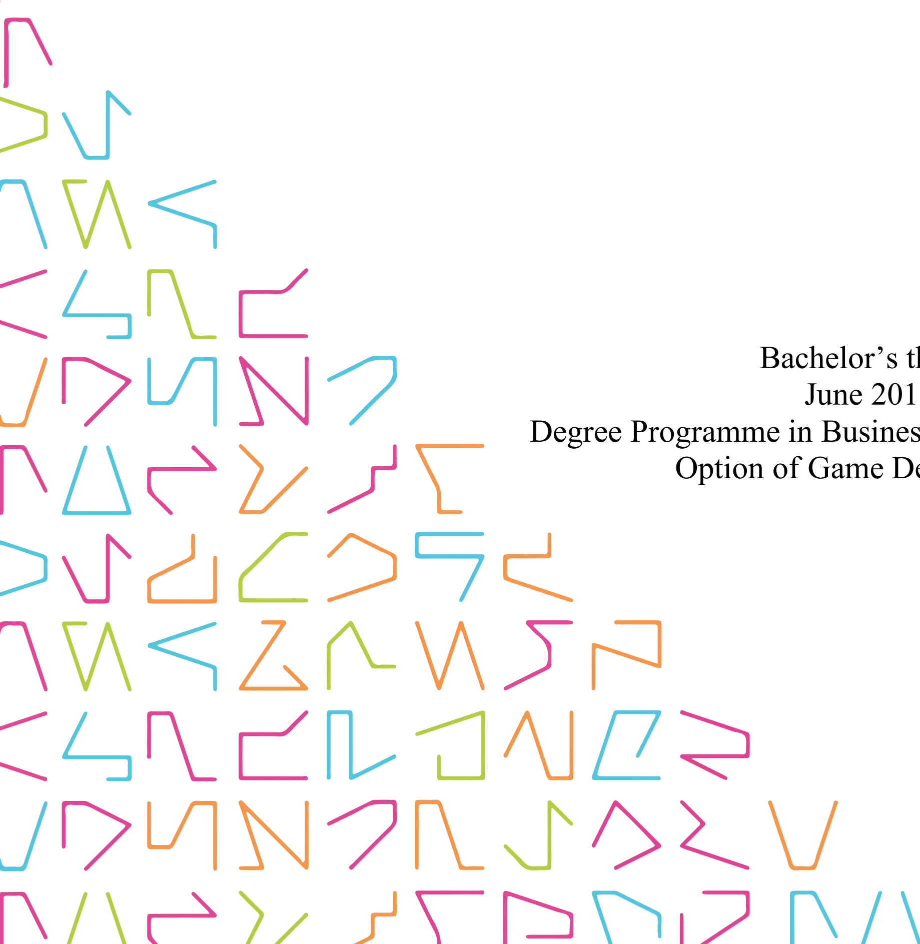
TAMPERE UNIVERSITY
OF APPLIED SCIENCES

# GAME DESIGN PATTERNS

## Utilizing Design Patterns in Game Programming

Anni Rautakopra



Bachelor's thesis
June 2018
Degree Programme in Business Information Systems
Option of Game Development

# ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Game Development

RAUTAKOPRA, ANNI:
Game Design Patterns
Utilizing Design Patterns in Game Programming

Bachelor's thesis 40 pages
June 2018

---

There is more to programming than the mere understanding of syntax. Badly planned and built code is difficult to understand or extend, making it invariably expensive to maintain. That is why it is important to be able to design code in an intelligent way. This skill is as valuable in game programming as it is to the development of any other software. The client of this thesis was the Option of Game Development of Tampere University of Applied Sciences: game development students would stand to gain from a study on code design, as there are no courses available on the subject.

The research problem was to find out what design patterns are and whether they could be used in game programming to help organize code: making it easier to read, modify and maintain. The objective of this thesis was to answer the questions of the research problem, to offer the students of the Option of Game Development information about design patterns that can be used in game programming, and how to utilize them with Unity game engine. The objective was also to encourage the students to utilize design patterns in their own game projects, and to inspire them to find out more about design patterns on their own. The research was implemented as a constructive research, using qualitative methodology. The research method was text analysis. The practical part of the thesis consisted of applying the theory into practice by refactoring a tangled game project that was made with Unity, with the help of design patterns. The purpose of this thesis was to create a study for the Option of Game Development of Tampere University of Applied Sciences about design patterns, and how they could be utilized in game programming, by providing examples from the Unity game project that was refactored using design patterns.

The results of the thesis confirmed that design patterns can also be used in game programming. It was also found that best programming practices play an important part in intelligent software design: many of the patterns rely heavily on them. The practical part of the thesis proved that design patterns can also be used in games made with a game engine like Unity, even though the structure of the development environment and the design of the code structure is different from the original frame of reference of design patterns.

The field of computing evolves constantly, and so do design patterns along with it, but one thing remains the same: design patterns play a key part in designing and building code that is easy to read, modify and maintain. Design patterns are a powerful tool in building games, but they have to be used intelligently. Which design patterns to use, is determined by the type of the game and the requirements it imposes.

---

Key words: game design patterns, programming patterns, best programming practices, Unity, C#

**TIIVISTELMÄ**

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

RAUTAKOPRA, ANNI:
Pelien suunnittelumallit
Suunnittelumallien hyödyntäminen pelien ohjelmoinnissa

Opinnäytetyö 40 sivua
Kesäkuu 2018

---

Ohjelmointi on muutakin kuin syntaksin ymmärtämistä. Huonosti suunniteltu ja rakennettu koodi tulee kalliiksi hankalan sekä virhealttiin muokattavuutensa vuoksi. Täten taito tuottaa helposti ymmärrettävää, muokattavaa ja ylläpidettävää koodia on sekä tärkeä että ajankohtainen niin pelien kuin muidenkin ohjelmistojen rakentamisessa. Opinnäytetyön toimeksiantajana toimi Tampereen ammattikorkeakoulun pelituotannon linja, jonka opiskelijoille tutkimus aiheesta olisi hyödyllinen, koska siitä ei ole kursseja tarjolla.

Tutkimusongelmana oli selvittää, mitä suunnittelumallit ovat ja voidaanko niitä hyödyntää pelien ohjelmoinnissa helpottamaan koodin luettavuutta, muokattavuutta ja ylläpidettävyyttä. Opinnäytetyön tavoitteena oli etsiä vastauksia tutkimusongelmassa asetettuihin kysymyksiin, tarjota pelituotannon opiskelijoille tietoa peleihin soveltuvista suunnittelumalleista ja siitä, kuinka hyödyntää niitä Unity-pelimoottorilla tehdyissä peleissä. Tavoitteena oli myös kannustaa opiskelijoita hyödyntämään suunnittelumalleja omissa projekteissaan ja innostaa etsimään niistä lisää tietoa itse. Tutkielma toteutettiin konstruktiivisena tutkimuksena kvalitatiivista tutkimusotetta hyödyntäen. Tutkimusmetodina käytettiin tekstianalyysiä. Opinnäytetyön tarkoituksena oli tuottaa tutkielma suunnittelumalleista ja niiden hyödyntämisestä pelien ohjelmoinnissa antamalla käytännön esimerkkejä suunnittelumalleilla korjatusta, Unity-pelimoottorilla toteutetusta pelistä.

Tutkielman tuloksena varmistui, että ohjelmoinnin suunnittelumalleja voidaan hyödyntää myös pelien ohjelmoinnissa. Suunnittelumallien lisäksi tärkeäksi osaksi hyvää ohjelmiston suunnittelutapaa nousi parhaiden ohjelmointikäytänteiden käsite, johon myös suunnittelumallit pohjaavat. Teoriaa soveltava käytännön osuus osoitti, että suunnittelumalleja voidaan soveltaa myös Unityllä toteutettavan pelin tekemisessä, vaikka kyseisen kehitysympäristön rakenne ja tapa rakentaa luokkia ja olioita poikkeavatkin suunnittelumallien alkuperäisestä viitekehyksestä.

Ohjelmointi on jatkuvasti kehittyvä ala ja suunnittelumallit kehittyvät sen mukana. Yksi asia ei kuitenkaan muutu: suunnittelumallit ovat tärkeä osa helposti ymmärrettävän, muokattavan ja ylläpidettävän koodin suunnittelua ja rakentamista. Suunnittelumallit ovat hyödyksi pelien ohjelmoinnissa, mutta niitä täytyy käyttää tarkkaan harkiten. Se, mitä suunnittelumalleja kussakin pelissä kannattaa hyödyntää, riippuu pelin tyypistä ja sen asettamista vaatimuksista.

