Comparisons on Game Development with Unreal Engine 4 using Object-Oriented Paradigm (OOP in C++) versus Notification Oriented Paradigm

Felipe dos Santos Neves
Graduate student at *PPGCA*[1] - *DAINF*[2] - *UTFPR*[3]
Report in article form to the course Programming Paradigms
(1st trimester of 2020) at *CPGEI*[4] - *UTFPR*[3] – Prof. Jean Marcelo Simão.
Curitiba, Paraná, Brazil
fneves@alunos.utfpr.edu.br

***Abstract*—This paper draws comparisons on the usage of the novel Notification Oriented Paradigm for game development with the traditional approach using Object-Oriented C++. For this purpose, a simple game is developed using the Unreal Engine 4, implementing the same game logic with both the Object-Oriented Paradigm and Notification Oriented Paradigm. A formal software development cycle, from requirements to design and implementation is used to ensure that both implementations have the same final result, enabling the comparisons of the time spent in development and number of lines of code necessary for each implementation. The software developed using the Notification Oriented Paradigm generated more lines of code, however less time was spent during development, showing that in this case using the Notification Oriented Paradigm was beneficial to reduce the software development time.**

***Index Terms*—Game Development Methodologies, Unreal Engine 4, NOP Versus OOP Experimentation**

I. Introduction

The development of electronic video games dates back to the early 1950s, where computers first began to be available to civilians, and the first graphical video game appeared in 1958, with the vastly know game called Pong, which was entirely hardware based [1]. Since then game Development has grown to a multibillion industry, reaching \$120.1 billion revenue in 2019 [2].

Game Development is a widely multidisciplinary field, involving field such as arts, design, business and software development. And as digital games grow in size and complexity each year the industry pushes forward the fields of both hardware and software development.

In the early days of computer assisted game development most games had software build from the ground up specifically for that purpose, but in the core of any modern game is a Game Engine, which is the core software responsible for handling the doing the rendering, calculating physics, lighting, playing sounds and much more. These engines provide the tools for the developers to bring their design to work without with much more ease than having to rebuild everything for each game. Of course, some cutting edge development still requires the development of a new engine from time to time to better fit the requirements of the game that is being developed [3].

Given the nature of such applications, as computer games can be roughly described as a group of objects that interact with each other and that have to be rendered on a screen on a fixed interval, both the games and the engines themselves are almost always implemented in the Object-Oriented paradigm. The engine used in this study is the Unreal Engine version 4, which has its API available in C++. This is one of the most used tools in professional game development, and shows how important the Object-Oriented paradigm is in the game industry.

Given this context this paper's goal is to show how the novel Notification Oriented paradigm can be used as a great asset to extend toolset developers have, where the Object-Oriented paradigm shows its limitations. In its current implementation there is a version of the Notification Oriented Paradigm available as a C++ library [4], which makes it suitable and easy to interface with the Unreal Engine API, providing a convenient way to study the viability of the application of the Notification Oriented Paradigm for game development.

Given the objective of this paper to compare the two approaches to the development of a game we establish a set of requirements that should be present in both implementations. Also, after developing for the initial requirements there will be a second set of modified requirements, where some of the requirements are modified and new ones are added, for the purpose of evaluating how easy it would be to modify the software after it is already done for the initial requirements.

II. Unreal Engine

The Unreal Engine was chosen for this project due to it being one of the most used engines in commercial game Development, as illustrated in Figure 1, from data gathered from Steam [5], the major games storefront for PC. Besides

[1]Programa de Pós-Graduação em Computação Aplicada
[2]Departamento Acadêmico de Informática
[3]Universidade Tecnológica Federal do Paraná
[4]Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

that, it also has its API in C++, making it easy to interface with the NOP C++ Framework 2.0 [4], which could be compiled as a library and loaded in the Unreal Engine project.

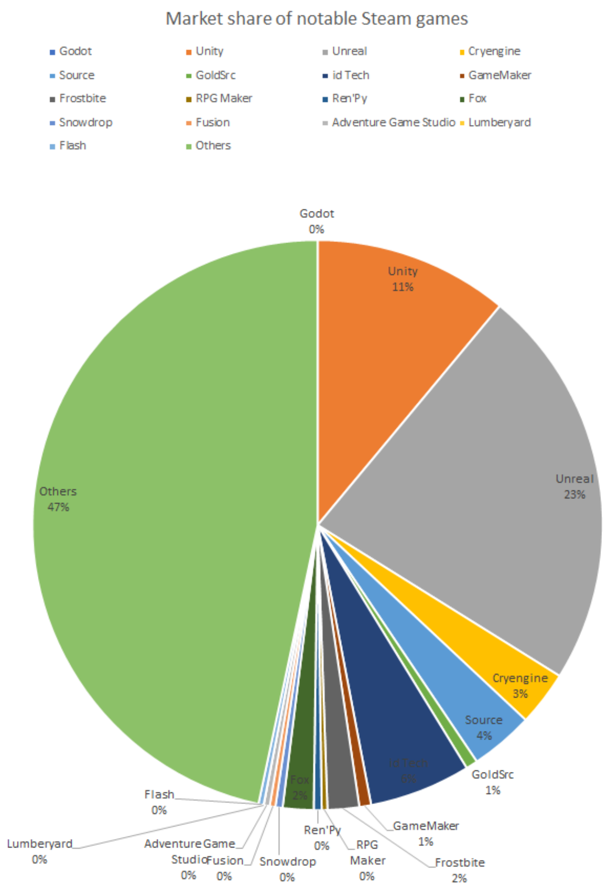

Fig. 1: Game engines market share from Steam [5]

