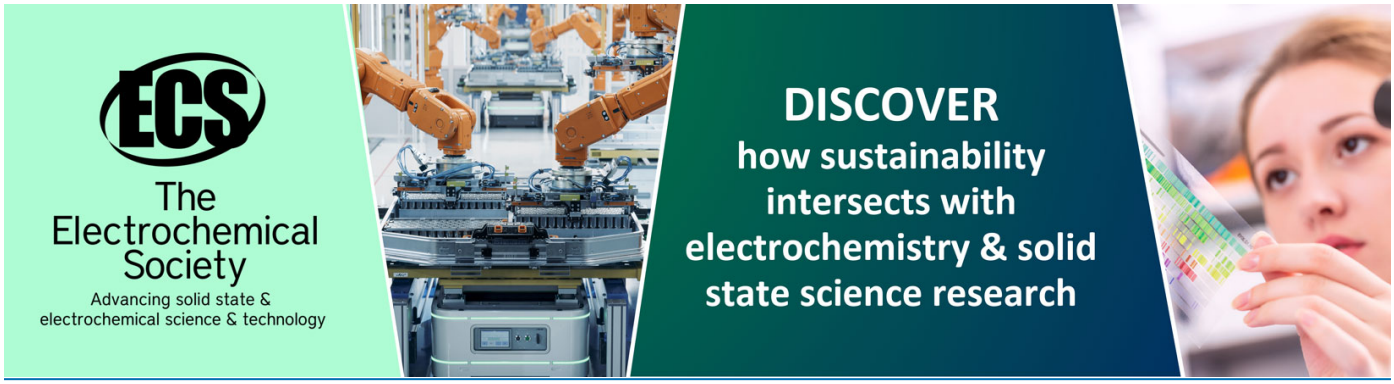
Application of Data-Oriented Design in Game Development



Journal of Physics: Conference Series **1694** (2020) 012035 doi:10.1088/1742-6596/1694/1/012035

Application of Data-Oriented Design in Game Development

K Fedoseev[1], N Askarbekuly[1], A E Uzbekova[1], M Mazzara[1]

[1]Innopolis Uninversity, Universitetskaya 1, Innopolis, Tatarstan, Russia

E-mail: k.fedoseev AT innopolis.ru, n.askarbekuly AT innopolis.university, e.uzbekova AT innopolis.university, m.mazzara AT innopolis.ru

Abstract. Game development is a big, human resource intensive industry. Object-Orientated Design (OOD) is the most popular software design paradigm in game development. However, it is not the optimal choice whet it comes to performance and resource utilization. To solve this problem, Data-Oriented Design (DOD) can serve as an alternative for resource-intensive games where performance is the highest priority. The latter approach is often perceived as something exclusive to highly qualified specialists. Thus, a question arises whether DOD can suit and be useful for smaller companies and projects, and what are the trade-offs of using the approach compared to more wide-spread object-orientation.

This paper examines the experience of a small game development team implementing two similar case study projects, one using OOD and another using DOD. After an overview of both implementations, the authors analyze and compare the two projects in regards to performance and maintainability. Recommendations are given on when each of the two design approaches should be applied and what are the corresponding skill sets for developers to use them.

1. Introduction

Data-Oriented Design (DOD) is gaining popularity in game development and is an alternative approach to the more traditional Object-Oriented Design (OOD).

DOD is a software-based approach aimed at efficiently utilizing the processor cache. The approach is to focus not on the relationships between the data, as in OOD, but on their location, aimed at separating and sorting the fields according to when they are accessed for reading or transformation.

The main advantage of DOD is that it allows you to design and develop game engines that efficiently use devices with a multi-core processor, due to the uniform distribution of the load on the processor across multiple threads, which is impossible when using an object-oriented paradigm. This, in turn, gives games created using DOD the more optimized resource utilization and performance that are so important in game development. In particular, games that are OODs typically use the same main thread for approximately 80% [[1]] of all CPU tasks, which is not very good at present when most devices have a multi-core CPU [[2]].

Despite its advantages and growing popularity, the use of DOD in game development has not been the subject of much scientific research.