---

Asiasanat: pelien suunnittelumallit, ohjelmoinnin suunnittelumallit, parhaat ohjelmointikäytänteet, Unity, C#

**FOREWORD**

A screenshot from Ember Breath, a Unity game project

## Thank You


- To the super talented development team of Ember Breath: to Silja for the wonderful game design and the masterful storytelling, and to Mika for the awesome, jaw-dropping graphics and amazing animations.
- To Jussi Pohjolainen for opening the door to the world of programming for me. When I began my studies at TAMK I honestly thought I would never be able to understand programming, and that I would surely fail the first-year compulsory programming class. But here we are now. Your courses were one of the best parts of my studies, with all the friendly bake-offs and shared jokes about classic old-school Sierra adventure games. You truly deserve the Best Teacher Award™!
- To Paula Hietala for the irreplaceable guidance and unfaltering support in my time of need. For that I will always be grateful.
- Last but not least: To Laura, Silja and Tuomo for the friendship and the uplifting moral support, and for the invaluable commenting and eagle-eyed proofreading of my work. You guys are the best! <3

**CONTENTS**


1 INTRODUCTION ..... 6
2 RELAX, SOMEONE HAS ALREADY SOLVED YOUR PROBLEM ..... 8
  2.1 What Are Design Patterns and Why Should You Use Them ..... 8
  2.2 Everything Begins with Change ..... 9
  2.3 Battle Change with Decoupling ..... 11
  2.4 What Is All This Talk About Best Programming Practices ..... 12
3 BEST PROGRAMMING PRACTICES 101 ..... 13
  3.1 The Four OOP Cornerstones ..... 13
    3.1.1 Abstraction ..... 13
    3.1.2 Encapsulation ..... 14
    3.1.3 Polymorphism ..... 14
    3.1.4 Inheritance ..... 15
  3.2 The Nine Design Principles ..... 15
    3.2.1 Identify the Aspects that Vary ..... 16
    3.2.2 Program to an Interface, Not an Implementation ..... 16
    3.2.3 Favor Composition over Inheritance ..... 17
    3.2.4 Loose Coupling ..... 18
    3.2.5 The Open-Closed Principle ..... 18
    3.2.6 The Dependency Inversion Principle ..... 18
    3.2.7 The Principle of Least Knowledge ..... 19
    3.2.8 The Hollywood Principle ..... 19
    3.2.9 Single Responsibility ..... 20
4 THREE DESIGN PATTERNS TO GET YOU STARTED ..... 21
  4.1 The Game Loop Pattern ..... 22
    4.1.1 Structure ..... 22
    4.1.2 How I Implemented the Pattern in Unity ..... 24
    4.1.3 Related Patterns ..... 27
  4.2 The Component Pattern ..... 27
    4.2.1 Structure ..... 28
    4.2.2 How I Implemented the Pattern in Unity ..... 30
    4.2.3 Related Patterns ..... 31
  4.3 The Singleton Pattern ..... 32
    4.3.1 Structure ..... 32
    4.3.2 How I Implemented the Pattern in Unity ..... 33
    4.3.3 Related Patterns ..... 35
5 WHERE TO GO FROM HERE ..... 36
REFERENCES ..... 40

# 1 INTRODUCTION

The codebase looks like a teetering house of cards on the verge of collapsing, held together only by massive amounts of duct tape and glue. If you try to build upon it or change some aspect about it, or even look at it in a wrong way, it will collapse under its own weight. Sound familiar? That is how I felt, looking at the codebase of Ember Breath, a Unity game project in which I was the programmer. Looking at the code I had written, I got the feeling that it would be easier to rig the whole thing with TNT and blow it to smithereens and start all over again from scratch, than to try to stop the horrid mess of hotfixes and prayers from either falling apart or getting even more tangled every time I tried to fix a problem or add a new feature. The real problem was that I did not know how to do it any better even if I did start from the beginning.

It was not a new predicament for me, I had pondered about how to construct my code in a better way since my first-year game project. Sure, sticking the code in one huge class might work for small programming class assignments, but it does not seem like a good idea when you are trying to build something bigger, like a game you actually have to ship. However, deciding how to divide the code in classes and objects *in a smart way* did not seem that straightforward either. It was time to really start figuring out what I could do to organize my code better. Pursuing this question led me to learn about design patterns, which in turn inspired me to choose them as the subject of my thesis. The subject also roused the interest of the teachers of the Option of Game Development of Tampere University of Applied Sciences (TAMK) – the results of my research might prove useful for those game development students who are battling with similar problems, especially since there are neither books written about design patterns in Unity, nor courses on the subject.

Thus, the object of my thesis became to establish whether design patterns could be used in game programming to help organize code; making it easier to read, maintain, and modify. The purpose of my thesis is to create a study that offers information about design patterns, and how they could be utilized when making games with Unity, for the students of the Option of Game Development of TAMK. It aims to encourage the students to use design patterns in their own game projects, and to inspire them to find out more about

design patterns on their own. This would also be beneficial to the Option of Game Development since the Option of Game Development follows a self-study-oriented style, where students learn by making games as project work.

The study is written in English because that is the new teaching language of the Option of Game Development. Another case for the choice of language is that English is the lingua franca of computing: it makes more sense to learn about design patterns and their terminology in their original language than to translate it to Finnish. That in turn makes it easier to communicate thoughts and ideas concerning software design with other programmers, since the terms and concepts can be discussed in a common, shared vocabulary.

In view of the intended target audience, the writing style of the thesis has been kept more conversational and personal, in order for it to be more easily approachable, thus being more likely to be read than an instruction manual.

During my research into design patterns it turned out they have a lot to do with best programming practices. You could say best programming practices are kind of building blocks that many of the design patterns are built upon. That is why in this thesis I will first take a quick tour introducing the distilled knowledge of best programming practices: The Four Cornerstones of OOP (Object-Oriented Programming) and the Nine Design Principles. After that I will introduce three design patterns that, in my opinion, would allow for an easy way into the world of design patterns. These patterns also helped me refactor and reorganize the codebase of Ember Breath. I will illustrate how I benefited from the design patterns and best programming practices by providing examples from Ember Breath in C#. With this, I hope to show you that design patterns are very useful, and that they can also be applied in practice with a game engine like Unity. There are many design patterns that can be used with games: I chose these three hoping that they would be the easiest ones to understand, in order to show you how design patterns work.

So, if you are like me and want to know more about programming beyond syntax, I hope to give you a short introduction to the world of design patterns and best programming practices. After all, why bang your head against the keyboard in frustration when you can take advantage of the well tested and time proven techniques of skilled software architects, and instead spend more time on all of the other cool stuff that coding games entails.

# 2 RELAX, SOMEONE HAS ALREADY SOLVED YOUR PROBLEM

Have you sat before the bright bluish light of the computer screen late at night, scratching your head until it bleeds, trying to figure out how to make the code bend to your will and make that measly little game object do what you want instead of it doing what it wants or, even worse, nothing at all? Frustrating, to say the least. Trying to reinvent the wheel usually is. That is why it is so relieving to realize, that someone has already done the heavy lifting for you, and you can benefit from the sweet, sweet results. That is, in essence, what design patterns are.

## 2.1 What Are Design Patterns and Why Should You Use Them

Design patterns are distilled knowledge of how to design object-oriented software. They are simple and elegant solutions to specific problems (Gamma et al. 1995, xi).

> Design patterns capture solutions that have developed and evolved over time. Hence they aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form. (Gamma et al. 1995, xi.)

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – later lovingly dubbed the Gang of Four, or, even shorter, GoF – were the first to document and catalogue software design patterns in their seminal book Design Patterns: Elements of Reusable Object-Oriented Software. Their aim was to give a voice to all of the silent knowledge and knowhow that had accumulated in the field of software development about how to write object-oriented programs in an intelligent way. Design patterns are created to solve design problems, like finding the appropriate objects, determining object granularity, and achieving intelligent code reuse (Gamma et al. 1995, 11–22). They show you how to build programs with good object-oriented design qualities (Freeman et al. 2004, 32). Game design patterns are design patterns that are especially useful for making games. Many of the design patterns introduced by Gamma et al. (1995) can also be used in making games, but Robert Nystrom (2009-2014) introduces in his book many other design patterns that are specifically useful when creating games.

As can be deducted from the quote above, because design patterns are not the designs that first come to mind when your fingers enthusiastically hit the keyboard in order to create that awesome game you have in mind, they usually bring with them a complexity that your initial code, written in a moment of great inspiration, might not have. That complexity might seem unnecessary when a simpler solution would have sufficed. And often that might very well be the case, for if you find yourself using a design pattern for writing a Hello World app, you know you have the Pattern Fever (Freeman et al. 2004, 27). So, as the KISS principle (Keep It Simple, Stupid) advices, it is often best to start from the simplest solution and only introduce complexity when it is absolutely necessary. That being said, sometimes design patterns are the easiest way to keep your program simple and flexible (Freeman et al. 2004, 594). After all, oftentimes hacking up code as you go can lead to a complexity of a different kind, that might otherwise be described as a mess.

