aspect system implementation (yet), such as the custom scripting languages mentioned in section 2.3.1 on page 21 .

To the best of our knowledge, no general solution exists that provides an aspect system to any language paradigm. The only solution to make the methodology work for any language for which there is no existing aspect system, would be an ad hoc implementation of the aspect system. In [42] two approaches to implementing AOP systems are given: modifying the runtime environment and transforming the code, leaving the runtime environment intact. The latter will be the most likely approach for scripting languages, as these are domain specific languages and as such will typically not provide the programmer with additional tools to modify the runtime environment.

5.1.3 Educational games domain limitation

The work was limited to the domain of existing educational games, as per requirement. Since GAAOP was designed based on the coarse-grained patterns observed in almost all game games (see section 2.3 on page 19), it is fairly safe to assume that GAAOP is also applicable to other high-level types of games such as shooters

[1] http://en.wikipedia.org/wiki/Aspect-oriented_programming#Implementations

or race games. It is our belief that quality of the support that can be provided by the core framework will not improve by restricting the game type to educational games only, as educational-specific support is provided by extensions rather than by built-in functionality.

5.1.4 Support for specific game genres

As done for high-level game types, providing support for specific game genres (such as arcade and third-person games, see section 2.3.2 on page 33) is delegated to extensions on the core framework, allowing us to make adaptations to any game genre. The game genre determines the requirements for the game engine that is used, but following the same argumentation as used for high-level game types, the same coarse-grained architectural patterns are found in game engines of all genres. If a new genre emerges or a genre for which no supporting extensions exist, the programmer will obviously have less support but the resulting code can be abstracted and reused by other programmers that want to apply adaptations to the same engine genre. This indicates that the methodology would clearly benefit from an open-source community where extensions and implementations are shared amongst the users of the framework.

Support for game engine frameworks

A more specific case of game genre engines is when the target game that is to be made adaptive is implemented using a framework that eases repetitive tasks for a specific genre of games, hiding the complexity of this task and therefore possibly restricting access to parts of the game context. As said in section 2.3.1 on page 30 , these frameworks are usually callback-driven or event-based.

For all frameworks, abstractions could be provided (as an extension to the core GAAOP framework) as pointcuts that expose context that is normally internal to the framework but that is necessary to make the required adaptations. This does imply that extension programmers could be required to research how the framework works internally, as would be the case if the game would be made adaptive manually. The difference is that manipulating a game engine's framework context can be extracted to reusable functions that provide an easier interface to modify the game context.

5.2 Adaptations

The core framework does not provide support for applying any specific adaptation, it only provides interfaces that allow the programmer to implement any possible

adaptation guided by five aspects that are each responsible for a sub concern of applying adaptations: prepare (`AdaptiveEngine`), (re)initialize (`Configuration`), assess (`Assessment`), adapt (`Adaptation`) and apply (`ApplySettings`).

Although it is not the case (yet), the framework could be extended with support for all possible adaptive interventions (see section 2.2.2 on page 14) and adaptation strategies (see section 2.2.3 on page 18). Although we claim that GAAOP allows us to implement every possible adaptation, there is a limited level of support that is provided by GAAOP in some components of the framework. We describe these shortcomings below.

5.2.1 Adaptation thresholds

In the Adaptation aspect, GAAOP does not aid to the programmer in finding the optimal values for upper and lower bounds for numeric (performance) thresholds such as the correctness and average height thresholds in the adaptive version of TuxMath. One possible approach to find good values is to conduct several studies on several test subjects to compare the learning rates using different values for the bounds and statistically determine the values that result in the best learning rates. Setting up these studies is outside of the scope of this thesis but concrete implementations of the methodology's aspect library could be extended with statistical utilities to support conducting the studies.

5.2.2 Applying settings that change the game-didactic ratio

The adaptations made in MathGame, TuxMath and Sumon focussed on game difficulty modification. No adaptations were made that change the ratio between game and didactic elements, because such adaptations can vary a lot in how they are applied (e.g. adding new sprites that show more emotions to make the player emerge better in the game). Typically the effects of game-didactic ratio settings are omnipresent and are applied continuously throughout the game, while difficulty settings are applied only for a discrete number of times. Since there is no example implementation for such adaptations, it is not verified that the core framework offers enough support to implement them.

5.2.3 Applying settings to component code

As already stated in section 4.1.5 on page 73 , in some cases the programmer will have to resort to the metaprogramming facilities provided by the component language to access and modify fields for which access was hidden or restricted by the game programmer. Although out of this thesis' scope, concrete implementations of the ApplySettings aspect could be extended with metaprogramming features such

as scope introspection for the context of a join point and external access to private or protected member variables.

5.2.4 User model DSL

GAAOP requires the programmer to initialize the user model (and the game settings) via method calls to <<T>> set(String name,T value) and addFeature(String name, Feature f). Initialization and reinitialization of properties is delegated to concrete implementations of the Configuration aspect. Determining the longevity of data for properties in the user model and resetting them when they become invalid, can differ for every property. In user models with a large set of properties this can become a very tedious task to implement a Configuration aspect for (re)initializing each property with a different longevity. It would be more concise and clearer to specify the user model in a domain specific language. Although one DSL grammar could be used, bindings will have to be made for translating DSL programs to constructs in the game component language of choice.

5.2.5 Inter-aspect dependencies

The GAAOP framework does not provide support for implementing inter-aspect dependencies (e.g. preparation before configuration before adaptation). Verifying the integrity of such dependencies are left as a task for the programmer. We believe it should be possible to verify dependencies of execution order between the different type of aspects (by adding runtime information about which aspect behaviour was applied and which not). However, since assessment and adaptation aspects can have multiple instances, providing the programmer with a mechanism to verify the integrity of dependencies between these instances is a more complex problem. For example, the game speed should not be altered if the difficulty of the questions was already altered or visa versa. Providing compile-time or runtime checks for ensuring such dependencies is an interesting research area, however for this thesis it is considered to be out of scope.

5.2.6 Generic aspect interface

As mentioned in section 4.1.5 on page 71 , some aspects can benefit from a generic interface that contains methods to be overridden such as the AdaptiveEngine aspect. However, if it cannot be specified in advance which behaviour is required and what context is required by this behaviour, a generic interface introduces some issues. Some of these issues are illustrated by appendix C. It shows the code of a generic aspect that provides such an interface for the statically typed language Java, using AspectJ as aspect language. The same appendix also includes an example for

tracking correct answers as a concrete implementation of an `Assessment` aspect that specifies the generic aspect. The disadvantages and issues introduced by this implementation include:

- In statically typed languages, to remain type safe, code-cluttering exception handling must be taken into account.
- In languages such as Java, native return types (e.g. int, boolean, double) are also tricky since they cannot be passed as class parameters and they would require runtime reflection to get the code accepted by the typechecker.
- When around advice is not needed by the concrete aspect, the default around behaviour can introduce a cost depending on the component language that is used (e.g. java requires us to call `proceed()` within a *try-catch* block).
- Providing advice methods rather than allowing the programmer to use advices directly also implies that the programmer should know the JoinPoint interface to obtain more contextual information such as call arguments and the target object. Using this interface rather than the syntactic sugar provided by aspect languages is yet another burden for the programmer.

Most of the disadvantages are related to statically typed component languages and statically woven aspect languages. Depending on the target component language and AOP system, more or less guidance can be given by the concrete implementation of the GAAOP framework.

5.3 Limitations Summary

In this chapter, we described the limitations and domain restrictions of GAAOP. It is our claim that the methodology is applicable for adding any type of adaptation to virtually every game, but the implementation effort can vary depending on what technologies are (already) available and which reusable extensions to the framework have already been written. A strong limitation of GAAOP is the availability of, or possibility to create, an AOP system that provides the basic concepts as described in chapter 2. Providing such an implementation for some third-party game engines could prove to be difficult, as these engines do not always provide documentation or easy access to meta-data that is provided by most AOP systems. Limitations regarding support for more specific games (e.g. games using the CAAT framework) can be refuted by writing core framework extensions.

Chapter 6

Related Work

In this chapter we discuss work, that was not discussed in chapter 2, and which relates closely to our goal of creating adaptive educational games. The motivation for work done in this area is always the same: providing a personalized learning experience that keeps the learner motivated by leveraging the motivation intrinsic to games. The same as with our work, related work is a combination of research from cognitive sciences and adaptive hypermedia. We discuss ELEKTRA, a European project that was targeted at revolutionising technology-enhanced learning. While the project ended in 2008, other research projects such as 80Days and ALIGN reused ideas and implementations from ELEKTRA. We also discuss both of these projects.

6.1 ELEKTRA

The ELEKTRA project[19], or Enhanced Learning Experience and Knowledge TRAnsfer, was funded by the European Commission, and ended in 2008. The main goal was to create an educational game that could compete with regular commercial games in terms of game experience.

