Towards an Application of Adaptive Programming to Gameplay Mechanics

Joshua Pritchard

University of Huddersfield

N421: BSc (Hons) Computing Science with Games Programming SW

CHP2524: Final Year Project

Dr. Minsi Chen & Dr. Gary Allen

May 4 2021

Abstract

This project deals with applying the paradigm of Adaptive Programming, popularized by *ScriptEase*, to the creation of gameplay mechanics, in the field of gameplay programming. Within, background research is undertaken, finding justification for tools to automate the construction of gameplay mechanics' functional implementations. Following the *Build*, *Process* and *Adaptive Software Development* methodologies, this report details a set of universal concepts to facilitate the construction of such tools. A partial example implementation is also constructed and tested, with critical evaluation of the technologies developed deeming their use significant compared against competing tools in the field. This report concludes that adaptive programming can be successfully applied to the creation of games mechanics, and recommends that further research and development be performed in order to realize the technology's potential to introduce standardization to the field of gameplay programming.

Contents

1. Introduction

The field of Software Engineering is broad, covering wildly different sub-fields and their comprising disciplines. Games Programming is one such sub-field, within which *Gameplay Programming* is a specific discipline.

Gameplay Programming is the process by which *Gameplay Mechanic Specifications* are converted into code. The applicability of common Software Engineering techniques to Games Programming (and therefore Gameplay Programming) is a highly debated topic, for which evidence suggests a lack thereof. Regardless – For any given Gameplay Mechanic Idea/Specification, there are a number of different Software-Engineering processes that can be used to model and create a functional [code] representation.

For example – *UML* & *MACHINATIONS*; while being very different tools, offer ways to bridge the gap between idea and execution. Both of these tools come with limitations that conflict directly with the evolutionary-prototyping-like approach commonly found in Gameplay Programming. *UML* is more suited to business software modelling, and *MACHINATIONS* is targeted towards economy mechanics[1].

Gameplay programming also carries a significant responsibility of code structure quality as, often, games are built around their core mechanics. This quality affects game development tasks further along the game development process (lower quality code structure will be harder and more time consuming to make changes/additions to). Combining this responsibility with current tool limitations presents a complex challenge to Gameplay Programmers who, often out of necessity, will choose to accept lower quality structure in favour of having a finished product.

This project attempts to remove this challenge by developing a new process that better fits the iterative nature of gameplay programming. Said process follows the principles of '*Adaptive*

[1] Consider how *MONOPOLY* can be modelled as an economic simulation.

Programming' – A programming paradigm by which pre-written, modular, blocks of code are combined via leverage of high-level patterns – Easy-to-understand, common, recurring 'use-cases' whose core implementations can be extended with additional, modular blocks; referred to as 'adaptations' (Cutumisu, et al., 2007).

This project is directly inspired by the most recognizable application of adaptive programming; *SCRIPTEASE* – A scripting tool developed for *NEVERWINTER NIGHTS*. With a visual interface, it applied the technique to trivialize the process of creating unique quests for the game – So much so that in a paper written by its creators, high school students with *no prior programming experience* were able to create unique quests in 1-2 hours (Cutumisu, et al., 2007). For designers, this was a way to quickly and easily create content; for the Gameplay Programmers, full access to the generated[2] source code is given, to which further unique modifications can be made and, perhaps most importantly, understanding and learning of the techniques used can be gained.

The process detailed in this work presents the idea of 'adaptation files'; written in a platform-agnostic, declarative language; in combination with 'pattern files' and a platform specific generation & adaptation system. With these, a user specifies a high-level gameplay mechanic pattern and zero or more adaptations. This specification is passed to the generation & adaptation system, which generates the high-level pattern code files and then modifies them with appropriate code for the adaptations. The generated code should be compile-able and ready to use.

Assuming the pre-written blocks are high quality, (regardless of whether the adaptation system generates implementation or just structural code) a prototype whose mechanics are created with this tool would have zero problems with code structure quality. This would present a significant improvement of workflow for prototypers, who could use the tool to focus on the implementation for prototypes without having to think about code architecture or engineering.

[2] This term is used sparingly here, as *Generative Programming* is a unique paradigm of its own, and is only a partial focus of this project.

This tool would also bring benefit to Gameplay Programming in education – At present, a student has few *specifically* gameplay programming related resources available to them. Many of the *actually* effective processes, methods and technologies are locked behind a protectivist games industry that prefers in-house training; and unexplored by an education system that prefers traditionally academic overlapping areas of games programming (such as artificial intelligence or real-time graphics).

Using this tool, students would be able to generate personalized exemplars from which they could learn the conventions and syntaxes of the target platform i.e. *UNREAL ENGINE*. Doing so presents a didactic learning method which would accelerate student comprehension and introduce a level of standardization throughout the field of Gameplay Programming. Such standardization would bring benefits comparative to that of standardization in 'mainstream' computer science, where the 'right way to do things' is more obvious and encouraged.

Since the initial specification of the product, the scope has been reduced. No attempt is made at a GUI frontend or actual adaptation of code files. Both of these are core components of the solution proposed by this project, however lie outside its time constraints. Instead, this project focuses on developing and demonstrating the logical process and language used to create and deduce intent from pattern and adaptation choices.

It is thought that this technology may bring about the accessibility and standardization discussed. The final product shown demonstrates clearly that the use of this process to generate gameplay mechanics would be useful and an improvement over existing methods. It also presents a recommendation for further, better funded research and development into not only the process, but the field in general. This is all done in a way independent of any specific programming language, in order to better display the universal principles behind the work. It is hoped that this serves as a prompt for further research and development to be undertaken to realize this solution's full potential.

The rest of this report is comprised as follows:

- A **Background Research** section, analyzing the justification for this tool and discussing the field of adaptive programming.
- A **Methodology** section, comprised of 3 chapters:
    - **Research Methodology**, discussing the way in which the experimental aspect of this project is approached.
    - **Software Development Methodology**, discussing the approach taken for managing the development portion of the project.
    - **PLESI Considerations**, providing a brief consideration of associated professional, legal, ethical and social issues.
- A **Developed Method** section, comprised of 2 chapters:
    - **Core Definitions & Process Explanation**, detailing the concepts and principles and of the developed process.
    - **Implementation Examples**, presenting an application of the process to *UNREAL ENGINE*.
- An **Implementation Results & Testing** section, detailing the tangible products of using the implementation, and the testing used to guide its development.
- A **Product Evaluation** section, critiquing the usefulness of the method & implementation.
- A **Project Evaluation** section; critiquing the project process, discussing challenges, and evaluating the author's performance as a whole throughout the project duration.
- **Conclusions**, including recommendations for future work.
- **References/Bibliography**.
- **Appendices**.

2. Background Research

2.1 – Game Development Difficulty

There is a general consensus that developing video games has a high level of difficulty. This theme is well-established, with a crucial article by Blow (2007) explaining how increasing consumer expectations, software framework sizes and lack of external tool support have brought on said difficulty.

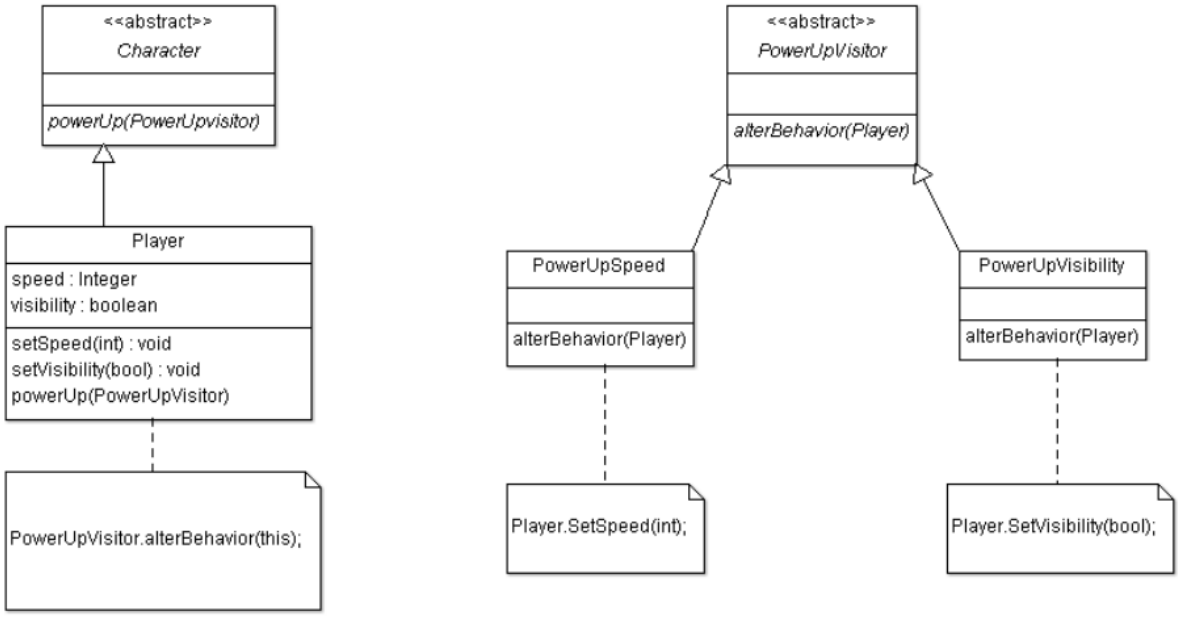
Despite the area of games development being one in which research quickly gains obsolescence, and an influx of the accessible software discussed within, the key issues raised still remain. Games development literature from all eras references difficulty in some form, of which multiple emphasize the creation of behaviour for games (Pellens et al., 2008; Roedavan et al., 2020; Kounoukla et al., 2016). Any usage of the GoF design patterns detailed by Kounoukla et al. (2016) (such as seen in figure 1) comes with significant implementation difficulty (Budinsky et al., 1996), and leads to a contribution towards the statistic of just 16% of projects being completed on time and under-budget (Kanode & Haddad, 2009).

Various studies highlight the extenuating considerations and skillsets required in game development (Aleem et al. 2016; Sarinho & Apolinário, 2009), while another key study uses a similar analysis to highlight the need for a streamlining of the entire process (Cutumisu, et al., 2007).

Despite this, there is a lack of research and standardization on key elements of games development, such as the creation of gameplay mechanics.

Figure 1

Visitor / Power-Ups



Note. This figure was produced by Kounoukla et al. in 2016, and details an application of the *Visitor* pattern to implementing Power-Ups. From “Implementing Game Mechanics with GoF Design Patterns”, by X.-C. Kounoukla, A. Ampatzoglou and K. Anagnostopolous, 2016. Copyright 2016 by ACM

2.2 – Learning Gameplay Programming

The discussed general difficulty of games development, and the author's own experiences present a more granular issue; It is hard to learn Gameplay Programming *specifically*.

Few studies deal with this exact topic, however Roedavan et al. (2020) details how 33% of students reported difficulty in creating gameplay with Unity C#, and that the majority of students find difficulty in implementing *mechanic* logic within *code* logic.

This is reinforced by Kounoukla et al. (2016), who explain that despite 'Game Design Patterns' being well documented (thanks in large part to the works of Björk (n.d.)), there is little to no guidance on their implementation in code. They justify the argument for design patterns by detailing how common GoF patterns can be used for common game mechanics, along with their benefits.

The topic is indirectly supported via lack of mention by Kanode & Haddad (2009), where much is written of iteration to 'find the fun' of a game, but nothing is written about simply the difficulties of Gameplay Programming from a software engineering perspective.

A potentially crucial explanation for this gap in knowledge comes from Aleem et al. (2016), drawing attention to the fact that researchers don't have the resources to create large games, and developers never publish the results of their own efforts. While the knowledge gap is clearly receding, there is still a critical lack of feedback loops between games development literature and practice.

2.3 – Existing Mechanic Definitions

There exists a pressing issue for specifically the topic of games mechanics; They are defined either too formally, or applied too simplistically in academia to represent any benefit to practitioners.

The work of Zook & Riedl (2014) best showcases the simplicity (regardless of the study's focus area) with which academia views game mechanics in the simplistic game environments that their mechanic generator is applied to, as seen in figure 2.

Maranhão et al. (2016) supports the notion with an observation of academic games mechanic representations, in that they are complex constructions and formally defined, resulting in diminished practical utility and increased obfuscation between a specification and its product.