Patterns are not meant to be wildly stamped around your code like funny animal stickers to a children's sticker book, they are meant to be used when the situation calls for a well-documented and tested solution to a specific problem. Design patterns do not offer code that you can just stamp in your codebase like that funky sticker, instead they give you a general solution to a specified design problem. You could say it is a matter of design reuse, instead of code reuse. Design patterns show how you can structure classes and objects to solve a certain type of a problem. It is then your job, as a programmer, to adapt these designs to fit your specific application. (Freeman et al. 2004, 29, 32.)

But if it is best to keep things as simple as possible, what is then the motivation to use something as complex as a design pattern?

## 2.2 Everything Begins with Change

Change is the one true constant in software development. Regardless of what you are building or which language you are programming with, you will always be stuck with issues of change. (Freeman et al. 2004, 8.) It is not about how well you design the software. Over time the application has to be able to grow and change or else it will face its end. In order to survive the changes that for example time, the needs of the client and customers, or changes in the API you are using cause, your application should be prepared

for change. Issues of change are, in fact, in the heart of most of the design patterns and principles (Gamma et al. 1995, 24; Freeman et al. 2004, 32).

Game developer Robert Nystrom explains that design patterns are more about how to organize code rather than about writing the code itself (Nystrom 2009-2014, Architecture, Performance, and Games). After all, all of the code you write is organized in some way, and sadly even bad organizing is still organizing, as I have experienced first-hand with the codebase of Ember Breath. Thus, it makes more sense to talk about how to organize your code *well* (Nystrom 2009-2014, Architecture, Performance, and Games).

According to Nystrom good architecture is such that when you need to make a change to the code, it is like the whole design of the codebase was prepared for it (Nystrom 2009-2014, Architecture, Performance, and Games). Making the change becomes easy and painless and it does not mangle the rest of the codebase or cause unexpected repercussions in some other part of the code. That does sound awesome, but how do you actually achieve such a flexible design? Nystrom continues:

> The first key piece is that architecture is about change. Someone has to be modifying the codebase. If no one is touching the code – whether because it's perfect and complete or so wretched no one will sully their text editor with it – its design is irrelevant. The measure of a design is how easily it accommodates changes. With no changes, it's a runner who never leaves the starting line. (Nystrom 2009-2014, Architecture, Performance, and Games.)

This brings to mind one of my mottos: Mutate and survive. A well-designed program will survive the changes the outside world forces on it. A badly planned codebase, on the other hand, can collapse under its own weight when, for example, you have to introduce a new cool feature, or change a game mechanic that was not as awesome as it was supposed to be. The problem often being – in my case at least – that even though I reasoned that there must be a better way to deal with the problem at hand, I could not figure out what that way was. I was trying to reinvent the wheel. Without the knowledge of design patterns, all I did was improvise code as I went along, adding yet another layer of duct tape on top of another as new requirements arose. Just think of the debugging and maintenance night-mare that approach results in!

> The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly. To design a system so that it's robust to such

> changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and reimplementation, client modification, and retesting. Redesign affects many parts of the software system, and unanticipated changes are invariably expensive. (Gamma et al. 1995, 23–24.)

When you code a personal project just for fun, or as a course assignment, it might not matter that much if under the hood your code is a hot mess. You might think that it is good as long as it goes through the compiler. But if you are trying to ship a game that needs to stay alive a long time and thus eventually evolve, and bring an income for the developers, bad code design becomes an active and expensive problem.

So, when exactly should you prepare for change? Sadly, there seems to be no right answer for that. It is more of a judgement call, a thing you learn to be better at with experience. Programming is also a form of art, and the more you learn about change and how it affects your design, the easier it gets to decide when to allow for it. (Holzner 2006, 27; Freeman et al. 2004, 594.) Learning about design principles and design patterns is a step toward that knowledge and experience.

### 2.3 Battle Change with Decoupling

Before you can actually make any changes in the code, you first have to understand what the existing code does. You do not have to be familiar with the whole codebase, but you do need to understand all of the relevant parts before you can confidently refactor the code. Nystrom explains that this step is often overlooked, even though it is often the most time-consuming part of programming (2009-2014, Architecture, Performance, and Games.) “If you think paging some data from disk into RAM is slow, try paging it into a simian cerebrum over a pair of optical nerves.” (Nystrom 2009-2014, Architecture, Performance, and Games).

That is why it really pays to keep your codebase as decoupled as possible. The less code you need to memorize in order to understand the code you are about to change, the easier it becomes to maintain that code. Less time spent pulling your hair because the codebase looks like a hairy monster from the sewers → more time spent implementing cool new features on your game. “While it isn’t obvious, I think much of software architecture is

about that learning phase. Loading code into neurons is so painfully slow that it pays to find strategies to reduce the volume of it." (Nystrom 2009-2014, Architecture, Performance, and Games). As a testament to the point, Nystrom explains that he has an entire section on decoupling patterns in his book, and that a large portion of the Gang of Four's design patterns focus on the same idea (Nystrom 2009-2014, Architecture, Performance, and Games).

> You can define 'decoupling' a bunch of ways, but I think if two pieces of code are coupled, it means you can't understand one without understanding the other. If you *de*-couple them, you can reason about either side independently. That's great because if only one of those pieces is relevant to your problem, you just need to load *it* into your monkey brain and not the other half too. To me, this is a key goal of software architecture: **minimize the amount of knowledge you need to have in-cranium before you can make progress.** (Nystrom 2009-2014, Architecture, Performance, and Games.)

Another way to look at decoupling is that a change to one part of the codebase does not necessitate a change to another. Of course, *something* needs to be changed, but the less coupling there is, the less that change affects the rest of the codebase. (Nystrom 2009-2014, Architecture, Performance, and Games.)

## 2.4 What Is All This Talk About Best Programming Practices

Why talk about best programming practices when this thesis is supposed to be about game design patterns? While researching about design patterns, it became clear to me that in order to understand the ideology behind design patterns, it is first necessary to discuss about best programming practices and design principles. These principles and practices are the groundwork upon which the structure and design of the software rest on. According to Freeman and Freeman, Sierra and Bates (2004, 32) "good OO designs are reusable, extensible and maintainable". Freeman et al. explain that design patterns show you how to build systems with good OO (object-oriented) design qualities. And to turn that around, many of the design patterns rely on OO basics and principles. (Freeman et al. 2004, 32.)

# 3 BEST PROGRAMMING PRACTICES 101

PhD Steve Holzner (2006, 9) explains in his book Design Patterns for Dummies that there is more to using design patterns than just memorizing them. Underneath design patterns there are OO principles and in order to understand how to work with design patterns you first have to understand the OO insights behind them. (Holzner 2006, 9.)

In other words, in order to be able to understand how, why and when to use design patterns it is important to first comprehend the OO principles behind them. And these principles – or best programming practices – are key in any software design, also for creating games. Many of the design patterns as well as game design patterns are built using best programming practices. That is why it is important to first refresh your memory on them – or learn about them, if they are new to you – before we move on to the actual design patterns in the next chapter.

## 3.1 The Four OOP Cornerstones

According to Holzner (2006, 19) there are four cornerstones of object-oriented programming – abstraction, encapsulation, polymorphism, and inheritance. These pillars of OOP are important building blocks that form the foundation of design patterns. Many of the patterns also make use of these building blocks.

### 3.1.1 Abstraction

Abstraction plays a big part in working with design patterns. Abstraction is not a programming technique, it is the process of conceptualizing a problem before applying OOP techniques. (Holzner 2006, 19.) An important part of using design patterns is to set up the way you attack the design problem correctly, and that often means spending more time working on the abstraction than on the concrete classes (Holzner 2006, 20). Spending enough time on carefully planning the structure of your codebase will most likely save you a lot of time and effort later when you need to maintain or change parts of your code. Just blindly coding away with the first idea you get and then later adding code hacks here

and there every time the specs change might lead to a codebase that is prone to bugs and no longer maintainable and extendable without huge refactoring efforts.

### 3.1.2 Encapsulation

Encapsulation means wrapping methods and data up into an object, much like how a pile of transistors, wiring and circuitry becomes, conceptually, a computer, or an electric kettle for example. Removing the complexity from view and making it into an easily graspable object is what makes objects so powerful. By encapsulating functionality into an object, you decide what sort of an interface that object exposes to the world (Holzner 2006, 20.) Your electric kettle, for example, might have all sorts of cool behind-the-scenes mechanics for heating the water just the right temperature for your choice of a mean cup of pu'erh, matcha or oolong tea, but the interface it offers to the tea connoisseur is just a simple dial. "In the same way, you decide what getter and setter methods and/or public properties your objects present to the rest of the application so that the application can interact with it." (Holzner 2006, 20).