1.1. Object-oriented design in games

As Nystrom R. describes, the most popular and default approach to software development in games is object-oriented design (OOD) [[3]]. A common reason for using OOD for games is that they are really easy to develop, maintain and find by developers. Thus, it is much cheaper and faster to develop. Thus, he is so popular in GameDev.

OOD's standard design patterns apply to adaptive games like any other software. The most commonly used templates in GameDev are Command, Observer, Prototype, Singleton, and State. Despite its popularity and usefulness in terms of the convenience of maintaining code, the above templates are not optimal when it comes to performance. Singleton is a good example of how OOD can be good in the context of readability and usability, but not when using memory at run time. Usually, this leads to a strong connection and a lot of clicks on links to objects. It is usually used to store links to other objects that contain data. To get the data, the program must jump 2-3-10 times before reaching it.

1.2. Data-oriented design in real-time systems and games

The main idea of DOD is to think about data and operations with it [[4]]. Of course, the OOD approach does the same successfully, but the DOD also focuses on data operations. Data locality and parallel computing are key features that significantly improve performance.

One of the main applications of DOD can be found in real-time systems because they always require stable performance with limited resources. [[5]] For example, we have a video recorder, it works on the recorded frame, which must be quickly processed and saved. And the operations are performed in real-time, that is, if he lost one frame due to the fact that he could not process it before another arrived, the video will be damaged, so stable performance is required. Of course, hardware improvements can reduce the likelihood of freezes, and when processing frames, you can use algorithms to skip errors. DOD systems in practice have much more stable performance than OOD systems because they have a lower chance of page skipping, which can lead to a freeze due to disk or memory latency due to the gap between memory and disk [[6]][[7]][[8]].

DOD is not commonly used in games, because, as mentioned earlier, it is believed to require a lot of effort from developers[[9]]. Game engines such as Unity provide frameworks like ECS [[10]] that provide developers with all the power of DOD. The ECS system is usually used in games with processor-intensive use, in which there is a lot of rendering of three-dimensional objects, physics, and any other calculations. Such projects are carried out by huge teams with professional qualifications. But in real life, for small teams, it is impossible to develop ECS from the very beginning, and usually, the result of their work is prototyping without any optimization. And then the project grows, starts to earn a lot of money, and the team also grows, then the developers are faced with the problem of optimizing the project [11].

2. Methodology, Case Study projects and Evaluation

To conduct the study, we have looked at the implementation of two hyper-casual games, which share the same game mechanics, location scene, and physics, but have different content.

The first case-study project was implemented using principles of Object-Oriented Design (OOD). The project's name is Droppy Kick. It is a simple hyper-casual game with a basic game mechanic and a lot of physics, animation and mathematics. Such traits are common for hyper-casual games.

The architecture is based on component inheritance, dependency injection, and other high-performance OOD design solutions. The game uses standard unity and physics rendering libraries. All data is stored in the components themselves.

The second case-study project is Justing sunset, which was implemented using principles of Data-Oriented Design (DOD). It is almost an identical project, that shares the same location scene and physics. The main difference is the software design.

This game was implemented using the LeoECS framework. The game consists of atomic systems that do something very small, for example, AddSpeedSystem.cs, which only adds speed to any object for which it was specified. No matter what the object is. In addition, data is stored in separate structures that are collected in SOA or AOS for filtering by a specific system. A particular system has a certain amount of data that needs to be filtered. The game is displayed according to the user system.

Both the OOD project and its DOD analog will be compared from the perspective of three evaluation criteria: performance, maintainability, and required programming skills, described below.

2.1. Performance evaluation

In order to evaluate the difference in performance evaluation comes down to the processor. Some method is needed to measure how well the processor works in each solution. Given that data-oriented principles are data-oriented for better localization, it is important to evaluate the time that the processor spends on the same type of operation. If a data-oriented implementation is really better, then it should be able to perform operations of the same type with fewer processor cycles. For this reason, the frame rate will be selected as the main evaluation criterion for this thesis. The frame rate is based on elapsed time, which will result in an indication of elapsed time for the processor. In addition, the frame rate is an important factor for video games, as it is also an indicator of image fidelity.