The project implements a 3D game that teaches physics classes from the European curriculum as a story in which players must uncover a conspiracy around Galileo Galilei. Screenshots of the game are shown in figure 6.1 . The game introduces the concept of Micro-level adaptivity as a continuous assessment by interpreting the learner's behaviour in the game and subsequent adaptations to learning situations. ELEKTRA capitalizes on non-invasive adaptations in order to maintain the balance between game immersion and learning experience. Just as with our methodology, ELEKTRA utilizes a learner model that is obtained through assessment, to apply both cognitive interventions (such as competence acquisition and activation interventions) and motivational interventions (e.g. adapting the game's narrative to use

Figure 6.1: Screenshots of the ELEKTRA game (source: [19])

emotions that trigger more brain-activity for the learner at hand).

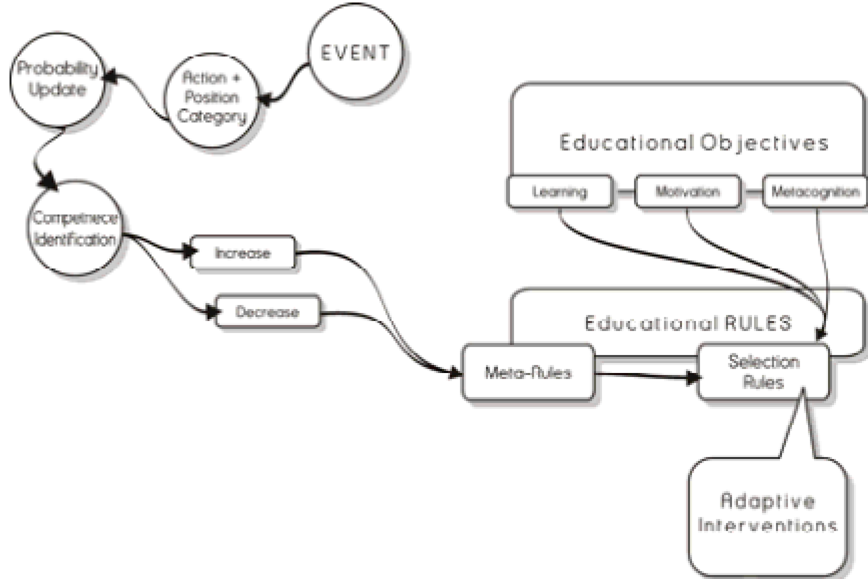


Figure 6.2: Summarization of the ELEKTRA framework (source: [19])

The micro adaptivity framework is summarized in figure 6.2 . It shows the process that leads to assessment and a subsequent adaptive intervention, triggered by an action performed by the learner. After updating the probabilities of the learner solving the possible questions, the learner's competence probabilities are increased or decreased accordingly. Finally a selection of rules is utilized to determine, based on the competencies probabilities, if a specific adaptive intervention should take place. Examples of these selection rules are checks to see if a specific competence probability has gone below or above a certain thresholds.

In many aspects, including the goals and motivation for the project, our project is comparable to the ELEKTRA project. The process leading to adaptations is also much alike, however we believe that our methodology brings the programmer more flexibility in terms of performance fine-tuning of the individual steps that lead to adaptations. Pointcuts can specify exactly for which actions or events assessments and adaptations are run. In ELEKTRA, the framework is applied for each action, having to filter out the unneeded assessments, selection rules and adaptive interventions over and over again. Performance can play a significant role in game programming, certainly when it is the goal to provide educational games which can compete with regular commercial games in terms of the gaming experience.

Another difference is the starting angle of our project. In the ELEKTRA project, a methodology is given to create a new game with adaptive interventions while our methodology allows us to add adaptivity to existing educational games as well as allowing us to build new games and adding adaptivity along the way (e.g. MathGame, see section 4.3 on page 74). Modifying existing games can be seen as an effective way to reduce the development costs of creating an adaptive educational game.

6.2 80Days

The 80Days project[18] is the successor of ELEKTRA, and aims to meld adaptive educational technology with interactive and adaptive storytelling in multiple games. Although the project only implemented one game, a secondary goal of the project is to provide a pool of (expensive) game assets (e.g. 3D models, game world models) that can be reused to create new games with possibly a different educational emphasis. The game aims at improving geography skills. The goal for the learner is to travel the world in a UFO with an alien companion and collect information for an intergalactic travel guide. The learner has to navigate the UFO through different destinations around the world and complete a variety of missions.

Based on the ideas from ELEKTRA, the game developed in the 80Days project performs a non-invasive assessment of learner's competence and motivation. Adaptive skill assessment must be non-invasive rather than explicit testing procedures. This can be achieved by embedding problem solving situations in a game's context and narrative.

A large part of the game's research focus is on providing adaptive storytelling. Whereas some digital educational games focus only on adaptive interactivity, 80Days tries to find a suitable and fair balance between the initially created story and the exceptions that modify the story-pace, caused by user interactions, or educationally inspired adaptations (such as revisiting a certain topic since it was not mastered yet by the learner). Concretely, this results in cognitive interventions such as problem

solving support (e.g. via hints), progress feedback and competence activation interventions (e.g. changing the level of difficulty or the game pace). The game is also adapted with motivational interventions, such as incitation interventions where the entire storyline is changed to be more adapted towards the learner's preferences. Unlike 80Days, the focus of our own methodology doesn't lie with supporting a number of specific interventions but rather on supporting all types of interventions using the same methodology.

Another part of the game's research is to make the adaptive storytelling reusable for different scenarios and educational topics, effectively allowing to create multiple games from the same (expensive) game assets and reducing the overall development cost of creating new educational games. This is in contrast with our methodology, where the starting point can be either an existing game or an entirely new game. If no existing game is available, it can be commercially more interesting to use the 80Days framework, depending on the type of game and the adaptative interventions that are required.

6.3 ALIGN

The ALIGN project[33], or Adaptive Learning In Games through Non-invasion, focuses around the implementation of non-invasive adaptive interventions that personalize the learning experience for the game that was developed in the ELEKTRA project, using their own ALIGN system. In contrast to the 80Days project, ALIGN promotes augmentation over intervention when adapting existing educational game content, in order to avoid the complexity of adaptive narratives. As with 80Days, the learner's motivation is also maintained through adapting the level of challenge that imposed to the learner.

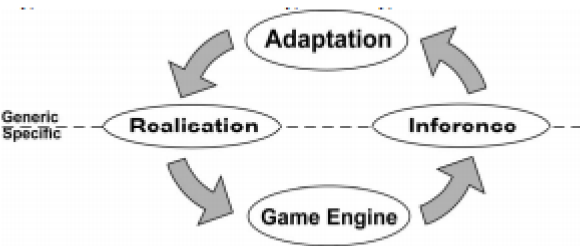


Figure 6.3: Conceptual separation of generic, abstract elements and game specific elements when using ALIGN (source: [33])

Much like our methodology, the ALIGN framework promotes adaptation reuse in multiple different games. To effectively use abstracted adaptations in ALIGN, a mapping must be made between game specificities and the abstracted logic, as is the case in our methodology where this is done by providing pointcuts and context-

modifying or context-inspecting advice behaviour. This separation between abstract and concrete elements is shown in figure 6.3 , where Realisation (ApplySettings aspect) and Inference (Assessment aspect) provide the bindings to apply the abstract Adaptation (adaptive settings adaptation from the Adaptation aspect) to the actual game.

The big difference between ALIGN and our system is in how the adaptations and the game engine communicate. In ALIGN all communication is done using TCP/IP sockets, while in our methodology, communication is done by weaving the aspect advice behaviour in the original code. Because of this reason, the ALIGN framework currently requires the adaptivity programmer to modify the original code to communicate with the recommendation engine (for the realisation of adaptations) and with the evidence interpretation engine (for user assessment). Our solution uses AOP to avoid having to modify the original code, but introduces a stronger dependency on the interface of functions that are referenced by pointcuts. Depending on how stable the interface of adaptation-triggering behaviour from the original game is, either ALIGN or GAAOP can be considered to be better suited for making an existing game adaptive.

6.4 Related Work Summary

In this chapter, we discussed three projects with the same main goal as our own methodology: adding adaptivity to a video game. The first project, ELEKTRA, implemented a framework creating adaptive games based on assessment components, selection rules and adaptive interventions.

The second project, 80Days, successor of ELEKTRA, divided the research focus between adaptivity and reducing the development cost of educational games. This resulted in a framework to create different educational games that provides an adaptivity storytelling engine as well as cognitive adaptive interventions. To reduce the cost, the framework also allows programmers to reuse game assets such as 3D models.

The third project, ALIGN, leverages the ELEKTRA framework to provide adaptivity to existing educational games. The conceptual separation between adaptivity and the existing game, is very much alike the separation that is provided by GAAOP. The big difference is how adaptivity is added to the game. In GAAOP, this is done using join points and aspect weavers, while in ALIGN this is done by manually inserting code that communicates over TCP/IP sockets with the adaptive engine.

Chapter 7

Future Work

This chapter highlights the areas of further research that can be explored starting from GAAOP. The future work is divided in three areas. The first area is about creating extensions for the core aspects of the framework that provide more support towards more specific types of educational games. The second area targets tools that can be used to facilitate the first four steps in the methodology. The third area expands on applying the method to other games in order to verify the genericity of the methodology.