### 3.1.3 Polymorphism

Polymorphism is the ability to write code that can work with different object types and decide on the actual object type at runtime (Holzner 2006, 20). For example, you might want to write code that handles all types of different animals that make sounds, for a cool "The Nature Sound of the Week" app. Although there is a myriad of different animals, they all have some common aspects, as far as your code goes, like making some sort of sounds. Rather than writing code like

```
Cat cat = new Cat();
cat.makeSound();
```

polymorphism allows you to make your code more flexible by deciding on the actual animal you want to use at runtime, rather than at compile time.

```
Animal animal = new Animal();
animal.makeSound();
```

This sort of polymorphic code will work with any animal that can make a sound, without having to be rewritten. All you have to do is create an Animal base class that has a makeSound() method and have all the different animals extend this class and create their own makeSound() implementation. This is called inheritance. Polymorphism then allows you to use animal.makeSound() to make any of the different animals make a sound since all the animals are derived from the Animal base class and implement the makeSound() method.

### 3.1.4 Inheritance

As mentioned above, inheritance is a process where a class can inherit methods and properties from another class. Inheritance is also an easy way to achieve code reuse by implementing features in the base class and having the derived classes then make use of them. This, however, can lead to unexpected rigidity and problems in your code, especially when it comes to maintaining it. Inheritance sets up a so-called IS-A relationship – a Cat *is an* Animal, for example. Design patterns, however, tend to favor composition over inheritance. When you use composition, your object contains other objects instead of inheriting from them. (Holzner 2006, 23.) For example, a Car *has* Wheels. This is called a HAS-A relationship. This allows for much more flexibility than the rigidity that inheritance forces. That is not to say that inheritance is a bad thing that should not be used. Inheritance is an important and powerful feature of OOP when used properly. And this is what many of the design patterns address.

## 3.2 The Nine Design Principles

What started with Gamma et al. (1995) in Design Patterns: Elements of Reusable Object-Oriented Software, evolved with Head First Design Patterns (Freeman et al. 2004). In their book Freeman et al. have crystallized the good OO practices Gamma et al. introduced into nine design principles. These principles form the basis of good OO programming practices, and they can also be found within many of the design patterns Gamma et al. introduced, as well as other design patterns that emerged later.

### 3.2.1 Identify the Aspects that Vary

Freeman et al. (2004) describe the principle as follows: "Identify the aspects of your application that vary and separate them from what stays the same." In other words, if you have a part of code that is changing a lot, for example with every new requirement, then you have behavior that needs to be separated and encapsulated from all of the stuff that does not change. This way you can later alter or extend the parts that vary without having to touch or affect the parts that do not. As simple as this concept sounds, it forms the basis for almost every design pattern: "All patterns provide a way to let *some part of a system vary independently of all other parts*." (Freeman et al. 2004, 9.)

### 3.2.2 Program to an Interface, Not an Implementation

> The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code. And we could rephrase 'program to a supertype' as 'the declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn't have to know about the actual object types!'. (Freeman et al. 2004, 12.)

Using the example from earlier, programming to an implementation would look like:

```
Cat cat = new Cat();
cat.meow();
```

Using a concrete implementation of the Animal base class forces the code to a concrete implementation, which is not what you want, whereas programming to an interface/supertype would look like:

```
Animal animal = new Cat();
animal.makeSound();
```

Even though we still know it is a Cat, we can now use the animal reference polymorphically. To go one step further, Freeman et al. (2004) present one more example:

```
animal = getAnimal();
animal.makeSound();
```

This implementation – rather than hard-coding the instantiation of the subtype, like new Cat(), into the code – assigns the concrete implementation object at runtime. This way the code does not have to know what the actual animal subtype is. All it needs to know is that the code knows how to respond to makeSound(). (Freeman et al. 2004, 12.)

### 3.2.3 Favor Composition over Inheritance

Inheritance gives you one way to reuse code, but often it is in the expense of a maintenance nightmare. Sure, it is very tempting to create a Monster base class for all the different opponents your player has to battle in your awesome RPG. Let's say the Monster base class has a health property and an attack() method. Then you create different monsters from orc to dragon that all derive from the Monster base class. Now, through inheritance, they all have health and an attack. Nice. But maybe you want the dragon to be a bit more powerful than the orc, so you then override the health property and the attack() method in the Dragon class. Then you start thinking that there could also be orcs that can shoot fiery arrows instead of just maiming the player with a rusty pike. So you make another Orc class that now overrides the attack() method with a flaming arrow attack. And so on, and so forth.

Suppose you have more types of monsters than just these two orcs and the dragon in your game, and they all derive from the same Monster base class. Imagine the amount of code overriding and maintenance work you need to do in order to create and maintain your cool RPG, and you start to see why inheritance might not have been such a good a choice for code reuse as it first seemed.

Instead of relying on inheritance – or an IS-A relationship – it is better to identify the aspects that vary and use composition – or a HAS-A relationship – to achieve code reuse, and a more flexible design that is easier to maintain. In the case of the example RPG, it might make more sense to encapsulate the different attack behaviors into their own classes, like BreathAttack and FlamingArrowAttack and so on, and then *compose* a monster with a suitable attack pattern. The Monster base class can then be composed of an attack behavior that now can also be changed dynamically at runtime. This way you can easily create an orc that has a pike attack, but that can also start shooting flaming arrows on the fly if it finds a set of bow and arrows lying on the ground. Or create a dragon with a

deadly fiery breath attack – or an orc with a deadly fiery breath attack for that matter – by simply composing the orc class with a different attack pattern. You cannot achieve that if you hardcode the attack into the monster class itself.

### 3.2.4 Loose Coupling

"Strive for loosely coupled designs between objects that interact." (Freeman et al. 2004, 53). As discussed in chapter two, decoupling is an important aspect of intelligent software design. Because loose coupling minimizes the interdependency between objects, it enables building flexible OO programs that can handle change (Freeman et al. 2004, 53).

### 3.2.5 The Open-Closed Principle

"Classes should be open for extension, but closed for modification." (Freeman et al. 2004, 86). It might seem a bit contradictory, but the idea is to allow classes to be easily extended with new behavior without having to modify existing code. Many design patterns are designed to protect your code from modification by giving a means of extension. However, like many of the design patterns, using the Open-Closed Principle adds a new layer of abstraction, which adds complexity to the code. That is why you need to carefully choose the areas of code that need to be extended. (Freeman et al. 2004, 86-87.)

### 3.2.6 The Dependency Inversion Principle

"Depend upon abstractions. Do not depend upon concrete classes." (Freeman et al. 2004, 139). This principle is similar to the "Program to an interface, not an implementation" principle, but makes an even stronger statement about abstraction. The principle states that high-level components, like classes, should not depend on low-level components, but instead they should both depend on abstractions. This means that, for example, you should not let a variable hold a reference to a concrete class, or that no class should derive from a concrete class, or that no method should override an implemented method of any of its base classes (Freeman et al. 2004, 139, 143).

This, of course, is quite impossible to achieve in practice since no matter what you do, *somewhere* in your code you will have to instantiate a concrete class. But that is not the end of the world if your class is not likely to change. If, however, your class is likely to change, then you should try to encapsulate that change as much as possible. The point of the principle is to try to avoid dependencies on concrete types and instead strive for abstractions (Freeman et al. 2004, 162).

### 3.2.7 The Principle of Least Knowledge

"Principle of Least Knowledge – talk only to your immediate friends." (Freeman et al. 2004, 265). The idea behind this principle – also known as the Law of Demeter – is to reduce the interaction between objects to just a few close "friends", so that when you design a system, you pay attention to how many classes your objects interact with, and in what way. It is another reminder to keep your classes as decoupled as you can, because if you have a lot of dependencies between many classes, your system will become fragile against change, making it expensive to maintain and complex for others to understand. (Freeman et al. 2004, 265.)

### 3.2.8 The Hollywood Principle

"Don't call us, we'll call you." (Freeman et al. 2004, 296). The Hollywood Principle was created to battle dependency rot. "Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on sideways components depending on low-level components, and so on." (Freeman et al. 2004, 296). When your code is infected with dependency rot, it becomes hard for anyone to understand how your system is designed. The idea with the Hollywood Principle is to allow low-level components to hook themselves into the system, but let the high-level components decide when and how they are needed. (Freeman et al. 2004, 296.)

### 3.2.9 Single Responsibility

“A class should have only one reason to change.” (Freeman et al. 2004, 339). Every responsibility a class has is an area of potential change. Having more than one responsibility means the class has more than one area of change. By now it is clear that change is something you want to avoid in a class as much as possible, because modifying code provides all kinds of opportunities for problems to creep in. A class that has more than one way to change has an increased probability to change in the future. And when it does, the change is going to affect more than one aspect of the design, so having only one responsibility per class makes sense. (Freeman et al. 2004, 339.)