2.1.1. Frame Rate The frame rate is used to describe the number of consecutive images appearing in one second. The frame rate is expressed in frames per second (fps)[[12]], where the frame is the image. A higher frame rate means that more frames can be displayed every second, providing smoother animations and therefore smoother gameplay. Frame rate is also an indicator of how fast the processor is. Frame rate can be calculated as:

$$fps = 1/framedt \tag{1}$$

where $framedt$ is the time elapsed between two frames.

Both minimum and average frame rates will be used. To get an accurate idea based on the frame rate, the compared implementations should have the same the game scene and mechanics. The only real difference in implementation should be only the fact that one is object-oriented and the other is data-oriented. In general, factors affecting performance must be consistent to provide an accurate case for comparison.

2.1.2. CPU usage time Despite the fact that the frame rate is the main indicator of productivity, it may not always be a sufficient idea of how a data-oriented project works in comparison with an object-oriented one. Some factors, such as rendering objects on the GPU[12], can adversely affect performance but are not related to software implementation. Therefore, such factors should be excluded, and instead, close attention should be paid to the performance costs directly related to software development. For this reason, a detailed analysis of the CPU usage time [[7]] will be carried out to better evaluate various solutions. This only applies to solutions run by Unity, as it supports the diagnostic tool, Unity Profiler. The Profiler gives a complete picture of the processor time for various operations of each frame in the process. Our goal is to understand the following processor usage characteristics:

- Distribution between threads/kernels: smoother, better ideal distribution
- Total Resource Usage: Less is Better

2.1.3. Distribution among threads Key indicators are the minimum and maximum percentage of CPU usage, as well as the average number of cores used, excluding inactive cores and threads [[8]] ($< 5\%$ of the maximum).

It is assumed that on different processors these will be different values, for example, 4 cores with a utilization factor of 0.2-5-5-10. Min = 5 %, max = 10 %, then 0.2 % less than 5 % of max ($< 0.5\%$), it seems that the distribution is good, but if the processors are not powerful, then we have a restriction on the use of 3 cores, which means that the distribution is not suitable for multi-core processors. In addition, twice as much load on the main core, even if it is only 10%, but with such a distribution it will not work on the processor 20 times weaker, but theoretically such a processor can hold it if the distribution is 5-5-5. -5 and on an abstractly 20 times weaker processor it will be 100-100-100-100. The idea is that such indicators are normalized for any processor power and measure only the distribution between threads/cores [[7]].

2.1.4. Overall CPU utilization This indicator is not relative but is highly dependent on the machine, so measurements should be carried out at the same level. The load can be calculated as the sum of the use of all cores. For example, the situation: 0.2-5-5-10. The sum is 20 because the inactive core can be removed from the estimates according to the same rule as in the Flow distribution indicator. That says that in one design method such performance, in the second another and the percentage difference between them is our final result.

2.1.5. Memory usage Allocated memory is also an important indicator for checking game performance. Memory has two important parameters for performance: size and number of skipped pages [[8]]. The first can simply be written with Unity Profiler, the second with internal OS tools. Of course, less memory allocation and page skipping correspond to better performance.

2.2. Maintainability evaluation

Maintainability usually is one the most important quality attribute of the SW [[13]] in particular for games too. The games, which are earning a lot of money, always is 3-5 years projects at least, so that maintaining simplicity of them can lead us to significant optimization of expenses.

2.2.1. Maintainability Index Maintainability Index is a programmatic measure that measures how much source code is supported (easy to maintain and modify). The Maintainability Index is calculated as a factor formula consisting of lines of code, cyclomatic complexity and the volume of Halstead. It is used in several automated metric toolkits, including the Microsoft Visual Studio 2010 development environment, which uses a derivative with an offset scale (from 0 to 100).

Calculation First, we need to measure the following metrics from the source code. The original formula:

$$MI = 171 - 5.2 * ln(V) - 0.23 * (G) - 16.2 * ln(LOC) \quad (2)$$

where

V — Halstead Volume [[14]] ;

G — Cyclomatic Complexity [[15]] ;

CM — percent of lines of Comment (optional). [[16]]

The derivative used by SEI[ref] is calculated as follows:

$$MI = 171 - 5.2 * log2(V) - 0.23 * G - 16.2 * log2(LOC) + 50 * sin(sqrt(2.4 * CM)) \quad (3)$$

The derivative used by Microsoft Visual Studio (since v2008) is calculated as follows:

$$MI = MAX(0, (171 - 5.2 * ln(V) - 0.23 * (G) - 16.2 * ln(LOC)) * 100/171) \quad (4)$$

In all derivatives of the formula, the most major factor in MI is Lines Of Code, which effectiveness have been subjected to debate.