While most of the game's implementations can be done in C++, the Unreal Engine also has its own visual programming language called *Blueprints*, that allows the users to use the API through visual blocks and connections. This is an excellent way to speed up the development, as it is very easy and intuitive, and also enables users without programming language to use the engine. There is extensive documentation provided for both the C++ and *Blueprints* and the Unreal Engine website [6], in this paper only the core concepts required in order to provide a basic understanding of the implementations of this project.

While being easier to use the *Blueprints* has a very robust interface to call methods from the native C++ API. The only downside is that it may be slower, and using the native API can result in a faster implementation, depending on how the code is written, and also allows for a more advanced use. For the most part for simpler games they can be made entirely using *Blueprints*, or used in complement with core implementations in C++, which the engine provides the tools to export to be used with *Blueprints*. This means that you can expand write your own classes in C++ and use the available tools from the engine, through some clever use of macros, and convert it to a pure *Blueprints* object.

A. *Core Concepts*

One of the core concepts of the Unreal Engine is that every object is derived from the base class UObject, which contains the base structure for every object that you want to be handled by the engine. This is because the engine expects to handle the two core loops for every object BeginPlay and Tick. Also, every object that is created is inserted into a list of all objects in the game world, so that the engine can handle all of them. The engine handles both graphical and non-graphical objects as well.

The BeginPlay function relates to a usual C++ class constructor. The BeginPlay function is called every time a instance of the class is invoked into the game world. It is very similar to what a constructor function would do in a normal C++ program, but the difference in this case is that the engine uses the constructor to pre-load assets, such as graphics and sounds, into memory. This is done to provide better memory efficiency and speed, as you cannot have heavy objects being loaded into memory during gameplay, so the engine pre-loads all the classes before the level starts, that would be what happens during a loading phase. And then there is the BeginPlay function that is only called when the object is effectively spawned into the world.

Another very important function is the Tick function. This function is actually optional, as it can be very resource intensive, as it is called for every tick of the engine, or for every frame that is drawn on the render. Objects that do not need to run every tick can disable this feature, but it is usually enabled for most objects that have any sort of interaction.

As it becomes evident from these concepts, for graphical objects in general, the code must run loops that update on every frame, what could contradict the principles of the Notification Oriented Paradigm, that would avoid those loops altogether. But as the game engine abstracts most of these graphical elements and the Notification Oriented Paradigm can be used, not to substitute the traditional Object Oriented approach, but to complement and improve the design of the other non-graphical related aspects of the game code, such as the enemy behavior and game systems and rules.

III. Notification Oriented Paradigm

A. *Objectives*

The main objectives of the Notification Oriented Paradigm are to solve some of the many problems present in the Imperative and Declarative Paradigms [7] [8]. The first problem that can be mentioned is the redundancy present on the Imperative Paradigm, since it is strongly dependent on loops (such as if, while and for loops) for evaluating variable states during the program execution, but most of these variables will never change during each interaction, resulting in a code that will run multiple times without achieving any different results [7].

Besides that one of the other main problems is distribution, since the same variables can be evaluated in multiple different sections of the code, and the passivity of its elements bring a close dependency to the data and commands within a program, making it harder to split into multiple execution units. In scenarios where you cannot control the execution order of the parts of your code it can cause many problems [8].

As such is the case of Unreal Engine objects, where the engine is responsible for choosing which elements to evaluate first, the developer has no input on that matter, and that can bring problems where there is a close dependency on variables that may change their states before a dependent state has its chance to be evaluated.

B. NOP Entities

This section presents in greater detail the structures that compose the Notification Oriented Paradigm implementation.

1) Fact Base Element: The Figure 2 describes the mains entity of the Notification Oriented Paradigm, the Fact Base Element (FBE). It is a class used to describe the states and services of our objects. An FBE contains attributes and methods, which are collaborative objects. The difference between an FBE and a traditional Object Oriented class is that the attributes and methods are not simple passive entities [8].

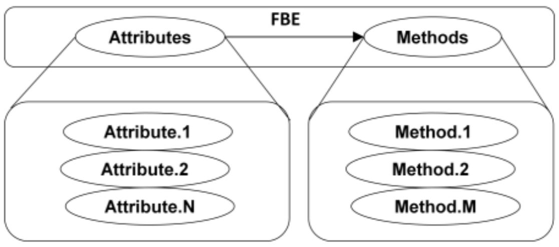

Fig. 2: FBE Structure
[8]

2) Attribute: The attributes store the values that represents the current states of the properties of given FBE. It is different from a traditional member variable from the point that it is able to notify its relevant premises when its state changes. The ability to notify the premises, that can be members of the same object or be part of a different entity is what gives the advantages in respect to parallel execution and decoupling.