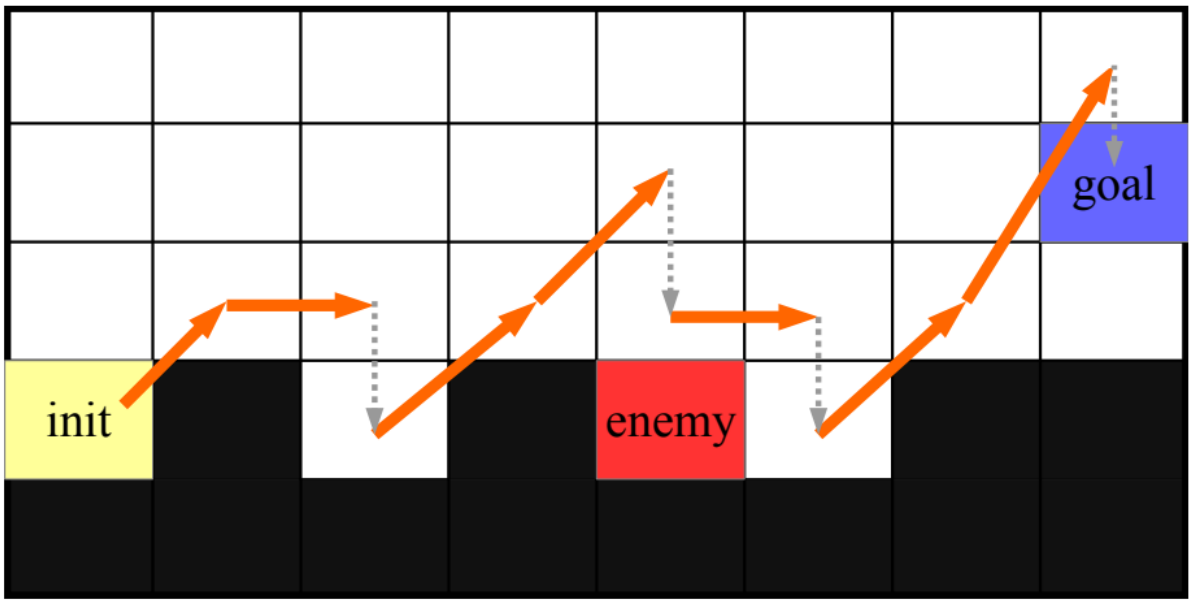
Thompson, et al. (2013) also support the notion of over-formality in their foundational criticisms of existing game description languages, which are unable to handle even simple arcade games. The date of this work adds further criticism to formal representations with the dates of the titles that they show GDLs could not support: *SPACE INVADERS (1978), ASTEROIDS (1979)*.

Despite the criticism, both works made significant strides in rectifying the issue, with Thompson, et al. (2013) developing the lacked game description language, and Maranhão et al. (2016) making a key observation around players viewing mechanics as abstracted and simplified forms of game rules, leading to their well-structured and seemingly obvious methodology for game mechanics analysis and definition, seen in figure 3.

Despite said key strides taken by the two works discussed, it is clearly an under-pursued area in literature, with large gaps between incremental research (to date, Maranhão et al. (2016) have but 3 citations). It is clear that a gap still exists in the field for a new method with which gameplay mechanics can be defined and analysed in a more natural way.

Figure 2

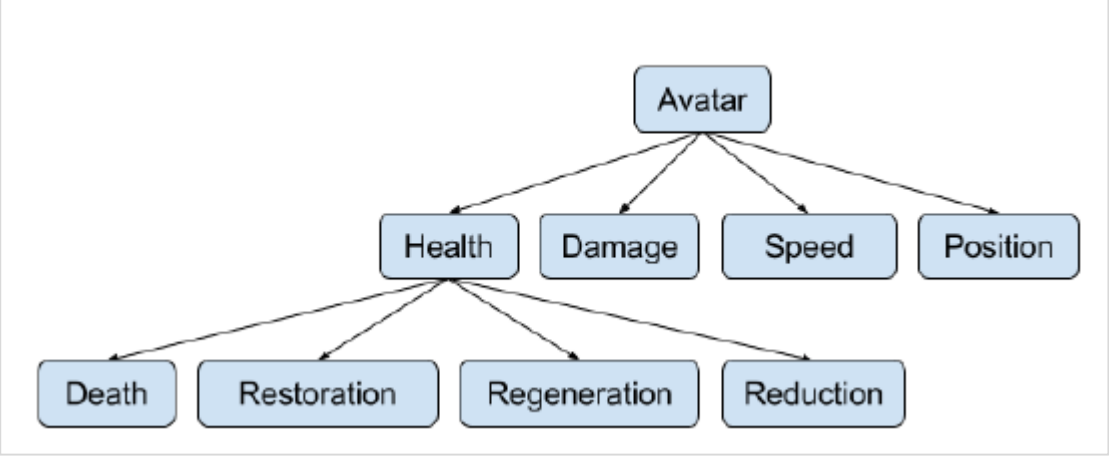
Platformer level showing a playtrace using a generated mechanic set. Arrows indicate generated mechanics, dotted arrows indicate gravity



Note. This figure was produced by Zook and Riedl in 2014, and shows an environment in which their mechanic generator has been applied to. From "Generating and Adapting Game Mechanics", by A. Zook and M. O. Riedl, 2014. Copyright 2019. CC BY-NC-SA 3.0

Figure 3

Visual Description of game mechanics on a hypothetical game. Note that the hierarchical structure of mechanics is clear and arguably easy to understand



Note. This figure was produced by Maranhão et al. in 2016, and shows an example application of their game description model. From "Towards a Comprehensive Model for Analysis and Definition of Game Mechanics", by D. M. Maranhão, Artur de Oliveira da Rocha Franco, G. M. M. Junior and J. G. R. Maia.

2.4 – Adaptive Programming for Games

A small number of studies have applied Generative/Adaptive Programming[3] to games development. Once again, the key study as relevant to this project is the work of Cutumisu et al. (2007), who detail the application of *ScriptEase* to the cRPG, *Neverwinter Nights*.

ScriptEase applies Adaptive Programming principles to game scripting by employing a 3 step process – Authors first select a high level story pattern, make small adaptations to it, and then click a button to generate the code necessary for that pattern to occur in the game world (seen in figure 4). The authors use a case study on non-programmers to justify their claims that Adaptive Programming is the future of programming, and that is has the potential to affect users in various domains and take steps towards eliminating programmers from many content authoring processes.

Within, they make two crucial points: One, a criticism of traditional Constructive Programming that states asking authors to create new scripts for every quest is akin to asking them to create new *patterns* every time they were to use *ScriptEase*. The second, an evidenced theory (through the case study) that Adaptive Programming allows non-programmers to create complex behaviour.

As already discussed, the date of this work is a limiting factor in the external observations and criticisms they make, however the novel aspect they identify is one of the future and in many ways *more* relevant now than it was in 2007.

Since then, just two studies seem to have applied even similar concepts to those discussed: Zook & Riedl (2014) (Figure 2), who touch on an application of AI mechanic generation in adaptation, wherein mechanics could be iterated upon to achieve adaptation requirements (in light of new content design requirements) – And Guana & Stroulia (2014), who define a code generation environment for physics based games, using a text editor backed by a domain specific language. This

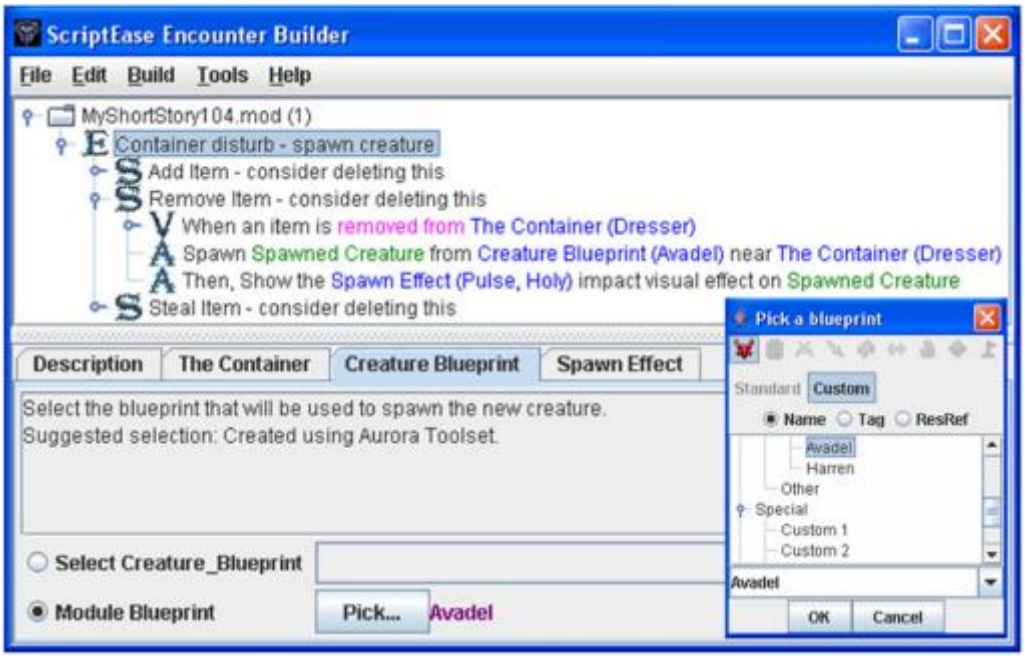
[3] A programming paradigm by which pre-written, modular, blocks of code are combined via leverage of high-level patterns – Easy-to-understand, common, recurring 'use-cases' whose core implementations can be extended with additional, modular blocks; referred to as 'adaptations' (Cutumisu, et al., 2007).

allows for adaptations to be made to game actors in terms of finite, predefined sets of properties in order to adapt gameplay at a high level.

In the Adaptive Programming paradigm, the last article discussed is the closest literature has come to documenting an application of Adaptive Programming to Gameplay Programming, making this an area of research ripe for innovation, 13 years after the inspiring *SCRIPTEASE* (Often cited as the only commercial application of Adaptive Programming to date).

Figure 4

A generative pattern, its description and a dialog being used to set an option. (For interpretation of the references to color in the figure legend, the reader is referred to the Web version of this article.)



Note. This figure was produced by Cutumisu et al. in 2007, and shows an example of a pattern within *ScriptEase*. From "ScriptEase: A generative/adaptive programming paradigm for game scripting", by M. Cutumisu, C. Onuczko, M. McNaughton, T. Roy, J. Schaeffer, A. Schumacher, J. Siegel, D. Szafron, K. Waugh, M. Carbonaro, H. Duff and S. Gillis. Copyright 2007 by Elsevier B. V.

2.5 – Adequacy of Current Mechanic Modelling Languages/Methods

Current methods of gameplay mechanic translation from concept to code are limited. *UML* for example, while considered the de facto standard of software modelling, doesn't support creative aspects of modelling very well (Kasurinen, 2016). Being more suited to developing business software, its meticulous documentation and modelling conventions often harbor hindrance behind initially attractive offers of 'correctness'. Take, for example, the lack of evident functional description in figure 5.

Montero Reyno & Á. Carsí Cubel's (2008) application of *Model-Driven Development* to games development doesn't address the lack of *low-level* creative ability in *UML* (on which it is based) – This makes the method inappropriate for translation of single game mechanics (especially in prototyping) in favour of improving high-level, abstract game comprehension (this makes sense, as it is first and foremost a methodology). *MDGD* makes relevant strides in bridging the gap between programmers and non-programmers (crucial for Gameplay Programming), however lacks application in commerical games development (suggesting better methods are being used) (Zhu & Inge Wang, 2019).

Cutumisu et al. (2007) highlight that visual scripting tools made for gameplay scripting [*KISMET*] are too low level and hard to use. Despite limitations of age and of *BLUEPRINTS* replacing *KISMET*, a crucial comparison: 'asking authors to create new scripts every time is akin to asking them to create new *patterns* every time they were to use ScriptEase', still holds true. Parallel to this, *BLUEPRINTS* retains the limitations of the programming language on which it is based – C++: A low level, granular tool; obstructive to any concise, high-level abstraction of functionality or structure.