7.1 Aspect extensions

As explained in 5, many of the limitations are related to a lack of support provided by the core framework. Providing better support can be done by restricting the domain of the target game or restricting the type of adaptivity that is required. With more restrictions, extensions to the core aspects can be created that provide abstraction for repetitive tasks in the given domain. In this section, we give a list of possible extensions to each of the core aspects without being exhaustive.

7.1.1 AdaptiveEngine aspect extensions

A first set of extensions are utility functions that would aid the programmer with subtasks for initializing and cleaning up resources that are used by assessments or adaptations. E.g. functions to access persistent storage with default initialization and cleanup behaviour or functions to communicate with external services such as knowledge databases that could be used during assessment or adaptation.

A second set of extensions are more specific towards the targeted game's implementation language or framework. E.g. predefined pointcuts that describe the

initialization and cleanup points for frameworks where fixed points could be found or depending on the available reflection tools, concrete implementations could inspect the runtime to find all implementations of user features and add them automatically.

Since the AdaptiveEngine holds the user model, a third set of extensions could focus on providing more expressiveness for declaring and modifying the user model. As described in section 5.2.4 on page 84, a DSL for describing the user model could bring more expressiveness. The ad hoc implementation for user models could also be replaced with existing services for tracking a user such as Personis[16].

7.1.2 Configuration aspect extensions

Configuration aspects extensions can be used to provide utility functions to initialize the user model and adaptation settings in batch. They can also provide predefined pointcuts that describe (re)initialization points for frameworks where fixed points can be found. A third possible extension would be to make the user model persistent (e.g. using JSON serialization to save and deserialization to load at some given points) to allow the game to be adapted towards the current user at all times. More functionality will probably have to be provided, such as user authentication, to allow multiple users to play the same game adapted towards themselves.

7.1.3 Assessment aspect extensions

Many types of assessment are possible, meaning many extensions can be given that provide support for each type. E.g. statistical analysis tools to determine which category a user property falls into, or interfaces to external hardware such as EEG-chips that can be used to measure attention level of the player.

The core aspect could also be extended to provide property manipulation functions (e.g. `minusOne` and `plusOne` for numeric properties). This is mainly relevant in statically typed languages as the generic interface of the user model would otherwise require a lot of typecasting to make the type checker accept the code.

If game frameworks are used, extensions can also provide predefined pointcuts to points in the framework's control flow where time is not crucial to the game (e.g. loading screens). These points can be used to perform computationally-heavy or time-consuming assessments.

If the game made adaptive using the methodology is used in a research study, some assessment extensions which have no direct impact on the game could also be desired. E.g. training the facial action unit by linking facial expressions to the results of SAM scale queries, see chapter 3, section 3.2 on page 50.

7.1.4 Adaptation aspect extensions

Finding reusable behaviour for adaptations is not that easy, as it is often very specific towards what adaptations are desired. Some reusable functions could be provided that (partially) handle the adaptive interventions and adaptive strategies explained in chapter 2, section 2.2 on page 12 .

Other extensions could target limitations related to the Adaptation aspect, such as statistical analysis tools to determine (sub)optimal adaptation thresholds or a mechanism to implement inter-adaptation dependencies without having to code them all together in one aspect.

7.1.5 ApplySettings aspect extensions

The only meaningful type of extensions to this aspect would be changing default behaviour in frameworks, based on a setting such as learning style (e.g. change the game's framework predefined avatar sprite to speak questions instead of printing them on the screen for players with an auditory learning style). We don't belief that the programmer could benefit from a more specific interface for this aspect. The core aspect could be extended with reflection tools that work on target game's component language.

7.2 Tool Support

The second area of possible future work is in creating tools that facilitate the first four steps of the methodology. Based on a categorization of the type of educational game, some suggestions towards what adaptivity for this game can be made as well as what data can usually be assessed for the game's category. A visual programming tool can be thought of to define adaptations by selecting the desired user features from the user model, generating the skeleton for an Adaptation aspect implementation. Code inspection tools can detect game loops or points where feedback is given to the user and generate pointcuts from this data.

7.3 Genericity Verification

To verify the genericity of the methodology, the methodology should be applied to more complex games. In chapter 2, section 2.1.3 on page 9 the game Monkey tales is introduced. It blends fun and learning for 2nd through 6th grade scholars, by providing them an interactive 3D world wherein mathematical games are embedded. At the time of writing, the game does not track the learner's behaviour at

runtime and can therefore not be classified under our definition of adaptive educational games. However, the game has the potential to be adaptive on both macro and micro level, just as the 80Days project (see chapter 7, section 6.2 on page 88) and is graphically more attractive, making it an excellent use case for testing our method.

Implementing different forms of adaptivity for this game would also confirm the genericity of the methodology. The game features a third-person three-dimensional view with an avatar to represent the player, allowing many possible adaptations that are not restricted to modifying the difficulty of the game, as was the case on previously tested games. For example making the avatar of the player show the same emotions as those observed from the player would emerge the player further in the game, possibly leading to a better learning rate.

Applying it to more complex games does not necessarily show the genericity of the method towards different component languages. To verify this, the implementation of the adaptive MathGame could be ported to fundamental different component languages (e.g. declarative languages, dynamically typed languages, . . .) using different AOP systems (e.g. dynamic aspect weaving). The work that is invested in porting the implementation can later be reused for adding adaptivity to an existing game that was written in the same component language.

7.4 Future Work Summary

This chapter described three areas of further research for GAAOP. The first area focuses on providing aspect extensions for the core framework that are more specialized towards the target game (e.g. CAAT framework bindings) or towards the required adaptation types (e.g. avatar emotional resemblance). The focus of the second area is on tool support such as code inspection and visualization tools. The focus of the third area is on the verification of the method's genericity, by applying it to other game genres or to games where different adaptations types and strategies are required.

Chapter 8

Conclusions

In the domain of education, much research indicates that learners benefit from a personalized teaching program, compared to the one-size-fits-all approach. Other research shows the benefits from leveraging game flow to keep a learner's motivation high when playing educational games. However, the development cost of high-quality adaptive educational games that can compete with regular, commercial games, is even higher than those of one-size-fits-all educational games.

We believe that GAAOP allows companies, who already invested in educational games, to provide an adaptive learning experience with minimal implementation effort (and thus lower development costs). This is done by following a 5-step methodology that guides the programmer in adding adaptivity to an existing educational game, as explained in 4.2.

GAAOP does not put any limitations on the type of information that can be obtained from the player, however, to reduce implementation costs, extensions can be created that are able to extract specific types of information under some well-specified circumstances (e.g. facial emotion recognition for any game written in C++).

As stated in 1.3, we used an inductive approach to develop the methodology. The implementation of the modified version of the educational game TuxMath, that was described in 3, illustrates the effort that is needed for making an existing educational game adaptive, and shows the cross-cutting nature of the adaptivity concern. AOP allows us to separate the adaptivity concern from the rest of the game code, increasing readability and maintainability on both sides.

In section 2.3 on page 19 we described the similarities and common patterns used to create games. We designed the AOP framework to be as generic as possible by only depending on the most coarse grained pattern that we can find in all games: the presence of a game loop, delegating support for every pattern that is more specific to extensions on the core framework. The patterns described in 2.3 can be

used as a basis to create many reusable extensions. Although we did not investigate a concrete example of a game that utilizes the client-server architecture, it is our belief that the effort required to implement adaptivity can be reduced by extensions that facilitate client-server communication. As for games that utilize the web technology stack, we showed in section 4.3 on page 74 that the framework is compatible with the programming language most commonly used for the web, JavaScript.

Since performance is an important aspect for creating high-quality games, the methodology explained in section 4.1 on page 60 , highlights the performance concerns that should be taken into consideration when providing concrete implementations for the abstract aspects. Again, reusable extensions could help the programmer in fine-tuning the performance (e.g. frequency of running assessment behaviour).

While extensions can be used to automate step 5 of the methodology, the process of adding adaptivity can be further automated by creating code inspection tools, as described in section 7.2 on page 93 .

To conclude, we would like to say that it is our belief that GAAOP can serve as a basis for adding adaptivity with minimal effort to existing educational games. The amount of effort can further be reduced by using extensions for the different components of the framework, each targeting a specific game feature, an adaptation type, or an assessment technique. In comparison to existing research, GAAOP stands out in the separation of the adaptive code and the game code, as well as in extensibility and genericity. The concepts, patterns and framework described in this dissertation can serve as a launching pad for extending the core functionality, and for further research towards even less implementation effort.

Bibliography

[1] Margaret M Bradley and Peter J Lang. Measuring emotion: the self-assessment manikin and the semantic differential. *Journal of behavior therapy and experimental psychiatry*, 25(1):49–59, 1994.

[2] Peter Brusilovsky. Methods and techniques of adaptive hypermedia. *User modeling and user-adapted interaction*, 6(2-3):87–129, 1996.

[3] Peter Brusilovsky. Adaptive hypermedia. *User modeling and user-adapted interaction*, 11(1-2):87–110, 2001.

[4] Mat Buckland. *Programming Game AI by Example*. Wordware Publishing, October 2004.

