UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Margus Luik

Programming Patterns in Computer Games Course

Master's Thesis (30 ECTS)

Supervisors: Dietmar Pfahl, PhD
Raimond-Hendrik Tunnel, MSc

Tartu 2016

Programming Patterns in Computer Games Course

Abstract:

This thesis describes the creation of a new course Programming Patterns in Computer Games (MTAT.03.315). The course covers the application of design patterns to solve recurring problems in game development. Course materials, tasks, grading criteria and an exam were designed as part of the thesis work. Experiential learning technique for teaching of the design patterns was explored through programming tasks. The course was conducted once and feedback from students was collected. Finally the feedback is analyzed and future improvements are proposed.

Keywords:

Design patterns, computer games, teaching, experiential learning

CERCS: P170 Computer science, numerical analysis, systems, control

Kursus "Programmeerimismustrid arvutimängudes"

Lühikokkuvõte:

Selles lõputöös on kirjeldatud uue kursuse Programmeerimismustrid Arvutimängudes (MTAT.03.315) loomist. Antud kursus õpetab tudengitele disainimustrite rakendamist korduvate probleemide lahendamiseks arvutimängude arendamisel. Lõputöö käigus koostati materjalid, ülesanded, hindamisjuhised ning eksam. Katsetati ka kogemuspõhise õppemeetodi rakendamist disainimustrite õpetamisel läbi ülesannete. Kursust viidi üks kord läbi ning koguti tudengitelt tagasisidet. Lõpetuseks esitatakse kogutud tagasiside ning pakutakse välja parendusi.

Võtmesõnad:

Disainimustrid, arvutimängud, õpetamine, kogemuspõhine õppe

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1. Introduction

Computer game development is a very popular subject among the students, yet there are not many ways to learn it. The University of Tartu does have a course named Let's Make Computer Games(P2TP.TK.057). Although main goal of the course is to teach programming using Python in an engaging fashion. The course is also entirely online.

In the past there has been a project based course Computer Game Development and Design (MTAT.03.263). In this course students would form groups and complete a game as a project, but it is currently not being conducted.

Only course to currently feature modern game development content is the Computer Graphics course (MTAT.03.015). During the course the students can choose an Unreal Engine 4 specialization module. The module takes place over eight weeks, of which two focus on scripting.

One reason for the lack of computer games related courses is, that they are difficult to create due to the multidisciplinary nature of the medium. Computer games contain, but are not limited to, sound design, art, storytelling, world design and software development. So to make a course on game development is a daunting challenge. As such instead of trying to create a course for encompass all facets game development, the goal was narrowed down to just teaching programming within the scope of computer games.

Games are some of the largest software projects when measuring the volume of code [1]. As the code is revisited and edited by many programmers, good coding practices and structure are very important to successfully launching a game.

Programming a single system within a game may often be quite simple, the complexity comes from having a large amount of systems constantly interact with each other in a simulation running at more than 60 frames per second.

Due to being unable to find out what exactly is good code, the next best option was chosen. It is difficult to teach writing good code directly, but teaching practices which often result in writing good code should be quite manageable. As such, design patterns made for a good subject matter through which computer games development could be explored.

Because the word design is rather ambiguous within the scope of game development the course was named as Programming Patterns in Computer Games (MTAT.03.315) as inspired by Robert Nystroms book Game Programming Patterns [2].

On the internet there are countless tutorials and libraries for just about any game feature imaginable. Programming Patterns in Computer Games is a course about writing the code that glues these features together into a game.

2. Course Design

The initial goal for the work done as part of this thesis was to reformat and improve the multidisciplinary course Computer Game Development and Design (MTAT.03.263) to better teach the programming part of a video game development. A public version of the course is available online [3].

As a multidisciplinary course featuring programming, modelling, art and other domains of videogames development the criteria for passing that course was varied from student to student. As such the final grading of the course depended mostly on a group project. This way was possible to pass the course entirely by working on art, as long as another student did in the same group did the programming part of the project.

Unfortunately, the course could only take place when instructors from multiple disciplines were able to together and organize it. Due to this and with limits of the grading criteria in mind, it became apparent that instead of reformatting Computer Game Development and Design, it would be better to create an entirely new course. As such a new 3 ECTS course named Programming Patterns in Computer Games (MTAT.03.315) was created.

The focus of the new course is to improve the programming ability of computer science students within the context of game development. The course is aimed at 3rd year bachelor's students who have passed the Object-oriented Programming (MTAT.03.130) course [4].

Design patterns are abstract and change little over time. They are relevant in different frameworks and game engines. Algorithms and data structures were also considered as part of the course, but simply focusing on patterns seemed like a better option, as the University of Tartu already has multiple courses dedicated entirely to algorithms.

The course was designed around the Game Programming Patterns by Robert Nystrom. This book is also available as an online format on the official webpage to everyone for free. The book was published in November 2nd 2014 and is scored 5 stars out of 5 in over 90% of the reviews on Amazon [5].

It was important not to have the course material overlap with the Computer Graphics course (MTAT.03.015), which also teaches topics relevant in the scope of game development. This did not become a problem, as Computer Graphics teaches working with mod-

ern graphics systems, such as OpenGL, and Programming Patterns in Computer games teaches how to implement and tie together different features in computer games by application of design patterns.

The name was chosen as programming patterns, due to design patterns in computer games development could be confused with working on art or game design.

The course was taught in English in order to both make it accessible to foreign students and also due to limited Estonian terminology for design patterns.

Due to this course being given by both me, Margus Luik, and my thesis supervisor Raimond-Hendrik Tunnel, it is important to outline our work allocations during the course.

As part of the thesis work I:

- Made decisions on the design of the course.
- Wrote all the course reading materials.
- Wrote all the tasks.
- Graded all the tasks and provided feedback to the students.
- Wrote and graded all the exams.
- Presented most of the topics in class.
- Assisted students during the practice sessions.

Raimond-Hendrik Tunnel:

- Assisted students during the practice sessions
- Instructed students in class
- Helped start and took part in discussions by asking questions or offering explanations when students were reluctant to participate
- Presented the double buffer and spatial partitioning topics in class.

The following section gives an overview of all work done for and during the course.

2.1 Structure and schedule

Programming Patterns in Computer Games is made up from practice sessions in class and individual work for the students. The course ended with an exam. Students could choose between two exam dates. First one took place few weeks after the end of last practice session and second in June.

The course takes place over eight practice sessions each of which are four hours long. In these sessions students are presented with the theoretical background, involved in discussions and given tasks to work on. Part of the sessions was planned for the students to work on the tasks and get instructions, for this both instructors were present in class at the same time. The details on structure of these sessions are elaborated on in section 2.2 – Didactic Considerations and each of the week's theme is discussed in 3.1 – Course Materials.

In both class and for individual work the course made use of the online learning environment CGLearn [6]. From there students could access materials and tasks. They could also download base code for the task, see instructions, submit solutions and get an overview on how well they are doing. Other online environments were also considered when making the course, for more details see section 2.5 Online Learning Environment.

Individual work effort for students was made up from working on tasks. For each topic presented in class students were also assigned to read the corresponding chapter from the textbook as preparation. Week six of the course, which focused on creational patterns was an exception, as this topic was not part of textbook and students had to rely entirely on class presentation and the written material in CGLearn. Each of the tasks is elaborated on in the section 3.1 Course Tasks.

During the course students were graded on a 100 points scale. 60 points were earned for solving the tasks and 40 for an exam at the end of the course. See Table 1 for the distribution of points.

Students could also receive bonus points that were outside the 100 points limit. The 4 bonus points were available to everyone for an optional task. Additional points could be earned for active class participation or exceptional solution for tasks that went beyond what was set by requirements. The active participation bonus point could be earned by starting discussions, doing a presentation or participating in the debate on week eight. Section 2.4 – Grading Scheme presents additional details and considerations on grading.

Compared to other courses and materials on learning design patterns Programming Patterns in Computer Games tried to innovate by structuring the course around experiential [7] learning methods or also known as learning by doing. For this students will have to complete tasks listed in Table 1.

Table 1: The schedule for Programming Patterns in Computer Games along with the tasks assigned on each of the weeks and the grading points for all parts of the course.

Week	Theme	Assigned tasks	Points
1	Introduction to patterns in Game Development	Hello Unity	6
2	Command	Move Command	6
		Switch Command	3
		Bonus task: Snake	4
3	Subclass Sandbox and Flyweight	Flyweight Replay	6
		Essay on Subclass Sandbox	8
4	State	Refactoring Replay	6
5	Type Object and Prototype	GameObject Clone	6
6	Creational Patterns	Factory Method	3
		Abstract Factory	3
7	Decoupling Patterns	Mixing Patterns	13
8	Optimization Patterns	-	-
-	The Exam		40

A typical online course or tutorial generally follows the didactic teaching route, as can be seen in the section 2.2 Existing Courses. It presents students with a recurring problem and explains why a given pattern should be used to solve it. It then proceeds to either present the code of the pattern or demonstrate implementing the code line by line. Finally, it would end with a conclusion and propose alternative patterns for the problem.

In Programming Patterns in Computer Games students are first presented with general problems the pattern would solve. They would also have to previously and independently read the textbook on a similar, yet different, problem from the one presented in class.

During the course or in the textbook student will never be presented with a complete code to a pattern. Instead they will have to figure the pattern out on their own while solving the task at hand and following provided instructions.

Example code is not provided because a too similar example to the task would trivialize the task's solution, as student could simply rewrite most of the code without having to understand it. Too different code on the other had may mislead or confuse the student.

The tasks were created in and solved within the Unity game engine using C#. See section 3.2 Tasks for elaborations on each of the tasks.

2.2 Existing Courses

In this section related courses on the topic of game development within University of Tartu are described. Due to difficulty of finding game development courses that focus on design patterns, simply other design pattern courses are looked at outside of the university.

Computer Graphics course (MTAT.03.015) teaches basic methods of computer graphics. The course also has an optional specialization module for Unreal Engine 4. This course also teaches the double buffer pattern, due to its importance in graphics libraries.

Commercial Videogames Design & Development (MTAT.03.263) has been previously conducted in the University of Tartu. Last time the course took place in the fall of 2014. In this course students used Unity game engine to complete a game project in groups of about 4 students. Weekly lectures alternated between game theory and Unity related practices.

A course Foundations of Programming: Design Patterns [8] is an online course available at Lynda.com [9]. The course is total of 2 hours and 19 minutes long, presented as video lectures and the source code for each pattern can be downloaded. It teaches the following patterns: strategy, observer, decorator, singleton, state, collection and factory method.

The design patterns are presented as standalone example. The intent of each pattern is discussed and implementation is shown as source code. Does not contain a practical exercise part.

There are also a similar video based course on Java Design Patterns and Architecture [10] on Udemy. This course consists of 19 video lectures over 4 and a half hours and also has no practical part for the students.

Oxford University has a course named Design Patterns [11]. The course teaches history of design patterns and has the following patterns listed as content: model-view-controller; observer; adapter; abstract factory; composite; command; iterator (note: also known as collection); visitor; strategy. Like Programming Patterns in Computer Games, this course also requires prior completion of an object-oriented programming course.

2.3 Didactic Considerations

The course has weekly four-hour long practice sessions, which include both theoretical and practical work. During the theoretical part new material was presented and discussed with the students. Practical part was for the students to solve related tasks and individual feedback.