This indicator has good reputation in OOD SW, but all such characteristics can be found in DOD SW, so that can be used in estimations.

2.2.2. Re-usability of code The main indicator is communication in the system, as well as the responsibilities of each class. If the connection is low, perhaps SW is more reusable, but if there are still several responsibilities, then the exact code cannot be used, it should be changed before use [[17]]. This is a fairly subjective indicator that consists of the percentage of reusable modules for all modules. To be more subjective, all developers from the project must be involved.

2.3. Required programmer level evaluation

The third required part to get a complete picture of the meaning of DOD is people in the process. Developers are one of the most expensive and scarce resources. So, the transition from one design concept to another should be carried out with a full understanding of the consequences.

2.3.1. Skill-set determining A complete set of skills of programmers with the required level in certain technologies and basic knowledge. For example, if DOD requires excellent knowledge of the OS, then this should be indicated in the list of skills with a level of "excellent". Skills tables will be discussed in more detail in section 3.5.1. The list itself should be used to understand how much time the developer should spend on training, that is, a minimum base. For comparison, for different approaches, you can take the total amount of time to understand the level of the minimum degree and the quality of education.

2.3.2. Learning curve This parameter(learning curve) is closely related to the total amount of time for understanding, plus future resources on the concept of learning. This can be compared using Skill-set data and developer feedback. It should also be assumed for different levels of developers: interns, juniors, middle peasants [[18]].

3. Results and Discussion

Analysing the projects' implementation the research team has collected the following results in regards to the three comparison criteria: Performance, Maintainability and Required programming skill.

3.1. Performance

The performance was collected from iPhone X with 6 core Apple A11 Bionic processor and 3GB RAM. For collecting performance data there was using Unity Profiler and for comparing that data and getting results as accurate as possible Xcode Instruments were included in process.

Allocating memory, which is also a good performance indicator for understanding how many objects were created, over the active 20 seconds: 128.8 MB.

General processor utilization using ECS is pretty good (Table 1). Moreover, as we see, the load distribution is smoother than with OOD. The difference between the minimum and maximum processor load is only 17%. Moreover, its minimum value is only 1.55 times less than the average, so we can conclude that there are no jumps in the load.

For FPS (Table 1), the ECS approach gives better results and a smoother situation. Please note that 60 frames per second is a hardware limitation on iOS devices. The average FPS is at the top level and does not freeze.

	Measurement Category	**Droppy Kick**	**Justing Sunset**
3***CPU**	Minimum CPU utilization	16%	16.2%
	Average CPU utilization	33.26%	25.11%
	Maximum CPU utilization	95%	42.11%
3***FPS**	Minimum FPS utilization	32	49
	Average FPS utilization	52.9	58.3
	Maximum CPU utilization	60	60

Table 1: Droppy Kick CPU and FPS utilization.

3.2. Maintainability

This section describes the most important aspects of maintenance, such as the code maintenance index, the reuse of this code in new projects, and the project development time. The Last aspect affects the overall performance of the programmer.

Visual Studio Code metrics were used for static code analysis. The table 2 shows the comparison of metrics for the two projects. As one can see the case of OOD project has overall better quality when it comes to maintainability.

	Droppy Kick (OOD)	Justing Sunset (DOD)
Maintainability index	82	82
Cyclomatic complexity	713	730
Inheritance depth	7	6
Class interdependence	206	399
LOC	4640	5920

Table 2: Maintainability metrics comparison.

For the DOD case, the quality of the code is relatively higher. The number of class interdependencies is twice as much, due to the fact that each component uses the System, and is something that can be expected. In addition, the code base is about 25% larger than for OOD, which can also be counted as a downside of using DOD.

However, the maintainability and cyclomatic complexity are similar. The depth of inheritance in the DOD is, of course, less, due to its being used only in particular cases and not as commonly as in OOD.