Tools often suffer from their specialism focus, with those focused on one speciality (e.g. *MACHINATIONS* for Game-Economy mechanics (Figure 6), or *LUDOCORE* for combinatorial mechanics)

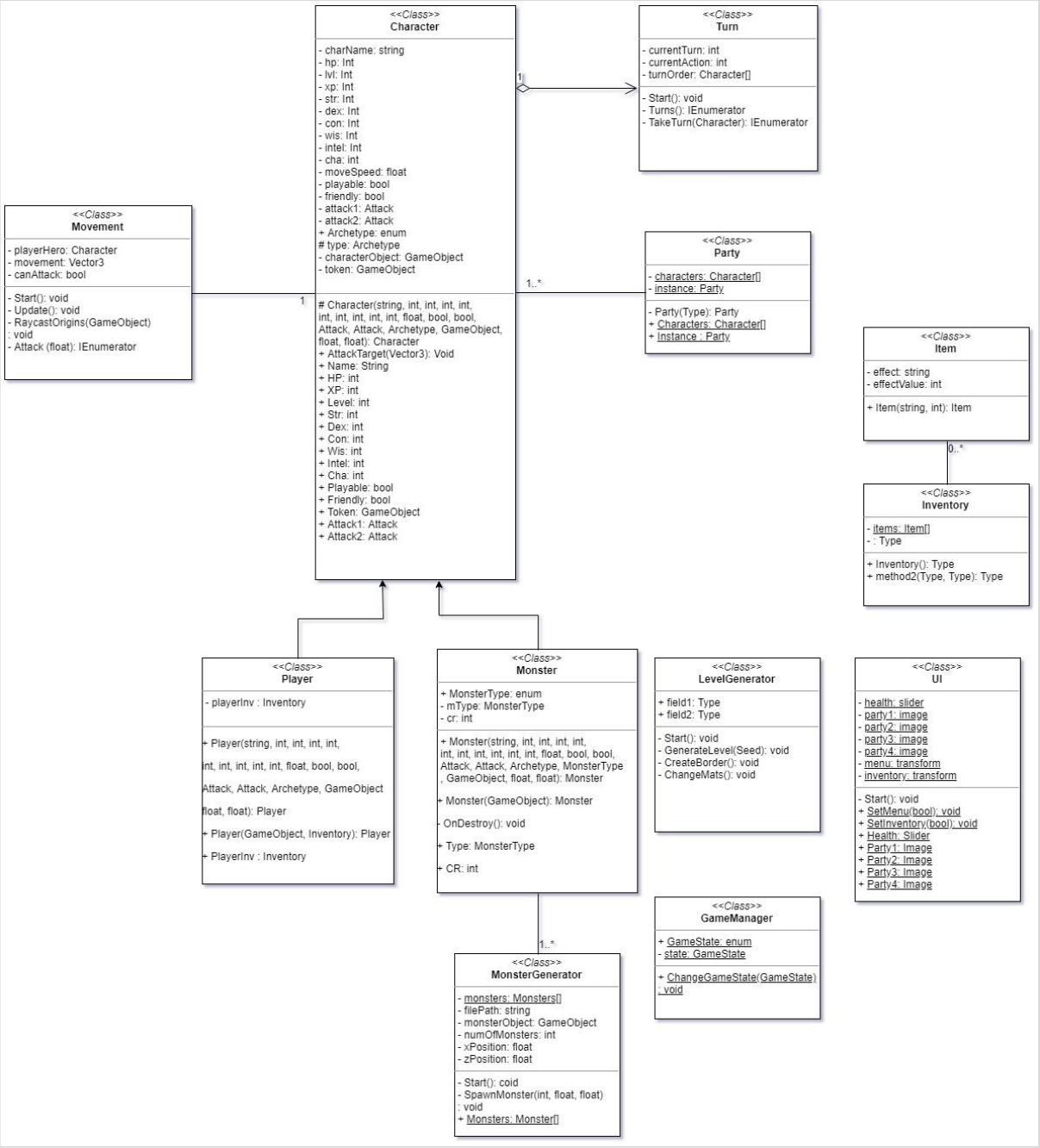
performing badly in other specialisms (like Avatar-Centric mechanics, which seems to be an accurate descriptor for the focus of this project) (Van Rozen, 2020).

MACHINATIONS was developed for modelling economic mechanics or large scale system dynamics, making it inherently unsuitable for dealing with granular, avatar-centric mechanics such as the grappling hook example from the introduction. Consider how figure 6 gives no physical description of how *MONOPOLY* pieces are moved around the board etc.

MECHANIC MINER, an existing avatar-centric gameplay mechanic tool, works backwards relative to Gameplay Programming, requiring a game's full source code to 'find' mechanics (Cook, 2013).

Figure 5

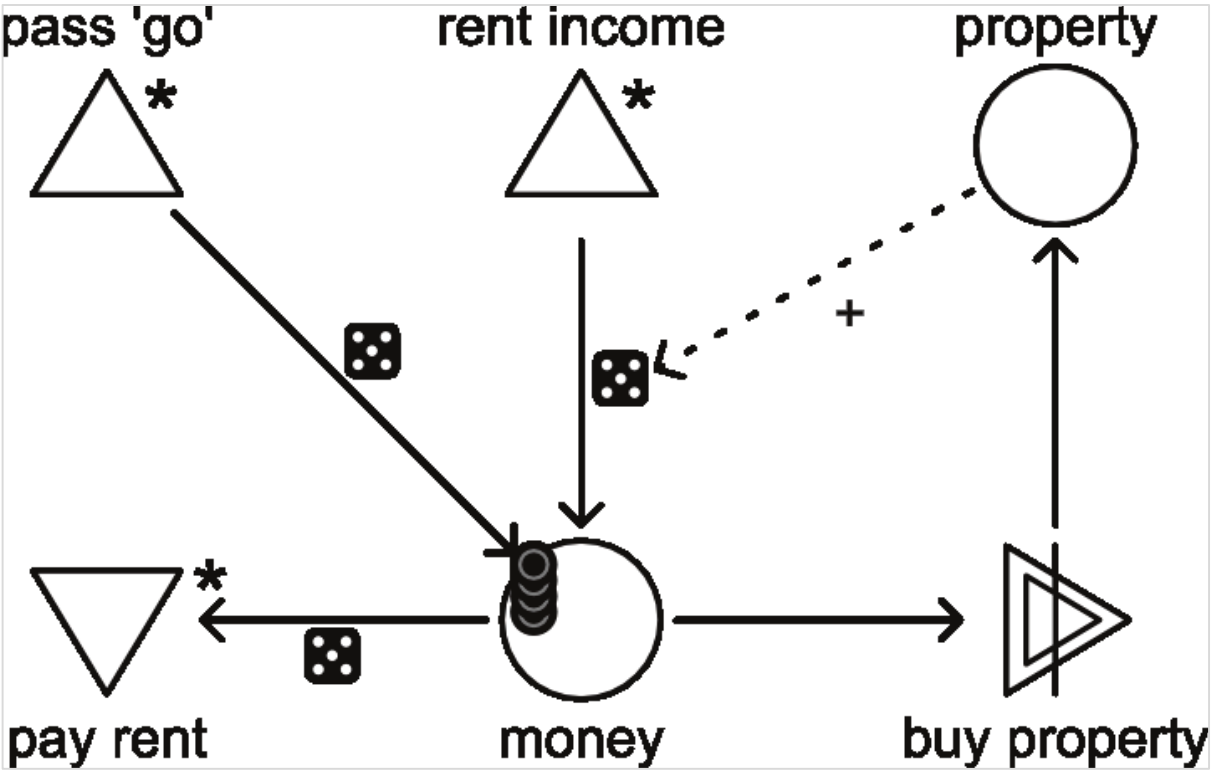
Class diagrams



Note. This figure was produced by Joshua Walker in 2018, and shows an example of game mechanic description via *UML* class diagram. (https://jjrwalker.github.io/general/UMLdesign/). Copyright 2020 by Joshua Walker.

Figure 6

A Machinations Diagram of Monopoly



Note. This figure demonstrates how *MACHINATIONS* can be used to model dynamic behaviour in games, with the example being *MONOPOLY*. Reprinted from "*Game Design Research: An Introduction to Theory & Practice"* (p. 87), by J. Dormans and A. Hook, 2017, ETC Press. Copyright 2017 by ETC Press. CC BY-NC-ND 4.0

2.6 – Applications of Adaptive Programming to Games Mechanics

This project deals with an application of the adaptive programming paradigm to the creation of *individual* gameplay mechanics, more akin to the creation of a method than an entire methodology. This core idea is supported and justified by many, however never referred to explicitly by name.

The most directly related of these suggests benefit in inclusive programming interfaces and the creation of mechanic stereotype catalogues & manuals with best practices (Maranhão et al., 2016).

Two studies in granular areas of game behaviour generation detail their workings and benefits, with the latter suggesting that distinct code generation environments can made for different game genres (Pellens et al., 2008; Guana & Stroulia, 2014).

Multiple studies re-affirm the prototyping, code re-use and software quality benefits of design patterns (both generative and descriptive), with the latter adding a second advocation for sample implementations of game mechanics that can be re-used as starting points for source code development (MacDonald, et al., 2002; Pellens et al., 2009; Kounoukla et al., 2016).

Aleem et al. (2016) provide a solid basis for justification of work with their observations around novel games being produced by simply iterating over gameplays etc. and that game development must follow an incremental model.

Finally, Cutumisu et al. (2007) provide an idyllic future outlook where an author can specify and test games directly, without ever using a programmer.

Ambitions are far from realisation, however the justification for such an application has been growing steadily for over a decade.

3. Methodology

3.1 – Research Methodology

As this is a research project in nature, consideration of *Research Methodology* is required. Amaral, et al. (n.d.) details five common research methodologies by which computer science research is performed, of which this project uses *Process* and *Build*:

1. *Formal* methodologies are largely mathematical, and are often concerned with proving facts or correctness about algorithms and systems.
2. *Experimental* methodologies are used in evaluation of solutions, where a project will generally ask and answer questions about a given system.
3. *Build* methodologies are concerned with the building of an artifact, often software. This is distinguished from a *Software Development Methodology* by virtue of the artifact's inclusion of novel features.
4. *Process* methodologies are used to understand processes in computer science, often tackling how humans build and *use* computer systems.
5. *Model* methodologies are used when seeking answers to questions asked of complex systems. Abstract models are used to 'simulate' a complex system and answer the desired questions. Often used in combination with *Experimental* methodologies.

3.1.2 – Process

As discussed in the introduction, *Gameplay Programming* is the conversion of *Gameplay Mechanic Specifications* to code. Independent of any modelling language or tool, a given *Gameplay Programmer* will formulate and follow a mental process (likely informal) in their implementation work. This project seeks to define and introduce standardization to this process, therefore aspects of a *Process* methodology will be used in its implementation.

As a *Gameplay Mechanic* is a slightly abstract concept, this project first seeks to define a simple cognitive model for defining gameplay mechanics, upon which a software process is formulated for its implementation automation. Born from a desire for improvement to workflow, this project assumes at the end of every task/development-cycle/addition that it is still as incorrect about the final requirements as when it started. In doing so, a strict framework of development, testing, critiquing and refinement is applied in its attempt to reach the proposed software process.

3.1.3 – Build

This project further attempts to implement an example of the developed software process, for which aspects of a *Build* methodology must be used. The best practices detailed by Amaral, et al. (n.d.) are thus dealt with as follows:

- The software process developed is modelled and remodeled as further requirements are discovered.
- While no reusable components exist for adaptive programming, *UNREAL ENGINE* is chosen as the target platform for its extensive library collection and large support community.
- C++ is chosen as the programming language. Alongside adequacy for the task, it brings relevancy to the product's evaluation via its complexity and widespread use.
- Developments are tested as they are made, a fail-fast process which uncovers requirements and identifies problems with the software process before they become too embedded to deal with.
- Code is documented as developed, and *GIT* with *SOURCEFORGE* source control used to manage versions & provide backup services.

Research Methodologies are quite different from *Software Development Methodologies (SDMs)* and, as this project follows a *Build* methodology, so must it follow an *SDM*. This is discussed in the next chapter.

3.2. Software Development Methodology

As a general preamble to this chapter – Discussed within is background research into potentially suitable Software Development Methodologies *and* discussion of the specific one chosen for this project.

3.2.1 – Software Development Paradigms for Prototyping

Every software development methodology is based upon a software development paradigm, such as *Agile*, *Waterfall*, *Incremental*, or *Prototyping* (W. Ambler, n.d.).

Waterfall is a sequential paradigm, consisting of development stages that must be completed in order (Avison & Fitzgerald, 2003). Formalized around 1970, its issues became apparent by the late 1970s with critique centred around *Waterfall's* inability to deal with requirements emergence/volatility (due to its requirements *freezing* at the first stage in development) (Boehm, 2006; McGregor & Korson, 1990; Rajlich, 2006).

In contrast – *Agile*, a paradigm born of the 4 values and 12 principles of *The Agile Manifesto* (Beck, et al., n.d.), is a customer focused paradigm based on valuing people, working software, collaboration and adaptability over other more traditionally sequential requirements (W. Ambler, n.d.). Multiple surveys confirm its adoption provably brings various benefits (State of Agile, 2014; Ambler, 2006), while another study shows precisely how *Agile* addresses *every* element of software quality (Jain, Sharma, & Ahuja, 2018).

Incremental is an evolution of *Waterfall*, often called *Iterative Waterfall*, that applies *Agile* values (Ruparelia, 2010). The software deliverable is broken into 'pieces' or 'components', each of which fulfil a subset of the final requirements (Hilburn & Townhidnejad, 2000). The expected benefits from working on smaller pieces of software are introduced, however the paradigm still suffers from rigid phasing and potential inflexibility in requirements volatility (Ghahrai, 2016; Guru99, n.d.).

Software Prototyping is a paradigm in which application prototypes displaying the functionality, but perhaps not containing the exact logic, of a final product are constructed (Tutorials Point, n.d.).

Multiple sub-types of prototyping fall under this umbrella, such as *Throwaway Prototyping*, wherein poorly understood requirements are made clear during production of a crude prototype, which is then 'thrown away' (Tutorials Point, n.d.; M. Davis & H. Bersoff, 1991). Provided the throw-away step is followed, this allows the final product to be developed with clear requirements, however can put strain on deadlines and time taken.

In contrast, *Evolutionary Prototyping* takes incomplete, but well understood requirements; to produce a prototype which will be added to and extended to create the final version (deemed satisfactory) (Tutorials Point, n.d.; M. Davis & H. Bersoff, 1991; Hugues, Zalila, Pautet, & Kordon, 2008). *Evolutionary Prototyping* can allow missing functionality to be identified during development (Guru99, n.d.), however can often lead to unstructured and difficult to maintain systems (Thompson, Heimdahl, & Miller, 1999).

3.2.2 – Software Development Methodologies For Prototyping

Software Development Paradigms can be seen as methodologies in their own right, however most *underpin* a set of methodologies based on their values and principles, such as *Adaptive Software Development, Feature Driven Development, Iterative and Incremental Development, Rapid Application Development,* and *Scrum*.