The classical one lecture and one practice session per week was considered, but that would have had the following problems:

- With practices and lectures running in parallel they may get out of sync. On some cases the lectures may even be a few weeks ahead. To keep the lecture and practice session topics synchronized sets an additional requirement, as all the practice sessions to take about same amount of time as the theoretical parts.
- When practical part does not immediately follow the theoretical part, then there is more time for students to forget the theory before getting to practice.
- Also listening to a 90-minute lecture with no practical work in between, is not recommended, as students able to keep attention [12].

A continuous practice session enables more flexible time sharing between theoretical part and practical part of the session. Also the length of the theoretical content does not need to be optimized for both the available time slot and optimal cover of the subject matter, but simply the latter. When the end of theoretical part is planned for discussion, then having the option carry the discussion over to practice time, removes the risk of having it unnaturally cut short due to time constraints as well.

Due to problems of one two-hour lecture and one two-hour practice session per, a continuous four-hour session was preferred. The sessions were simply called practice session.

General time planning for each practice session:

- 20 minutes revisiting last previous topics or present some points of interest for tasks which deadline has just passed.
- 70 minutes of theoretical part. This is further broken down into topics, with discussions in between. Every 10 to 15 minutes should either be a new topic, a discussion or simply a round for questions.
- 90 minutes of work on tasks. The end of theoretical part and introduction of each task can have an overlap. In the end of the theoretical part the corresponding task description should also serve as an example of the presented material. This way there is no sudden change into the practical part.

2.4 Grading Scheme

The grade was decided on the 100-point scale. Of these 60 points were assigned to tasks and 40 points to the exam. Points for individual tasks are listed in in Compared to other courses and materials on learning design patterns Programming Pat-terns in Computer Games tried to innovate by structuring the course around experiential [7] learning methods or also known as learning by doing. For this students will have to complete tasks listed in Table 1.

Topics covered during each week usually included one to three tasks. Each of the tasks were introduced in the corresponding session.

Each task had two weeks to be solved. This enabled the students to work on a task for a week. After which they could receive feedback or guidance during the following week and still have one week left to submit the final solution.

The task submissions were locked at the start of the practice sessions. This way a correct solution could be presented in class directly after the submission. At that point the problem would still be fresh for the students who recently submitted their final solution. The presentation may also lead to discussion and better understanding of the subject. As most tasks built on previous tasks, the correct solution was also made available to the students as a fallback base code.

2.5 Online learning environment

Before deciding on an online learning environment, the following requirements were set:

1. Students should be able to download base code for the tasks.
2. Students need to be able to read task instructions and other course materials.
3. Students need to be able to submit tasks.
4. Tasks should have deadlines
5. Teachers need to be able to grade tasks
6. Students need to be able to track points standing
7. Teachers need to be able to leave feedback for submitted tasks.

The first requirement can be achieved with a hyperlink and a file hosting service, e.g. Dropbox [13]. Although some environments may make it more convenient by providing a direct upload option to the study system.

Three different options were considered for the online environment of the course. The following subsections discuss the pros and cons of the different environments that were considered.

2.5.1 The Courses page

Courses.cs.ut.ee [14] is the wiki page available to all Institute of Computer Science courses in the University of Tartu. This option was enticing solution due to its simplicity.

For tasks a wiki page could be created for each task or the instructions could be provided as downloadable PDF files. Submitting tasks is also possible and uploaded tasks on the course page come with a timestamp, so a deadline could be enforced. As a downside the task submission has to be manually closed.

The requirements 5 and 6 for teachers to be able to grade students and students being able to get an overview of their current points standing could have been achieved with a Google Spreadsheet [15], which is linked to from the courses page. The sheet would have each student's number of study book along with points for each task.

2.5.2 Moodle

Moodle [16] is a widely used open source online course management system. It has a modular design and the functionality can be extended with a large variety of plugins. Uni-

versity of Tartu maintains its own Moodle service, which is also integrated with the UT Study Information System [17].

Moodle easily covers all the requirements and much more. Although on the other hand the large feature set is not completely positive due to making the development of course more complicated and time consuming, as noted by a number of user reviews [18]. As such Moodle seemed slightly daunting for the given small set of requirements.

As a downside each Moodle course is reused every year with all student related data simply deleted. The teachers are responsible for backing up and storing the data themselves. This is especially bad when a course would be taken over by another teacher, as in this case being able to access previous years for guidance becomes especially important and actually having access to it is not guaranteed by Moodle [19].

Due to not having a need for most of the features offered in Moodle it was decided not to use it.

2.5.3 CGLearn

CGLearn is a learning environment created by Raimond-Hendrik Tunnel to teach the Computer Graphics course.

The system backs up data on Amazon Simple Storage Service(S3) [20] automatically. New courses are created as clones from the old ones. This is a clear advantage over Moodle, as being able to browse previous versions of the course is helpful when trying to improve the course. It also offers statistics and additional feedback for both the teacher and the students during the course.

In the end the decision was made to go with CGLearn as the study online environment. An additional courses page [14] was also added to make the course easier to find for the students. This page contained organizational information of the course and a link to CGLearn.

Decision to go with CGLearn was also influenced by the following reasons:

- I had previously used CGLearn as part of the course Computer Graphics [21] (MTAT.03.315), which had left me with a positive opinion of the learning environment.

- Raimond-Hendrik Tunnel, supervisor of this thesis, is also the author [22] of CGLearn. Due to feedback and experience from this course could be used to further improve the CGLearn system.
- As a personal opinion I found the layout of materials to be better in CGLearn than it would have been on the courses page or as pdf documents that students could download from the courses page.

2.6 Game engine

It was decided that the course should be about programming games and not game engines, although the line between game and a game engine is rather vague [23]. It is best described that systems, which most or all games need are part of the engine. As few examples such things would be a graphics library for rendering visuals, a sound engine for audio, image and 3d model importers, library for reading input from controllers, animation systems, particle effects and a physics engine.

Implementing a custom game engine by integrating preexisting libraries could fill a 3 ECTS course on its own. Due to that, in scope of the goals of this course, it made sense to use a preexisting game engine.

Other reasons for using a game engine:

- For the patterns to be useful in modern game development, they should be applicable inside modern game engines, which are general way of developing games.
- Writing a game on top of a large number of libraries is very time consuming. Integrating and initializing a library for audio, video, input etc.
- Should be open source if possible.

2.6.1 Unity

Unity [24] game engine is a popular modern game engine. According to Unity Technologies itself, they control 45% of the full featured game engine market [25].

Scripting in Unity is done using the C# language, which is then compiled to Mono assemblies. Mono [26] is an open source cross platform implementation of Microsoft's .NET Framework [27]. It is known that Unity Technologies is also working on their own IL2CPP compiler for years, which would compile C# code into C++ and from there use

native compilation for the platform. The compiler is currently used only for iOS 64 and WebGL builds [28].

It is the authors subjective opinion, that Unity has fairly good documentation and due to large community following most problems that student may run into should already be answered on message boards.

2.6.2 Unreal Engine 4

Unreal Engine 4 (UE4) [29] is one of the most visually stunning modern game engines, which has published its full source code and is also available for free [30].

Scripting in UE4 is done in C++, which makes it a good fit for design patterns as many of them can use the friend access modifier [31]. This modifier is unfortunately not available in other languages (e.g. C#).

UE4 also features a very powerful Blueprints Visual Scripting system [32]. Due to BVS being much simpler than C++ much of the online materials for beginners focuses more on BVS. This could be a hindrance as most of the students would be new to UE4, but the nature of the course would have them using C++.

One of the main disadvantages of using UE4 during the course are its compile times. Even when making minor changes these could take up about 20 to 100 seconds [33] . This along with high system requirements of UE4's editor could mean a lot of extra waiting time for the students while working on the exercises.

2.6.3 Monogame

One of the engines considered was Monogame [34]. It is an open source [35] game engine written in C#. It implements the Microsoft XNA game engine API. Microsoft XNA is no longer developed since January 2013 [36], although Monogame is still in active development.

Scripting in Monogame is done using C#, which is then compiled into a Mono assembly.

Monogame's own documentation is not very good and much of it depends on old XNA documentation. E.g. a link to getting started [37] is a completely blank page. Working with this documentation may make the tasks more difficult than they should be.

The advantage of using Monogame for the course, is that the engine is much more lightweight and students both get and have to do more through scripting.

The low quality of documentation for Monogame ruled it out as a reasonable choice. As UE 4 is already taught in Computer Graphics, then for sake of offering more variety to the students Unity seemed like a good pick. So the decision was made to create the tasks within the Unity game engine.

3. Course Materials and Tasks

The following sections describe the main content created for the course Programming Patterns in Computer Games. As additional materials also an exam and grading criteria for the tasks were created, but are intentionally left out, as the thesis is public.

The course materials section describes materials written for CGLearn, as well as how these materials were followed along practice sessions. The purpose of these were to support the text book and better link the content from the book to the tasks.

The content in CGLearn is mostly condensed version of what was being presented during the theoretical parts of each practice session.

The second section describes tasks. The tasks gave students a base code and instructions, which when followed would help the student complete a pattern and implement a new game mechanic.

3.1 Course Materials

The materials played integral part in tying the course together. On entering CGLearn students would land on the materials page, where they would see the course formatted into eight themes in a sidebar. Each of the themes represented a single week and were further broken down into topics. Students would go through the themes in a sequential order at a rate of one per week.

The ordering of the patterns during the course did not follow the textbooks ordering. Instead it was designed around the tasks so that students would mostly be working on a single code base and improve it with additional features throughout the course.

All the pattern specific topics follow a strict format. They would start by presenting the intent of the pattern as it is defined in the course textbook. In chapters where the course textbook is missing the intent, it is presented according to the book Design Patterns: Elements of Reusable Object-Oriented Software [38] (GoF book from here on).

The intent succinctly gives purpose to the pattern and hints at possible method of implementation. Students are not expected to learn much from it at first. Rather they should understand the intent better while going through the rest of the topic and related tasks. When

revisiting the materials page, reading the intent should act as a quick way to refresh the students' memory on the subject.

Following the intent, there would be about two to five paragraphs of text breaking down the intent and giving motivation to use the pattern. This would be achieved by presenting the implementation of a frequently used feature in computer games as a problem and then describing how use of the pattern at hand to solve it.

After the motivation student find a set of characteristics describing when this pattern should be applied. During the course it is often emphasized, that knowing when to apply a pattern is just as important as knowing how to apply it. This section is there to help students better identify a problem the given pattern can be used to solve. The characteristics are based on the applicability sections from GoF book.

After that a class diagram of the pattern is presented along with descriptions of the roles of each of the participants in the diagram. In the builder pattern topic, the class diagram is replaced with a sequence diagram, as a sequence diagram does a much better job at illustrating the pattern.

During the first week there were also topics that were not on patterns and as such did not follow the described structure. Also not all patterns from the course textbook were part of the course and not all the topics in the course were from the textbook.

The following subsections of the thesis will go through each week's theme and topics, and explain the relevance of each in the scope of game development.

The sections also aim to elaborate on the schedule of the course as it was presented in Table 1, by explaining what happened in class, which tasks were assigned and which deadlines had passed.