3) Premises: The premises represent the evaluation of the state of a given attribute. When the attribute notifies the premises, it is then evaluated against a given value or against another attribute. The premises can be evaluated as true or false, and when its value changes it notifies the relevant conditions.

4) Condition: The condition is composed from premises, that are evaluated with either a conjunction or disjunction operation. It notifies the relevant rules when it is approved.

5) Rule: A rule, as illustrated in Figure 3, is composed by a condition and an action, in a causal manner, when the rule's condition becomes true it then the rule becomes approved for execution, executing its action. Indirectly, the rule is composed by premises, and instigations.

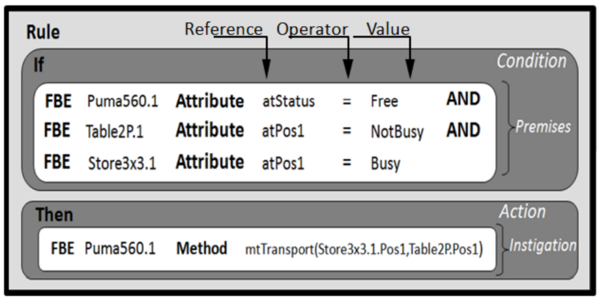
Rule
Reference
Operator
Value
If
Condition
FBE Puma560.1 Attribute atStatus = Free AND
FBE Table2P.1 Attribute atPos1 = NotBusy AND
FBE Store3x3.1 Attribute atPos1 = Busy
Premises
Then
Action
FBE Puma560.1 Method mtTransport(Store3x3.1.Pos1,Table2P.Pos1)
Instigation

Fig. 3: Rule Structure
[8]

6) Action: The action may contain multiple instigations. Its activated when the rule is approved. It is responsible for calling the connected instigations.

7) Instigation: The responsibility of the instigation is to call the methods of the objects, passing their appropriate parameters, when required. These methods can either be NOP attribute changes or also regular C++ function calls. By allowing the call of the regular code functions it gives the flexibility to integrate with other frameworks such as the Unreal Engine API.

The chain of events from the attribute changes to the method execution as described in the Section III-B is also illustrated in the Figure 4.

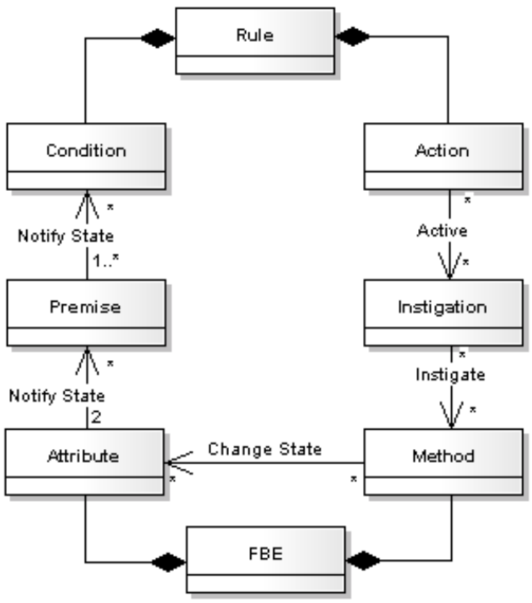

Fig. 4: NOP Notification Mechanism Diagram
[8]

IV. Programming Concepts

As defined in by Peter Van Roy [9], programming paradigms are composed of smaller conceptual units, called programming concepts. He defines 4 concepts that he considers the most important.

1) Record: A data structure, as a group of references that can be accessed through indexes to each item. Is ins present in the PON C++ framework as inherited from C++. It can be seen in the FBE structure, which is composed of other structures such as attributes and methods.

2) Lexically Scoped Closures: In the PON C++ framework, as all the entities described in Section III-B are implemented as classes from C++ we inherit this concept.

3) Independence (Concurrency): A program can be constructed as independent parts. When two parts do not interact, they are concurrent, as opposed to when the order of execution is defined and they are sequential. All the PON entities are by definition independent and support parallelism, as their behavior is determined by the notification mechanism, which supports concurrency. In this project it is very clear to observe this behavior as all the objects constructed in the Unreal Engine act as separate and independent threads therefore are executed concurrently.

4) Named State: Is the capacity of storing information in a computational element with a name, so that it can be accessed and modified through the code. It is very important in the PON framework so that attributes, premises, conditions and rules can be defined, and the instances of these elements can be shared through references to their names.

These concepts where later analyzed and classified in relation to the Notification Oriented Paradigm [10]. And these concepts are then validated and discussed bellow derived from the observations applying the Notification Oriented Paradigm through the C++ Framework in this project in Table I.