Iterative and Incremental Development is largely a blanket term to describe any methodology variant that performs the key development steps in *more* than one development cycle, each building upon the last (Moazeni, Link, & Boehm, 2013). Having been informally used as early as 1957 (Gosling & Bollella, 2003), it enables *Agile* development through continuous learning and application between development cycles (Farcic, 2014).

Rapid Application Development is another somewhat blanket term (lesser than *IID*) for processes characterized by proritization of rapid prototype releases and iterations (Singh, 2019). Coined by James Martin around 1991, it embodies *Agile* development through iterative cycles composed of: *Requirements Planning, User Design, Construction,* and *Cutover* (Agarwal, Prasad, Tanniru, & Lynch, 2000).

Adaptive Software Development, based on *RAD* principles and formalized in 1997 by Jim Highsmith (Highsmith, 1997), assumes that every aspect of development is highly volatile, and that complex systems are difficult to achieve without special consideration taken to enable collaboration between developers and stakeholders (Ramsin & Paige, 2008). Through its Speculate-Collaborate-Learn iteration cycles, it facillitates *Agile* development through continuous adaptation, learning and requirements discovery (Xie, Shen, rong, & Shao, 2012).

Scrum, introduced in 1986 by Hirotaka Takeuchi and Ikujiro Nonaka (Takeuchi & Nonaka, 1986), is based around 'sprints': Time boxed development cycles usually around 2 weeks long (Sutherland & Schwaber, 2020). It enables *Agile* development through team introspection, learning and feedback in reviewing sprint goals and setting new ones (Scrum Alliance, n.d.).

Feature Driven Development, often based around *Scrum* and created in 1999 by Jeff DeLuca and Peter Coad, is focused on delivering individual 'features' – Pieces of functionality that underpin user stories/use cases. By forcing features to be no larger than 2 weeks worth of work, it enables *Agile* development and accurate progress monitoring (Ramsin & Paige, 2008; Abrahamsson, et.al, 2003).

3.2.3 – A Note On Software Development Methodology Usage

Any chosen paradigm cannot be assumed to be the only governing factor in software development success. Avison & Fitzgerald (2003) discuss various contributors to this opinion, including, critically, how methodologies are rarely flexible to the size of a project. However, they offer a critique of this opinion in that this shortcoming lies in the inadequacy of *current* methods, something the publication date corroborates.

Despite this, other studies offer similar critiques: Ruparelia (2010) highlights the quantity of documentation required by *Waterfall*, and LaToza, et.al. (2006) discusses how *Waterfall's* prediction of where developers spend their time is inaccurate.

Regardless of critique and advocation, a reasonable conclusion seems to be that the proposed benefits of any methodology will depend on the individual environment and procedures actually used by the adopter (Fuggetta, 2000; Ruparelia, 2010).

3.2.4 – Software Development Methodology Followed

As discussed in previous chapters, the software product this project deals with the creation of is experimental and based in research *Build* methodology. This makes any methodology based on *Waterfall & Incremental* unsuitable, as they both rely on comprehensive requirements analysis that is unfeasible to possess before development begins.

Agile appears a good fit, with its focus on customer satisfaction and working software oferring an attractive route towards a developed software artifact. However, the benefits of *Evolutionary Prototyping* really shine in the context of this project – Its formal, incremental approach to the well understood but incomplete requirements of this project allow for flexibility and requirements discovery throughout the development process. It is in *that* sense that a continual feedback loop is

used during development, quite akin to *Adaptive Software Development*, which can be considered the methodology for this project.

Another benefit of this methodology is that it makes no assumptions about the size of the development team, something that *Waterfall* and *Agile* often attract criticism for. Choosing a methodology with a focus on requirements discovery instead of team introspection puts a more coherent focus into the *product* (as opposed to the *team*), giving a better chance of success.

The last chapter in this section is a brief discussion of professional, legal, ethical and social issues, before section 4, where the developed method is detailed and discussed.

3.3. Professional, Legal, Ethical and Social Issues

In any ambitious and/or speculative *Build* project, fidelity can become a concern that permeates every action taken. Measures are taken to ensure that pitches/descriptions of capability etc. are truthful and accurate in order to guarantee honesty and faithfulness in the software development community.

The work of Cutumisu, et al. (2007) and this project's direct inspiration presents an issue of intellectual property. However, taking inspiration from and applying similar technology to a different area is not considered a copyright infringement on *SCRIPTEASE*.

A completed PLESI analysis form can be found in Appendix I.

4. The Developed Method

4.1 – Core Definitions & Process Explanation

4.1.1 – What Is a Pattern?

The developed method is based on the assumption that across many different games, the same high-level gameplay mechanics are present, but with unique or different functionality. With this assumption, the concept of *Patterns* can be applied to gameplay mechanics.

The concept of patterns is an abstract one, with varying definitions and mental models in each field it is relevant in (i.e. Software Engineering Design Patterns vs. Game Design Patterns vs. *ScriptEase's* Quest Patterns). Each field's definition of what a pattern is & how they're used is moulded to its specific needs, however one property remains the same across all – A pattern is a high-level '*thing*' that can be copied/reused and tweaked to fit a particular situation.

This idea of what a pattern *is* allows the term to be applied to gameplay mechanics: A Gameplay Mechanic Pattern is a basic, high-level mechanic implementation, capable of being introduced to any game and adapted to fit design requirements. To put this into context, consider the example of *Hades* and *The Binding of Isaac* (Appendix A) – Both games have a gameplay mechanic resembling projectile shooting, therefore we could consider *ShootProjectile* to be a gameplay mechanic pattern. In *Hades*' case, the pattern is adapted by adding hold-to-charge, release-to-fire mechanics. In *The Binding of Isaac*, the pattern is adapted by adding auto-fire mechanics.

From a technical perspective, a pattern is a collection of classes (object-oriented implementation is assumed), each of which have a collection of components/objects representing their internal structure, and a set of mechanics that describe how that class responds to things happening – discussed in the next section.

4.1.2 – What Are Adaptations?

As discussed, in order to make a gameplay mechanic pattern unique, one has to add additional functionality to it. In the games industry, this is often referred to as 'putting a twist' on an existing mechanic. As an example: One could choose the '*ShootProjectile*' pattern and add a '*Ricochet*' mechanic (in that, whenever a *Projectile* collides with something that is not an *Enemy*, it will reflect off that surface instead of exploding and/or dealing damage). This, in its purest form, is an *Adaptation* to a *Pattern*.

Next, consider how an Adaptation can be described in the same way as a singular mechanic. As found during background research, many methods for defining a singular mechanic exist (many overly formal). To adhere to this project's aim of creating a method that is first and foremost easy to use, a mechanic, and thus an adaptation, is defined in the most straightforward way possible: As a set of *Events* which, when they occur, will trigger a set of *Effects*[4].

To put this in context, refer back to the example of *ShootProjectile* in *Hades* and *THE BINDING OF ISAAC*. To accomplish the hold-to-fire, release-to-shoot mechanics shown in *HADES*, one could state two adaptations:

1. **Events**: Fire Button Held Down **Effects**: After a Certain Amount of Time, Allow Firing

2. **Events**: Fire Button Released **Effects**: If Firing is Allowed, Fire, Then Disallow Firing again

Similarly, to accomplish the auto-fire mechanics shown in *THE BINDING OF ISAAC*, one could state a different two adaptations:

1. **Events**: Fire Button Held Down **Effects**: Start a Recurring Timer That Shoots a Projectile

2. **Events**: Fire Button Released **Effects**: Stop the Recurring Timer That Shoots a Projectile

[4] This is directly inspired by work on the *VGDL*, which presents a straightforward way of responding to input. (Thompson, et al., 2013)

4.1.3 – The Developed Process/Method

The two prior definitions allow the overall developed process/method for creating and adding gameplay mechanics to a game to be explained. In order to do this, hypothesize the existence of a large library of patterns – Some examples may include the aforementioned *ShootProjectile*, a generic *PickUp* pattern, maybe a *Dash* pattern, or perhaps an *Inventory* pattern etc. etc.

A user of the process may either have an idea of, or some design requirements for, a gameplay mechanic they want to create. The user could then search the pattern library to find a high-level pattern that most closely represents the essential functionality of what they want. Next, assuming they had access to information about the existing individual mechanics of the pattern, they could write/select/create/specify[5] adaptations to the pattern in order to create a description of the gameplay mechanic they require.

Next, assume that each of these patterns in the library has a tangible, functional code representation for a set of target platforms (i.e. *UNITY, UNREAL ENGINE, An In-House Engine, etc.*). It is entirely possible that a tool could be created to automate the *actual* code modification required to transform the pattern code into code whose functionality matches that of the user's described gameplay mechanic.

Code for each event specified by an adaptation can be generated by overriding functions or generating methods whose signatures allow binding to a message dispatcher of the target platform's event system[6] – Something already possible with tools like *JETBRAINS RESHARPER C++*. Assuming that a similar library of code snippets for effects in a given target platform exists, the code for the effects could be generated as well.

[5] The process by which a user specifies adaptations can be implemented in a wide variety of ways.
[6] For more details on event systems in game engines, the reader is referred to the *UNREAL ENGINE* documentation for message dispatchers.

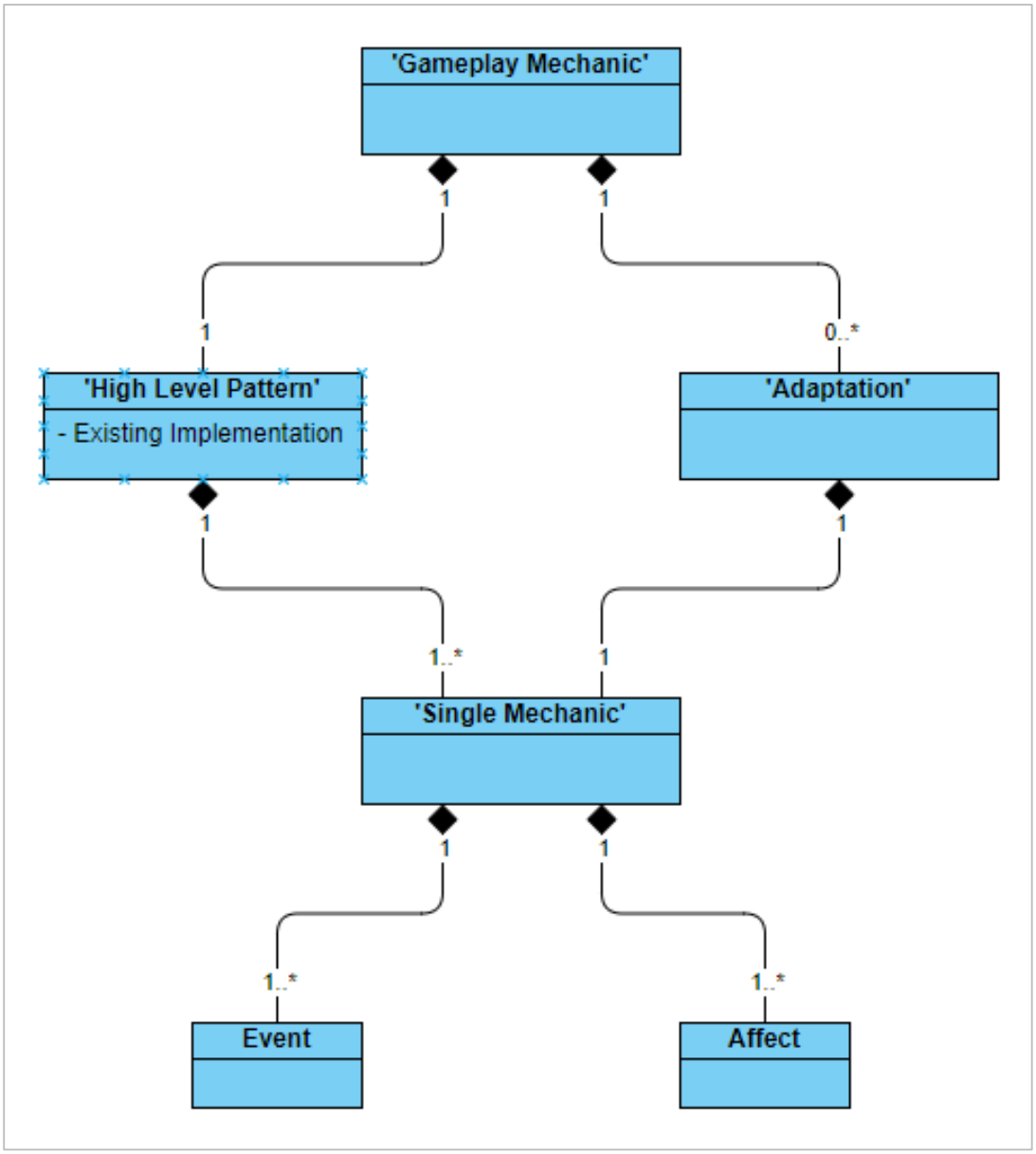
Note that, even if an implementation could only generate the structure (events) and not the implementation (effects), this would still be a tool of significant use.