3.1.1 Introduction to Patterns in Game Development

The theme of the first week was introduction to the design patterns.

About one third of the total time was allocated to revisiting the basics and principles of object-oriented programming (OOP from here on), as working with design patterns otherwise would not be possible. It was expected that students knew all of this from Object-oriented programming course.

This section was also necessary as Object-oriented programming is taught using Java, but Programming Patterns in Computer Games uses C#. As such the revisiting of OOP was tied together with introduction to C#. As both languages syntaxes are based off of C syntax, it is an easy transition to make.

As the course tries to follow experiential learning methodologies the OOP material was presented as a live demo and time was given for students to follow along. The presentation focused on following features:

- Encapsulation,
    - Classes and structures in C#
    - Using constructors and destructors
    - Access modifiers
- Inheritance
    - Using new and override keywords
- Polymorphism
    - Virtual and abstract keywords

For each of the subjects there is also a summary written up in CGLearn, along with references to appropriate tutorials on Microsoft Developer Network [39] .

Following the introduction to C# the second half of the session was dedicated to introduction to patterns and game development. In this four patterns were explained, which are relevant to game development, but are usually implemented at game engine or graphics library level.

The presentation of the patterns was tied together with introduction to the Unity game engine. Examples were made of how a game developer can make use of these already being implemented patterns within the framework.

The first pattern to be presented was the game loop. This is the general structure of almost every computer game. In essence it is a large while loop, that allocates time to each subsystem and entity during each frame.

Most available game engines have already implemented this pattern as part of the framework, as it is some of the most executed part of the code in a game. Due to that, it has to be well optimized to run fast.

CGLearn material for game loop simply referred students to the course book, as this is a pattern that in its simplest form is trivial while loop, but gets progressively more complex the more systems it has to maintain. Modern game engines as a general rule also make use of multiple threads within the game loop, which introduces further complexity through synchronization.

Next pattern was the component pattern. This is used by game engines to structure game logic and game entities across multiple domains. In component pattern a single entity is composed of multiple systems. The goal is to decouple code and dependencies between objects that ought not to be together. In class an example was presented how a player character in game could have one component which handled sound, one for input and one for physics. While they all simulate the single character in game, there is no reason for physics or sound components to interact or share any code.

In class the component pattern classes of Unity engine were also presented. This was necessary, as all game logic code that students would write during the course was going to be created as components.

To tie the component and game loop patterns together the update method pattern was presented. This pattern hooks up individual components into the game loop by holding a collection on references to all component objects. A virtual update method is also added to the base component class, which extending classes can override. Game loop can make use of this by calling the update method on all components to allocate time to them during each frame.

During this class Raimond also made a short presentation on the double buffer pattern. This is slightly different from the other three, as it is instead implemented within the graphics API that is responsible for drawing (*E.g.* OpenGL [40]). The pattern also has alternate uses to simulate multiple changes that should happen simultaneously. Conway's Game of Life [41] was presented as an example.

At the end of practice session students were assigned with Hello Unity task to finish on their own and were also presented how to submit tasks for the course as Unity packages [42]. The instructions for task submission were also included in the end of first weeks CGLearn materials.

3.1.2 Command

As the first week was mostly theory and very little actual practice, the second week aimed to get quickly to a point where students could start writing code on their own. For this reason, the practice was committed to a single pattern with wide array of uses. This way the theory part could be completed in about quarter of the total length of the session and the rest was left to practice.

Command was chosen as first pattern for the course, as it is easy to implement and the benefits are easy to understand. It is also the first pattern in the course textbook.

The intent of this pattern is simply to take a method call and turn it into an object – a command. In games some uses for this pattern are: implementation of key rebinding; replay systems; time manipulation mechanics to reverse the simulation of the game world; logging. It is also beneficial when sharing logic code between player and AI, as input and AI actions can both issue the same commands.

The materials and task instructions were tied together as a single example of making rebindable key bindings for a game. During this session the tasks Move Command, Switch Command and Bonus: Snake were assigned. Students could also receive assistance on the Hello Unity task that was assigned on previous week.

3.1.3 Subclass Sandbox and Flyweight

The theme of this week was to show to illustrate combining of patterns.

In the session the meaning of extrinsic and intrinsic states was described and additional examples for uses in games were made. Attention was also brought to examples of flyweight uses within the Unity engine itself for data that could be shared between multiple objects, such as 3D meshes and textures.

CGLearn materials for flyweight followed the previously outlined structure. The motivation paragraphs of this pattern shared the same example as the Flyweight Replay task. This created a very smooth transition from the theoretical part of the session into practice.

Subclass sandbox was also shortly introduced in the practice. The CGLearn material consisted of only the intent and motivation part. The goal of both were to introduce the Subclass Sandbox Essay task described in section 3.2.6. As an exception the deadline of this task was in five weeks, when there would be a debate on the topic in class.

During the class students were assigned with the Flyweight Replay task and could also receive assistance on the MoveCommand, SwitchCommand and Snake tasks.

3.1.4 State

The fourth week was dedicated to state pattern. The pattern describes how to implement finite state machines (FSM) in an object oriented fashion. As the students at this level are already familiar with FSM in general, the theory part focused mainly on example use cases within games.

CGLearn materials for this pattern followed the standard structure. The example explained the use of state machines for controlling of animation states and also referenced the use of FSMs in the Unity Mecanim [43] animation system.

This week was also the deadline of the three command practice session tasks. The bonus task Snake solution was also presented in class, as it was quite short. It was also made available as a fallback code to students.

Students were assigned with the Refactoring Replay task.

3.1.5 Type Object and Prototype

The fifth weeks had a split theme. The type object pattern was used to emphasize the idea, that some patterns differ based on the problem they tackle, yet the way it is solved may be the same. The prototype pattern was used as intro to creational patterns, which would follow on the next week.

The three patterns type object, strategy and state are nearly identical when it comes to the actual implementation. The theory part of the session was used to do a quick recap on the state pattern and then compare it with type object and strategy pattern.

The CGLearn material is slightly lengthier on this subject, as the strategy pattern is not part of the textbook. Neither is it an actual topic, but a few additional paragraphs were written on it as part of introduction, before getting to the type object material.

To better draw attention to the similarities a class diagram was also added for the strategy pattern. This was expected to immediately draw students' attention, as the previous week's material ended with a class diagram of the state pattern.

For CGLearn an example was created where prototype pattern would be used in a fantasy game to track characters' races and professions while avoiding the creation of an unmanageable class hierarchy.

The second half of the theory was on the prototype pattern, which unfortunately relates to the type object only through similarity of the name.

The theory part of prototype focused on beneficial use cases in games. As an example a Minecraft [44] inspired concept was presented with an additional feature of being able save and copy player created buildings.

Overall the week was intended to recap and tie up old topics and introduce the next. The single exercise, GameObject Clone, was also planned as an easy one. During the practice most students would still be working on the Refactoring Replay task, so it was necessary that the GameObject Clone task would be something that could be finished fast, as the following weeks would have tasks that required a bit more effort to complete.

The CGLearn material for prototype mainly focused on illustrating the differences of deep and shallow copies and use of prototypes in Unity. It also included the example game concept, which was presented during the practice session.

This week was also the deadline of Flyweight Replay. As the Refactoring Replay builds on the previous exercise, a fallback base code could have been provided for students that required it. Although this turned out to be unnecessary.

3.1.6 Creational Patterns

This week focused on creational patterns. The goal was to teach students the abstract factory and builder pattern, but as the abstract factory is quite complicated we first worked through simple factory idiom and factory method. None of the patterns covered in this week are part of the course textbook, so the written materials in CGLearn were more detailed than on average.

First pattern to be presented was the simple factory. It is debated whether or not this is actually a pattern and as such is often referred to as simple factory idiom instead. The main criticism of simple factory is, that it goes against the open/closed principle [45] of OOP. Whenever an additional product is added then the factory class has to be changed as well. Students are recommended to combine the factory method with abstract factory in-

stead of simple cases. For more complex cases where runtime configuration is required then abstract factory should be combined with prototype pattern covered in previous week.

Builder pattern – important as it enables building up composite objects from smaller parts. Very good for all things that are created procedurally. Decouples the building blocks for the building algorithm. Can mix procedures with different materials.

The motivation for all the creational patterns was explained through use of a tile based world generation feature. The different benefits of each pattern were illustrated by comparison of the features of the generator.

Students received two new tasks during this session: Factory Method and Abstract Factory.

3.1.7 Decoupling Patterns

This week focused mainly on the observer and singleton patterns.

Singleton is widely used in games for all kinds of manager classes, such as game manager, sound manager, input manager etc. It is also a controversial pattern due to its wide use and the that it provides two unrelated features. A singleton provides global access point to an object and also limits it to single instance. As often only the global access point is relevant, use of static methods or static reference to an active instance were proposed instead.

The observer pattern was also presented during the session. The relevance of observer pattern is mainly its use in user interface systems, although it is viable in any situation an unknown number of objects needs to be coupled with to a game system during runtime.

The presentation first introduced the pattern as it is described in the GoF book and followed that up with the C# events, which is a language level implementation of the observer pattern. The Observer and Observable [46] interfaces of Java were also mentioned.

During the session students started working on the final task – Mixing Patterns. The deadline for this task was set as exam, which the first sitting took place four weeks after the practice session.

Students were also assigned the service locator and event queue patterns from the course textbook on their own.

Most of the practice session was spent working on the Mixing Patterns task described in section 3.2.11.

3.1.8 Optimization Patterns

This was the final week of the course. The topics chosen this week were things that would be good to know, yet exactly aren't essential while developing a game.

Presented data locality based on the course textbook. This is a pattern that is mainly relevant when developing a game engine, rather than a game. It plays an important part when designing the component and update method system which were discussed on the introduction week. The components have to be arranged with data locality in mind for the update methods to be efficient.

We also did a live presentation of implementing an object pool in Unity and Raimond presented the spatial partitioning topic.

Following this there was time planned for assistance on the mixing patterns task, but students did not currently have any questions or problems, so we moved on to the next activity.

The second half of the session was planned for the subclass sandbox debate. Original plan was to divide the students into two groups, but due to very low attendance – only 3 of the 11 students took part in last practice, the plan was changed on the fly. This was partly due to multiple students being ill and also likely partly due to it being the last practice session, where no new tasks were assigned.

The debate was changed into a story based exercise. In the exercise a game was being developed with a magic system. One of the students had to present a general overview of the subclass sandbox pattern. The second had to make a case for how this pattern could be applied for the system at hand and the third had to propose an alternative solution. All the students present made a convincing case to that they had understood the pattern.

3.2 Tasks

The goal of the tasks was to act as a code examples of patterns, which students will have to complete on their own. The approach was inspired by experimental learning techniques. [7]

Most of the tasks during the course added up to a single project, which the students continuously improved during the course.

On the second week students would receive a simple isometric like game scene where a character can move from tile to tile (*Illustration 2*). The tasks will make it possible for input to be rebound to different keys, turn the single character movement into a snake or column formation like movement. Make the actions inside game recordable and replayable and finally explore creational patterns by cloning in game elements and procedurally generating new terrain.

When constructing the task one of the main goals was always, that the effort that students have to put into this would be on the subject. In other words, students should not have to spend time on things, which are not specifically about improving their knowledge of patterns, game development or programming ability. This means mainly that students should not have to spend hours on figuring out some exception due to the tools not working.