# 4 THREE DESIGN PATTERNS TO GET YOU STARTED

Having covered the design principles and best practices many of the design patterns are based on, we can now move on to the actual star of the show. Design patterns can be defined as a tool that should only be used when needed. As discussed previously, the design of software should always be first started from design principles, creating the simplest code that does the job. Design patterns should only be introduced when the need for one emerges, or when you are quite sure that in the future there will be a change in the requirements of your application that a design pattern can deal with. (Freeman et al. 2004, 596.) That being said, only knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object-oriented designer. To be one, you need to think about how you can create flexible designs that are maintainable and can cope with change, and that is where design patterns truly shine. (Freeman et al. 2004, 31.)

In this chapter I will introduce three design patterns that I think are amongst the easiest patterns to understand. They work well in game programming and also with Unity. These design patterns also helped me fix the mangled codebase of duct tape and interdependencies galore Ember Breath had become. The design patterns that are right for you depend entirely on the type of game you are making, and the types of code structures it entails. The following design patterns were useful in solving the types of problems I had with the code and structure of Ember Breath.

The Game Loop Pattern is an essential pattern in any game. It is the heart of the game: its job is to run continuously during gameplay, processing user input and updating and rendering the game state. To put it simply: if you intend to make a game, you will need some sort of a game loop. In Unity you cannot create your own implementation of a game loop from the ground up, Unity has its own game loop mechanisms that you have to use. But there are things to consider when using Unity’s game loop, that I will discuss.

The Component Pattern aims to keep different domains of the code separate from each other, to avoid code coupling. By keeping code as loosely coupled as possible, it will be easier to comprehend, manage and extend. The pattern also showcases how Unity is built: every game object is composed of one or more components.

The Singleton Pattern offers a solution when you need to guarantee that there exists only one instance of a class. The pattern has gained criticism because of misuse, but when used properly it works wonders on stopping multiple instances of you game manager class existing, for example.

By introducing these design patterns and showing how they helped me in reorganizing a codebase that I thought was beyond repair, I hope to show you what design patterns are all about, and how they can be used with Unity. With these three patterns you can get a good head start into realizing the possibilities that design patterns offer, and to use them as a stepping stone to find out more about design patterns yourself. There is no way – for me at least – to determine which design patterns are the absolute best there are for game programming. As I said, that depends entirely on what you are making. But I can get you started with these three that are amongst the simplest to understand, showing you a case study on how I fixed my code design troubles with them.

### 4.1 The Game Loop Pattern

“Game loops are the quintessential example of a ‘game programming pattern’. Almost every game has one, no two are exactly alike, and relatively few programs outside of games uses them.” (Nystrom 2009-2014, Game Loop.) The job of the game loop is to run continuously during gameplay. During each turn of the loop, it processes user input without blocking the game, updates the game state, and renders the game. The game loop also tracks the passage of time in order to control the rate of gameplay. (Nystrom 2009-2014, Game Loop.)

#### 4.1.1 Structure

In its most simplified form a game loop handles user input, advances the game simulation one step and draws the result on screen so the player can see what happened. This process is then repeated indefinitely, until the game ends. A very simple example of a game loop might look something like:

```
while(isGameOn) {
    processInput();
    update();
    render();
}
```

But if you use an approach as simplistic as this, you will most likely run into performance problems. Your game loop will execute as fast as it can, meaning that when your game is running on a powerful computer or a mobile device it will run fast, but on an older or slower device it will run slower. This will create a difference in gameplay, making the player move in different speed depending on the underlying platform used for instance, making network gaming quite unfair.

In the early days of video games this was not a problem since the developers knew exactly which CPU the game was running on and they coded specifically for that setup. All they had to worry about was how much work should be done each frame. By coding the game to do just enough work each frame would guarantee the game ran at the speed the developers wanted. But playing that same game on a faster or slower machine would then make the game run faster or slower. (Nystrom 2009-2014, Game Loop.) This is also why it is necessary to have the ability to change the CPU speed in emulators such as Dosbox. As an example, while playing Gabriel Knight – an old PC adventure game – it became impossible to solve a puzzle that hanged on the movements of a doughnut-craving police officer. On the regular speed of the emulator the chubby officer was simply too fast to get his daily dose of beignet for Gabriel to sneak past him. The game loop was running as fast as it could and the default CPU settings for Dosbox were simply too high for the old game, effectively making the puzzle impossible to solve. The solution was to slow the CPU down so that the police officer would not be quite so fleet-footed.

> A game loop has two key pieces: non-blocking user input and adapting to the passage of time. Input is straightforward. The magic is in how you deal with time. There are a near-infinite number of platforms that games can run on, and any single game may run on quite a few. How it accommodates that variation is key. (Nystrom 2009-2014, Game Loop.)

With all the different platforms and hardware today, it is impossible as a game developer to know exactly what hardware setup your game will be running on. That is why it is crucial to make sure your game will run equally well on slower or older devices as well as newer ones. “This is the other key job of a game loop: it runs the game at a consistent

speed despite differences in the underlying hardware." (Nystrom 2009-2014, Game Loop).

### 4.1.2 How I Implemented the Pattern in Unity

"For me, this is the difference between an 'engine' and a 'library'. With libraries, you own the main game loop and call into the library. An engine owns the loop and calls into *your* code." (Nystrom 2009-2014, Game Loop.) This is the difference between the games coded in Java during first-year studies and the games coded with Unity during game development studies. During my first-year spring game project we made a game for the Nokia Asha feature phone (R.I.P.) using Java and Nokia Asha SDK Game API as framework. The game and its game loop were hand coded from the ground up. In a game engine like Unity the game loop is dictated by the engine, so most of the work is already done for you. But it is still possible – and important – to make sure your game runs equally on all of the different gaming platforms Unity supports out-of-the-box. You can do this by smartly utilizing the three different game loop methods – Update(), FixedUpdate(), and LateUpdate() – and the delta time functionality Unity offers.

Unity's regular game loop, Update(), is used for most game loop activities. Its placement within Unity's script lifecycle can be seen in figure 1 (Unity Manual, 2018b). Update() is called before the frame is rendered and animations are calculated (Unity Manual, 2018a). That makes it the main place to put most of your code that needs to update on each frame. But if you add code that changes the position of a game object for example, you should multiply your values by Time.deltaTime in order to make the game run evenly on different platforms and framerates.

The physics engine of Unity updates in discrete time steps, and FixedUpdate() is called just before each of the engine's physics updates, as can be seen in figure 1 (Unity Manual, 2018b). Because frame updates and physics updates do not necessarily occur with the same frequency, you will get more accurate results from physics code if you put it in FixedUpdate() instead of Update(). (Unity Manual, 2018a.) Also, because FixedUpdate() is called on a reliable timer, independent of the frame rate, you do not need to multiply your values by Time.deltaTime (Unity Manual, 2018b).