3.3. Reused code

Reusing code is a very important thing that reduces budget and time[[17]][[13]]. So, in the two compared projects, frameworks, reusable code was used. The first was the 6th iterative project in a team with such a structure. And the second was the first iteration with LeoECS, an open-source environment that goes through many projects.

For the first project, only the structure itself was used. All other code was written from scratch and was not used after. Of course, some parts, such as moving behavior, can be transferred to another project, but there are many dependencies that take longer to resolve than recording from scratch, as we saw in the code diagram for Droppy Kick **??**.

For the second project, the results were much better. About 75-80% of the code was reused in the next project, which can be considered the same. This effect was achieved by the atomic design of the system. Most systems, such as ballistic movement, rotation system are really the same for any project and require only parameter settings. In addition, another advantage is that all the code is pure C# code, which is used without dependencies even on LeoECS (the creator

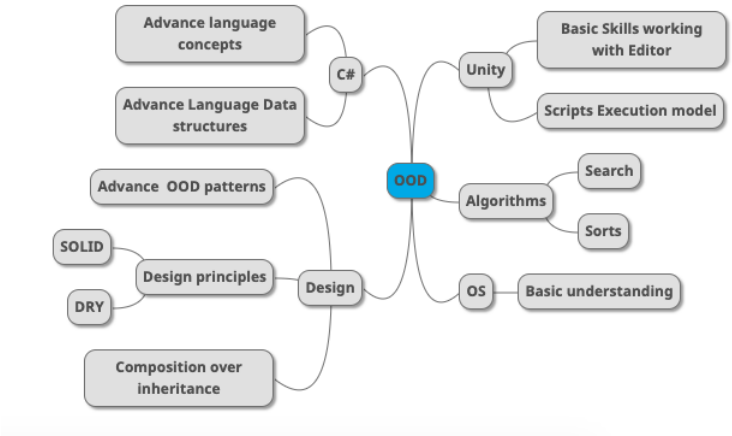


Figure 1: Table of skills with deep categories for OOD

creates and adds all systems). The code can be moved from one project to another without any changes. This greatly accelerated the development of the next project from 2.5 to 1.5 weeks.

3.4. Game Development Time Estimation

About 2.5 weeks were spent on both projects. This amount of time was spent only on development. The projects were completed, but some fixes appeared.

It was not difficult to fix the OOD project, but the architecture was not so good, because the project was created within 2.5 weeks, and only the structure and its flexibility helped to make it faster. In general, corrections took another 1.5 weeks.

The fixes in the DOD project were also simple, but there were several more. The architecture of any DOD system is fairly atomic, and corrections were made a little faster. In general, corrections were also made within 1.5 additional weeks.

In the process of fixing the first project and the second were fixed by programmers who were associated with the development, but after some time the programmer left these projects. As a result, a third-party programmer fixed the OOD project a little easier than DOD under the same conditions.

3.5. Required programmer level

The previous indicators are very useful in terms of final product and its maintaining, but not about programmers and the requirements for them. And in this section will be covered all aspects about people and how they feel the process.

3.5.1. Used skill-set Games that are used in research are hyper-casual. These are simple games and in most cases contain only the main or main gameplay. There is no complicated meta mechanics. Thus, the skill set is not wide and requires only basic things.

Everything requires a good knowledge of Unity and C#. Due to the fact that these are mobile games, a programmer should know what are the limitations and nuances of mobile platforms such as Android and IOS[[19]].

For OOD, you need to understand Object-Oriented Programming (OOP), know basic patterns, dependency injection, SOLID principles, and a component approach to inheritance. The list also includes some other complex skills, such as a degree in computer science, but in practice, this gives a little impetus because everything can be found on Google.