The course had a total of 11 tasks. Of these tasks 10 were mandatory and one was an optional bonus exercise.

Students usually had two weeks to work on each task, although multiple tasks could be worked on at a time. The exceptions to this rule are the tasks Subclass Sandbox Essay and Mixing Patterns. As students would often have multiple tasks assigned at the same time, care was taken not to have too many tasks, which are either difficult or take a lot of effort to complete at the same time. For the full task schedule see Illustration 1.

Subclass Sandbox Essay was the only task, where students did not have to write code. It was given out early, so students would have time to do their own research. The tasks goal was to have students prepare for a debate during the last practice session.

Mixing Patterns was the final exercise, which was slightly more complicated than other tasks, but also gave by far the most points.

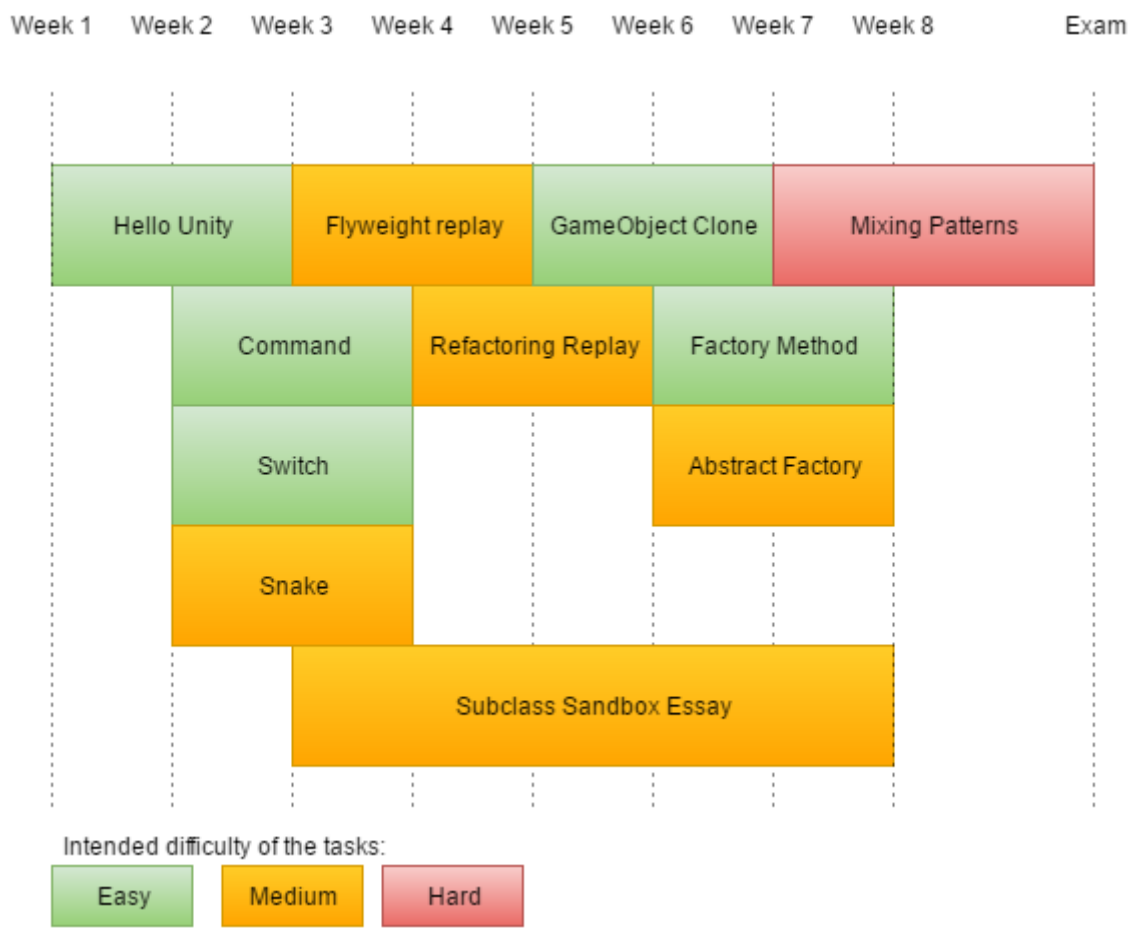


Illustration 1: The assignment, deadlines and intended difficulty of tasks.

The following subsections explain the study goals of each task and describe what students had to do to complete them.

3.2.1 Hello Unity

Students are tasked to create a component script inside the Unity game engine. When attached to a game object, the script would have to translate the position of this object by 1 unit per second. The task would take about one or two lines of code from the student to complete.

For this task there is no base code to start out from.

The goals of this task are to:

- For students who intend to use their personal computers to set up the development environment.
- Get students accustomed to the Unity game engine, so that they would know how to write and run code within the framework.

- Illustrate the benefit of game loop, update method and component patterns discussed in class.
- Get students accustomed to writing C#.

Students were also given a hint to use `Time.deltaTime` variable to make sure the speed of movement is independent from framerate.

3.2.2 Move Command

This task illustrates use of the command pattern by turning keyboard input method calls into command objects. Students had to add a layer of abstraction between input and the game characters' movement. At start of the exercise, character movements are tied directly to set keyboard buttons. At the end the keyboard buttons would first create command objects. Each of the command objects have an execute method, which when invoked would make the characters move. Creation of these command objects can be tied to different keyboard buttons in a dictionary.

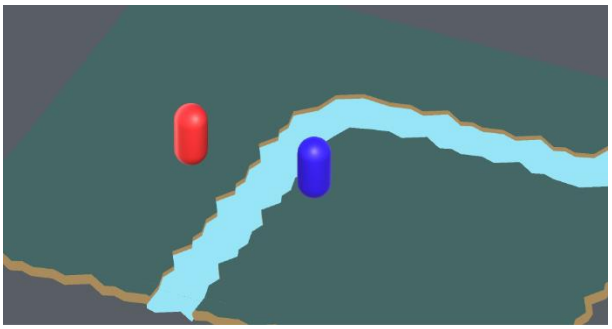


Illustration 2: The start project.

The task had a base project consisting of:

- An `InputHandler` script.

  The script consumes Unity legacy, pre version 4.6, GUI events to implement a `ReadKey` method, which returns a `KeyCode`. This is a work around to not having to call `Input.GetKey()` against all possible `KeyCodes` to determine if a key is pressed.

  Doing this enabled the students to use a dictionary of `KeyCode` and `Command` pairs to create input rebinding. For an actual application better alternatives exist, yet this solution was chosen to make the code students have to write shorter, less complex and cleaner.

- Movement component script which moves a `GameObject` that it is attached to. The movement distance depends on an input `Vector3` and the movement is completed over a positive `float` duration, which can be set from a local variable. At start of the exercise input is directly tied to this script. Students had to remove the inpud handling from this class, as it the movement would be controlled by command objects.
- A Unity scene with two movable characters and background terrain. As characters have the same movement component that both read and act on input, then both characters will move at the same time. At end of the exercise only one of the characters should move.
- A camera movement script that can follow a `GameObject` target.

The tasks instructions also partially set up part of the flyweight exercise solution, by having students only have a single instance of each command. Although no attention is specifically drawn to it during this exercise.

3.2.3 Switch Command

Switch task continues from the Move Command task. To complete this task, student had to:

- Create a `SwitchCommand` class.
- A method that redirects movement commands and camera target to another character.

The task illustrates, that command objects can also be reused on different in game entities. As well as, that not all commands have to control in game characters, but could instead be directed to an in game system, such as `InputHandler`.

3.2.4 Bonus: Snake

As this is a bonus task, no direct instructions were included. The task was presented as a set of requirements for the final outcome.

The idea was to have the characters used in Move Command task follow each oth-

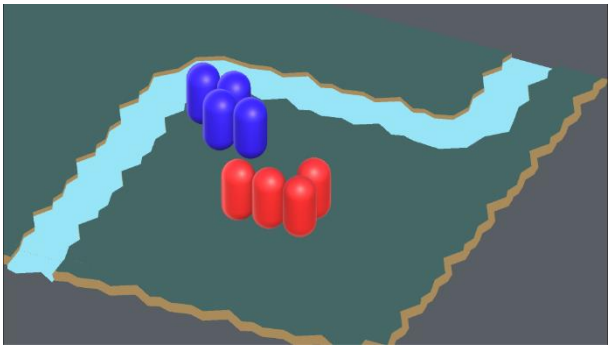


Illustration 3: Completed Snake task

er in snake like formation, the played input would direct the head of the snake and the rest of the parts would follow it.

The full solution from last task was quite short if the student has already completed previous tasks and understood how to use command pattern.

To get maximum points the following changes had to be made in `TileBasedMovement` script:

- Give each part of the snake a reference to the next. A simple way of doing this was to make a public variable inside the `TileBasedMovement`, which could be accessed inside the Unity editor, and assign the next part of the snake to it.
- Save each incoming command in a local variable. This could either be a `Command` type variable to save the incoming command or simple a movement vector from the command.
- Add an if statement to the move method that checks if this character has been assigned a follower and if it has saved an incoming command. In case both of these two checks are true, then send the saved command to the following snake part.

A hint was provided to start with all pieces of snake stacked on top of each other. By doing that, there is no need to define a special behavior for the first few moves of the snake.

The task was worth total of 4 points and could be submitted multiple times over the two weeks' period. As an optional bonus task, the points are not part of the 60 task points, meaning that the completion of this task could take the final score over 100 points.

3.2.5 Flyweight Replay

This idea takes the command pattern further by mixing it together with flyweight pattern. The task was intended to both teach the flyweight pattern and illustrate that new functionality can be gained by mixing together multiple patterns. This is idea was more closely visited in the last task.

The task starts out familiarly by having student bind even more new commands to input keys, namely record and play.

The `RecordCommand` would tell the input handler that all following commands should be recorder and `PlayCommand` would order the input handler to play back all recorded commands.

Third command that students had to add was the actual flyweight class with both intrinsic and extrinsic states. This was a command, which would encapsulate any other command, much like a decorator pattern. It also had a timestamp variable to determine when the command should be executed during playback.

Recording a playback would mean that all issued commands would also be encapsulated inside a playback command with a timestamp and then that playback command would be added to a list of recorded commands.

The important part for students to take away is that flyweight pattern can be used to share data that does not change, or changes for all instances at the same time, can be shared. In this task all the different playback commands, which save the same type of command (E.g. move 1 unit along x axis), would refer to the same command objects. This is the *intrinsic state*. The playback object along with the timestamp is the *extrinsic state*.

At this point student would, hopefully, also realize why they were instructed to store only one instance of each command in a dictionary and reuse it.

This task had no additional base code and could be continued from Move Command, Switch Command or the Bonus: Snake task.

The solved version of the task was expected to cause a situation, which could elegantly be solved by implementing the state pattern. This was caused by having the input recorder to essentially have three exclusive states: *playing*, *idle* and *recording*. As the states were not explicitly in requirements, then the solutions mostly used multiple Boolean variables and if-else statements.

3.2.6 Subclass Sandbox (Essay)

Subclass Sandbox was an unusual task compared to the others. This did not require students to write any code. Instead they had to read the textbook, do their own research and find more information outside book on this pattern, formulate an opinion and write an essay on it.