4.1.4 – An Ontology for Gameplay Mechanics

For clarity, referencing use, and to encapsulate the prior discussions – An ontology diagram can be created to define gameplay mechanics, seen overleaf in figure 7.

Figure 7

An Ontology for Gameplay Mechanics



Note. This figure shows how gameplay mechanics can be represented ontologically.

- A gameplay mechanic is composed of 1 high level pattern, and 0 to many adaptations.
- A high-level pattern is composed of its existing implementation (required because the scope of gameplay mechanics is so broad), and 1 to many single mechanics. (These single mechanics are representative of the existing implementation, and exist solely to represent the pattern in terms of 'mechanics').
- Similarly, 1 adaptation is comprised of 1 'mechanic'
- Each mechanic is comprised of 1 or more events (including preconditions), and 1 or more effects, both of which are platform agnostic.

4.2 – Implementation Examples

Where the last chapter discussed the universal principles of the process, this chapter will discuss The Author's example of how these ideas and concepts can be implemented.

4.2.1 – Description of a Pattern & a Set of Adaptations

Assuming that an implementation will include the automation features discussed, a method of describing both a pattern and a set of adaptations has two requirements:

1. It must be readable by humans.
2. It must be capable of computer interpretation.

The obvious solution here is to use a descriptive language, such as *XML* or *JSON*. Both of these languages allow objects to be described in a generic way, however *JSON* currently holds the lead in terms of human legibility, with *XML* a little more suited to web applications.

With this choice of technology, a *JSON Schema* can be created for what this implementation terms as '*Adaptation Files'* (figure 8). It specifies that each adaptation file must describe exactly one pattern, and a set of zero or more adaptations, each of which consist of a set of events, and a set of effects. This represents the Gameplay Mechanic seen in figure 7.

To put this back into context, consider again the examples of *ShootProjectile* in *HADES* and *THE BINDING OF ISAAC*, which could be represented in adaptation files as seen in figures 9 and 10, respectively.

Figure 8

AdaptationFileSchema.txt

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "",
  "title": "Adaptation File",
  "description": "A Description of a Gameplay Mechanic",
  "type": "object"
  "properties": {
    "pattern": {
      "description": "The High Level Gameplay Mechanic Pattern",
      "type": "string"
    }
  },
  "adaptations": {
      "description": "Adaptations to the Pattern",
      "type": "array",
      "items": {
        "type": "object"
        "properties": {
            "events": {
                "description": "Events List for Adaptation",
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "effects": {
                "description": "Effects List for Adaptation",
                "type": "array",
                "items": {
                    "type": "string"
                }
            }
        },
        "required": [ "events", "effects" ]
      },
      "minItems": 0,
      "uniqueItems": true
    }
  "required": [ "pattern" ]
  "required": [ "adaptations" ]
}
```

Note. This figure shows the contents of a *JSON Schema* file specifying the composition of an adaptation file

Figure 9

HADES' hold-to-charge, release-to-fire projectile mechanic represented in an *Adaptation File*

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["Fire Button Held Down"],"effects":["After a Certain Amount of Time, Allow Firing"]},
        {"events":["Fire Button Released"],"effects":["If Firing is Allowed, Fire, Then Disallow Firing Again"]},
    ]
}
```

Note. This figure shows how the hold-to-charge, release-to-fire projectile mechanic found in *HADES* could be represented in an adaptation file.

Figure 10

THE BINDING OF ISAAC'S auto-fire projectile mechanic represented in an adaptation file

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["Fire Button Held Down"],"effects":["Start a Recurring Timer That Shoots a Projectile"]},
        {"events":["Fire Button Released"],"effects":["Stop The Recurring Timer That Shoots a Projectile"]},
    ]
}
```

Note. This figure shows how the auto-fire projectile mechanic found in *THE BINDING OF ISAAC* could be represented in an adaptation file.

4.2.1.1 – Description of Events

A problem is immediately evident in figures 9 and 10 – The natural language used to describe the events and effects violates the second requirement of the adaptation files, and cannot be interpreted by computers using current techniques[7]. Thus, to re-enable computer interpretation of the files, specification of a declarative scripting language for events and effects is required. This language shares the requirements of the adaptation files, and introduces one more:

3. The terminology used should be platform agnostic

To understand this requirement better, consider the case of *UNREAL ENGINE* and *UNITY*. Both provide events within their collision systems, however use different terminology – `OnTriggerEnter` in the case of *UNITY*, and `OnOverlapBegin` for *UNREAL*. Both provide exactly the same functionality, therefore an event named something like `OnCollisionBegin` may be better suited for platform agnosticism. Note that this is not a concrete requirement, as a given user may prefer the style of one target platform. So long as a label/symbol/term in the language can map to exactly 1 or less[8] events in a target platform, the language is valid and unambiguous (a property required for interpretation).

The *HADES* and *BINDING OF ISAAC* adaptation files could now look something like figures 11 and 12.

[7] Natural language processing is not currently an accessible technology.
[8] If a specified event maps to zero engine events during generation, it can be considered user made, and generated thusly.

Figure 11

HADES' hold-to-charge, release-to-fire adaptation file with reworked events

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["Pressed(Fire)"],"effects":["After a Certain Amount of Time, Allow Firing"]},
        {"events":["Released(Fire)"],"effects":["If Firing is Allowed, Fire, Then Disallow Firing Again"]},
    ]
}
```

Note. This figure shows how the events of the original *HADES* adaptation file can be rewritten as map-able events

Figure 12

THE BINDING OF ISAAC'S auto-fire adaptation file with reworked events

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["Pressed(Fire)"],"effects":["Start a Recurring Timer That Shoots a Projectile"]},
        {"events":["Released(Fire)"],"effects":["Stop The Recurring Timer That Shoots a Projectile"]},
    ]
}
```

Note. This figure shows how the events of the original *THE BINDING OF ISAAC* adaptation file can be rewritten as map-able events

4.2.1.2 – Description of Effects

Following on from the redefinition of how events are described in the language, it is evident that the same problem exists with the description of effects. This can be solved in the same way, but the question of what constitutes a valid effect presents a problem: The list of possible effects is theoretically infinite. This same problem exists with events, but is solved by mapping to override-able/bind-able events first, then considering everything else as user made. There are no map-able effects in game engine, so how is an effect considered valid?

Answering this requires looking ahead to what an implementation would do considering *any* implementation is valid. It turns out that, in most cases, a database mapping effects to functional code implementations or guidance on implementation is the only simple solution. Therefore, creating an implementation requirement of this database allows map-able labels to said database, with any missing effect deemed as 'not created yet' for the target platform's implementation.

With this in mind, the adaptation files of *HADES* and *THE BINDING OF ISAAC* may look something like figures 13 and 14. Note the additional event in the example of *HADES* – `IsAllowed` – Which can be considered as a precondition, implying a logical *AND* connecting the two events.

Figure 13

Hades' hold-to-charge, release-to-fire adaptation file with reworked effects

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["Pressed(Fire)"],"effects":["SetTimer(Allow(Firing))"]},
        {"events":["Released(Fire)","IsAllowed(Firing)"],"effects":["Fire","Disallow(Firing)"]},
    ]
}
```

Note. This figure shows how the effects of the original *Hades* adaptation file can be rewritten as map-able effects

Figure 14

The Binding of Isaac's auto-fire adaptation file with reworked effects

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["Pressed(Fire)"],"effects":["SetTimerLoop(Fire)"]},
        {"events":["Released(Fire)"],"effects":["StopTimerLoop(Fire)"]},
    ]
}
```

Note. This figure shows how the effects of the original *The Binding of Isaac* adaptation file can be rewritten as map-able effects

4.2.1.3 – Specifying Which Class Handles What

As discussed in 4.1.1, a pattern (in technical requirements) is composed of a collection of classes. Suppose that the *ShootProjectile* pattern being discussed contains two classes: A *Projectile* class, and a *ShootProjectileManager*. Currently, the language has no way of specifying which class is responsible for which event or effect.

In order for code to follow object oriented principles, functionality contained within a given class should be relevant to that class only, unless it is a class with the expressed purpose of managing or interacting with other classes. For this language to fulfil its aims of producing code with no structural defects, it must adhere to these object oriented principles.

This is an easy problem to find a solution for, and simply by specifying which class is responsible for each event with a common *member* operator, the *HADES* and *THE BINDING OF ISAAC* examples may then look like figures 15 and 16.

Note that in both cases, this adds the ability to distinguish between a parameter:

(aka. *Fire* in *Pressed(Fire)*)

and an event:

(aka. *ShootProjectileManager.Allow(…)*)

but also allows an event to be passed *as* a parameter:

(aka. *ShootProjectileManager.Allow(…)* in *ShootProjectileManager.SetTimer(…)*).

With this final addition, a user can accurately describe a pattern and a set of adaptations. Note that the adaptation files are now very verbose, a negative to human legibility. For this, a GUI representing choices in a more readable way is desired. The example implementation does not include one, however its benefits are discussed in more detail shortly.

Figure 15

HADES' hold-to-charge, release-to-fire adaptation file with specified classes

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["ShootProjectileManager.Pressed(Fire)"],"effects":["ShootProjectileManager.SetTimer(ShootProjectileManager.Allow(Firing))"]},
        {"events":["ShootProjectileManager.Released(Fire)","IsAllowed(Firing)"],"effects":["ShootProjectileManager.Fire","ShootProjectileManager.Disallow(Firing)"]},
    ]
}
```

Note. This figure shows how the adaption file for *HADES* can be rewritten, specifying which class handles what

Figure 16

THE BINDING OF ISAAC'S auto-fire adaptation file with specified classes

```
{
    "pattern":"ShootProjectile",
    "adaptations":
    [
        {"events":["ShootProjectileManager.Pressed(Fire)"],"effects":["ShootProjectileManager.SetTimerLoop(ShootProjectileManager.Fire)"]},
        {"events":["ShootProjectileManager.Released(Fire)"],"effects":["ShootProjectileManager.StopTimerLoop(ShootProjectileManager.Fire)"]},
    ]
}
```

Note. This figure shows how the adaptation file for *THE BINDING OF ISAAC* can be rewritten, specifying which class handles what

4.2.2 – Anatomy of A Pattern

The only requirement of an implementation of patterns is that they must contain a functional implementation and a *specification file* (discussed shortly). Any other supplementary files used by the target platform generation and adaptation file are irrelevant to the platform agnostic language side of the implementation.

4.2.2.1 – Using Template Files for The Functional Implementation

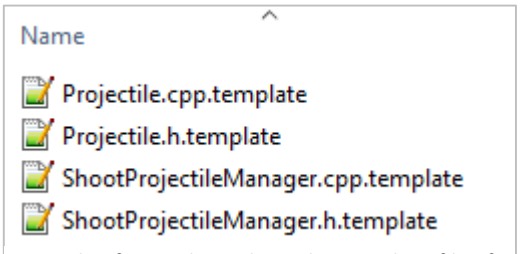
As with any implementation of an abstract concept, the use of standard conventions and technologies improves community understanding and potential use of said implementation. For that reason, a good way to approach the inclusion of a functional implementation is via use of template files.

Template files bring many benefits to the table, perhaps the biggest of all: their widespread usage. At a technical level, they are indistinguishable from C++ source and header files, save for tags, usually surrounded by a pair of %'s (e.g. %CLASS_NAME%) – Intended to be replaced when the template files are copied into a project's source code collection.

Continuing with the *ShootProjectile* example, consider how the two assumed classes (*Projectile* and *ShootProjectileManager*) may look like figure 17.

Figure 17

Template files for *ShootProjectile*



Note. This figure shows how the template files for the *ShootProjectile* pattern may look in context.

4.2.2.2 – Specification Files to Describe the Functional Implementation

Automating the direct reading and interpretation of meaning from code files is a highly complex problem – One that may be unfeasible given the size of community this method is aimed at. Instead, *specification files* can be created, describing the technical elements of a pattern – The component/objects comprising structure, and mechanics, of each class. With these, the complex operation of deducing exactly how to modify a pattern's code files to fit a described set of adaptations can be brought into the realm of achievability.

Specification files contain an inherently platform specific element, in the sense that target platforms may provide different elements for composing the structure of a class. As the specification files must describe the structure of each class in the pattern *exactly,* to be useful in code adaptation, so must they specify *exactly* what type each object in a class' structure is, introducing terminology from the target platform.

Again continuing with the *ShootProjectile* example, a specification file following these rules may look like figure 18. Note that, as is evident in the figure, the target platform for this specification file is *UNREAL ENGINE*.

Note here the benefit a GUI would bring to users when describing unique mechanics, as this would allow for suitable visual obfuscation of platform specific details while providing knowledge of the existing mechanics of the pattern. Additionally, a GUI could provide visual tools instead of written, meaning a user would never have the need to interact with a specification file *or* an adaptation file.