TABLE I: Programming Concepts in the NOP

Concept	**Present**
Record	Yes
Lexically Scoped Closures	Yes
Independence (Concurrency)	Yes
Named State	Yes

V. Implementation

Regarding the design of the game itself the was to reduce the complexity related to the graphics so that it would be possible to focus more on the software development itself. For that purpose the game was designed as a top-down shooter, which is composed of a player and enemies that are able to move in a two-dimensional environment. This is a appropriate approach for this study as it gives two degrees of freedom for the movement of the entities while still remaining fairly easy to implement when compared to full three dimensional environment. The graphics themselves are also implement just as necessary to provide a minimum viable demo, and to give suitable player feedback so that the game can be enjoyed.

As for the NOP implementation specifics, it is still constrained to the Object-Oriented approach that comes from the engine for the basic objects structure, but these objects are now also Fact-Based Entities. To comply with the NOP concepts, the basic iteration loops of the Unreal Engine are still used, but variable evaluation and function calls during these loops are avoided, begin only used for the attribution of updated values to the FBE's attributes.

In Figure 5 is shown a screenshot of the implemented game, with the player in the center, surrounded by enemies with behavior following the game logic implemented using the NOP.

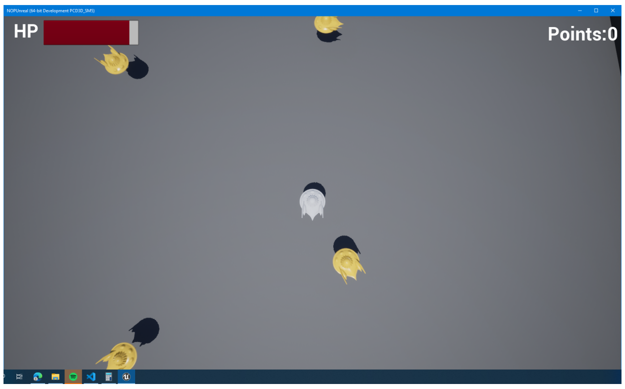

Fig. 5: In-Game screenshot

A. Requirements Specification

Given the fact that the two implementations have the same objective it was clear that writing a good set of requirements to guide both of them. Having a strict set of Functional and Non-Functional Requirements allows us to ensure that both implementations would have the same final functionality. These full set of requirements is described in Table II and III.

The initial set of requirements in Table III was designed to achieve a minimum viable product, a simple but functional game. We do not enter in much detail about the graphical aspects of the project, such as models, textures, sounds and effects as they are outside of the scope of the study.

As one of our objectives was to evaluate how both implementations could be modified a second set of modified requirements was defined with the objective to improve upon the original game design and is presented in Table IV. With these modifications it is possible to evaluate how easy is to modify an already finished code with the different approaches.

One of the benefits of developing with NOP based on functional requirements is how easy it is to relate the requirements with the code, due to how the program is structured, it more clearly reproduces the rules of a requirement into the code, making it easier to understand and to develop. This relationship between rules and requirements is extremely beneficial to the system, as it facilitates and encourages better traceability of the requirements during the whole product life cycle [11].

B. Class Diagram

The classes were designed to respect a similar hierarchy in both implementation, as well as reutilizing elements that

TABLE II: Non-Functional Requirements

Requirement	Description
NFR-1	The game shall be implemented using C++, the NOP C+++ Framework 2.0 and blueprints with Unreal Engine 4
NFR-2	All elements shall be implemented using both only C++ and C++ with the NOP Framework while keeping the same functionality
NFR-3	The game shall be be composed of a single level that can be played with both implementations

did not change in both implementations, such the classes for the abilities, derived from UAbility. The Object-Oriented only design is presented in Figure 6 and the NOP design in Figure 7. The only difference in both class diagrams is that in the NOP design the Player and Enemy classes derive from both AActor class and the FBE class and the names are different to separate both implementations, as they run simultaneously in the same application. The similar structure also allows to reutilize some of the code used to implement basic functionality, such as controlling the player, moving the objects and using weapons, only requiring minor modifications between each implementation and thus keeping both codes as similar as possible, and this is important to be able to evaluate the differences in the amount of lines of code as well.

C. Rules Diagram

For the NOP implementation there are a total of 8 classes that are derived from the FBE class, meaning that they have a set of rules in their implementation, and it is present below the textual description of all the rules and their conditions. These rules were designed in order to achieve the requirements presented in the section V-A. The rules are described in tables V to XXI.

One of the advantages of a software developed using NOP is that it can become very easy to trace the requirements into the code. As, ideally, all of the logic must be contained within the rules are linked directly to a requirement and the code where it is implemented, via the rules, premises, attributes and methods involved. In Table XXV it is shown the link between the requirements and the rules used for their implementation. Some requirements may not link to any rule as they were not implemented using rules, an therefore these requirements become harder to trace in the code. It can be observed that the requirements that describe the enemy and generic game behavior are the ones that have a rule, because these are the features that were more appropriate to be implemented with rules.