The goal was to encourage thinking on their own as being able to decide when to apply a pattern is just as important as knowing how to apply it.

The subclass sandbox was chosen, as this pattern is presented only in GPP book and has received some interesting criticism [47]. Students were expected to find similarities with the strategy pattern, due to both being used for similar problems.

3.2.7 Refactoring Replay

This task continues from flyweight replay. By not being aware of the state pattern, it is expected that most students create a code with many if-else statements and Boolean variables in Flyweight Replay. This task refactors the replay system into a state machine.

In Refactoring Replay task the students had to continue from the end of the Flyweight replay task. The instructions presented as a class diagram and a state diagram for students to implement.

The final outcome should have worked exactly the same as the last task.

3.2.8 GameObject Clone

As the GameObject Clone task between two difficult tasks in the schedule, then it was intended be slightly easier. Although due to it being only task of the week, it would still give 6 points.

As the standard way of adding game objects into a scene in Unity is to clone a prototype using the instantiate method, this task also serves to illustrate the idea of using correct tools for the job.

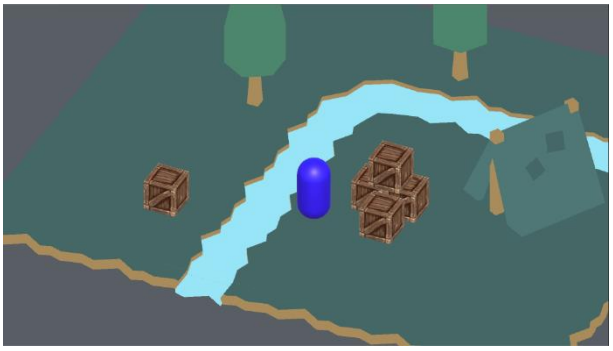


Illustration 4: Completed GameObject Clone task. The player cam stack crates using the mouse buttons.

In this task the students have to ray casting to select objects inside the game scene by left clicking on them with the mouse cursor. After that right clicks should create clones of the selected object at the location of the mouse cursor. Additional limitation was also that only certain objects in the scene should be cloneable.

During this task, students did not implement a pattern on their own, but made use of the build in support for prototypes in Unity.

3.2.9 Factory Method

This task illustrates the use of factory method pattern. For the base code of this task students were provided with a scene with a tile based map and a player character used in previous tasks. The map rooms consisted of two rooms and portal tiles, which moved the player character between rooms when entered. The layout of the map was generated inside a layout script.

Student's task was to replace all the calls to the ground, river, room and portal map tile constructors with virtual factory methods. This was done inside the class responsible for layout of the map. They also had to create a subclass for the layout class, in which they would override the factory method responsible for creation of portal tiles. The modified portal tile had to be visually different.

This task also acted as an introduction of more complicated abstract factory and builder pattern used in following tasks. During this task the students were familiarized with the map data classes, which are also used in following tasks.

3.2.10 Abstract Factory

The abstract factory is frequently used pattern in game development for insanitation of objects. In this task students had to defer creation of map tiles to a factory object, create two factories with visually distinct tile sets and pick one of the factories at random at start of the game.

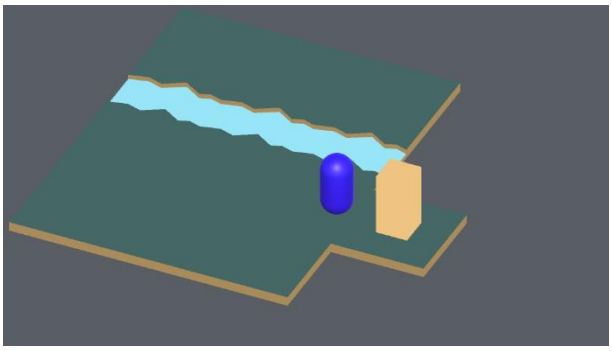


Illustration 5: Factory tasks featured small rooms, which could be traversed between by walking through portals.

The concrete version of factory was to be implemented using factory methods for context access and prototypical instances for configuration of the factories. With this approach it was expected to make the abstract factory pattern seem quite simple, as most of the steps were already completed in the GameObject Clone and Factory Method tasks. Compared to last task, students only had to split off the factory methods into a different class and load each concrete factory with different prototypical instances for tiles.

3.2.11 Mixing Patterns

The final task was intended as more advanced exercise. Mixing patterns did not focus on a single pattern, but instead of making use of multiple patterns to implement a procedurally generating world around a player character.

The task has students tie the movement and input handler code together with new map creation code. This is by far the largest task they had to complete.

The task was set up as a list of specific requirements and the patterns these would have to be implemented with.

To hold the map data students were given requirements, which described the map data used in previous factory tasks, with minor modifications. The map tile types were to be set up as flyweight objects, which students had experience with from the Flyweight Replay task.

As a new pattern students had to use builder pattern for generation of 10x10 chunks of map tiles. This was slightly different from the factories used in previous tasks, as builder decouples combination logic from the objects it is combining. Factories simply create objects and the combination has to be done by a context class.

The world class to hold the map data and reference the builder was to be created and accessed using the singleton pattern. This class was also an observer.

The observable class was to be the familiar movement component students had used since second week. This had to be modified so, that every time it finished movement it would notify all observers of its position.

As this was a larger and more complicated task, the completion of this task was worth 13 points.

4. Results and Discussion

During the course some feedback was collected from students through the online learning environment and in the end of the course students also filled out quite lengthy questionnaire. The received feedback is presented in the following chapters, which is then by discussion and potential future improvements.

4.1 CGLearn feedback

CGLearn offers a few built in options for students to submit feedback. When submitting a task, they had an option to leave a comment, select from a dropdown list how much time the task took and how difficult they found the task on a five-point scale.

As students were not directly instructed to leave feedback in the comments about the task, then they used it mostly to elaborate on what they did in the task. From teaching point of view this is actually a good approach. The submissions would sometimes include comments stating that some the student was not entirely sure on a specific part of the solution they submitted. This enabled to clarify the confusing parts in task feedback.

Based on the difficulty and time feedback CGLearn also generated graphs. These do not however give very good overview, as all the submissions are considered. As students had the option to submit tasks multiple times. On one submission student would mark down how long the task took to complete. If the task is then graded and given feedback before deadline, the students still had an option to resubmit a better solution with another time feedback.

At this point CGLearn does not direct if the time spent on second submission should be the time spent making changes or time spent for the whole task. Should the student submit the second solution as time spent making changes, then compared to other submissions this may look like a very fast solution, if the changes made were minimal.

Alternatively, the student makes minimal changes that take very little time and chooses the same time spent as on the first submission. In this case this students task solution time rating would have double the weight in the aggregated statistics, compared to the students who only submitted once.

As CGLearn considers the time from both submission as independent time assessments, then not much can be analyzed based on these statistics. This is further skewed by

CGLearn filling in feedback with default values of 1 hour and medium difficulty, so a lot of submissions from students were sent with that.

The CGLearn graphs are included in the Appendix A.

4.2 Questionnaire

After the end of the last practice session students were asked to fill out an online feedback form. The feedback could be submitted until the first exam date on 19.04. The questionnaire was mandatory for students who wished to attend this exam. From 11 students attending the course 9 filled out the questionnaire form. To be able to enforce that students had filled out the questionnaire a name field was included.

The form was closed after the exam, so that the final grade of the student could not affect the feedback. As the questionnaire was not anonymous, none of the feedback was reviewed before grading of the exams. While it is unlikely that grades would have a direct impact on the feedback or that the feedback would have a direct impact on grades, it was safer to rule out possibility of either happening at all.

The questionnaire was split into three sections. The first section collected feedback on course didactics, online learning system and the technologies used during the course. The second focused on the quality of tasks and the third on the relevance of chosen patterns in context of game development.

Most of the selectable answers for each question were presented on a five-point scale.

On the longer grid formatted questions students were also presented with a sixth option to opt out of any answers. It was considered possible, that a student might not remember every single pattern or task. As the questionnaire was mandatory, by providing the possibility to opt out, they would not be forced to pick an answer at random in case they do not hold a strong opinion on a specific question.

Following the longer grid formatted questions there was always also an open ended optional feedback box, where a student could clarify any of the chosen answers should they feel like it. In the following sections, answers for text based questions are marked with IDs. These IDs only refer to a specific answer and not the students who wrote the answers.

Some of the answers in these sections are shortened for presentation. The full answers are included as an ods format spreadsheet file as an appendix.

4.2.1 Section 1 – Q&A – General

Section 1 consisted of the following questions:

1. Name
2. How did you like CGLearn?
3. How did you feel about having a 4-hour continuous practice session?
4. What is your opinion of having 2 teachers?
5. What is your opinion on the format of practice session?
6. How suitable did you find Unity for this course?
7. How suitable did you find C# for this course?
8. Was the introduction to OOP useful?

The question 2 was an open text field question. The question was asked to see, if students also find CGLearn suitable for this course and to possibly find out if something could be improved in the environment. Majority of the answers to this question could be summarized as either "OK" or "liked it". The longer more interesting answers can be seen in Table 2.

Questions 3 to 5 compared practice session model presented in the thesis section 2.3 against the more common model of having one practice session and one lecture per week model. Answer option 1 was that the student liked the use course format and answer option 2 described the alternative model. For question 4 the alternate was that student would prefer one teacher in class at a time. Students also had third option, in which case student would have a textbox to write their answer in.

Table 2: Other answers for section 1 question 2

ID	Answer
1	The topics were rather interesting, though sometimes a bit confusing.
2	It does what is intended to do pretty well, even though the materials section is strangely empty for a landing page, a link to the programming patterns book at least would have sufficed.
3	The online environment? Pretty great, its design is better than whatever most other courses have (moodle, 90's style web pages...). Or if it stands for the course itself, I liked that too. It was the only option I had in this university for pursuing my dream of becoming a computer game dev.
4	its easy to use and user does not get lost there.