Figure 18

A specification file for the *ShootProjectile* pattern

```
{
    "classes":
    [
        {
            "name":"Projectile",
            "parent":"Actor",
            "components":
            [
                {
                    "name":"SphereComponent",
                    "parent":"SphereComponent",
                    "components":
                    [
                        {
                            "name":"StaticMesh",
                            "parent":"StaticMeshComponent"
                        }
                    ]
                },
                {
                    "name":"ProjectileMovementComponent",
                    "parent":"ProjectileMovementComponent"
                }
            ],
            "mechanics":
            [
                {
                    "events":["Projectile.Tick"],"effects":["SphereComponent.Move(Forward)"]
                },
                {
                    "events":["SphereComponent.OnBeginOverlap(Any)"],"effects":["Projectile.Destroy","Any.Damage"]
                },
                {
                    "events":["Projectile.Destroyed"],"effects":["Projectile.Spawn(Explosion_Particle)"]
                }
            ]
        },
        {
            "name":"ShootProjectileManager",
            "parent":"ActorComponent",
            "mechanics":
            [
                {
                    "events":["ShootProjectileManager.Pressed(Fire)"],"effects":["ShootProjectileManager.Spawn(Projectile)","ShootProjectileManager.Cooldown"]
                }
            ]
        }
    ]
}
```

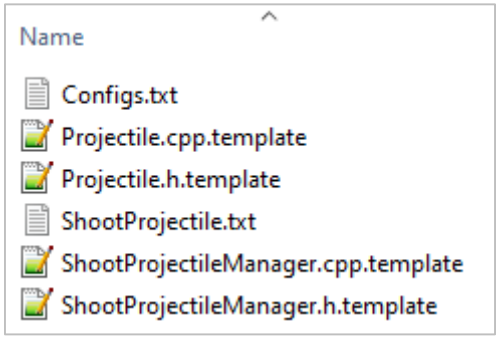
Note. This figure shows how the *ShootProjectile* pattern can be represented with a specification file.

4.2.2.3 – Supplementary Files

Starting with figure 19, showing a pattern in its entirety, note the supplementary *Configs.txt* file present.

Figure 19

The example *ShootProjectile* pattern files



Note. This figure shows the pattern files for *ShootProjectile* in their entirety

Each target engine implementation is unique, so the data requiring pre-specification by a pattern's creator will differ as well. In this case, the inputs required by the *ShootProjectile* pattern are described in a config file, shown overleaf in figures 20 and 21, which the *UNREAL ENGINE* implementation uses to edit its input settings during pattern code generation (discussed shortly). The *ShootProjectile* pattern only requires the input shown overleaf, however this format could be used for other configurations, such as the collision system's *Object & Trace Channels.*

Figure 20

Configs.txt

```
{
    "type":"input",
    "name":"Fire",
    "action":true,
    "mappings":
    [
        "LeftMouseButton"
    ]
}
```

Note. This figure shows the *Configs.txt* file, how a set of configuration file edits can be expressed in *JSON* notation.

Figure 21

The Required *ShootProjectile* Pattern Input

Note. This figure shows the Fire input required by the *ShootProjectile* pattern's code.

4.2.3 – A Generation & Adaptation System

4.2.3.1 – Pattern Code Generation

Provided with an adaptation file, the first step is to ‘generate’ the pattern code. As the template files already exist, this is a trivial copying operation from the pattern to the source code directory. The user does this by clicking a ‘Generate Code’ button (the adaptation file chosen is hardcoded currently). Secondly, the configs file is read and handled (discussed prior). At this point, the adaptations described in the adaptation file can be read and handled.

4.2.3.2 – Deduction of Intent from Adaptations

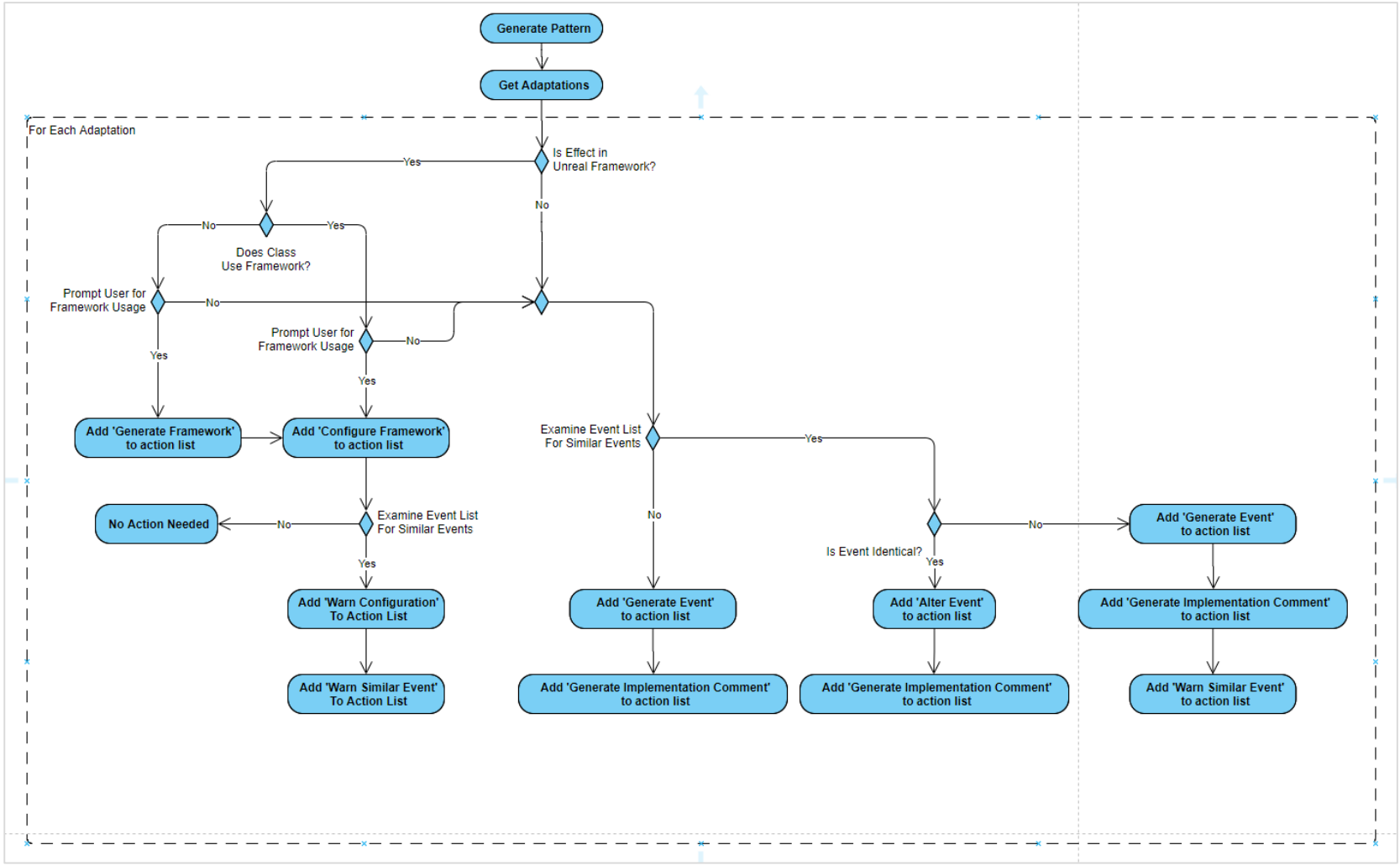
Following the feedback loop methodology discussed previously, a final version of a logical process by which intent can be deduced from adaptations is shown in figure 22. (A full evolution of the process can be found in Appendix B). It should be noted that the activity diagram does not represent the actual code implementation 1:1, as further requirements were discovered during implementation. Code for the function modelled can be found in Appendix C.

UNREAL ENGINE has a rare property of Game Engines in that it offers a small set of application specific functionality (such as the *ProjectileMovementComponent*), owing to its heritage as the engine behind *UNREAL TOURNAMENT*. This functionality, hereafter referred to as *UNREAL Framework*, is often the 'preferred' way to do things, which means the demonstration must apply the overarching principle of using the 'best-practices' approach and use them. This is accomplished by prompting the user to select whether they would like to use the framework or not (done with popups), followed by adding relevant items to the action list (discussed shortly).

The right side of the activity diagram shows the actions to be taken if the framework cannot or should not be used, wherein the specific actions to be taken depend on the presence of similar or identical events.

Figure 22

The Final Logical Process to Deduce Intent from Adaptations



Note. The above activity diagram shows how a list of individual actions can be deduced from a set of adaptations and a specification file.

4.2.3.3 – The Action List

In the development of any gameplay mechanic, the process taken by its programmer could follow this format:

1. Create base mechanic.
2. Add unique functionality.

By mapping the base mechanic creation to the pattern generation, the addition of unique functionality can be considered to be the adaptation stage – A task suitable to be broken down into smaller, logical steps (as may be done in practice by a programmer). The developed implementation uses this probable thought process as the basis for its code modification.

For an adaptation that can be created using the *UNREAL Framework*, it may first have to be added to the class, configured, then possible pre-existing configurations (of the same framework) and other events checked for conflicting functionality.

Similarly, a non-framework adaptation could be as simple as creating the event and writing the functionality within. It could be the case that the event needed already exists, therefore instead of generating the event, it can be altered to include the new functionality. Finally, if just a similar event were to exist, the new event could be generated and implementation written, then the similar events checked for conflicting functionality.

The final action list, created as a result of following this process, is then sanitized to remove any duplicate items, and followed/executed in order to create the adaptations desired. The crucial property of this action list is that each item is a single, concise task, capable of varying degrees of automation, the highest of which is still achievable.

For full explanations of each action list item in detail, see Appendix D.

4.2.3.4 – Code File Modification

The penultimate step taken by the implementation must be modifying the code files. The higher complexity, more ambitious route is to modify the code files with functional implementations for each effect, however as previously discussed, generating just the events (skeleton code) and placing comments within them detailing implementation guidance for the effects would still prove to be a useful tool. Each of the action list items (full explanations in Appendix D) encapsulates a single task, described below.

Note that these are specific to the *UNREAL ENGINE* implementation, and will differ for an alternative target platform – They also detail a system that generates comments only, as time constraints did not allow for full code modification work to take place.

Generate Framework

Add the specified *UNREAL* framework component to the class' header file, and generate a default initialization within the class' constructor for said component.

Configure Framework

Add a comment in the class' constructor, underneath the specified *UNREAL* framework component's initialization, explaining how to configure it to achieve the effect specified.

Warn Configuration

Add a comment in the class' constructor, underneath the specified *UNREAL* framework component's initialization, indicating to the user that the new configuration for the specified effect may conflict with an existing, specified effect.

Warn Similar Event

Add a comment in the class' source file definition for the event specified, indicating to the user that the effect implemented in the function may conflict with the new, specified effect.

Generate Event

Generate/override the specified event in the class' header and source file. If the event requires binding to a delegate, then do so in the class' constructor.

Generate Implementation Comment

Generate a comment in the class' source file definition for the event specified, informing the user what functions they can use to implement the specified effect.

Alter Event

Generate a comment in the class' source file definition for the event specified, informing the user that this event must be modified to implement the specified effect.

4.2.3.5 – Compilation & Integration

In any compiled program, such as most game engines, code introduced to a source code directory will not be visible or usable by the program until it is recompiled. In order to make the workflow as seamless as possible, a developed implementation is encouraged to perform this step automatically after code file modification, or at least instruct the user to do so.

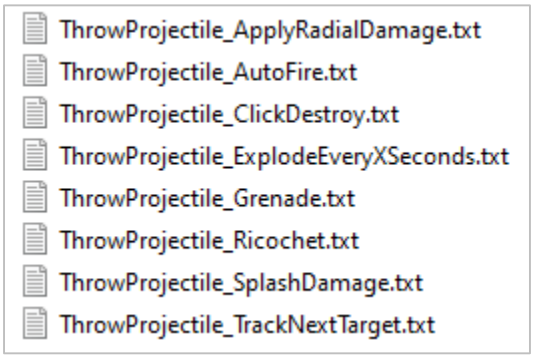
Finally, the user of the workflow must be instructed on how to utilize the new code within their game. In the *UNREAL ENGINE* implementation, this is done by instructing the user to make *BLUEPRINT* sub-classes of the generated classes. With these, the user can easily add them to a different class of their choosing, making for efficient integration. A developed implementation is encouraged to make use of the target engine's best practices, but make this process as smooth as possible.

5. Implementation Results & Testing

Where the last section discussed opportunities for and recommendations of implementations for the developed process, this section will detail the workings of and testing of the sample implementation created for demonstration. The sample implementation was created around 8 distinct use cases, formed of adaptation files (figure 23), which were used to test the final version of the system.

Figure 23

The 8 use-cases the implementation was tested with.



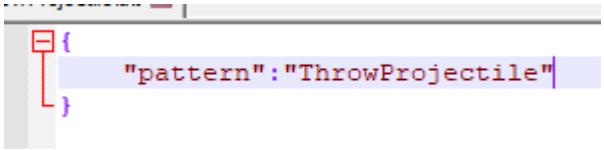
Note. This figure shows the 8 use-case adaptation files the sample implementation was tested with.

Overall, the testing for this project was done as part of each *Speculate-Collaborate-Learn* cycle performed for each task during development. Each tasks' testing, performed in the *Learn* segment, served to: Test the functionality works as expected; Inform any new discovery of overall project requirements; And inform the subsequent *Speculate* segment and task.