TABLE III: Initial Functional Requirements

Requirement	Description
FR-1	The player shall be able to choose from the C++ and NOP versions in the menu
FR-2	The level shall be composed of a player versus enemies inside a closed arena
FR-3	All the objects shall be able to move in two dimensions
FR-4	The player shall win when all the enemies are dead
FR-5	The player shall lose when it dies
FR-6	The player shall be able to move with WASD keys
FR-7	The player shall be able to shoot with directional keys
FR-8	The enemy type 1 shall follow the player while its HP is below half its maximum value
FR-9	The enemy type 1 shall flee from the player while its HP is at or bellow half its maximum value
FR-10	The enemy type 1 shall fire a projectile always when it has a clear line of shot towards the player and it is not on cooldown
FR-11	The enemy type 1 shall fire a missile always when it has a clear line of shot towards the player and it is not on cooldown
FR-12	The enemy type 2 shall flee from the player while its HP is at or bellow half its maximum value
FR-13	The enemy type 2 shall follow the player while its over 500 units away from the player
FR-14	The enemy type 2 shall follow a straight line its at or under 500 units away from the player
FR-15	The enemy type 2 shall place a bomb when the player has a clear path towards its current position and it is not on cooldown
FR-16	The enemy type 3 shall always follow the player
FR-17	The enemy type 3 shall explode when it collides
FR-18	The enemy type 3 shall cause 2 damage in a 200 units radius when exploding
FR-19	An entity shall not cause damage to itself
FR-20	The projectile ability shall have a cooldown of 2 seconds to the enemies
FR-21	The projectile ability shall have a cooldown of 1 second to the player
FR-22	The missile ability shall have a cooldown of 5 seconds to the enemies
FR-23	The bomb ability shall have a cooldown of 5 seconds to the enemies
FR-24	A projectile shall cause 1 damage to the object it collides
FR-25	A missile shall cause 2 damage to the object it collides
FR-26	A bomb shall detonate 3 seconds after being created
FR-27	A enemy type 3 shall cause 2 damage in a 200 units radius when exploding
FR-28	The player shall move at a speed of 500 units per second
FR-29	The enemy type 1 shall move at a speed of 300 units per second
FR-30	The enemy type 2 shall move at a speed of 250 units per second
FR-31	The enemy type 3 shall move at a speed of 400 units per second
FR-32	The objects shall not be able to move over each other
FR-33	The projectile shall follow a straight line with a speed of 1000 units per second
FR-34	The missile shall follow the player with a speed of 1000 units per second
FR-35	The level shall spawn a enemy of each type in 3 corners of the arena
FR-36	The player shall be able to pause the game
FR-37	The player shall be able to continue the game from the pause menu
FR-38	The enemy type 1 shall have 2 HP
FR-39	The enemy type 2 shall have 5 HP
FR-40	The enemy type 3 shall have 1 HP
FR-41	The player shall have 20 HP