Table 3: Answers for section 1 questions 3 to 5

Question	Model A	Model B	Other
3	4	2	3
4	9	0	0
5	8	0	1

Table 4: Other answers for section 1 question 3

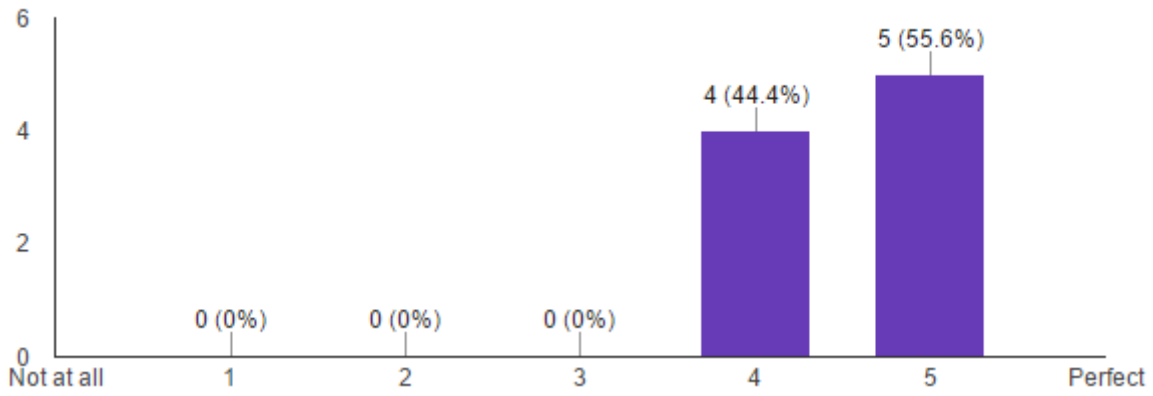
ID	Answer
1	No strong opinion - it suited my schedule, but in general I don't think there would have been any loss in quality if there had been 2 shorter sessions.
2	I think a continuous 3 hour session would be better from the infrequent times I was present.
3	It would be alright if only one half wouldn't overlap with another course (a late time in the evening could be perfect)

Table 5: Other answers for section 1 question 5

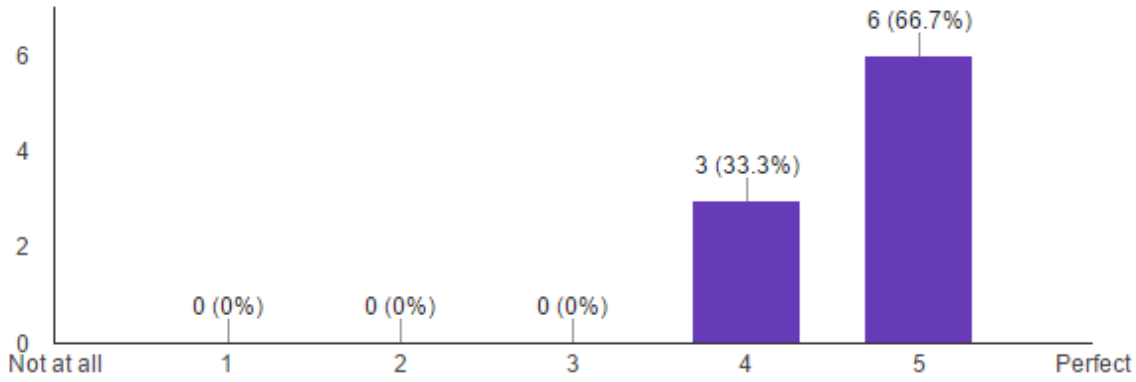
ID	Answer
1	It was good because even if student did not get the idea at first in the lecture part, then doing the practical part it was easy to ask about stuff we did not understand

Questions 6 to 8 were presented on a 5-point scale. 1 represented the least favorable answer “not at all” and 5 represented “perfect”.

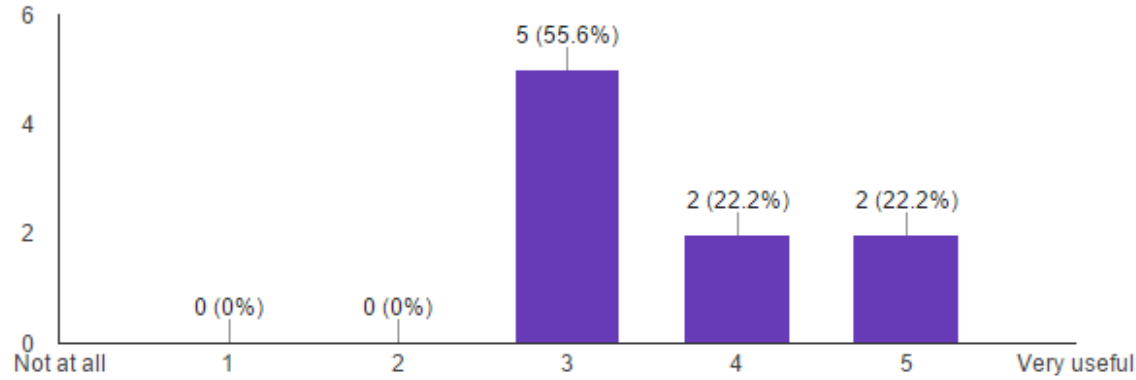
5. How suitable did you find Unity for this course?



6. How suitable did you find C# for this course?



7. Was the introduction to OOP useful?



4.2.2 Section 2 – Q&A – Tasks

Section 2 consisted of the following questions:

1. Rate the tasks based on how well they illustrated the pattern they were about.
2. Comments or clarifications on the ratings.
3. Rate the tasks based on clarity of the instructions.
4. Comments or clarifications on the ratings.

Questions 1 and 3 of this section were grid layout questions about the tasks. Each line on the grid represented one of the tasks and students could select one answer from each line. The possible answers were to pass this questions or to rate it on a 5-point scale. 1 represented not at all and 5 represented very well. Following each of the grid layout questions was an optional question where student could clarify their choices if he or she wished to do so.

Table 6: Answers for question 1 "Rate the tasks based on how well they illustrated the pattern they were about."

Task	Opt out	1	2	3	4	5
Hello Unity	1	0	0	1	3	4
Move Command	0	0	0	0	2	7
Switch Command	0	0	0	0	2	7
Bonus: Snake	4	0	0	0	2	3
Flyweight Replay	1	0	0	0	2	6
Subclass Sandbox	2	0	0	1	2	4
Refactoring Replay	1	0	0	1	2	5
GameObject Clone	1	0	1	1	2	3
Factory Method	2	0	1	1	2	3
Abstract Factory	3	0	1	1	1	3
Mixing Patterns	3	0	0	2	0	4

Table 7: Students comments on section 2 question 1 answers.

ID	Answer
1	Factory homework tasks were quite confusing, possibly due to combined base package. Mixing patterns task is significantly larger and more complex than any other task. Effort needed for it affects illustration aspect negatively.
2	All the tasks I worked on seemed to carry their points across very well.
3	the factory method doesn't really feel like a stand-alone task. the snake task was a very good example of how the pattern helps simplify code (i realized the same idea that was shown in class, but my code had a couple of errors so i didn't submit it)
4	There could be some additional material given in the practice session to complement the essay on Subclass Sandbox.

Table 8: Answers for question 3 "Rate the tasks based on clarity of the instructions."

Task	Opt out	1	2	3	4	5
Hello Unity	0	0	0	1	1	7

Move Command	0	0	0	0	1	8
Switch Command	0	0	1	1	0	7
Bonus: Snake	2	0	0	0	1	6
Flyweight Replay	0	0	0	0	5	4
Subclass Sandbox	1	0	1	0	2	5
Refactoring Replay	1	0	0	0	4	4
GameObject Clone	1	0	1	0	3	4
Factory Method	1	0	0	5	2	1
Abstract Factory	2	0	0	4	3	0
Mixing Patterns	3	0	0	1	4	0

Table 9: Students comments on section 2 question 3 answers.

ID	Answer
1	The more complex tasks needed some additional help to understand.
2	The Abstract Factory task description was somewhat indistinct. Or maybe it's just me being blonde.

4.2.3 Section 3 – Q&A – Patterns

The questions in this section focused on the selection of design patterns for the course. The following questions were asked in Section 3:

1. In your opinion, should the following patterns be part of the "Programming Patterns in Computer Games" course.
2. Comments or clarifications on the ratings.
3. Rate how much time was allocated per pattern.
4. Comments or clarifications on the ratings.
5. Do you think the patterns will be useful in your future work?
6. Is there a pattern the course did not cover, but you wish it had?
7. Completely open text box for you to write whatever your heart desires.

Number of answers does not add up to 9, as one student noticed, that these questions were missing the required answer checkbox and skipped the questions. As students were allowed to skip questions anyway using the skip option on individual question, then this does not change the outcome much.

Table 10: Answers for question 1 "In your opinion, should the following patterns be part of the "Programming Patterns in Computer Games" course.

Pattern	Opt out	No	Yes	Pattern	Opt out	No	Yes
Game Loop	1	0	8	**Factory Method**	0	2	6
Double Buffer	1	1	7	**Abstract Factory**	0	0	9
Command	0	0	9	**Event Queue**	1	0	7
Flyweight	0	0	9	**Builder**	0	0	9
Component	1	0	8	**Service Locator**	2	0	6
Update Method	1	0	8	**Singleton**	1	1	6
Subclass Sandbox	1	1	6	**Observer**	1	0	7
State	0	0	9	**Object Pool**	2	0	6
Type Object	0	0	9	**Data Locality**	1	1	6
Prototype	0	0	9	**Spatial Partitioning**	2	1	5

Table 11: Students comments on section 3 question 1 answers.

ID	Answer
1	Double buffer and spatial partitioning seem like they'd suit more into a computer graphic oriented course. Other non-checked options, I'd categorize into more general programming topics, rather than computer game programming.
2	This course was mighty short. I wouldn't mind a whole semester of computer game programming theory.
3	All of them because they are game programming patterns

Table 12: Answers to *"Rate how much time was allocated per pattern."*

Pattern	Opt out	Too little	Slightly too little	About right	Slightly too much	Too much

Game Loop	3	1	0	4	0	0
Double Buffer	3	0	1	0	0	0
Component	2	0	0	6	0	0
Update Method	2	0	1	5	0	0
Command	2	0	0	6	0	0
Flyweight	2	0	0	6	0	0
Subclass Sandbox	2	1	1	3	1	0
State	2	0	0	6	0	0
Type Object	2	0	0	6	0	0
Prototype	3	0	0	5	0	0
Factory Method	4	0	0	3	1	0
Abstract Factory	4	0	0	4	0	0
Builder	4	0	0	4	0	0
Event Queue	4	0	3	1	0	0
Service Locator	5	1	1	1	0	0
Singleton	3	0	2	3	0	0
Observer	3	1	1	3	0	0
Object Pool	3	1	1	3	0	0
Data Locality	3	1	1	3	0	0
Spatial Partitioning	4	1	1	2	0	0

5. Do you think the patterns will be useful in your future work?

This was an yes or no question. All of the students answered yes.

Table 13: Answers to "Is there a pattern the course did not cover, but you wish it had?" All different phrasings of no omitted from the table.

ID	Answer
1	Definitely game loop (especially the parts about extrapolation and using that together with i.e the physics system manually)

Table 14: Answers to "Completely open text box for you to write whatever your heart desires."

ID	Answer
1	The exercises were well designed and I learned a lot, but I missed being able to work on an actual game of my own. […] it could be reasonable to have an opportunity to solve the tasks within the context of your own project […] Of course this is all to say, thank you for an interesting course, hope the next iteration is even cooler!
2	Please speak up a little. Or turn off the air conditioner that blows right onto a lamp, making it rattle. Sometimes it was truly challenging to hear you even from the second row.
3	Feedback after each task would give most accurate results. After the course it is too difficult to remember and recall. Great course overall.

4.3 Discussion

The answers in section 1 were mostly positive and supported the choices made for organization of this course.

Students found that CGLearn worked well as an online environment. Although one student pointed out that the landing page of CGLearn for this course was strangely empty. This should certainly be improved for the next year. Currently all organizational information was on the courses page and the introduction and along with general topics, such as submitting tasks, was under the first week. As students may not revisit the courses page after being introduced to CGLearn, then the opening page in CGLearn could potentially duplicate some of what is posted on courses page. Also some information from the first week may fit better on the landing page. Alternatively, the landing page could be set to be the first week of the course.

Overall the students seemed to like the four-hour practice session format as well, as seen by the answers of section 1 question 5. Although it did cause some issues due to scheduling, as pointed out by the other answers. It is difficult to have a four-hour slot for an elective course, without overlapping with students' mandatory courses. This could be somewhat remedied by choosing the time so, that it would not overlap with 3rd year computer science bachelors' obligatory courses.