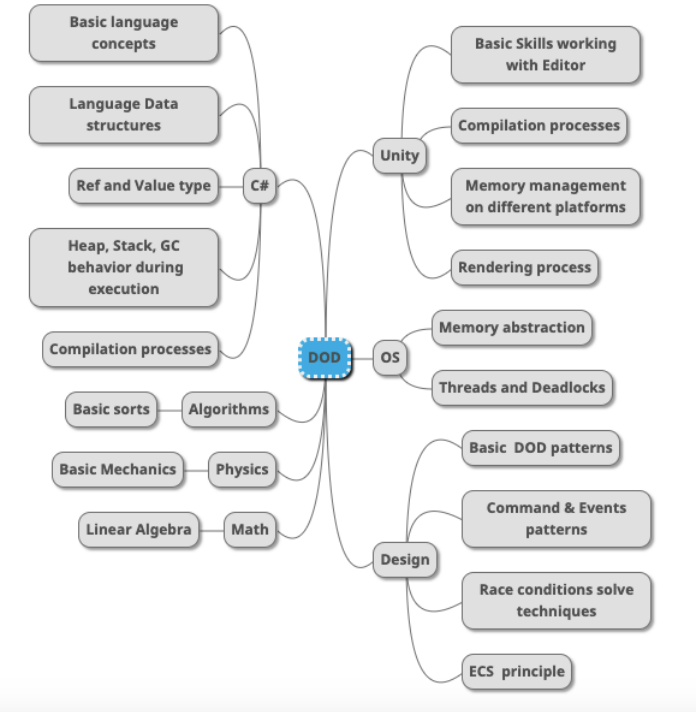


Figure 2: Table of skills with deep categories for DOD

DOD requires a degree in computer science because all problems are new and cannot be found as questions about OOD. It is also necessary to have an idea of the problem of the state of the race conditions, deadlocks and how to solve it. Knowledge of design patterns and templates is not required, since there are no complex templates in DOD, and their number is 10 times less than in OOD.

The skillset for DOD is smaller if it is assumed that having a degree is a normal situation for programmers. But in fact, the degree can be poor and will not have any impact on working with DOD projects. The rest of OOD has a minimal set of skills because there are a lot of skills in computer science that require much more time than just learning patterns.

3.5.2. Junior and intern performance in both The study was conducted on two different juniors[[18]]. The first was without any base in computer science and knew only about unity[[20]] and C# [[21])]. The second has a solid base in computer science: operating systems, computer architecture, compilers, algorithms, and so on. Both have the same level of knowledge of Unity (or game engines in general [[22]]) and C#.

The first has good performance as a junior in the OOD project (Droppy Kick). Most of the assignments that were provided by management were completed. The density of questions was fine for the younger. The second has a slightly better result for the OOD project. The density of the question was half as much due to some experience working with the component on the principle of inheritance.

For DOD, the situation was completely different. Both had no experience in such projects. For the first performance, it was terrible. The density of questions was higher, and the moderator spent more time on questions than if the task had been written by himself. The problem was the lack of computer knowledge. Due to the fact that most of the questions concerned understanding how to write the correct data structures, another part of the questions concerned errors and understanding how the system works. For the second junior, the situation was more positive.

The density of questions was less than in the OOD project since the architecture was more atomic, and the logic was less cohesive and the locality of logic in one system. For the second, it was much better to work with DOD because of an understanding of how the system works at a low level in memory. It was also easier for him to work with DOD due to the lack of frameworks like GameCore, which required additional work experience.

As a result, both juniors have different results, this, of course, is a subjective conclusion. However, DOD, like any other technology, required basic knowledge in the field of computer science, which greatly accelerates the learning process for the second junior.

3.5.3. Learning curve As in the previous section, the data obtained from two juniors were used in the evaluation. Both had no experience in the GameCore and LeoECS Framework for OOD and DOD, respectively. But it is assumed that GameCore is an OOD framework and its structure is very familiar to juniors since OOD is used in most programming courses.

For the OOD GameCore platform, its use was required about a week before the project and two per project. It was a little different for the first and second juniors, but they can be neglected. The density of questions about the structure has been declining since today. Last week, it was one or two questions about the framework only. The learning outcomes were such that both can write games on their own.

As for the LeoECS DOD platform, the juniors had no experience, and he did not even hear about it. It was a game for both. The first did not complete the training after 3 weeks of training and left the post of developer. The second developer was trained in 2 weeks. The first week was spent reading and viewing training materials about DOD and ECS. The second was carried out in practice.