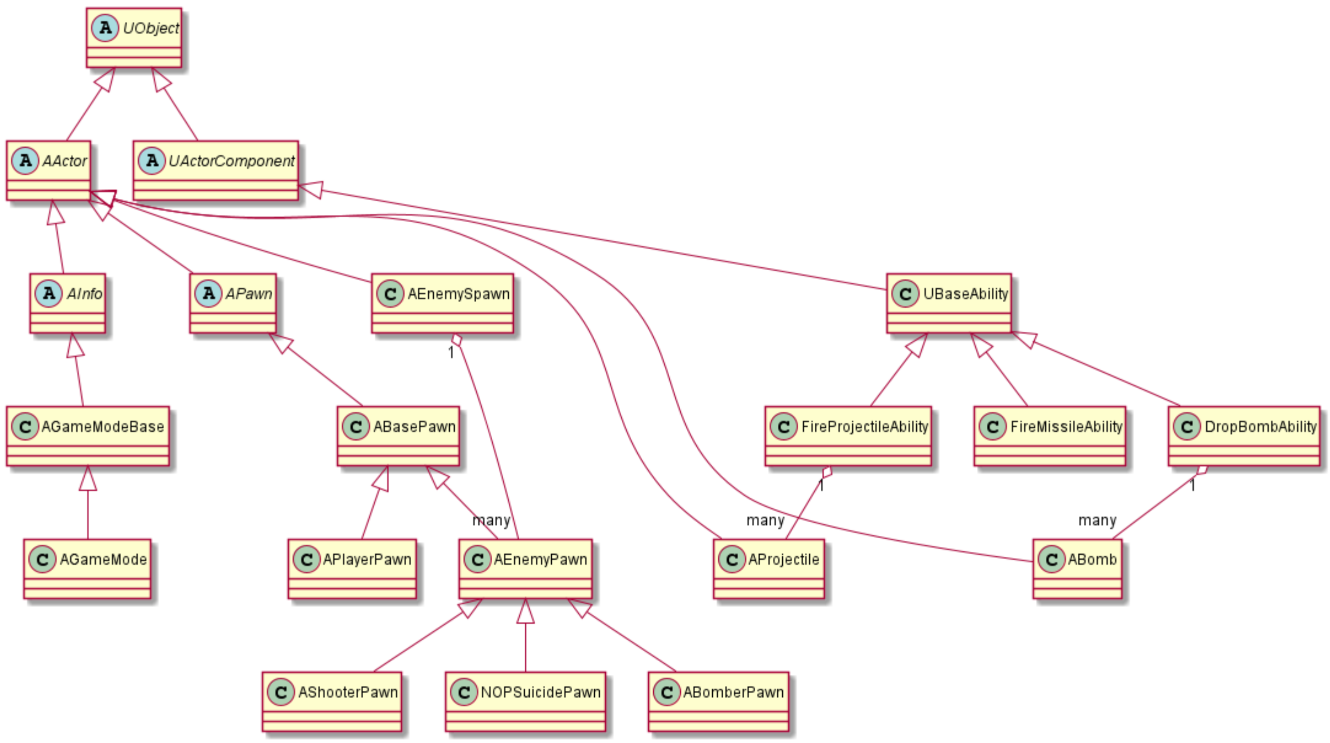

Fig. 6: OO Class Diagram

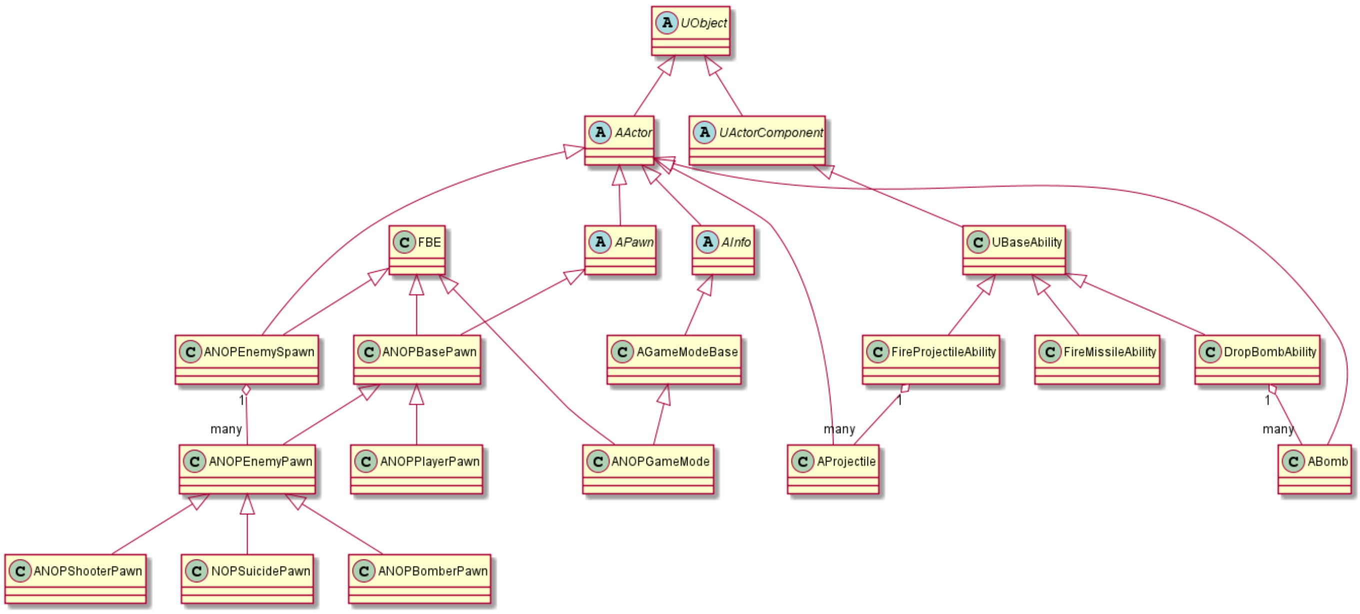

Fig. 7: NOP Class Diagram

TABLE IV: Modified Functional Requirements

Requirement	Description
FR-8	The enemy type 1 shall follow the player while it is over half of its maximum HP and it is the closest type 1 enemy to the player or it is over a distance of 800 units
FR-35	The level shall spawn enemies dynamically
FR-42	The player shall win when reaching 20 points
FR-43	The player shall be awarded 2 points when destroying a enemy type 1
FR-44	The player shall be awarded 3 points when destroying a enemy type 2
FR-45	The player shall be awarded 1 point when destroying a enemy type 3
FR-46	The player shall not be awarded any points when enemies destroy each other
FR-47	The level shall only spawn new enemies on the corners of the arena
FR-48	The level shall spawn a new enemy type 1 every 10 seconds until a maximum of 5 simultaneous enemies type 1 exist
FR-49	The level shall spawn a enemy type 3 for each two enemies of other types destroyed
FR-50	The level shall spawn a enemy type 2 every 5 seconds, while there are less than 5 enemies of any type in the arena
FR-51	The level shall initially spawn a enemy type 1 in each of the corners of the arena
FR-52	The enemy type 1 shall circle around the player with a 800 units radius while its HP is over half of its maximum HP and it is not the closest type 1 enemy to the player