Students also liked having two instructors in class for this course. In class both of the instructors were mostly busy during the earlier practices, as when learning design patterns

students are often not quite sure if they are on the right track. Unlike as with most programming subjects, often the errors are not something that the compiler can catch. In the task Refactoring Replay, the starting and finishing functionality had to be exactly same as well, while a single large class was being split into six much smaller ones.

The format was also very good fit for me, as I had not given a full course before. Having someone more experienced in the room was very reassuring. This worked out very well for starting discussions and also, due to computer science students being somewhat reluctant to ask questions. Although once the discussion has started they seemed to be more likely to join in. The question asked by the second instructor should probably be a simple one, which any student may have on their mind as well.

The question 7 of section 1 “Was the introduction to OOP useful?” had somewhat interesting answers. Only four answers out of nine could be considered that students found the section useful for them.

Reason for this could be, that as Object-oriented programming is the prerequisite subject for this course. As all of the material should be already known to the students, so a revisit to this may not be necessary at all.

On the other hand, four students did find the part useful and they might not have been able to attend the course if this part was missing. As the topic was only about 45 minutes long, then it is likely worth it.

There were also students who unsubscribed from the course after first two weeks, as they found it too difficult. Some of these students that unregistered were attending the Object-oriented Programming course in parallel, so they were also able to register to Programming Patterns in Computer Games. Revisiting part to teach all from the OOP course was clearly not enough for them. This shows that having OOP as prerequisite course is good choice for registration requirements.

Feedback section 2 focused on tasks. The answers in this section were also overall fairly positive.

The students that completed the Snake task had a very positive opinion of it. In class one student also commented, that after figuring out this task the value of command pattern became much clearer. This notion is also supported by the comment 3 in Table 7.

The Snake task was made optional, as the feature can take a lot of time and effort to implement, if the student does not see the easy solution. As an example, one of the submissions was solved with about 100 lines of code and did still not work quite correctly. Although the correct solution could be implemented in just about 6 lines of code by making use of the commands created in the previous tasks.

It would be nice to include this task as part of the mandatory tasks, but command pattern should then be moved into a later week. Possibly some other task should then be removed from course altogether.

A good candidate for removal could be the Factory Method task, as the lowest rated task based on clarity of the instructions. The comments 1 and 3 in Table 7 also point out that this task was confusing and did not feel like a standalone task. The latter is a very good observation, as the Factory Method task was designed to be a warm up for the Abstract Factory task. Solving the first task was supposed to make solving the Abstract Factory easier. The rationale behind taking this approach was due to considering abstract factory as the most complicated pattern in this course. In hindsight it looks like a good idea to merge the two tasks into one.

The intent of the third section was to figure out, if the design patterns chosen for this course worked as a good set of game programming patterns. Overall students seemed to think that most patterns did fit the course.

Spatial partitioning and double buffer were pointed out to be more about computer graphics than game programming. I tend to agree with this as well. Both of these topics already had very little time assigned to them and were not included in any of the tasks.

Table 12 shows that most patterns seemed to have too little time assigned to them, this notion is also supported by comment 2 in Table 11. As both the topics spatial partitioning and double buffer come up in the course Computer Graphics, it would make sense to exclude these patterns next year from Programming Patterns in Computer Games. Doing this would free up a little bit of extra time for the rest of the topics.

To conclude the feedback, students seemed to like the course overall, yet it is also clear that improvements can be made.

4.4 Future ideas

In this section some future ideas for the course organization are presented. These are in addition to the proposed changes from discussion part.

Bachelor's students are often interested creating a game related thesis. The base project that the students worked on could potentially be made much more interesting. As an example the map in the background scene could be improved, better shading, lighting and animations could be added. Alternatively, new tasks could be designed as replacements and improvements on the old or as optional bonus tasks. As the course was currently slightly lacking in time, by adding more content it could be grown into a full semester 6 ECTS course. These improvements could be considered as potential bachelors' theses

Currently students are working in a single scene with minor modifications until they get to the creational patterns near the end. At that point the tasks continues using the same scripts, but they will have to build a new map for the background. By moving the creational patterns to the start of the course, students would first create the scene. All the following tasks would then take place within the students' own scenes.

The difficulty with this change is that, abstract factory is a very complicated pattern to start with, which is the reason it was near end. Although a solution where the creational patterns are broken up, with a Factory Method task at start and Abstract Factory near the end could be considered. Another positive aspect of this change is that command pattern could be moved further away from start and then the Snake task could be included as a mandatory task. This idea however is contradictory to the change proposed under discussion, where Factory Method and Abstract Factory would be merged into a single task.

In hindsight the Mixing Patterns task was a bit too large. A subtask of using the observer pattern would make a good standalone task for early part of the course. The pattern is certainly above average in complexity, but as it is already implemented on the language level for C# it could be made into a task that is manageable on one of the first weeks. Some interesting combinations with command pattern may also be possible.

5. Summary

During this thesis work a new 3 ECTS course Programming Patterns in Computer Games (MTAT.03.315) was designed. For the course a set of design patterns that are useful for computer games development were chosen. The patterns in the course were taught based on the books Game Programming Patterns by Robert Nystrom and Design Patterns: Elements of Reusable Object-Oriented Software by Gamma et al.

The thesis starts with introduction of why such new course is necessary. Next in the second chapter a general outline of the course is given. This is followed by more detailed overview of the course structure and schedule. In didactic consideration the organization of a practice session was described along with the innovative approach for teaching design patterns through tasks, where students figure out the implementation of patters by writing code. The second chapter also compared different online learning environments and game engines for the course. CGLearn and Unity were decided upon as the learning environment and the game engine respectively.

Third chapter presented the main parts of the thesis work. It was split into Course Materials and Tasks. Course Materials described the written materials on CGLearn along with how these materials were used throughout the course. The Tasks described each of the tasks created for this course, of which there were total of 11.

The course was conducted with 11 students in the spring of 2016 in the University of Tartu. At the end of the course feedback was collected via a questionnaire. The results were also presented and analyzed. Based on the feedback the course could overall be considered a success and can hopefully be conducted again next spring. The questionnaire, results and analyzes is presented in chapter 4 Results and Discussion.

Designing and conducting this course has given me a lot of valuable experience. Being able to teach design patterns to others, gave me far better understanding of the subject matter itself. It brings me great joy, that students found the course useful.

I would like to thank all the students who participated in this experimental course. I would also very much like thanks to my supervisor Dietmar Pfahl. Special thanks as well go to my good friend, the thesis co-supervisor and the second course instructor Raimond-Hendrik Tunnel. Finally, I would like to especially thank the University of Tartu for trusting me to conduct this course in the first place.

6. References

[1] R. P. P. 28/09/2012, “Games are arguably the most sophisticated and complex forms of software out there these days,” *Eurogamer.net*. [Online]. Available: http://www.eurogamer.net/articles/2012-09-28-games-are-arguably-the-most-sophisticated-and-complex-forms-of-software-out-there-these-days. [Accessed: 19-May-2016].

[2] R. Nystrom, “Game Programming Patterns,” 08-May-2016. [Online]. Available: http://gameprogrammingpatterns.com/. [Accessed: 08-May-2016].

[3] M. Luik, “CGLearn - Programming Patterns in Computer Games.” [Online]. Available: https://cglearn.codelight.eu/pub/programming-patterns-in-computer-games. [Accessed: 19-May-2016].

[4] TÜ haridustehnoloogiakeskus, “Object-oriented programming - Courses - Institute of Computer Science.” [Online]. Available: https://courses.cs.ut.ee/2016/OOP/spring. [Accessed: 16-May-2016].

[5] Amazon, “Amazon.com: Customer Reviews: Game Programming Patterns.” [Online]. Available: http://www.amazon.com/Game-Programming-Patterns-Robert-Nystrom/product-reviews/0990582906. [Accessed: 16-May-2016].

[6] R.-H. Tunnel, “Computer Graphics Learning - Courses.” [Online]. Available: https://cglearn.codelight.eu/pub/courses. [Accessed: 18-May-2016].

[7] J. Neill, “What is Experiential Learning?” [Online]. Available: http://www.wilderdom.com/experiential/ExperientialLearningWhatIs.html. [Accessed: 19-May-2016].

[8] E. Robson and E. Freeman, “Foundations of Programming: Design Patterns | Lynda.com,” *Lynda.com - A LinkedIn Company*. [Online]. Available: https://www.lynda.com/Developer-Programming-Foundations-tutorials/Foundations-Programming-Design-Patterns/135365-2.html. [Accessed: 19-May-2016].

[9] Lynda.com, “Lynda.com: Online Video Tutorials & Training,” *Lynda.com - A LinkedIn Company*. [Online]. Available: https://www.lynda.com/. [Accessed: 19-May-2016].

[10] J. Purcell, “Learn Java Design Patterns and Architecture,” *Udemy*. [Online]. Available: https://www.udemy.com/java-design-patterns-tutorial/. [Accessed: 19-May-2016].

[11] Oxford, “Software Engineering at Oxford | Design Patterns (DPA).” [Online]. Available: https://www.cs.ox.ac.uk/softeng/subjects/DPA.html. [Accessed: 19-May-2016].

[12] S. Khan, “Why Long Lectures Are Ineffective,” *Time*.

[13] “Dropbox,” *Dropbox*. [Online]. Available: https://www.dropbox.com/. [Accessed: 10-May-2016].

[14] “Courses - Institute of Computer Science.” [Online]. Available: https://courses.cs.ut.ee/. [Accessed: 08-May-2016].

[15] “Google Sheets - create and edit spreadsheets online, for free.” [Online]. Available: https://www.google.com/sheets/about/. [Accessed: 08-May-2016].

[16] “Tartu Ülikooli Moodle´i õpikeskkond.” [Online]. Available: https://moodle.ut.ee/. [Accessed: 08-May-2016].

[17] University of Tartu, “Study information system of the University.” [Online]. Available: https://www.is.ut.ee/pls/ois/!tere.tulemast. [Accessed: 19-May-2016].

[18] “Reviews of Moodle : Free Pricing & Demos : Learning Management System Software.” [Online]. Available: http://www.capterra.com/learning-management-system-software/spotlight/80691/Moodle/Moodle. [Accessed: 08-May-2016].

[19] “11. Kursuse lõpetamine Moodle’is - Moodle - TÜ wiki.” [Online]. Available: https://wiki.ut.ee/pages/viewpage.action?pageId=17114243. [Accessed: 08-May-2016].

[20] “Amazon Simple Storage Service (S3) - Cloud Storage,” *Amazon Web Services, Inc.* [Online]. Available: //aws.amazon.com/s3/. [Accessed: 08-May-2016].

[21] “Computer Graphics - Courses - Institute of Computer Science.” [Online]. Available: https://courses.cs.ut.ee/2015/cg/fall. [Accessed: 08-May-2016].

[22] R.-H. Tunnel, “Arvutigraafika õppematerjal.”

[23] J. Gregory, *Game Engine Architecture, Second Edition*. CRC Press, 2014.

[24] “Unity - Game Engine.” [Online]. Available: https://unity3d.com/. [Accessed: 08-May-2016].