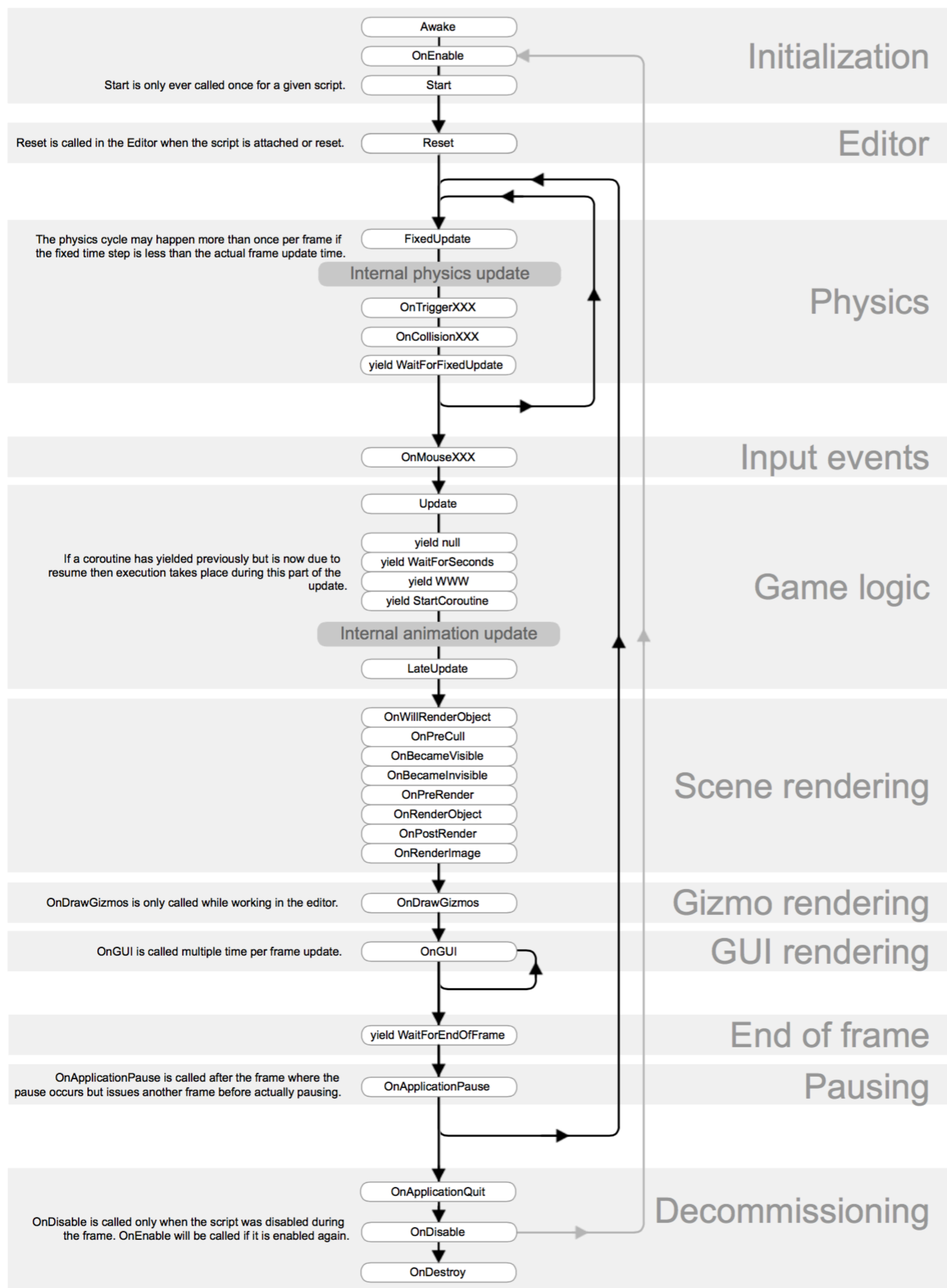



FIGURE 1. The script lifecycle flowchart of Unity (Unity Manual, 2018b)

LateUpdate() (figure 1) comes in handy if you want to make additional changes after FixedUpdate() and Update() have been called for all game objects in the scene and after all animations have been calculated (Unity Manual, 2018a; Unity Manual, 2018b). A character-following third-person camera is a common example. You put your character movement code inside the Update() method, and the camera movement and rotation code

in the LateUpdate() method. This will ensure that the character movement has been fully completed before the camera tracks its position. (Unity Manual, 2018b.)

My codebase for Ember Breath consists of many different scripts, but only a few of them actually require a game loop. Most of them rely on a different design pattern, called the Observer, that allows the scripts to activate only when certain conditions are met. The method OnTriggerEnter2D() in Unity is an example of the Observer pattern. The code inside the method is waiting for Unity to inform it that something has triggered the collider of the game object the script is attached to. When this trigger is activated, Unity calls the method and the code is executed. This is good for those game objects that do not need to update their status on every loop of the game, such as collectables and doors, but instead need to be activated only when the player interacts with them, for example. But for any playable character, on the other hand, that will not suffice. The character needs to receive commands from the player, needs to update its animation and behave according to the rules of the physics engine, and it needs to do that every single loop of the game.

For the playable character in Ember Breath I utilized both Update() and FixedUpdate() in the manner discussed above. The Move() method, that makes the character move according to the button presses received, is called in a fixed time step in FixedUpdate() while, for more accurate results, the button presses for the jump command are read in Update() that loops as fast as it can (picture 1).

```
16      void Update () {
17
18          // read jump input in Update() so that button presses aren't missed:
19          if (!isJumping) {
20              isJumping = Input.GetButtonDown ("Jump");
21          }
22      }
23
24      void FixedUpdate() {
25
26          // read the button presses for moving:
27          float move = Input.GetAxis ("Horizontal");
28
29          // pass the parameters to the PlayerController, to make the character actually move:
30          player.Move (move, isJumping, false);
31      }
```

PICTURE 1. The division between Update() and FixedUpdate()

### 4.1.3 Related Patterns

The Update Method is a very closely related pattern. It is defined as: “Simulate a collection of independent objects by telling each to process one frame of behavior at a time.” (Nystrom 2009-2014, Update Method). Game Loop and Update Method are like bread and butter, they often work together. If your game has dragons, cats, ninjas or other interactive actors, you will most likely use the Update Method pattern. If, however, your game is more like chess or Go, where the actors are passive, Update Method might be a poor fit. Although, even if you do not need to update the behavior of the pieces each frame, you might still want to update their animation every frame with the help of the Update Method pattern. (Nystrom 2009-2014, Update Method.)

## 4.2 The Component Pattern

“Allow a single entity to span multiple domains without coupling the domains to each other.” (Nystrom 2009-2014, Component.)

Best programming practices teach us that it is good to have one class for one behavior (Single Responsibility) and also that it is best to avoid coupling as much as possible (Loose Coupling, Favor Composition over Inheritance, The Principle of Least Knowledge). The Component Pattern aims to solve these sorts of problems by keeping different domains, like AI, physics, rendering and sound isolated from each other. If you crammed all of those into a single class, you would be creating a dumping ground of code that makes you wince when you even think about having to go through or edit it. Trust me, my expression was far from happy when I fired up the old version of the Ember Breath player control script in the editor. The best solution I could think of was to flip the table, set the whole codebase ablaze, and dance on top of its smoldering remains. Which incidentally is pretty much what I did – figuratively of course – after reading about the Component Pattern and the State Pattern.

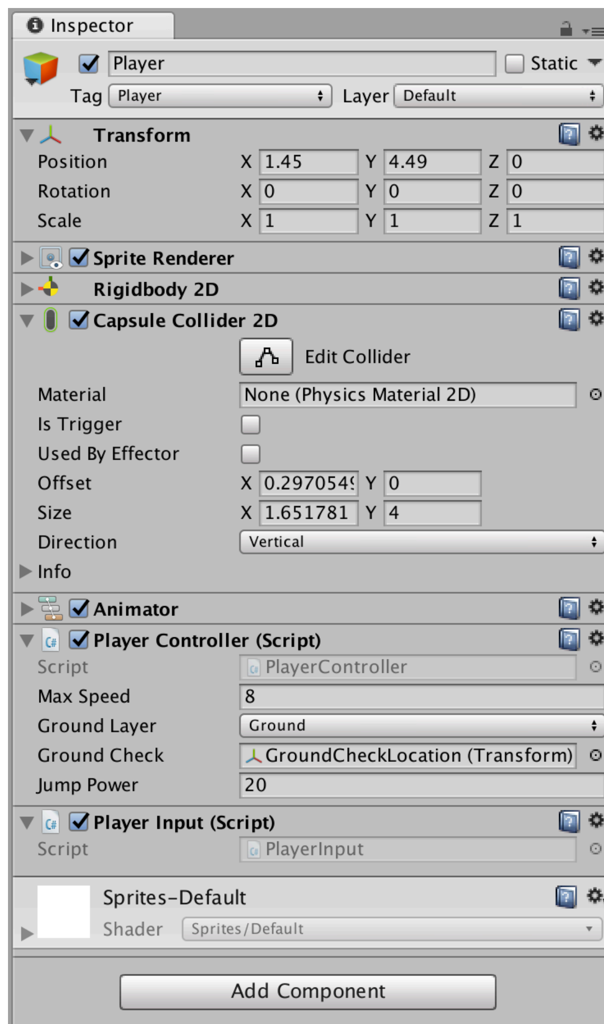
If you cram everything together like that, any programmer trying to make a change in one aspect of the code will have to know something about all of the domains just to make sure they don’t break anything along the way. Just imagine what would happen if the game uses concurrency, like the trend is today. With everything tightly coupled it would be

disastrous when introduced to multiple threads. “Having a single class with an UpdateSounds() method that must be called from one thread and a RenderGraphics() method that must be called from another is begging for those kinds of bugs to happen.” (Nystrom 2009-2014, Component.)

### 4.2.1 Structure

The solution is simply to divide the class into separate parts along domain boundaries. For example, the code for handling user input can be separated from the player’s movement logic and moved into a separate InputComponent class. The player class will then own an instance of this component (IS-A vs. HAS-A). Then it is a matter of repeating the process for each of the domains the player class touches. As an end result, almost everything is moved out of the player class and all that remains is a thin shell that binds the components together. This solves the huge tangled hairball class, but also solves the coupling problem by decoupling the components from each other. Even though the player class can have many components, like a physics component, and a graphics component, they do not know about each other. This means, that the programmer working on physics can modify the physics component without needing to know anything about graphics and vice versa.

This is also how Unity’s game objects are laid out. Each game object can have multiple components, like for example collider components, animation components and script components, but they do not know about each other. This way you can tweak one component, like the collider, without having to touch the other components. The game object itself acts as the shell that binds together all of the components attached to it. You can see an example of a Unity game object and the components attached to it in picture 2. The components are: Transform, Sprite Renderer, Rigidbody 2D, Capsule Collider 2D, Animator, Player Controller, and Player Input. As you can see, also scripts are treated as components of a game object in Unity.