[5] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. In *ACM SIGPLAN Notices*, volume 34, pages 325–339. ACM, 1999.

[6] Siobhán Clarke and Robert J Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd international conference on Software engineering*, pages 5–14. IEEE Computer Society, 2001.

[7] Ben Corne. Internship report: Making existing educational games adaptive. Technical report, 2013.

[8] Olga De Troyer, Frederic Kleinermann, and Ahmed Ewais. Enhancing virtual reality learning environments with adaptivity: lessons learned. In *HCI in Work and Learning, Life and Leisure*, pages 244–265. Springer, 2010.

[9] William A Drago and Richard J Wagner. Vark preferred learning styles and online education. *Management Research News*, 27(7):1–13, 2004.

[10] The Dojo Foundation. Dojo: The javascript toolkit. `http://dojotoolkit.org/`. Visited on March 20, 2013.

[11] Susan Gauch, Mirco Speretta, Aravind Chandramouli, and Alessandro Micarelli. User profiles for personalized information access. In *The adaptive web*, pages 54–89. Springer, 2007.

[12] Jason Gregory. *Game engine architecture*. Taylor & Francis Ltd, April 2009.

[13] Koen Homblé. Cade project: Protocol for sint-anna study. Technical report, 2012.

[14] Waynne Blue James and Daniel L Gardner. Learning styles: Implications for distance learning. *New directions for adult and continuing education*, 1995(67):19–31, 1995.

[15] Charalampos Karagiannidis and Demetrios Sampson. Adaptation rules relating learning styles research and learning objects meta-data. In *Workshop on Individual Differences in Adaptive Hypermedia. 3rd International Conference on Adaptive Hypermedia and Adaptive Web-based Systems (AH2004), Eindhoven, Netherlands*, 2004.

[16] Judy Kay, Bob Kummerfeld, and Piers Lauder. Personis: a server for user models. In *Adaptive Hypermedia and Adaptive Web-Based Systems*, pages 203–212. Springer, 2006.

[17] Diane Kelly and Jaime Teevan. Implicit feedback for inferring user preference: a bibliography. In *ACM SIGIR Forum*, volume 37, pages 18–28. ACM, 2003.

[18] Michael D Kickmeier-Rust, Stefan Göbel, and Dietrich Albert. 80days: Melding adaptive educational technology and adaptive and interactive storytelling in digital educational games. In *Proceedings of the First International Workshop on Story-Telling and Educational Games (STEG08)*, 2008.

[19] Michael D Kickmeier-Rust, Cord Hockemeyer, Dietrich Albert, and Thomas Augustin. Micro adaptive, non-invasive knowledge assessment in educational games. In *Digital Games and Intelligent Toys Based Education, 2008 Second IEEE International Conference on*, pages 135–137. IEEE, 2008.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[21] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *Lecture Notes in Computer Science 1357*, pages 48–3. Springer Verlag, 1998.

[22] Eric Klopfer, Scot Osterweil, Katie Salen, et al. Moving learning games forward. 2009.

[23] Uirá Kulesza, Vander Alves, Alessandro Garcia, Carlos JP De Lucena, and Paulo Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *Reuse of Off-the-Shelf Components*, pages 231–245. Springer, 2006.

[24] Ramnivas Laddad. *AspectJ in action: practical aspect-oriented programming*, volume 6. 2003.

[25] Sam Lantinga. Simple directmedia layer. http://www.libsdl.org/. Accessed December 10, 2011.

[26] Eugene Lazutkin. dojox.lang.aspect - js library to support aop techniques. http://dojotoolkit.org/reference-guide/1.9/dojox/lang/aspect.html. Visited on March 20, 2013.

[27] Eugene Lazutkin. Aop aspect of javascript with dojo. http://lazutkin.com/blog/2008/may/18/aop-aspect-javascript-dojo/, May 2008. Blogpost. Visited on March 20, 2013.

[28] Ibon Tolosana Ludei. Sumon. https://github.com/hyperandroid/Sumon. [Computer Software], first accessed on October 8, 2012.

[29] Anurag Mendhekar, Anurag Mendhekar, Gregor Kiczales, Gregor Kiczales, John Lamping, and John Lamping. Rg: A case-study for aspect-oriented programming. Technical report, 1997.

[30] Graz University of Technology. 80days. http://www.eightydays.eu/. Accessed on November 12, 2012.

[31] Tim OReilly. Web 2.0: compact definition. *Message posted to http://radar.oreilly.com/archives/2005/10/web_20_compact_definition.html*, 2005.

[32] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[33] Neil Peirce, Owen Conlan, and Vincent Wade. Adaptive educational games: Providing non-invasive personalised learning experiences. In *Digital Games and Intelligent Toys Based Education, 2008 Second IEEE International Conference on*, pages 28–35. IEEE, 2008.

[34] Prodiagnostiek. Tempotoets hoofdrekenen tot 20 dudal. http://www.prodiagnostiek.be/downloads/Diagnostisch%20materiaal_Tempotoetsen%20hoofdrekenen%20tot%2020%20Dudal.pdf. Visited on March 24, 2013.

[35] CR Snyder and Shane J Lopez. Handbook of positive psychology. pages 89 – 91, 2002.

[36] Torus Knot Software. Ogre3d - object-oriented graphics rendering engine. http://www.ogre3d.org. Visited on February 25, 2013.

[37] Olaf Spinczyk. Aspectc++ website. http://www.aspectc.org/. Visited on March 20, 2013.

[38] Olaf Spinczyk. Aspectc++ - execution model overview. http://www.aspectc.org/fileadmin/documentation/ac-exec-survey.pdf, 2005. Prepared for the European AOSD Network of Excellence.

[39] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 53–60. Australian Computer Society, Inc., 2002.

[40] Christina M Steiner, Michael D Kickmeier-Rust, Elke Mattheiss, and Dietrich Albert. Undercover: Non-invasive, adaptive interventions in educational games. In *Proceedings of 80Days 1st International Open Workshop on Intelligent Personalisation and Adaptation in Digital Educational Games*, pages 55–65, 2009.

[41] Larian Studios. Monkey tales. http://www.monkeytalesgames.com. [Computer Software], first accessed on November 10, 2011.

[42] Éric Tanter and Jacques Noyé. Motivation and requirements for a versatile aop kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004), Berlin, Germany*, 2004.

[43] Ibon Tolosana. Caat - canvas advanced animation toolkit. http://labs.hyperandroid.com/static/caat/. Visited on February 25, 2013.

[44] Tux4Kids. Tux of math command (version 2.0.1, april 2011). http://tux4kids.alioth.debian.org/. [Computer Software], first accessed on November 10, 2011.

[45] Richard Van Eck. Digital game-based learning: It's not just the digital natives who are restless. *EDUCAUSE review*, 41(2), 2006.

[46] Wikipedia. Aspect-oriented programming — Wikipedia, the free encyclopedia. Visited on March 10, 2013.

[47] Benedict Wydooghe, Evelien De Pauw, Stefaan Pleysier, Jan Van Looy, Jeroen Bourgonjon, Kris Rutten, Steven Vanhooven, and Ronald Soetaert. Game on! we krijgen er niet genoeg van. *viWTA dossiers*, 2008.

Appendices

Appendix A

MathGame and GAAOP in Java

A.1 be.bennit.mg

This package contains the code for running MathGame and the test script.

```
1  package be.bennit.mg;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.io.PrintStream;
7
8  public class Game {
9
10         public static int Q_PER_ROUND = 2;
11
12         public static void main(String[] args) {
13
14                 writeln("Welcome␣to␣MathGame!");
15                 writeln("Let's␣start␣counting.");
16                 gameloop();
17                 writeln("Count-on!");
18
19         }
20
21         // Start the gameloop until the player wants
              to stop
22         static void gameloop() {
```

```
23                  int i = 1;
24                  do {
25                          round(i);
26                          i++;
27                  } while (askContinue());
28          }
29
30          // Start a round of questions
31          static void round(int nr) {
32                  writeln("Round "+nr);
33                  int correct = 0;
34                  for(int i = 0; i < Q_PER_ROUND; i++) {
35                          if(question(i+1))
36                                  correct++;
37                  }
38                  writeln("Round grade: "+correct+"/"+
                       Q_PER_ROUND);
39                  writeln("
                       ---------------------------------")
                       ;
40          }
41
42          // Ask a question and check the answer
43          static boolean question(int nr) {
44                  Question q = new Question();
45                  write("Question "+nr+": "+q.
                       computation+" = ");
46                  if(q.check(read())) {
47                          writeln("Correct!");
48                          return true;
49                  }
50                  else {
51                          writeln("Wrong, the correct 
                               answer was "+q.answer);
52                          return false;
53                  }
54          }
55
56          static boolean askContinue() {
57                  write("Do you want to go for another 
                       round (Y/n)? ");
58                  return ! read().equals("n");
59          }
60
```

```
61          // Reading & Writing
62      static BufferedReader in =
63        new BufferedReader(new InputStreamReader(System.
            in));
64      static PrintStream out = System.out;
65      static void write(String s) {
66          out.print(s);
67      }
68      static void writeln(String s) {
69          out.println(s);
70      }
71      static String read() {
72          try { return in.readLine();      }
73          catch (IOException e) { e.printStackTrace();
              return ""; }
74      }
75
76  }
```