[25] Unity Technologies, “Unity - Fast Facts.” [Online]. Available: https://unity3d.com/public-relations. [Accessed: 18-May-2016].

[26] Mono Project, “Home | Mono.” [Online]. Available: http://www.mono-project.com/. [Accessed: 18-May-2016].

[27] Microsoft, ".NET - Powerful Open Source Cross Platform Development." [Online]. Available: https://www.microsoft.com/net/. [Accessed: 18-May-2016].

[28] A. Davis, "IL2CPP," *What could possibly go wrong?*, 08-Jul-2015. [Online]. Available: http://www.what-could-possibly-go-wrong.com/il2cpp/. [Accessed: 18-May-2016].

[29] "What is Unreal Engine 4." [Online]. Available: https://www.unrealengine.com/what-is-unreal-engine-4. [Accessed: 08-May-2016].

[30] "Unreal Engine FAQ." [Online]. Available: https://www.unrealengine.com/faq. [Accessed: 10-May-2016].

[31] "Friends, C++ FAQ." [Online]. Available: https://isocpp.org/wiki/faq/friends. [Accessed: 10-May-2016].

[32] "Blueprints Visual Scripting." [Online]. Available: https://docs.unrealengine.com/latest/INT/Engine/Blueprints/. [Accessed: 08-May-2016].

[33] nesis, "Comparing Unity 5, Unreal 4 and CryEngine 3." 20-Mar-2014.

[34] "MonoGame | Write Once, Play Everywhere." [Online]. Available: http://www.monogame.net/. [Accessed: 08-May-2016].

[35] MonoGame Team, "mono/MonoGame," *GitHub*. [Online]. Available: https://github.com/mono/MonoGame. [Accessed: 18-May-2016].

[36] Microsoft, "Microsoft XNA," *Wikipedia, the free encyclopedia*, 08-May-2016. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Microsoft_XNA&oldid=719271035. [Accessed: 18-May-2016].

[37] MonoGame Team, "Getting Started | MonoGame." [Online]. Available: http://www.monogame.net/documentation/?page=Getting_Started. [Accessed: 18-May-2016].

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[39] Microsoft, "Learn to Develop with Microsoft Developer Network | MSDN." [Online]. Available: https://msdn.microsoft.com/en-us/default.aspx. [Accessed: 17-May-2016].

[40] SGI, "Default Framebuffer - OpenGL.org." [Online]. Available: https://www.opengl.org/wiki/Default_Framebuffer. [Accessed: 17-May-2016].

[41] Wikipedia, “Conway’s Game of Life,” *Wikipedia, the free encyclopedia*. 16-May-2016.

[42] Unity Technologies, “Unity - Manual: Asset Packages.” [Online]. Available: http://docs.unity3d.com/Manual/AssetPackages.html. [Accessed: 17-May-2016].

[43] Unity Technologies, “Unity - Manual: Mecanim Animation System.” [Online]. Available: http://docs.unity3d.com/460/Documentation/Manual/MecanimAnimationSystem.html. [Accessed: 17-May-2016].

[44] Mojang, “minecraft.net - Home.” [Online]. Available: https://minecraft.net/en/. [Accessed: 17-May-2016].

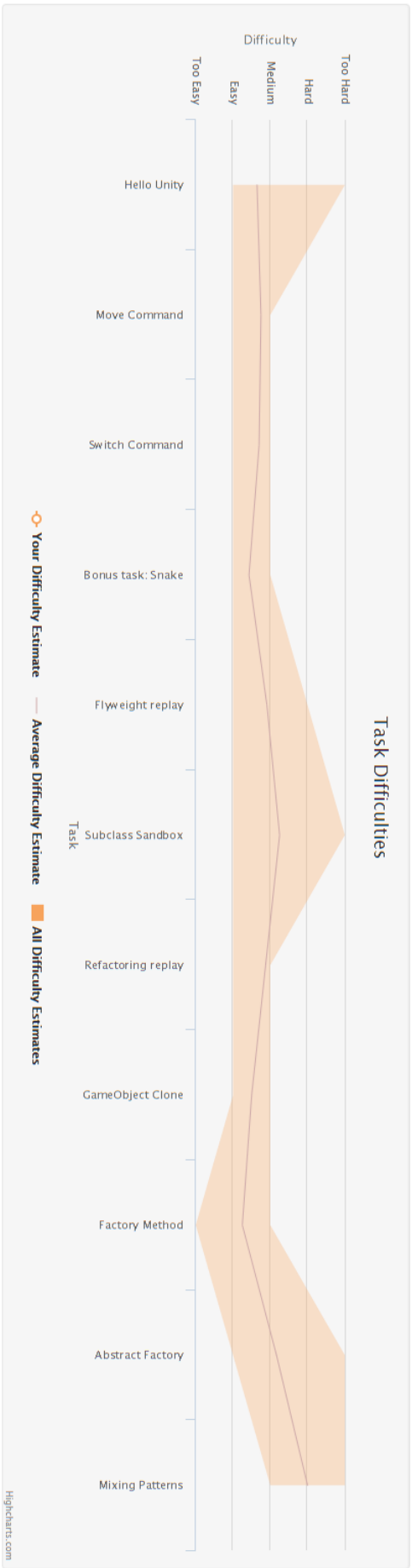
[45] Wikipedia, “Open/closed principle,” *Wikipedia, the free encyclopedia*. 29-Mar-2016.

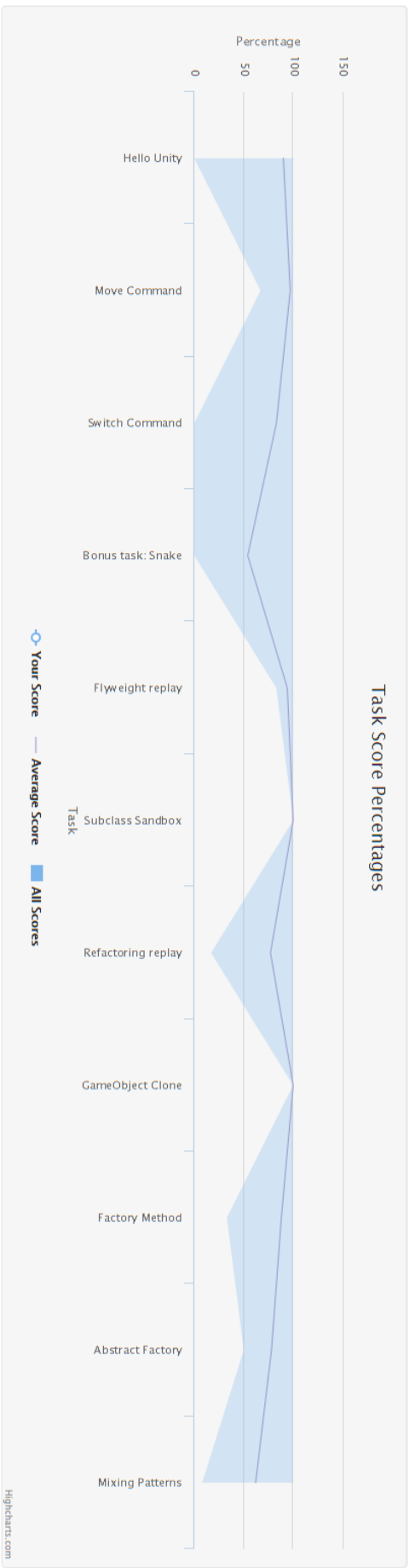
[46] Oracle, “Observable (Java Platform SE 7).” [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Observable.html. [Accessed: 17-May-2016].

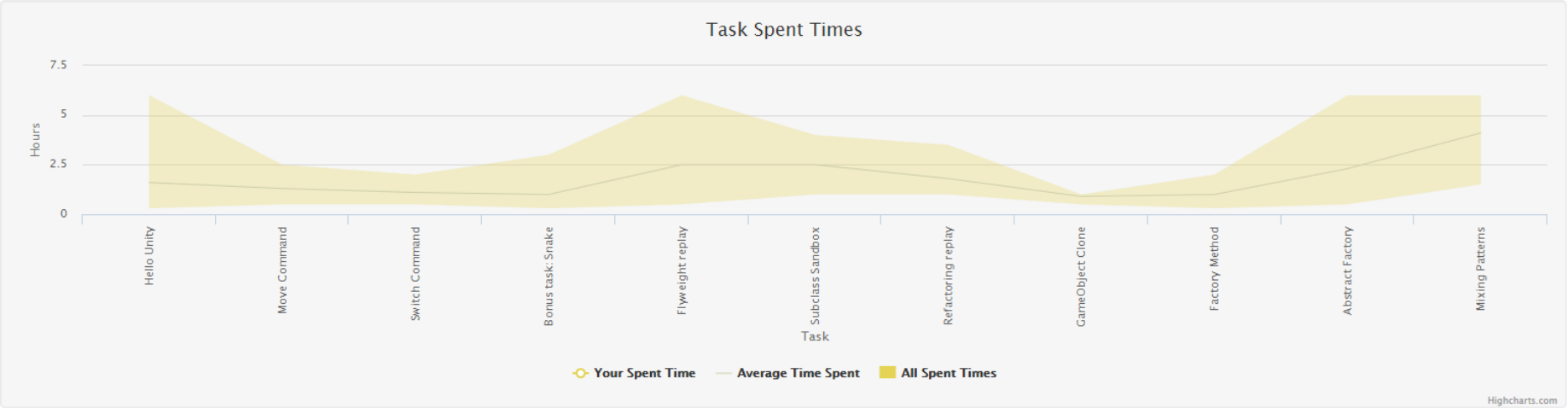
[47] Diamonde, “Design question: Is Subclass-sandbox an overcomplicated strategy pattern?,” *GitHub*. [Online]. Available: https://github.com/munificent/game-programming-patterns/issues/286. [Accessed: 19-May-2016].

Appendix

The following pages of appendix present graphs generated by the CGLearn learning environment. The accompanying archive file contains full results from the questionnaire, without the students names.


Task Difficulties
Difficulty
Too Easy
Easy
Medium
Hard
Too Hard
Task
Hello Unity
Move Command
Switch Command
Bonus task: Snake
Flyweight replay
Subclass Sandbox
Refactoring replay
GameObject Clone
Factory Method
Abstract Factory
Mixing Patterns
Your Difficulty Estimate
Average Difficulty Estimate
All Difficulty Estimates
Highcharts.com


Task Score Percentages
Percentage
0
50
100
150
Task
Hello Unity
Move Command
Switch Command
Bonus task: Snake
Flyweight replay
Subclass Sandbox
Refactoring replay
GameObject Clone
Factory Method
Abstract Factory
Mixing Patterns
Your Score
Average Score
All Scores
Highcharts.com


Task Spent Times
7.5
5
2.5
0
Hours
Hello Unity
Move Command
Switch Command
Bonus task: Snake
Flyweight replay
Subclass Sandbox
Refactoring replay
GameObject Clone
Factory Method
Abstract Factory
Mixing Patterns
Task
Your Spent Time
Average Time Spent
All Spent Times
Highcharts.com

License

Non-exclusive licence to reproduce thesis and make thesis public

I, Margus Luik,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Programming Patterns in Computer Games Course,

supervised by Dietmar Pfahl and Raimond-Hendrik Tunnel,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 19.05.2015