PICTURE 2. A Unity game object with various components attached

Of course, the components will have to have *some* interaction between themselves. For example, a collision script will need to receive collision data from the collider component in order to correctly trigger a huge explosion with cool sound effects, or to get the player character to pick up a shovel. However, it is better to restrict this to the components that *do* need to talk instead of just tossing everything into the aforementioned huge-hairball-of-doom class and watch the world burn.

This also neatly makes the resulting components work as reusable packages. In Unity, you can add, for example, the collider component to as many game objects that you want. The component is made as a nice and neat reusable package that can easily be attached to game objects as needed. When designing your scripts with this component style, you can create reusable scripts that can be attached to different kinds of game objects, thus achieving sensible code reuse without having to result to inheritance, terrible hard-coded or boilerplate code. You could, for example, write one script for a door trigger that you can

attach to as many doors as you want, without having to write a new special script for each one of them separately.

### 4.2.2 How I Implemented the Pattern in Unity

As I mentioned before, the original player controller script I wrote for Ember Breath was quite nightmarish, to say the least. Everything was crammed into one huge script that coupled every imaginable part of the system together in a hard-coded shiver-inducing way. For example, the input axes were hard-coded in the script, making any changes to the input button layout tedious at best, having to go through the long script searching for places where the input axes were called. As a solution I separated the whole input logic from the player script into a separate script that would only handle that and nothing else (picture 3). As a result, if the button layout needs to be changed, it would only affect the input script, but changes there would in no way affect the player movement logic script (picture 4). Easier maintenance, less chance of new bugs being created when tweaking stuff and less hair loss from frustration → win!

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class PlayerInput : MonoBehaviour {
6
7     private PlayerController player; // reference for the PlayerController script
8     private bool isJumping; // flag for checking if the character is already jumping
9
10    void Awake () {
11        // fetch the PlayerController script that is attached to the game object:
12        player = GetComponent<PlayerController>();
13        isJumping = false;
14    }
15
16    void Update () {
17
18        // read jump input in Update() so that button presses aren't missed:
19        if (!isJumping) {
20            isJumping = Input.GetButtonDown ("Jump");
21        }
22    }
23
24    void FixedUpdate() {
25
26        // read the button presses for moving:
27        float move = Input.GetAxis ("Horizontal");
28
29        // pass the parameters to the PlayerController, to make the character actually move:
30        player.Move (move, isJumping, false);
31    }
32 }
```

PICTURE 3. The script for handling the input logic of the player character

```
35      // Move the character according to the button presses received from PlayerInput:
36      public void Move (float move, bool jump, bool climb) {
37
38          // calculate whether the player is grounded or not by testing the overlapping of the
39          // groundCheck point set to the player game object and the layer defined as ground in the game:
40          isGrounded = Physics2D.OverlapCircle (groundCheck.position, groundCheckRadius, groundLayer);
41          // and update the result to the animator of the player, so the animator
42          // can transition from the falling state animation to the idle state animation):
43          playerAnimator.SetBool ("isGrounded", isGrounded);
44
45          // set a velocity for the player game object using the max speed multiplier, keeping y axis as is:
46          playerRigidBody.velocity = new Vector2 (move * maxSpeed, playerRigidBody.velocity.y);
47          // and pass the move value to the animator as an absolute value, so that the animator
48          // can transition from the idle state animation to the walk state animation:
49          playerAnimator.SetFloat ("MoveSpeed", Mathf.Abs (move));
50
51          // jumping:
52          if (isGrounded && Input.GetAxis ("Jump") > 0) {
53              // let the animator know that the player is no longer grounded:
54              playerAnimator.SetBool ("isGrounded", false);
55              // reset the player's velocity y to zero to make all jumps equal height:
56              playerRigidBody.velocity = new Vector2 (playerRigidBody.velocity.x, 0);
57              // add a force only on the y axis, with the amout of jumpPower and as an rapid impulse:
58              playerRigidBody.AddForce (new Vector2 (0, jumpPower), ForceMode2D.Impulse);
59              // also let the script know the player has left the ground:
60              isGrounded = false;
61          }
62
63          // flipping the player's sprite to face the direction it's moving:
64          if (move > 0 && isFacingLeft) { // moving right but facing left:
65              Flip ();
66          } else if (move < 0 && !isFacingLeft) { // moving left but facing right:
67              Flip ();
68          }
69      } // Move ()
```

PICTURE 4. The method for handling the movement logic of the player character

### 4.2.3 Related Patterns

The Strategy Pattern, introduced by the Gang of Four, bears resemblance to the Component Pattern. They both take a part of an object’s behavior and delegate it to a separate subordinate object. “The difference is that with the Strategy Pattern, the separate ‘strategy’ object is usually stateless – it encapsulates an algorithm, but no data. It defines how an object behaves, but not what it is.” (Nystrom 2009-2014, Component.) Components, on the other hand, often hold state that describes the object and helps define its identity. It is not always that black and white, however: “You may have some components that don’t need any local state. In that case, you’re free to use the same component instance across multiple container objects. At that point, it really is behaving more akin to a strategy.” (Nystrom 2009-2014, Component.)

The official definition of the Strategy Pattern is: “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” (Gamma et al. 1995, 315.)

## 4.3 The Singleton Pattern

The Singleton Pattern is used when you need to guarantee that there exists only a single instance of a certain class. It is probably one of the most misused and misunderstood of patterns. Many even feel that the pattern should not be used at all. However, most likely a great deal of the misconceptions concerning the Singleton Pattern are due to the abuse and misusage of the pattern. After all, if you tried to use a hammer to screw on a screw you might get the misconception that the hammer is a totally useless tool, whereas in reality the hammer is an irreplaceable tool when you use it right. You just can't use a hammer to fix everything. Freeman et al. (2004) summarize it well:

> There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphic cards. In fact, for many of these types of objects, if we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results. (Freeman et al. 2004, 170.)

Gamma et al. (1995) define the Singleton Pattern as follows: "Ensure a class only has one instance, and provide a global point of access to it" (Gamma et al. 1995, 127). In a nutshell, you create a class and let it manage a single instance of itself. It is also crucial to prevent any other classes from creating new instances of the Singleton class on their own. To get an instance of the Singleton, you have to go through the class itself. It is also important to provide a global access point to the said instance so that whenever you need an instance, you just query the class and it will hand you back the single instance. (Freeman et al. 2004, 177.)

### 4.3.1 Structure

In figure 2 you will see the UML class diagram of the Singleton Pattern. It consists of just one class, the Singleton itself. According to the definition of the Pattern, the Singleton has a private constructor and a static class variable that holds the one and only instance of Singleton. The getInstance() method is also a static class method. This allows you to conveniently access the method from anywhere in your code using Singleton.getInstance(). This is just as easy as accessing a global variable, but unlike a global variable,

you cannot instantiate multiple objects from a Singleton. The private constructor ensures that no other class can instantiate the Singleton class.

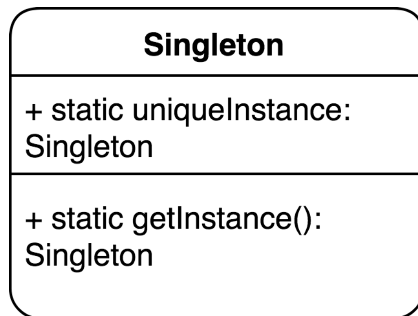



FIGURE 2. UML class diagram of the Singleton Pattern

#### 4.3.2 How I Implemented the Pattern in Unity

Already in the first version of Ember Breath, I used the Singleton Pattern to create a game manager that holds a reference to all of the game data that needs to persist between the changing of the scenes. My original version of the game manager did not conform to best programming practices however, so I refactored it to better suit the modularity Unity strives for. In the old version there was so much coupling among the scripts, that it would have surpassed most soap operas with its intertwined dependencies. Good luck trying not to cause a butterfly effect on the whole codebase when tweaking one of the interdependent variables "just a little bit".

My new implementation of the Singleton pattern with Unity is quite simple. Ember Breath has different rooms the player can traverse, and changing the room loads another scene in Unity. There is critical game data that needs to persist through these scene changes, like which collectable objects have already been collected, and which doors unlocked. Normally all of this data gets deleted when a new scene is loaded, so I need a way to store it somewhere so that the player cannot, for example, pick – or see – the same shovel more than once.

If I only needed to make a game object persist through scene changes, I could just use:

```
DontDestroyOnLoad(gameObject);
```

This would be enough if there are only a couple of scenes in the game. If you create your game manager this way in the first scene of the game, it will persist throughout the game. Cool. But what if your game has a hundred scenes and there is a nasty bug in room 99. You would have to play test the whole game until room 99 to even get to the bug, let alone test it. You could solve this by creating an instance of the game manager in each room, which would make it a lot easier for testing and debugging purposes. But that gives rise to another problem. Now, when you change the scene in the game you will be creating multiple instances of the game manager, which is less than desirable and will create a whole set of other problems.

To address this predicament, I decided to make use of the Singleton Pattern for my game manager. This way I am able to make one game object responsible for holding various critical game data throughout the game, but at the same time making sure there ever is only one instance of it in the scene. This way I can also have a game manager in each of the scenes, making testing and debugging easier, but in the same time ensuring that when I move to another scene in the game I will still have the right game controller with me.

You can see how my solution looks like, with simple example data, in picture 5.