The generation portion of the system was the first developed and had the clearest requirements, therefore the testing took the form of simple use cases which were developed for and tested. Attention was not given to edge cases beyond what was needed for the demonstration, as these had the potential to consume far more development time than was feasible. Take, for example, figures 24 and 25, which show *Unreal Engine's* output log being used to test the reading of adaptation files correctly:

Figure 24

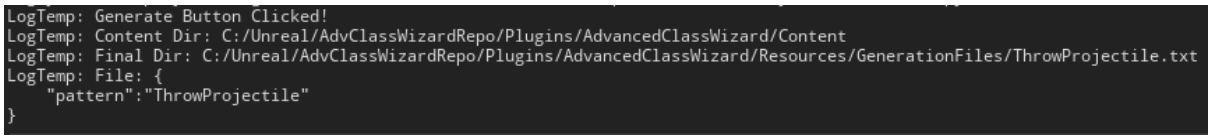
The tested Adaptation File



Note. This figure shows an early version of an Adaptation File, containing only a pattern specification, to test the *Json* reading of the Pattern field.

Figure 25

The Output Log used to test reading the pattern field.

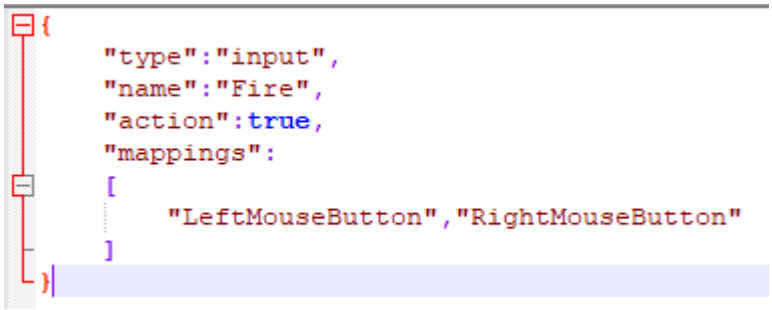


Note. This figure shows the output log's usage in testing the above Adaptation File.

This method of testing served the development of the generation portion well, and the frequency with which it was conducted allowed bugs to be found and fixed often as quickly as in the next development cycle. For example, an issue that caused the configuration file edits to not save (whose testing is shown in figures 26 and 27) was caught during the next development cycle.

Figure 26

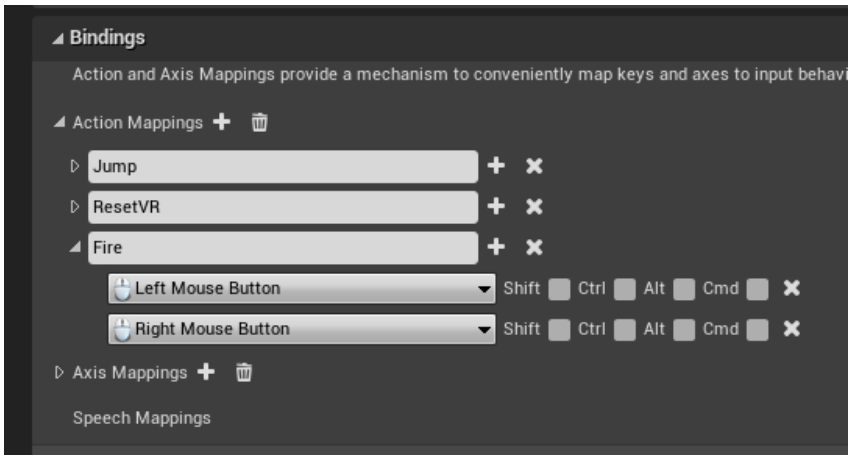
The tested Configs file



Note. This figure shows a version of a Configs file, used to test the automated configuration file editing.

Figure 27

The post-generation inputs configuration, used to test reading the Configs file.



Note. This figure shows the input configurator inside *Unreal Engine's* use in testing the automated configuration file editing.

Much of the early testing of the developed process was based around ensuring the technical specification for what had to be created was both feasible and useful. Long *Speculate* and *Collaborate* portions were used to perform exploratory 'deep-dives' into a potential route. For example – One of the first 'deep-dives' was done around how a complete C++ source and header file might be generated from just a specification file, seen in figure 28 (for the full development journal excerpt, see Appendix G). It was this process that helped refine the requirements of the project away from code generation/adaptation & towards logical deduction.

Figure 28

The Specification File Used for the Deep-Dive

```
{
    "classes":
    [
        {
            "name":"Projectile",
            "parent":"Actor",
            "elements":
            [
                {
                    "inputs":["Tick"],"effects":["Move(Forward)"]
                },
                {
                    "inputs":["Collision(Any)"],"effects":["Destroy(This)","Spawn(Explosion_Particle)","Damage(Any)"]
                }
            ]
        },
        {
            "name":"ThrowProjectileComponent",
            "parent":"ActorComponent",
            "elements":
            [
                {
                    "inputs":["Press(Fire)","Delay(0.25)"],"effects":["Spawn(Projectile)"]
                }
            ]
        }
    ]
}
```

Note. This figure shows an early Specification File used to inform and perform a deep-dive into how a C++ source and header file might be generated for the classes specified within.

The actual logical deduction process developed for the *Unreal Engine* generation and adaptation system was created by designing a hypothetical activity diagram for the *ThrowProjectile_Ricochet* use case (similar to *ShootProjectile*). This initial diagram (figure 26) then went through 7 (8 in total) development cycles, each consisting of a different use case (e.g. *ThrowProjectile_ExplodeEveryXSeconds, ThrowProjectile_AutoFire*). In these cycles, the logical process was 'tested' by comparing a manually deduced action list for the adaptation's creation against the result of blindly following the activity diagram. In cases where the activity diagram failed to identify necessary actions or perhaps broke altogether, this information was used to refine and produce evolutions of the activity diagram (see Appendix B for the full evolution process). This resulted in the final logical process seen in figure 22.

Note also that these evolutions informed changes to the syntax of specification and adaptation files, seen in figures 29 & 30, respectively.

Figure 29

The First Version of *ThrowProjectile_SplashDamage*

```
{
    "pattern":"ThrowProjectile",
    "adaptations":
    [
        {"events":["OnBeginOverlap(Projectile, Any)"],"effects":["Destroy(This)","Spawn(Explosion_Particle)","RadialDamage"]}
    ]
}
```

Note. This figure shows the first version of the *ThrowProjectile_SplashDamage* adaptation file. Note the differences in syntax between this and figure 30

Figure 30

The Final Version of *ThrowProjectile_SplashDamage*

```
{
    "pattern":"ThrowProjectile",
    "adaptations":
    [
        {"events":["Projectile.OnBeginOverlap(Any)"],"effects":["Projectile.Destroy","Projectile.RadialDamage"]}
    ]
}
```

Note. This figure shows the final version of the *ThrowProjectile_SplashDamage* adaptation file. Note the differences in syntax between this and figure 29

In each use-case, the required adaptation file was passed as a parameter to the generation & adaptation system. In all cases, after fixing bugs highlighted as a result of said testing, the system generated base pattern code, asked users for any prompts on *UNREAL* framework components, and generated a logically correct action list (relative to a manual deduction of action list creating during the evolution of the activity diagram), outputted to the log. In some cases, the system identified additional action list items that were not found during manual deduction, suggesting immediate benefit to non-automated workflows.

The full testing excerpts for this portion can be found in Appendix H.

The generated pattern code can be found in Appendix J.

6. Product Evaluation

As a general preface to evaluation of the developed product, User Testing was considered unsuitable. The exploratory nature of the project means that the concepts requiring evaluation at the point of development's conclusion are more abstract than would be suitable for any Human-Computer-Interface testing that could be performed. For this reason, the product's evaluation is centered around its applicability and usefulness as a process, and not its 'traditional' usability.

6.1 – The Declarative Language

Despite being taken for granted throughout the rest of this report, the declarative language introduced within the adaptation & specification files is novel, and requires evaluation of its usefulness relative to existing methods.

The first point of note is that the developed language successfully avoids the overly formal approach present in the work of Zook & Riedl (2014) – And provides a declarative approach, contrary to the constructive like syntax of *PHYDSL* (Guana & Stroulia, 2014). In doing so, the language embraces the accessibility of *MINIGAME* (Maranhão, Franco, Junior, & Maia, 2016) and the clear Input/Output structure found in *VGDL* (Thompson, et al., 2013).

While the former 2 works have significant uses in their fields, the developed declarative language combines the aforementioned aspects of the latter 2, and presents a simple, easy to use language that is (crucially) capable of and suitable for not just the definition and analyzing of existing mechanics, but for the *construction* of new mechanics – Something that prior methods have somewhat lacked.

6.2 – The Process As A Workflow

Despite its incompleteness, the developed demonstration presents a somewhat novel workflow, therefore its perceived usefulness must be evaluated relevant to existing workflows.

The major relevant point of evaluation is that of comparison to *ScriptEase* (Cutumisu, et al., 2007) – As the concept of Adaptive Programming applied to content authoring processes was concretely proven in their case study. While following the principles set forth in said work, the developed product falls short in a few areas.

Firstly, the lack of a graphical user interface impacts not just the usability, but the obfuscation with which platform specific terminology can be abstractly dealt with by a user in a platform *agnostic* fashion. Despite an explanation in time constraints, this point can't be omitted as a negative for evaluation purposes.

Secondly, *SCRIPTEASE* supports the modification of generated code after its creation. This is undoubtedly a crucial aspect of it's success, considering the agreed incremental nature of games development. The developed product does not include this due to 2 barriers: One, the aforementioned lack of a GUI with which to implement editing – And two, the lack of development in the area of *actual* code manipulation and editing, which was left unexplored in this product. Both barriers could be solved with additional research and development, with the second verging on a critical success factor should this process and tool ever be widely adopted.

6.3 – The Process as a Tool

An alternative area of evaluation comes from comparison to existing *tools*, such as *UML* and *MACHINATIONS*.

The benefits over *MACHINATIONS* is easily identifiable, with the developed workflow and language far more capable of describing granular/individual or avatar-centric mechanics. The criticism of specialism focus in economic mechanics can't be ignored however, as the developed workflow is certainly incapable of modelling even the basic types of economic situations *MACHINATIONS* can.

Similarly, the benefits relative to *UML* are fairly easy to identify. Considering its researched lack of suitablility for creative applications, the developed process and workflow offers a considerable improvement towards modelling gameplay mechanics. In partcular, the ability to describe both structure *and* functionality in the same model (adaptation files) cannot be overstated in importance where usefulness relative to *UML* is concerned.

6.4 – An Additional Example of Use

In an effort to further prove the usability of the workflow and process, this report presents an alternative example to the *ThrowProjectile* pattern produced.

Consider how a *PickUp* pattern might work, with a *PickUp Actor* and a *PickUpComponent ActorComponent* to handle them. A possible specification file may be similar to figure 31 – With a possible adaptation file adding a *Glowing* effect every few seconds and a period of *Invulnerability* shown in figure 32.

Along with showing clearly how structure and functionality can be described and edited quickly, this presents a few more items for evaluation:

Firstly, one might question the prior knowledge of the pattern's workings required by the user before writing an adaptation file. The developed product has no answer to this, other than re-affirm the aim of process demonstration – A GUI would clearly allow for knowledge of the pattern to be presented to the user during adaptation file creation, akin to *ScriptEase*.

Secondly, the developed product is incapable of dealing with many complex language features, such as the parameter passing shown in figures 31 & 32. This would be solvable with the use of a Recursive Descent Parser when interpreting the language, however development of this was far outside time constraints.

Finally, many will notice the Object-Oriented Programming violations in figures 30 and 31, with the *PickUpComponent* being asked to, among others, *AddHealth* and *AddInvulnerability* – Seemingly irrelevant operations for the component. To this, the developed product has no answer, as the consideration of interfacing with classes the pattern has no knowledge of would have consumed unfeasible amounts of development time.

Figure 31

PickUp.txt

```
{
    "classes":
    [
        {
            "name":"PickUp",
            "parent":"Actor",
            "components":
            [
                {
                    "name":"SphereComponent",
                    "parent":"SphereComponent",
                    "components":
                    [
                        {
                            "name":"StaticMesh",
                            "parent":"StaticMeshComponent"
                        }
                    ]
                }
            ],
            "mechanics":
            [
                {
                    "events":["SphereComponent.OnBeginOverlap(Pawn)"],"effects":["PickUpComponent.PickUp(This)"]
                }
            ]
        },
        {
            "name":"PickUpComponent",
            "parent":"ActorComponent",
            "mechanics":
            [
                {
                    "events":["PickUpComponent.PickUp(PickUp)"],"effects":["PickUpComponent.AddHealth"]
                }
            ]
        }
    ]
}
```

Note. The above figure shows a hypothetical specification file for a Pick-Up pattern.

3d v – Evaluation Conclusion

The developed process and product both have clear and obvious benefits over the methods and tools it is designed to challenge, however is limited in scope due to the development time it could be allocated. This will be discussed partially in the next chapter, after which the project's conclusions will be presented.