Listing A.1: mg/Game.java

```
1   package be.bennit.mg;
2
3   public class Question {
4           public Integer lh;
5           public Integer rh;
6           public Operator op;
7           public String computation;
8           public String answer;
9
10          public Question() {
11                  op = Operator.randomOperation();
12                  lh = Operand.randomOperand();
13                  rh = Operand.randomOperand();
14                  computation = lh+"␣"+op+"␣"+rh;
15                  answer = ((Integer) op.apply(lh, rh)).
                        toString();
16          }
17
18          public boolean check(String attempt) {
19                  return answer.equals(attempt);
20          }
21  }
```

Listing A.2: mg/Question.java

```
1  package be.bennit.mg;
2
3  import java.util.Random;
4
5  public class Operand {
6
7          public static Random r = new Random();
8
9          public static int
10                 MIN_OPR = 0,
11                 MAX_OPR = 20;
12
13         public static int randomOperand() {
14                 return MIN_OPR + r.nextInt(MAX_OPR -
                      MIN_OPR);
15         }
16
17 }
```

Listing A.3: mg/Operand.java

```
1  package be.bennit.mg;
2
3  import java.util.Arrays;
4  import java.util.Collections;
5  import java.util.List;
6  import java.util.Random;
7
8  public enum Operator {
9          ADDITION("+") {
10                 public int apply(int lh, int rh) {
11             return lh + rh;
12         }
13         },
14         SUBTRACTION("-") {
15                 public int apply(int lh, int rh) {
16             return lh - rh;
17         }
18         },
19         MULTIPLICATION("*") {
20                 public int apply(int lh, int rh) {
```

```
21                  return lh * rh;
22              }
23              };
24
25              private final String text;
26              private Operator(String s) {
27                      text = s;
28              }
29
30      public String toString() {
31          return text;
32      }
33
34      public abstract int apply(int lh, int rh);
35
36      private static final List<Operator> VALUES =
37          Collections.unmodifiableList(Arrays.asList(
              values()));
38      private static final int SIZE = VALUES.size();
39      private static final Random RANDOM = new Random();
40
41      public static Operator randomOperation()  {
42          return VALUES.get(RANDOM.nextInt(SIZE));
43      }
44
45  }
```

Listing A.4: mg/Operator.java

```
1   package be.bennit.mg;
2
3   public aspect Script {
4
5           static {
6                   // set the amount of questions per
                       round
7                   Game.Q_PER_ROUND = 10;
8           }
9
10          // track the round nr
11          static int round = 0;
12          before() : call(void be.bennit.mg.Game.round(
                int)) {
13                  round++;
```

```
14          }
15
16          // track the questions
17          after() returning (Question q) : call(Question
                .new()) {
18                  String answer = q.answer;
19                  switch(round) {
20                  case 1: break;
                                            // always correct
21                  case 2: answer+= "5"; break;     //
                        always wrong
22                  case 3: break;
                                            // always correct
23                  }
24                  input(answer);
25          }
26
27          // continue for 3 rounds, then stop
28          before() : call(boolean be.bennit.mg.Game.
                askContinue()) {
29                  if(round == 3) { input("n"); }
30                  else { input("Y"); };
31          }
32
33          // input emulation
34          static String input = "";
35          static void input(String s) { input = s; }
36          String around() : call(String be.bennit.mg.
                Game.read()) {
37                  Game.writeln(input);
38                  return input;
39          }
40
41  }
```

Listing A.5: mg/Script.aj

A.2 be.bennit.gaaop

This package contains the generic GAAOP adaptivity framework.

```
1   package be.bennit.gaaop;
2
```

```
3  public abstract aspect AdaptivityBaseAspect {
4
5          public static User user() {
6                  return AdaptiveEngineAspect.user;
7          }
8
9          public static Assoc cfg() {
10                 return AdaptiveEngineAspect.cfg;
11         }
12
13         public static void debug(String msg) {
14                 if(java.lang.management.
                      ManagementFactory.
15         getRuntimeMXBean().getInputArguments().
16         toString().indexOf("-agentlib:jdwp") > 0) {
17                         System.out.println("[GAAOP]␣"+
                              msg);
18                 }
19
20         }
21
22 }
```

Listing A.6: gaaop/AdaptivityBaseAspect.aj

```
1  package be.bennit.gaaop;
2
3  import org.aspectj.lang.JoinPoint;
4
5  public abstract aspect AdaptiveEngineAspect
6    extends AdaptivityBaseAspect {
7
8          public static User user = new User();
9          public static Assoc cfg = new Assoc();
10
11         // method that gets called to add the features
12         protected abstract void addFeatures();
13
14         // adaptive engine initialization interface
15         protected abstract pointcut init();
16         protected abstract void doInit(JoinPoint jp);
17         before() : init() {
18                 addFeatures();
19                 doInit(thisJoinPoint);
```

```
20          }
21
22          // adaptive engine finalization interface
23          protected abstract pointcut cleanup();
24          protected abstract void doCleanup(JoinPoint jp
                );
25          after() : cleanup() {
26                  doCleanup(thisJoinPoint);
27          }
28
29  }
```

Listing A.7: gaaop/AdaptiveEngineAspect.aj

```
1   package be.bennit.gaaop;
2
3   import java.util.Set;
4
5   import org.json.JSONObject;
6
7   public abstract aspect ConfigurationAspect
8     extends AdaptivityBaseAspect {
9
10          protected static void cfgFromJSON(JSONObject
                json, Assoc target)
11    {
12
13      @SuppressWarnings("unchecked")
14                  Set<String> keys = json.keySet();
15
16                  for(String key : keys) {
17                          target.set(key,json.get(key));
18                  }
19
20          }
21
22          protected abstract pointcut pc();
23          protected abstract void configure(User u,
                Assoc cfg);
24          before(): pc() {
25                  configure(user(), cfg());
26          }
27
28  }
```

Listing A.8: gaaop/ConfigurationAspect.aj

```
1  package be.bennit.gaaop;
2
3  public abstract aspect AssessmentAspect
4    extends AdaptivityBaseAspect {
5
6          protected void plusOne(String prop) {
7                  user().set(prop,(int) user().get(prop)
                       + 1);
8          }
9
10         protected void minusOne(String prop) {
11                 user().set(prop,(int) user().get(prop)
                       - 1);
12         }
13
14 }
```

Listing A.9: gaaop/AssessmentAspect.aj

```
1  package be.bennit.gaaop;
2
3  public abstract aspect AdaptationAspect
4    extends AdaptivityBaseAspect {
5
6          abstract protected pointcut pc();
7          abstract protected boolean threshold(User u);
8          abstract protected void adapt(Assoc cfg);
9
10         after() : pc() {
11                 if(threshold(user())) {
12                         adapt(cfg());
13                 }
14         }
15
16 }
```

Listing A.10: gaaop/AdaptationAspect.aj

```
1  package be.bennit.gaaop;
2
3  import org.aspectj.lang.JoinPoint;
```

```
4
5 public abstract aspect ApplySettingsAspect
6   extends AdaptivityBaseAspect {
7
8         protected abstract pointcut pc();
9         protected abstract void apply(Assoc cfg,
              JoinPoint jp);
10
11        after() : pc() {
12                this.apply(cfg(), thisJoinPoint);
13        }
14
15
16 }
```

Listing A.11: gaaop/ApplySettingsAspect.aj

```
1 package be.bennit.gaaop;
2
3 public interface Feature<T> {
4         // Compute the feature based on the user
5         public T compute(User u);
6 }
```

Listing A.12: gaaop/Feature.java

```
1 package be.bennit.gaaop;
2
3 import java.util.HashMap;
4
5 public class User extends Assoc {
6
7         HashMap<String,Feature<?>> features;
8
9         public User() {
10                super();
11                features = new HashMap<>();
12        }
13
14        public void addFeature(String name, Feature<?>
              ft) {
15                features.put(name,ft);
16        }
17
18        @SuppressWarnings("unchecked")
```

```
19          public <T> T compute(String ft) {
20                  return (T) features.get(ft).compute(
                       this);
21          }
22
23
24  }
```

Listing A.13: gaaop/User.java

```
1   package be.bennit.gaaop;
2
3   import org.json.JSONObject;
4
5   public class Assoc {
6
7           JSONObject properties;
8
9           @SuppressWarnings("unchecked")
10          public <T> T get(String prop) {
11                  return (T) properties.get(prop);
12          }
13
14          public <T> void set(String prop, T val) {
15                  properties.put(prop,val);
16          }
17
18          public Assoc() {
19                  this.properties = new JSONObject();
20          }
21
22          public String toString() {
23                  return properties.toString();
24          }
25
26  }
```