TABLE V: Rule 1

RL-1	
Class	NOPBasePawn
Type	Conjunction
Description	Handle pawn' death

Premises
Pawn's HP is at or below 0

TABLE VI: Rule 2

RL-2	
Class	NOPBasePawn
Type	Conjunction
Description	Gives rewards on pawn's death

Premises
Pawn's HP is equal ow less than 0

TABLE VII: Rule 3

RL-3	
Class	NOPBomberPawn
Type	Conjunction
Description	Use the drop bomb ability

Premises
Ability cooldown is ready
Player has free path to current position

TABLE VIII: Rule 4

RL-4	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to flee

Premises
Pawn's HP is equal ow less than the maximum

TABLE IX: Rule 5

RL-5	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to follow

Premises
Pawn's HP is greater than half the maximum
Distance to player is greater than 500 units

TABLE X: Rule 6

RL-6	
Class	NOPBomberPawn
Type	Conjunction
Description	Set strategy to roam

Premises
Pawn's HP is less or equal half the maximum
Distance to player is less or equal 500 units

TABLE XI: Rule 7

RL-6	
Class	NOPEnemySpawn
Type	Conjunction
Description	Spawn an enemy type 1

Premises
Spawn cooldown is ready
Number of simultaneous enemies of type 1 is less than 5

TABLE XII: Rule 8

RL-7	
Class	NOPEnemySpawn
Type	Conjunction
Description	Spawn an enemy type 2

Premises
Spawn cooldown is ready
Number of simultaneous total enemies is less than 5

TABLE XIII: Rule 9

RL-8	
Class	NOPEnemySpawn
Type	Conjunction
Description	Spawn an enemy type 3

Premises
Spawn cooldown is ready
Number of simultaneous total enemies is less than 5

TABLE XIV: Rule 10

RL-9	
Class	NOPShooterPawn
Type	Conjunction
Description	Fire a projectile

Premises
Has clear line of shot to player
Projectile ability cooldown is ready

TABLE XV: Rule 11

RL-10	
Class	NOPShooterPawn
Type	Conjunction
Description	Fire a missile

Premises
Has clear line of shot to player
Missile ability cooldown is ready

TABLE XVI: Rule 12

RL-11	
Class	NOPShooterPawn
Type	Conjunction
Description	Set strategy to flee

Premises
Pawn’s HP is less or equal half the maximum

TABLE XVII: Rule 13

RL-12	
Class	NOPShooterPawn
Type	Disjunction
Description	Set strategy to follow

Premises	
Sub condition 1	Type: Conjunction
Pawn’s HP is greater than half the maximum	
Pawn is the closest of its type to the player	
Sub condition 2	Type: Conjunction
Pawn’s HP is greater than half the maximum	
Pawn over 800 units away from the player	

TABLE XVIII: Rule 14

RL-13	
Class	NOPShooterPawn
Type	Conjunction
Description	Set strategy to roam

Premises
Pawn is not the closest of its type to the player
Pawn’s HP is greater than half the maximum
Pawn under 800 units away from the player

TABLE XIX: Rule 15

RL-14	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Player wins the game

Premises
Game has started
Player has 20 or more points
Player is alive

TABLE XX: Rule 16

RL-15	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Player loses the game

Premises
Game has started
Player is dead

TABLE XXI: Rule 17

RL-16	
Class	NOPUnrealGameMode
Type	Conjunction
Description	Set game as started

Premises
Game has not started
Game startup is finished

VI. Evaluations and Comparisons

For an objective perspective in comparing the amount of effort required to develop the software it was chosen to compare both implementations by evaluating the difference in lines of code and time required to develop the software using both approaches, as they can be useful as indicators of how much effort was required during the development. Given that the developer was not experienced with neither the Unreal Engine API nor the NOP C++ Framework it was chosen to only evaluate their time used in the development of the second set of requirements, since most of the time spent during the development of the first version was actually spent learning how to use the tools as well as writing basic functions used for both versions and therefore would not reflect the real difference in the amount of effort required.

For the time evaluation each requirement, either new or modified, is linked to the amount of time spent in the Object-Oriented and Notification Oriented approaches, detailed in Table XXII. For this purpose it is only considered the time effectively spent writing code.

TABLE XXII: Time spent for development

Requirement	**State**	**NOP**	**OOP**
FR-35	Modified	0 min	0 min
FR-42	New	5 min	20 min
FR-43	New	15 min	5 min
FR-44	New	5 min	5 min
FR-45	New	5 min	5 min
FR-46	New	0 min	0 min
FR-47	New	5 min	5 min
FR-48	New	10 min	30 min
FR-49	New	10 min	30 min
FR-50	New	10 min	30 min
FR-51	New	10 min	10 min
FR-8	Modified	30 min	90 min
FR-9	Modified	30 min	30 min
Total		135 min	220 min

In the same manner the total lines of code are compared using the *cloc* tool [12]. This tool can count the total lines of code while eliminating comments and blank lines. Only the source code used for each implementation is considered for this purpose, and code that is common for both implementations is accounted for in each one of them. The results are shown in Table XXIII. The number of files is the same for both implementations.