4. Conclusion

Research has shown that Object-Oriented Design (OOD) and Data-Oriented Design (DOD) both have their advantages and disadvantages in regards to performance, maintainability, and entry-barrier level. Based on the results, the following are the aspects one should consider when choosing a design paradigm, technology stack, and team formation:

- OOD is much easier for juniors and maybe the only way for those who do not know computer science.
- DOD requires a knowledge base in computer science, in particular, operational systems and how memory works.
- DOD has better performance. This is confirmed in our study, and the related literature.
- OOD and DOD in the case-study projects have similar maintainability metrics. However, in practice, the team's experience indicates that it took more time to understand the code structure and flow in the case of DOD for engineers joining the project at later stages.
- The effectiveness of training interns or juniors for each design approach depends, in particular, on the experience of programmers, whether a programmer has a knowledge of computer science fundamentals. If this is the case, then DOD can be simpler than OOD, due to fewer templates, general structure, and simple principles.
- DOD facilitates reuse through its simple and atomic structure of the project modules.
- The atomic structure and simplicity of code modules in DOD make it easy to distribute tasks between developers. Based on this, the ECS community [[23]] argues that DOD projects are easier to develop and maintain.

4.1. Limitations and Future Work

Amongst the limitations of this study, the main one is the number of case-projects (only two). Moreover, both of them were of relatively small size. To further verify the conclusions, there is a need for more case-studies involving projects and development teams of different size. Another possible limitation comes from the fact that the research development team had previous experience with OOD and were absolute beginners in DOD, which makes the results more applicable to development teams with a similar skill-set and experience. Lastly, the reusability of DOD source modules across projects can be a topic of interest to game developers and should be further research, especially in the context of mass product line development.

References

[1] tpi2007 2012 Reference on forum post-article about cpu usage in games URL `https://www.overclock.net/forum/78-pc-gaming/1229915-how-cpu-gpu-usage-along-fps-game.html`
[2] Gough C 2020 Reference on the amount of cpus in pc distribution URL `https://www.statista.com/statistics/265031/number-of-cpus-per-pc-on-the-online-gaming-platform-steam/`
[3] Nystrom R 2014 *Game programming patterns* (Genever Benning)
[4] Fabian R 2018 *Data-oriented design* (R. Fabian)
[5] Joshi R 2007 *whitepaper, Aug*
[6] Carvalho C 2002 *Proc. of IEEE International Conference on Control and Automation*
[7] Andrew S Tanenbaum T A 2011 *Computer Architecture*
[8] Tanenbaum A S 2014 *Operating Systems*
[9] Faryabi W 2018 *Data-oriented Design approach for processor intensive games* Ph.D. thesis NTNU
[10] Unity The guide for ecs system by unity URL `https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/index.html`
[11] Hu P and Zhu K 2014 *J Chem Pharm Res* **6** 785–791
[12] Engel W 2013 *GPU Pro 4: Advanced Rendering Techniques*
[13] Naik K and Tripathy P 2011 *Software testing and quality assurance: theory and practice* (John Wiley & Sons)
[14] ProjectCodemeter About gl halstead URL `http://www.projectcodemeter.com/cost_estimation/help/GL_halstead.htm`
[15] ProjectCodemeter About cyclomatic comlexity URL `http://www.projectcodemeter.com/cost_estimation/help/GL_cyclomatic.htm`
[16] ProjectCodemeter About sloc URL `http://www.projectcodemeter.com/cost_estimation/help/GL_sloc.htm`
[17] Anguswamy R and Frakes W B 2012 *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (IEEE) pp 161–164
[18] Soft A 2018 Software engineer qualification levels: Junior, middle, and senior URL `https://www.altexsoft.com/blog/business/software-engineer-qualification-levels-junior-middle-and-senior/`
[19] Sheikh A A, Ganai P T, Malik N A and Dar K A 2013 *The SIJ Transactions on Computer Science Engineering & its Applications (CSEA)* **1** 141–148
[20] Hocking J 2015 *Unity in action: Multiplatform game development in C# with Unity 5* (Manning Publ.)
[21] Joseph Albahari B A 2017 *C# 7.0 in a Nutshell: The Definitive Reference* (o'reilly)
[22] Gregory J 2018 *Game engine architecture* (crc Press)
[23] ECSCommunity Reference on telegram community channel about ecs URL `https://t.me/ecschat/417`