Listing A.14: gaaop/Assoc.java

A.3 be.bennit.mg.gaaop

This package contains the concrete implementations of the aspects provided by the GAAOP framework.

```
package be.bennit.mg.gaaop;

import org.aspectj.lang.JoinPoint;

import be.bennit.gaaop.AdaptiveEngineAspect;

public aspect MGAdaptiveEngine
  extends AdaptiveEngineAspect {

        protected pointcut init() :
    call(void be.bennit.mg.Game.gameloop());
        protected void doInit(JoinPoint jp) {
                debug("MGAdaptiveEngine␣init");
                // nothing to do
        }

        protected pointcut cleanup() :
    call(void be.bennit.mg.Game.gameloop());
        protected void doCleanup(JoinPoint jp) {
                debug("MGAdaptiveEngine␣cleanup");
                // nothing to do
        }

        protected void addFeatures() {
                user().addFeature("accuracy", new
                   MGAccuracyFeature());
        }

}
```

Listing A.15: mg/gaaop/MGAdaptiveEngine.aj

```
package be.bennit.mg.gaaop;

import be.bennit.gaaop.ConfigurationAspect;
import be.bennit.gaaop.Assoc;
import be.bennit.gaaop.User;

public aspect MGRoundResetUserInit
  extends ConfigurationAspect {

        protected pointcut pc() :
    call(void be.bennit.mg.Game.round(int));
```

```
12
13          protected void configure(User u, Assoc cfg) {
14                  u.set("questions", 0);
15                  u.set("correct", 0);
16          }
17
18  }
```

Listing A.16: mg/gaaop/MGRoundResetUserInit.aj

```
1   package be.bennit.mg.gaaop;
2
3   import be.bennit.gaaop.Assoc;
4   import be.bennit.gaaop.ConfigurationAspect;
5   import be.bennit.gaaop.User;
6   import be.bennit.mg.Operand;
7
8   public aspect MGSettingsUserInit
9     extends ConfigurationAspect {
10
11          boolean first = true;
12
13          protected pointcut pc() :
14      call(void be.bennit.mg.Game.round(int));
15
16          protected void configure(User u, Assoc cfg) {
17                  cfg.<Integer>set("max", Operand.
                       MAX_OPR);
18          }
19
20  }
```

Listing A.17: mg/gaaop/MGSettingsUserInit.aj

```
1   package be.bennit.mg.gaaop;
2
3   import be.bennit.gaaop.AssessmentAspect;
4
5   public aspect MGCorrectAssessment
6     extends AssessmentAspect {
7
8           // correct answers
9           boolean around(String s) :
10                  cflow(call(boolean be.bennit.mg.Game.
                       question(int))) &&
```

```
11                  execution(boolean be.bennit.mg.
                       Question.check(String)) &&
12                  args(s) {
13
14                  boolean answer = proceed(s);
15                  if(answer) {
16                          debug("Attempt␣correct");
17                          plusOne("correct");
18                          return true;
19                  }
20                  else {
21                          debug("Attempt␣incorrect");
22                          return false;
23                  }
24
25          }
26
27  }
```

Listing A.18: mg/gaaop/MGCorrectAssessment.aj

```
1   package be.bennit.mg.gaaop;
2
3   import be.bennit.gaaop.*;
4
5   public aspect MGQuestionsAssessment
6     extends AssessmentAspect {
7
8           // questions
9           before() :
10      execution(boolean be.bennit.mg.Game.question(int))
11    {
12                  debug("Question␣asked");
13                  plusOne("questions");
14          }
15
16  }
```

Listing A.19: mg/gaaop/MGQuestionsAssessment.aj

```
1   package be.bennit.mg.gaaop;
2
3   import be.bennit.gaaop.*;
4
5   public aspect MGEasierAdaptation
```

```
6    extends AdaptationAspect {
7
8          Double limit = 0.5;
9          Double factor = 0.1;
10
11         protected pointcut pc() :
12     call(void be.bennit.mg.Game.round(int));
13
14         protected boolean threshold(User u) {
15                 debug("Running threshold for easier 
                      adaptation");
16                 return (Double) u.compute("accuracy")
                      <= limit;
17         }
18
19         protected void adapt(Assoc cfg) {
20                 debug("Adapting: making it easier");
21                 int max = cfg.get("max");
22                 cfg.set("max",(int) Math.ceil(factor*
                      max));
23         }
24
25 }
```

Listing A.20: mg/gaaop/MGEasierAdaptation.aj

```
1  package be.bennit.mg.gaaop;
2
3  import be.bennit.gaaop.*;
4
5  public aspect MGHarderAdaptation
6    extends AdaptationAspect {
7
8          Double limit = 0.8;
9          Double factor = 10.0;
10
11         protected pointcut pc() :
12     call(void be.bennit.mg.Game.round(int));
13
14         protected boolean threshold(User u) {
15                 debug("Running threshold for harder 
                      adaptation");
16                 return (Double) u.compute("accuracy")
                      > limit;
```

```
17          }
18
19          protected void adapt(Assoc cfg) {
20                  debug("Adapting:␣making␣it␣harder");
21                  int max = cfg.get("max");
22                  cfg.set("max",(int) Math.ceil(factor*
                        max));
23          }
24
25  }
```

Listing A.21: mg/gaaop/MGHarderAdaptation.aj

```
1   package be.bennit.mg.gaaop;
2
3   import org.aspectj.lang.JoinPoint;
4
5   import be.bennit.gaaop.ApplySettingsAspect;
6   import be.bennit.gaaop.Assoc;
7   import be.bennit.mg.Operand;
8
9   public aspect MGApplySettings
10    extends ApplySettingsAspect {
11
12          protected pointcut pc() : call(void adapt(
                Assoc));
13          // run after adaptations to the configuration
                were made
14          protected void apply(Assoc cfg, JoinPoint jp)
                {
15                  debug("Applying␣settings");
16                  Operand.MAX_OPR = cfg.get("max");
17          }
18
19  }
```

Listing A.22: mg/gaaop/MGApplySettings.aj

```
1   package be.bennit.mg.gaaop;
2
3   import be.bennit.gaaop.Feature;
4   import be.bennit.gaaop.User;
5
6   public class MGAccuracyFeature
7     implements Feature<Double> {
```

```
8
9         public Double compute(User u) {
10                Integer correct = u.get("correct");
11                Integer questions = u.get("questions")
                     ;
12
13                if(questions == 0)
14                        return 1.0;
15                else
16                        return correct / (double)
                             questions;
17        }
18
19 }
```

Listing A.23: mg/gaaop/MGAccuracyFeature.java

Appendix B

MathGame and GAAOP in JavaScript

B.1 MathGame code

This section contains the code for the logic behind MathGame as well as the code for the user interface.

```
1  <html>
2    <head>
3      <script src="operand.js"></script>
4      <script src="operator.js"></script>
5      <script src="question.js"></script>
6      <script src="game.js"></script>
7      <script src="http://ajax.googleapis.com/ajax/libs/
          jquery/2.0.0/jquery.min.js"></script>
8
9      <!-- Entry point for AOP -->
10     <script src="http://ajax.googleapis.com/ajax/libs/
          dojo/1.9.0/dojo/dojo.js"></script>
11     <script src="gaaop/gaaop.js"></script>
12     <script src="gaaop/Assoc.js"></script>
13     <script src="gaaop/User.js"></script>
14     <script src="gaaop/AdaptivityBaseAspect.js"></
          script>
15     <script src="gaaop/AdaptiveEngineAspect.js"></
          script>
16     <script src="gaaop/ConfigurationAspect.js"></
          script>
17     <script src="gaaop/AssessmentAspect.js"></script>
```

```
18      <script src="gaaop/AdaptationAspect.js"></script>
19      <script src="gaaop/ApplySettingsAspect.js"></
           script>
20
21      <script src="adaptivity/MGAccuracyFeature.js"></
           script>
22      <script src="adaptivity/MGAdaptiveEngine.js"></
           script>
23      <script src="adaptivity/MGRoundResetUserInit.js
           "></script>
24      <script src="adaptivity/MGSettingsUserInit.js"></
           script>
25      <script src="adaptivity/MGCorrectAssessment.js"></
           script>
26      <script src="adaptivity/MGQuestionsAssessment.js
           "></script>
27      <script src="adaptivity/MGEasierAdaptation.js"></
           script>
28      <script src="adaptivity/MGHarderAdaptation.js"></
           script>
29      <script src="adaptivity/MGApplySettings.js"></
           script>
30
31    </head>
32    <body>
33      <div id="question" style="border:1px black solid;
           width:400px;height:200px;overflow-y:scroll;"></
           div>
34      <input id="answer" type="text" style="width:400px
           ;"/>
35      <script>
36        GAAOP([MGAdaptiveEngine,
37               MGRoundResetUserInit,MGSettingsUserInit,
38               MGCorrectAssessment,MGQuestionsAssessment
                   ,
39               MGEasierAdaptation,MGHarderAdaptation,
40               MGApplySettings],function() {
41          Game.start();
42        });
43      </script>
44    </body>
```