TABLE XXIII: Lines of code written

Requirement Version	**OOP**	**NOP**
Initial	1272 lines	1407 lines
Final	1583 lines	1776 lines

Another analysis was to count the number of words in the same manner, this time using the command *wc* from *Linux*. The results are shown in Table XXIV.

TABLE XXIV: Words written

Requirement Version	**OOP**	**NOP**
Initial	54800 words	59315 words
Final	67682 words	75329 words

VII. Conclusion and Future Works

The results presented clearly show that working with NOP can bring great improvements to the process of game development, and other types of software as well, whether by reducing the development time due to a more comprehensive code structure, specially when writing code for behavior that can be translated well into rules, which is the case in many aspects of game development, since a lot of the effort is dedicated to writing the game logic and non-playable characters behavior.

As shown in the results from Table XXII, the time spent modifying the NOP code was 38.62% when compared with the OOP only code. And another major improvement is that all the NOP rules can be directly traced into specific requirements, meaning that any changes in the project can be more easily translated into code, as it reduces the risk of having the logic spread among to many places in the code, which makes it harder to do any changes without breaking other behaviors or leaving deprecated code in some places, which may easily lead to bugs and inconsistency in the software behavior.

There are however some clear drawbacks, that come from working with a novel technology. The framework itself can be very verbose, as it is shown in the increased amount of code written necessary for the same logic, as show in Table XXIII and Table XXIV, as the NOP code has 12.16% more lines and 11.13% more words, however this is small problem, since this comes from having to copy the initialization and declaration from the NOP structure, and doing so still took less time than the OOP approach.

Further development of the NOP technology and the C++ framework itself can help yield even better results, reducing the total time and amount of code required for software development. The development of more projects with the NOP framework can also be useful to ground the statement that the use of the technology can help reduce the effort required in software development.

References

[1] J. Gold, *Object-oriented game development*. Pearson Education Limited, 2004.

[2] D. Takahashi, "Superdata: Games hit $120.1 billion in 2019, with fortnite topping$1.8 billion," Dec 2019. [Online]. Available: https://venturebeat.com/2020/01/02/superdata-games-hit-120-1-billion-in-2019-with-fortnite-topping-1-8-billion/

[3] J. Gregory, *Game engine architecture*. CRC Press, 2019.

[4] A. F. Ronszcka, "Contribuição para a concepção de aplicações no paradigma orientado a notificações (pon) sob o viés de padrões," Master's thesis, UTFPR, 2012.

[5] L. D. Wen, "Steam-engines," https://github.com/limdingwen/Steam-Engines, 2018.

[6] E. Games. (2020) Unreal engine 4 documentation. [Online]. Available: https://docs.unrealengine.com/en-US/index.html

TABLE XXV: Correlation Matrix

Requirement	RL-1	RL-2	RL-3	RL-4	RL-5	RL-6	RL-7	RL-8	RL-9	RL-10	RL-11	RL-12	RL-13	RL-14	RL-15	RL-16	RL-17
FR-1																	
FR-2																	
FR-3																	
FR-4																	
FR-5	x															x	
FR-6																	
FR-7																	
FR-8													x				
FR-9												x					
FR-10										x							
FR-11											x						
FR-12				x													
FR-13					x												
FR-14						x											
FR-15			x														
FR-16																	
FR-17																	
FR-18																	
FR-19																	
FR-20										x							
FR-21																	
FR-22											x						
FR-23			x														
FR-24																	
FR-25																	
FR-26																	
FR-27																	
FR-28																	
FR-29																	
FR-30																	
FR-31																	
FR-32																	
FR-33																	
FR-34																	
FR-35																	
FR-36																	
FR-37																	
FR-38																	
FR-39																	
FR-40																	
FR-41																	
FR-42															x		x
FR-43		x															
FR-44		x															
FR-45		x															
FR-46		x															
FR-47																	
FR-48							x										
FR-49									x								
FR-50								x									
FR-51																	x
FR-52														x			

[7] J. M. Simão, "Proposta de uma arquitetura de controle para sistemas flexíveis de manufatura baseada em regras e agentes," Master's thesis, UTFPR, 2001.

[8] R. F. Banaszewski, "Paradigma orientado a notificações: Avanços e comparações," Master's thesis, UTFPR, 2009.

[9] P. Van Roy, "Programming paradigms for dummies: What every programmer should know," 04 2012.

[10] R. D. Xavier, "Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações," Master's thesis, UTFPR, 2014.

[11] S. H. P. Y. L. P. C. S. J. M, "A notification-oriented approach for systems requirements egineering," *Advances in Transdisciplinary Engineering*, vol. 4, pp. p. 229–238, 2016.

[12] A. L. Danial, "Count lines of code," https://github.com/AlDanial/cloc, 2020.