```
 1 using System.Collections;
 2 using UnityEngine;
 3
 4 public class Singleton : MonoBehaviour {
 5
 6     public static Singleton uniqueInstance;
 7
 8     // simple example data to verify that the Singleton operates correctly:
 9     public int health;
10
11
12     void Awake () {
13
14         Debug.Log ("Singleton: Awake called.");
15
16         if (uniqueInstance != null) {
17             Debug.Log ("There already is a unique istance of this class, destroying this instance.");
18             // if an instance of this class already exists, destroy it:
19             Destroy (gameObject);
20         } else {
21             Debug.Log ("No previous instance, making this the unique instance.");
22             // there is no instance yet, so assign it:
23             uniqueInstance = this;
24             // prevent the automatic destruction of the object target when loading a new scene:
25             DontDestroyOnLoad (this);
26         }
27     }
28 }
```

PICTURE 5. A Unity script demonstrating the Singleton Pattern

#### 4.3.3 Related Patterns

According to Gamma et al. (1995), many patterns can be implemented using the Singleton Pattern. Abstract Factory, Builder and Prototype are mentioned. (Gamma et al. 1995, 134.)

Robert Nystrom (2009-2014, Singleton) reveals that he has never used the full Gang of Four implementation of the Singleton Pattern, he instead prefers to use static classes to ensure single instantiation. “If that doesn’t work, I’ll use a static flag to check at runtime that only one instance of the class is constructed” (Nystrom 2009-2014, Singleton). He gives a couple of other options for the classic Singleton Pattern: The Subclass Sandbox and the Service Locator, both of which can be found in his book.

# 5 WHERE TO GO FROM HERE

You have reached the end of my short introduction into the world of design patterns. It is my sincere hope that you have found at least some insight within these pages, perhaps a new tangent to propel your way forward through the vast sea of software development. My journey with design patterns has been a long and an interesting one, and I doubt the end is near, there is still so much to learn.

There are no books written – as far as I know – about design patterns in Unity, like there are books about design patterns in Java, C# or C++. It has therefore been an intriguing expedition to research about design patterns and to try to apply the gained knowledge in practice in a Unity game project. Since there are neither books about the subject, nor courses about it taught in TAMK, this study is hoped to be useful for the game development students.

The process of reading books about design patterns in general, to applying design patterns in practice on a game engine with its own rules and constraints, was slow and sometimes rather difficult, but it was also the most rewarding part of making this thesis. Sometimes I really wished that there was a Design Patterns in Unity for Dummies book. But if there was, I would have had to change the subject of my thesis. After all, the idea behind this thesis was to learn more about design patterns and how to utilize them in Unity, and to share that knowledge with other game development students who battle with giant hairball codebases of doom.

All those hours spent reading about design patterns really did pay out in the end, when I finally found the right pattern that would help me solve a problem of the codebase of Ember Breath, and when I figured out a way to utilize the pattern in Unity. Of course, I cannot be sure of the supremacy of my solutions since there was no For Dummies book about design patterns in Unity. But finding the right pattern and figuring out a good way to utilize it in Unity felt gratifying like finally solving a difficult puzzle, and I really learned a lot during the undertaking. I hope that in the process I managed to demonstrate that also your code puzzles can be solved with the help of design patterns. You just have to pick the right ones for your purpose.

I set out to find out whether design patterns could be used in game programming, whether they could be used to help organize code; making it easier to read, maintain, and modify. And also, whether design patterns could be used with a game engine like Unity, with its own restrictions and style of coding. During the process I discovered that best programming practices are an important part of design patterns. I used Ember Breath – an unfinished game project that had ended up in a deadlock code-wise – as an example project to refactor and fix with the help of design patterns and best programming practices. Although a game engine like Unity already makes use of many design patterns behind the scenes, and coding with Unity is different than when coding something from the ground up, I found that design patterns can nevertheless be used, and that design patterns are as useful in game programming as they are in any other software development.

In this thesis, I introduced three design patterns that I felt were amongst the easiest ones to start learning about design patterns, and that were useful in my particular circumstances, but of course there are many more you can use. Which ones you choose to use will depend entirely on the kind of game or software you are making. As stated before, the value of the design of your code is not in the amount of design patterns you managed to cram in it. The design of software should always be first started from best programming practices and design principles: design patterns should be added only when they are meaningful in solving a design problem.

Since there are a lot of design patterns that were left outside of the scope of this thesis, as a development suggestion I propose researching and testing other design patterns, to see if they could be used with Unity. For starters, another design pattern that could have been useful also in my game project is the State Pattern.

With the State Pattern you can, for example, create states for you player character to be in, like jumping and climbing. The character can only be in one state at a time. This allows the character to alter its behavior when the state changes. Sure, you can write it as a compilation of if statements instead, like I did originally. But if you need to add more behavior, like crouching, shooting, shooting while crouching, a special dash attack that can only be done while wielding a bazooka while double jumping, etc., it can quickly escalate to a horrid error prone hot mess, where each additional behavior you add will require another flag to be tested in the massive maze of conditional statements, most likely causing some nifty behavior bugs along the way. With the State Pattern you can clear out a lot of those

conditional statements and flags for checking, for example, whether the player character is already jumping when the player presses the jump button, or whether the player tries to use a shovel while the player character is climbing. This makes the code easier to read and helps solve weird bugs and eliminate situations that do not make sense – like shoveling while climbing – without having to create those complicated and error prone conditional statements with lots of flags. It also makes adding more of those cool, action packed behaviors a lot easier, because they will not mess up the behaviors you already have. So, if you would rather spend your time playing Monster Hunter World and eating chocolate, for example, it is worth to consider using the State Pattern instead.

If I succeeded in piquing your interest about design patterns, and you want to know more about them and how to utilize them in your projects, I have a couple of salient books to recommend. The seminal book on design patterns is of course Design Patterns: Elements of Reusable Object-Oriented Software (Gamma et al. 1995). Even though it is a really old book – especially by the standards of the quickly advancing field of computing – the fundamentals still hold today. The book also features a catalogue of all the 23 GoF patterns, with cross references on related patterns. A word of warning though: the examples are in C++ and Smalltalk, so you might have to brush up on your skills in those first, or at least in C++.

For a more recent approach to the subject, I cannot recommend enough Head First Design Patterns by Freeman et al. (2004). The book is written in a witty, easily approachable style that is more geared to teaching the design patterns than merely cataloguing them. The examples are in Java, so they are easy to understand for a student in TAMK. After reading the book it became a lot easier for me to understand the more fact of the matter style of Gamma et al. that was too cumbersome for me at the beginning. That is why I really recommend this book as the next step on your journey towards learning about design patterns.

For the freshest insights on the matter, and especially designed for game programming, I suggest you check out Game Programming Patterns by Robert Nystrom (2009-2014). His book – free to read on the website – is definitely something not to be missed. Nystrom has the expertise of an industry veteran, having worked as a game developer at Electronic Arts for eight years. In his book, he revisits some of the old Gang of Four design patterns, and introduces many new design patterns, created especially to solve recurring problems

in game programming. Another word of warning: the examples are in C++ in this book as well, so quickly checking out the syntax might help.

Design patterns are an essential part of intelligent software design, useful equally for making games as for making any other software. Although many of the design patterns have stood the test of time, they are a constantly evolving part of software development. As programming languages evolve, so do design patterns. But one thing does not change: no matter what engine or language you choose to code your next amazing game with, you are bound to profit from the use of design patterns.

**REFERENCES**

Freeman, Er., Freeman, El., Sierra, K. & Bates, B. 2004. Head First Design Patterns. 1st printing. Sebastopol: O'Reilly.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. 44th printing. Massachusetts: Addison-Wesley.

Holzner, S. 2006. Design Patterns for Dummies. 1st printing. Indiana: Wiley Publishing, Inc.

Nystrom, R. 2009-2014. Game Programming Patterns. Read 14.2.2018. http://gameprogrammingpatterns.com/

Unity Manual, version 2018.1. 2018a. Event Functions. https://docs.unity3d.com/Manual/EventFunctions.html

Unity Manual, version 2018.1. 2018b. Execution Order of Event Functions. https://docs.unity3d.com/Manual/ExecutionOrder.html/