Listing B.1: game.html

```
var Game = (function() {
  var m = {};

  m.qPerRound = 2;

  m.write = function(str) {
    var qE = document.getElementById('question');
    $(qE).append(str);
    qE.scrollTop = qE.scrollHeight;
  };

  m.writeln = function(str) {
    m.write(str+'<br />');
  };

  m.read = function() {
    var answer = $('#answer').val();
    $('#answer').val('');
    m.writeln(answer);
    return answer;
  };

  m.reader = function() {};
  m.noread = function() { m.reader = function() {} };

  m.askContinue = function(cb) {
    m.reader = function(str) {
      cb(str != 'n');
    }
    m.write("Do you want to go for another round (Y/n)
       ? ");
  };

  m.question = function(nr,cb) {
    m.writeln('Question '+nr);
    var q = new question();
    m.write(q.computation + ' = ');
    m.reader = function(answer) {
      m.noread();
      if(q.check(answer)) {
        m.writeln("Your answer was correct!");
        cb(true);
      }
      else {
```

```
44          m.writeln("Your answer was incorrect");
45          cb(false);
46        }
47      }
48    };
49
50    m.round = function(nr,cb) {
51      m.writeln("Round "+nr);
52      var correct = 0;
53      var correctplus = function(result) { if(result)
          correct ++; };
54      var loop = function(i) {
55        if(i == m.qPerRound) {
56          m.writeln("Round grade: "+correct+"/"+m.
              qPerRound);
57          cb();
58        }
59        else {
60          m.question(i+1,function(result) {
61            correctplus(result);
62            loop(i+1);
63          });
64        }
65      };
66      // start the loop
67      loop(0);
68    };
69
70    m.gameloop = function(i,cb) {
71      m.round(i,function() {
72        m.askContinue(function(cnt) {
73          if(cnt) { m.gameloop(i+1,cb); }
74          else { m.noread(); cb(); }
75        })
76      });
77    };
78
79    m.start = function() {
80      // bind reader
81      $('#answer').keypress(function(e) {
82        if(e.which == 13) {
83          m.reader(m.read());
84        }
85      });
```

```
86      m.writeln("Welcome to MathGame!");
87      m.writeln("Let's start counting.");
88      m.gameloop(1,function() {
89        m.writeln("Count-on!");
90      });
91    };
92
93    return m;
94
95  })();
```

Listing B.2: game.js

```
1  function operand() {
2    return operand.MIN + Math.floor(Math.random() * (
       operand.MAX - operand.MIN + 1));
3  }
4
5  operand.MIN = 0;
6  operand.MAX = 20;
```

Listing B.3: operand.js

```
1  function operator(symbol,apply) {
2    this.symbol = symbol;
3    this.apply = apply;
4  };
5
6  operator.oprs = [
7    new operator("+",function(a,b){return a+b;}),
8    new operator("-",function(a,b){return a-b;})
9  ];
10
11 operator.random = function() {
12   return operator.oprs[Math.floor(Math.random()*
       operator.oprs.length)];
13 };
```

Listing B.4: operator.js

```
1  function question() {
2    this.lh = operand();
3    this.rh = operand();
4    this.op = operator.random();
```

```
5     this.computation = this.lh+" "+this.op.symbol+" "+
         this.rh;
6     this.answer = this.op.apply(this.lh,this.rh);
7   }
8
9   question.prototype.check = function(attempt) {
10    return this.answer == attempt;
11  }
```

Listing B.5: question.js

B.2 GAAOP framework

This section contains the gaaop framework code and the code that starts the framework in a browser environment.

```
1   function AdaptivityBaseAspect () { };
2
3   AdaptivityBaseAspect.prototype.debug = function(s) {
4     console.log("[GAAOP] "+s);
5   };
6
7   AdaptivityBaseAspect.prototype.user = function() {
8     return AdaptiveEngineAspect.user;
9   };
10
11  AdaptivityBaseAspect.prototype.cfg = function() {
12    return AdaptiveEngineAspect.cfg;
13  };
14
15  AdaptivityBaseAspect.prototype._apply = function() {
16    // override
17    console.error("_apply not overriden for "+this.
         __proto__);
18  };
```

Listing B.6: gaaop/AdaptivityBaseAspect.js

```
1   function AdaptiveEngineAspect() {
2     AdaptiveEngineAspect.user = new User();
3     AdaptiveEngineAspect.cfg = new Assoc();
4
5     //this.init = null; // pointcut
```

```
6    //this.doInit = function() { }; // before init pc
       advice
7
8    //this.cleanup = null; // pointcut
9    //this.doCleanup = function() { }; // before cleanup
        pc advice
10
11   //this.addFeatures = function() { };
12 }
13 AdaptiveEngineAspect.inheritsFrom(AdaptivityBaseAspect
     );
14
15 AdaptiveEngineAspect.prototype._apply = function() {
16   var aspect = this;
17   aop.advise(this.init.obj,this.init.pattern,{before:
       function() {
18     aspect.doInit(aop.getContext().joinPoint);}});
19   aop.advise(this.cleanup.obj,this.cleanup.pattern,{
       after: function() {
20     aspect.doCleanup(aop.getContext().joinPoint);}});
21 }
```

Listing B.7: gaaop/AdaptiveEngineAspect.js

```
1  function ConfigurationAspect() {
2    // this.pc = null; // pointcut
3    // this.configure = function(user,cfg) { /* ... */ }
4  }
5  ConfigurationAspect.inheritsFrom(AdaptivityBaseAspect)
     ;
6
7  ConfigurationAspect.prototype._apply = function() {
8    var aspect = this;
9    aop.advise(this.pc.obj,this.pc.pattern,{before:
       function() {
10     aspect.configure(aspect.user(),aspect.cfg());
11   }});
12 };
13
14 ConfigurationAspect.prototype.initialize_json =
     function(assoc,json) {
15   $.extend(assoc,json);
16 };
```

Listing B.8: gaaop/ConfigurationAspect.js

```
1  function AssessmentAspect() {
2    // this.pc // pointcut
3    // this.advice // aspect advice
4  }
5  AssessmentAspect.inheritsFrom(AdaptivityBaseAspect);
6
7  AssessmentAspect.prototype.plusOne = function(p) {
8    this.user().set(p,this.user().get(p)+1);
9  }
10
11 AssessmentAspect.prototype.minusOne = function(p) {
12   this.user().set(p,this.user().get(p)-1);
13 }
14
15 AssessmentAspect.prototype._apply = function() {
16   aop.advise(this.pc.obj,this.pc.pattern,this.advice);
17 }
```

Listing B.9: gaaop/AssessmentAspect.js

```
1  function AdaptationAspect () {
2    // this.pc = null; // pointcut
3    // this.adapt = function(user,cfg) { /* ... */ }; //
        before advice
4    // this.threshold = function(user) { return true/
       false; };
5  }
6  AdaptationAspect.inheritsFrom(AdaptivityBaseAspect);
7
8  AdaptationAspect.prototype._apply = function() {
9    var aspect = this;
10   aop.advise(this.pc.obj,this.pc.pattern,{before:
       function() {
11     if(aspect.threshold(aspect.user())) {
12       aspect.adapt(aspect.user(),aspect.cfg());
13     }
14   }});
15 };
```

Listing B.10: gaaop/AdaptationAspect.js

```
function ApplySettingsAspect () {
  // this.pc // pointcut
  // this.apply(cfg,jp) // before advice
}
ApplySettingsAspect.inheritsFrom(AdaptivityBaseAspect)
   ;

ApplySettingsAspect.prototype._apply = function() {
  var aspect = this;
  aop.advise(this.pc.obj,this.pc.pattern,{before:
     function() {
    aspect.apply(aspect.cfg(),aop.getContext().
       joinPoint);
  }});
};
```

Listing B.11: gaaop/ApplySettingsAspect.js

```
function Assoc() {
  this.properties = {};
}

Assoc.prototype.get = function(p) {
  return this.properties[p];
}

Assoc.prototype.set = function(p,v) {
  this.properties[p] = v;
}
```

Listing B.12: gaaop/Assoc.js

```
function User() {
  this.features = {};
};
User.inheritsFrom(Assoc);

User.prototype.compute = function(ft) {
  var feature = this.features[ft];
  return feature(this);
}

User.prototype.addFeature = function(name,ft) {
  this.features[name] = ft;
```

```
13 }
```

Listing B.13: gaaop/User.js

```
1  /**
2   * This function was coppied from:
3   * http://phrogz.net/JS/classes/OOPinJS2.html
4   */
5  Function.prototype.inheritsFrom = function(
      parentClassOrObject ){
6    if ( parentClassOrObject.constructor == Function )
7    {
8      //Normal Inheritance
9      this.prototype = new parentClassOrObject;
10     this.prototype.constructor = this;
11     this.prototype.parent = parentClassOrObject.
         prototype;
12   }
13   else
14   {
15     //Pure Virtual Inheritance
16     this.prototype = parentClassOrObject;
17     this.prototype.constructor = this;
18     this.prototype.parent = parentClassOrObject;
19   }
20
21   return this;
22 }
23
24 function Pointcut(obj,pattern) {
25   this.obj = obj;
26   this.pattern = pattern
27 }
28
29 var aop = null;
30 var adas = [];
31
32 function GAAOP(aspects,cb) {
33   dojo.require("dojox.lang.aspect");
34   dojo.ready(function() {
35     aop = dojox.lang.aspect;
36     for(var i = 0; i < aspects.length; i++) {
37       var aspect = new aspects[i]();
38
```

```
39        if(aspect.hasOwnProperty('addFeatures')) {
40          aspect.addFeatures();
41        }
42        aspect._apply();
43        adas.push(aspect);
44        console.log("loaded and applied "+aspect.
              constructor.name);
45      }
46      cb();
47    });
48  }
```

Listing B.14: gaaop/gaaop.js

B.3 Adaptive MathGame

This section contains the concrete implementations of the abstract aspects from the GAAOP framework.

```
1   function MGAdaptiveEngine() {
2
3     this.init = new Pointcut(Game,"start");
4     this.doInit = function(jp) {
5       this.debug("MGAdaptiveEngine init");
6       // nothing to do
7     };
8
9     this.cleanup = new Pointcut(Game,"start");
10    this.doCleanup = function(jp) {
11      this.debug("MGAdaptiveEngine cleanup");
12      // nothing to do
13    };
14
15    this.addFeatures = function() {
16      this.user().addFeature("accuracy",
            MGAccuracyFeature);
17    };
18  }
19  MGAdaptiveEngine.inheritsFrom(AdaptiveEngineAspect);
```

Listing B.15: adaptivity/MGAdaptiveEngine.js

```
1   function MGRoundResetUserInit() {
```

```
2    this.pc = new Pointcut(Game,"round");
3    this.configure = function(user,cfg) {
4      this.debug("Resetting user profile");
5      user.set("questions",0);
6      user.set("correct",0);
7    };
8  }
9  MGRoundResetUserInit.inheritsFrom(ConfigurationAspect)
     ;
```

Listing B.16: adaptivity/MGRoundResetUserInit.js

```
1  function MGSettingsUserInit() {
2    var first = true;
3    this.pc = new Pointcut(Game,"gameloop");
4    this.configure = function(user,cfg) {
5      if(first) { first = false; }
6      else { return; }
7      this.debug("Setting max");
8      cfg.set("max",operand.MAX);
9    };
10 }
11 MGSettingsUserInit.inheritsFrom(ConfigurationAspect);
```

Listing B.17: adaptivity/MGSettingsUserInit.js

```
1  function MGCorrectAssessment() {
2    this.pc = new Pointcut(question,"check");
3    var aspect = this;
4    this.advice = {around:function() {
5      var answer = aop.proceed(arguments[0]);
6      if(answer) {
7        aspect.debug("Attempt correct");
8        aspect.plusOne("correct");
9      } else {
10       aspect.debug("Attempt incorrect");
11     }
12     return answer;
13   }};
14 }
15 MGCorrectAssessment.inheritsFrom(AssessmentAspect);
```

Listing B.18: adaptivity/MGCorrectAssessment.js

```
1  function MGQuestionsAssessment() {
```

```
2    this.pc = new Pointcut(Game,"question");
3    var aspect = this;
4    this.advice = {before:function() {
5      aspect.debug("Question asked");
6      aspect.plusOne("questions");
7    }};
8  }
9  MGQuestionsAssessment.inheritsFrom(AssessmentAspect);
```

Listing B.19: adaptivity/MGQuestionsAssessment.js

```
1  function MGEasierAdaptation() {
2    var limit = 0.6;
3    var factor = 0.01;
4    var first = true;
5
6    this.pc = new Pointcut(Game,"round");
7    this.threshold = function(user) {
8      if(first) { first = false; return false; }
9      this.debug("Running threshold function for easier:
           "+user.compute("accuracy")+" <= "+limit);
10     return user.compute("accuracy") <= limit;
11   };
12   this.adapt = function(user,cfg) {
13     this.debug("Making the game easier");
14     var max = this.cfg().get("max");
15     this.cfg().set("max", Math.ceil(max*factor));
16     this.debug('NewMax='+this.cfg().get('max'));
17   };
18 }
19 MGEasierAdaptation.inheritsFrom(AdaptationAspect);
```

Listing B.20: adaptivity/MGEasierAdaptation.js

```
1  function MGHarderAdaptation() {
2    var limit = 0.1;
3    var factor = 100.0;
4    var first = true;
5
6    this.pc = new Pointcut(Game,"round");
7    this.threshold = function(user) {
8      if(first) { first = false; return false; }
9      this.debug("Running threshold function for harder:
           "+user.compute("accuracy")+" > "+limit);
10     return user.compute("accuracy") > limit;
```

```
11    };
12    this.adapt = function(user,cfg) {
13      this.debug("Making the game harder");
14      var max = cfg.get("max");
15      cfg.set("max", Math.ceil(max*factor));
16      this.debug('NewMax='+cfg.get('max'));
17    };
18  }
19  MGHarderAdaptation.inheritsFrom(AdaptationAspect);
```

Listing B.21: adaptivity/MGHarderAdaptation.js

```
1  function MGApplySettings () {
2    this.pc = new Pointcut(Game,"round");
3    this.apply = function(cfg) {
4      this.debug('Applying settings');
5      operand.MAX = this.cfg().get("max");
6    }
7  }
8  MGApplySettings.inheritsFrom(ApplySettingsAspect);
```

Listing B.22: adaptivity/MGApplySettings.js

```
1  function MGAccuracyFeature(user) {
2    correct = user.get('correct');
3    questions = user.get('questions');
4
5    if(questions == 0)
6      return 1.0;
7    else
8      return correct / questions;
9  }
```

Listing B.23: adaptivity/MGAccuracyFeature.js

Appendix C

Generic aspect interface in Java

This appendix illustrates the issues that come with a generic aspect interface in the statically typed Java.

```
1  public abstract aspect GenericAspect<T> {
2
3          protected User user() { return AdaptiveEngine.
               user; }
4          protected Assoc cfg() { return AdaptiveEngine.
               cfg; }
5
6
7          // The pointcut where the advice is applied on
8          protected pointcut pc();
9
10         // We define an annotation @XOROverrideSet(
               setName)
11         // to specify that concrete subclasses should
12         // override at least one of the advises
13         // @see .NET @mustoverride
14
15         @XOROverrideSet("advice")
16         protected void beforeAdvice(JoinPoint jp) {}
17
18         @XOROverrideSet("advice")
19         protected void afterAdvice(JoinPoint jp) {}
20
21         // Can't type check against generic parameter
```

```
           T
22         @SuppressWarnings("unchecked")
23         @XOROverrideSet("advice")
24         protected T aroundAdvice(ProceedingJoinPoint
              pjp) {
25                 try {
26                         return (T) pjp.proceed(pjp.
                              getArgs());
27                 } catch (Throwable e) {
28                         e.printStackTrace();
29                         return null;
30                 }
31         }
32         // We could leave the programmer with an
              interface that
33         // declares throwing Throwable, but this also
              clutters the
34         // code, as concrete classes will also have to
               specify this.
35
36         // Extend this class with even more types of
              advice
37         // such as afterReturning, beforeReturning,
              afterThrowing, ...
38
39         // Note: This aspect causes a performance hit
              each time an
40         // advice is applied, as all the advices have
              to be run and some
41         // of the default advice implementations like
              around need to
42         // ensure type safety to get accepted by the
              type checker
43
44         around() : pc() {
45                 beforeAdvice(thisJoinPoint);
46                 afterAdvice(thisJoinPoint);
47                 return
48                         aroundAdvice((
                              ProceedingJoinPoint)
                              thisJoinPoint);
49         }
50
51 }
```

Listing C.1: GenericAspect.aj

```
1  public abstract aspect GenericAssessment<T>
2          extends GenericAspect<T> {
3
4          protected void plusOne(String prop) {
5                  cfg().set(prop,((Integer) cfg().get(
                      prop)) + 1);
6          }
7
8          protected void minusOne(String prop) {
9                  cfg().set(prop,((Integer) cfg().get(
                      prop)) - 1);
10         }
11 }
```

Listing C.2: GenericAssessment.aj

```
1  public aspect ConcreteAssessment
2          extends GenericAssessment<Boolean> {
3
4          // Question.check's signature can't return non
              -object type
5          // boolean, but has to be modified to return
              Boolean as we
6          // can only pass Object types as generic
              parameters to
7          // classes and aspects.
8
9          protected pointcut pc() : call(Boolean
              Question.check(String));
10
11         protected Boolean aroundAdvice(
              ProceedingJoinPoint pjp) {
12                 try {
13                         // capture the answer
14                         Boolean answer = (Boolean) pjp
                              .proceed(pjp.getArgs());
15                         // check what it was
16                         if(answer) {
17                                 // it was correct
18                                 this.plusOne("correct"
                                      );
```

```
19                          }
20                          return answer;
21                  } catch (Throwable e) {
22                          e.printStackTrace(); return
                               null;
23                  }
24          }
25
26          // The before, after and possible other
               advices are all run,
27          // even though only the aroundAdvice is needed
               .
28
29  }
```

Listing C.3: ConcreteAssessment.aj