Figure 32

PickUp_GlowAndInvuln.txt

```
{
    "pattern":"PickUp",
    "adaptations":
    [
        {"events":["PickUp.BeginPlay"],"effects":["PickUp.StartTimerLoop(PickUp.Glow)"]},
        {"events":["PickUpComponent.PickUp(PickUp)"],"effects":["PickUpComponent.AddInvulnerability","PickUpComponent.StartTimer(PickUpComponent.RemoveInvulnerability)"]},
    ]
}
```

Note. The above figure shows how the *PickUp* pattern might be used and adapted to have the pickup glow every few seconds, and apply a brief period of invulnerability.

7. Overall Project Evaluation

7.1 – Encountered Problems

As with any project, problems will be encountered and their resolutions impact the course of progression – This one was no exception.

Perhaps the largest problem throughout development was a recurrent one from the very beginnings of proposal and development: Scope. In retrospect, the initial project proposal of a tool to generate and adapt *any* functional mechanic is very clearly out of reach for a 6 month, 1-person project. The short *Speculate-Collaborate-Learn* development cycles followed as part of the *ASD* methodology followed helped with identifying out-of-scope routes of development quickly – However, it was from collaboration with and guidance from this project's supervisor's experience in the field that scope was most effectively curtailed. (Note that all specialist support facilities were deemed unsuitable for this project's topic).

The limiting of scope was instrumental in ensuring the project produced something meaningful, however had it been done sooner, more might have been produced as part of the demonstration. This problem in particular has served to cement the author's knowledge and experience in project planning and scoping, particularly where larger projects are concerned.

A second problem, one that any *UNREAL ENGINE* developer will be acquainted with, is the lack of support available for *specific* areas of development within the engine. For example, there are next to no tutorials or documentation available for *SLATE* development – A mandatory technology used by *UNREAL ENGINE Editor Plugin* User Interfaces. In light of this challenge, experience had to be quickly gained and utilized in searching the engine source code for examples in how to accomplish things, something that even experienced *SLATE* developers recommend as the only support available (ben, 2017).

7.2 – Methodology Used

Having followed the *Process, Build* and *Adaptive Software Development* methodologies throughout the project's development, it is clear that they brought numerous benefits.

In particular, the *Speculate-Collaborate-Learn* development cycles of *ASD* combined with the comprehensive development journal kept allowed for extensive requirements discovery throughout the project's duration. This enabled the research methodology focused portion of the project to shine in the creation of the novel workflow and process discussed.

However, the project may well have fallen foul of the extended time requirements known to be caused by *Evolutionary Prototyping* based methodologies – Evident in the partially finished state of the demonstration product. In retort, it is unlikely the project would have been able to present the novel ideas it did had one not been followed.

As with any project, additional pre-existing requirements knowledge may have enabled more quantity of product to be produced, however re-iteration of the impossibility of this knowledge is key here. It was in this sense that a *MoSCoW* analysis *could* have been produced, but would likely have rapidly become irrelevant in light of the aforementioned requirements discovery.

Discrepancy exists in comparison to the initial planning of the project. What seemed at first glance a difficult but easily definable project transformed into incomplete requirements and complex problems, causing the development gantt chart created to become irrelevant very quickly. It was the following of *ASD* that allowed this project to bear fruit at all.

In this sense, the project planning could be considered a mild failure, but not in a way that conveys any negativity towards the project – This was the author's first experience with *ASD* and research project methodologies.

7.3 – The Final Product

As discussed in the product evaluation, the developed process presents novel ideas and offers a potential improvement over existing methods. In terms of broad research area, this project sits in the niche created by the works of Cutumisu, et al. (2007) – It deals squarely with the concept of Adaptive Programming, applies it to a content authoring process (Gameplay Programming) and avoids straying into the more formal research areas, such as the AI mechanic tools discussed by Zook & Riedl (2014).

Somewhat disappointingly, but absolutely expected given the 1-person workforce and time available, is this project's failure to produce a tangible piece of software that could be user tested instead of evaluated theoretically. This leads to the project final status as a 'Towards …' project, providing research and development into its areas, but requiring *further* research and development to fully realize its potential.

Numerous avenues for improvement exist within both the product and the process.

Firstly, the developed declarative language, while a tool with significant arguable benefits, would benefit greatly from formalization and specification detailing exactly what is and isn't possible. It's entirely possible that a GUI would lead towards this anyway, due to the discussed requirement for platform-agnostic terminology. However, considering language features such as the *<object.method/member>* or *<method(parameter)>* conventions, formal definitions would push the declarative language in the direction of having a quick guide for users to understand exactly how to use it.

A point worth noting here is that scripting languages often balloon with features until they are indistinguishable from the languages they were meant to simplify. Future work would have to exercise caution with every new feature added, testing against a non-programmer user-base every time. This

would further lead to concrete requirements specifications for any target engine generation and adaptation system, something that the product demonstration had to contend without.

The obvious future work opportunity presented is of finishing the *Unreal Engine* implementation. The creation of functionality for interpreting each action list item would allow for tangible code modification, pushing the technology of this project towards viability. It may also inform further changes to the action list deduction process presented. Combining this with investigation into different patterns would be a worthwhile investigation into the suitability of this workflow for different archetypes of mechanics and also provide an opportunity to test how this tool would work in open source.

From the beginnings of this project's conception, it was known that the greatest benefit to be gained is from open source, where the gameplay programming community can come together to: Improve the technology, submit patterns, continually update platform's effects databases with either recommended implementations or actual code. It's in this vision that the benefit to the community really lies, with an ambitious view of a tool that could potentially automate the work of a gameplay programming mentor.

The final chapter of this report will present conclusions from its entirety, and present the recommended courses for future work.

8. Conclusions

This project set out to perform research and development into an application of adaptive programming to gameplay programming. As part of this, extensive research was undertaken into where the aforementioned technology sits at the present moment, including drawing extensive inspiration from the works of Cutumisu, et al. (2007) and Thompson, et al. (2013). From this, clear evidence of advocation and justification for a tool fulfilling this project's aims was discovered, with ample gameplay mechanics definition work to underpin the technology. Along with research into research and software development methodologies, a plan was created and followed to produce the work detailed within.

Of that, a novel, declarative language was created and discussed – It's ease of use and clarity of expression for both functional and structural aspects of gameplay programming evident in comparison to *UML* and *MACHINATIONS*. To demonstrate its applicability and practicality, a partial implementation of a generation and adaptation system for *Unreal Engine* was created, reaching the point of deducing an action list – A concrete list of actions to take – From adaptation files specifying a pre-existing high-level mechanic pattern and a set of adaptations. A user could follow this action list manually to produce the unique mechanic specified by the adaptation file, or in a hypothetical future scenario, gain the result of this automatically from a fully finished implementation.

While the project succeeded in the context of producing novel ideas and technologies with which to advocate for further development, it did not achieve its full, ambitious aim of actually producing the technology and testing it on users. Whilst tempting to look for blame in the project methodologies followed, the researched domain complexity of gameplay programming and foray into research methodology oriented development suggests success of the project overall, especially where product evaluation shows the benefits of what *was* produced to be significant.

The project presents the concepts, language, schemas for adaptation files, and the sample target engine implementation for *UNREAL ENGINE* as its major contributions, and encourages that future work be done to advance and utilize the technology and process discussed, in order to achieve the project's more ambitious aims of standardizing the field of gameplay programming:

- The completion of the sample implementation for *UNREAL ENGINE*, with which user testing and an experiment into open sourcing the technology could be performed.
- A formalization of the declarative language created, allowing for concrete validity checking of adaptation files, and finite requirements specifications for target engine implementations.
- Investigations into other mechanic archetypes, testing the robustness of the process and language across a broader slice of the gameplay programming domain.

9. References

Abrahamsson, P., Warsta, J., Siponen, M. T., & Ronkainen, J. (2003). New Directions on Agile Methods: A Comparative Analysis. *Proceedings of the 25th International Conference on Software Engineering* (pp. 244-254). Portland: IEEE Computer Society.

Agarwal, R., Prasad, J., Tanniru, M., & Lynch, J. (2000). Risks of Rapid Application Development. *Commun. ACM*, 1.

Aleem, S., Capretz, L. F., & Ahmed, F. (2016). Critical Success Factors to Improve the Game Development Process from a Developer's Perspective. *Journal of Computer Science and Technology*, 925-950.

Amaral, J. N., Buro, M., Elio, R., Hoover, J., Nikolaidis, I., Salavatipour, M., . . . Wong, K. (n.d.). About Computing Science Research Methodology.

Ambler, S. (2006, August 2006). *Survey Says: Agile Works in Practice*. Retrieved from Dr. Dobb's The World of Software Development: https://www.drdobbs.com/architecture-and-design/survey-says-agile-works-in-practice/191800169?queryText=agile+survey

Avison, D. E., & Fitzgerald, g. (2003). Where Now for Development Methodologies? *Commun. ACM*, 78-82.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . Thomas, D. (n.d.). *Manifesto for Agile Software Development*. Retrieved from Agile Manifesto: https://agilemanifesto.org/

ben. (2017, May 21). *Creating new UWidgets in C++*. Retrieved from benui: https://benui.ca/unreal/ui-cpp-uwidget/

Björk, S. (n.d.). *Main Page*. Retrieved from Game Design Patterns: http://virt10.itu.chalmers.se/index.php/Main_Page

Blow, J. (2007). Game Development: Harder Than You Think. *Association for Computing Machinery*, 28-37.

Boehm, B. (2006). A View of 20th and 21st Century Software Engineering. *Proceedings of the 28th International Conference on Software Engineering* (pp. 12-29). Shanghai: Association for Computing Machinery.

Cook, M. (2013). Creativity in Code: Generating Rules for Video Games. *XRDS*, 40-43.

Cutumisu, M., Onuczko, C., McNaughton, M., Roy, T., Schaeffer, J., Schumacher, A., . . . Gillis, S. (2007). ScriptEase: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 32-58.

Farcic, V. (2014, January 21). *Software Development Models: Iterative and Incremental Development*. Retrieved from Technology Conversations: https://technologyconversations.com/2014/01/21/software-development-models-iterative-and-incremental-development/

Fuggetta, A. (2000). Software Process: A Roadmap. *Proceedings of the Conference on The Future of Software Engineering* (pp. 25-34). Limerick: Association for Computing Machinery.

Ghahrai, A. (2016, September 3). *Software Development Methodologies*. Retrieved from DevQA: https://devqa.io/software-development-methodologies/

Gosling, J., & Bollella, G. (2003). Iterative and Incremental Development: A Brief History. *Computer*, 47-56.

Guana, V., & Stroulia, E. (2014). PhyDSL: A Code-generation Environment for 2D Physics-based Games.

Guru99. (n.d.). *Incremental Model in SDLC: Use, Advantage & Disadvantage*. Retrieved from Guru99: https://www.guru99.com/what-is-incremental-model-in-sdlc-advantages-disadvantages.html

Guru99. (n.d.). *Software Engineering Prototyping Model*. Retrieved from Guru99: https://www.guru99.com/software-engineering-prototyping-model.html

Highsmith, J. (1997, January). *MESSY, EXCITING, AND ANXIETY-RIDDEN: ADAPTIVE SOFTWARE DEVELOPMENT*. Retrieved from AdaptiveSD:

https://web.archive.org/web/20171004140236/http://www.adaptivesd.com/articles/messy.htm

Hilburn, B., & Townhidnejad, M. (2000). Software Quality: A Curriculum Postscript? *SIGCSE Bull.*, 167-171.

Hugues, J., Zalila, B., Pautet, L., & Kordon, F. (2008). From the Prototype to the Final Embedded System using the Ocarina AADL Tool Suite. *ACM Trans. Embed. Comput. Syst.*, Article 42.

Jain, P., Sharma, A., & Ahuja, L. (2018). The Impact of Agile Software Development Process on the Quality of Software Product. *International Conference on Reliability, Infocom Technologies and Optimization* (pp. 812-815). ICRITO.

Kanode, C. M., & Haddad, H. M. (2009). Software Engineering Challenges in Game Development. *2009 Sixth International Conference on Information Technology: New Generations*, 260-265.

Kasurinen, J. (2016). Games as Software - Similarities and Differences beteween the Implementation Projects. *International Conference on Computer Systems and Technologies*, (pp. 33-40).

Kounoukla, X.-C., Ampatzoglou, A., & Anagnostopoulos, K. (2016). *Implementing Game Mechanics with GoF Design Patterns.*

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining Mental Models: A Study of Developer Work Habits. *Proceedings of the 18th International Conference on Software Engineering* (pp. 492-501). Shanghai: Association for Computing Machinery.

M. Davis, A., & H. Bersoff, E. (1991). Impacts of Life Cycle Models on Software Configuration Management. *Commun. ACM*, 104-118.

Maranhão, D. M., Franco, A. d., Junior, G. M., & Maia, J. G. (2016). Towards a Comprehensive Model for Analysis and Definition of Game Mechanics. *SB Games*, (pp. 581-590). São Paulo.

McGregor, J., & Korson, T. (1990). Understanding Object-Oriented: A Unifying Paradigm. *Commun. ACM*, 40-60.

Moazeni, R., Link, D., & Boehm, B. (2013). Incremental Development Productivity Decline. *Proceedings of the 9th International Conference on Predictive Models in Software Engineering* (p. Article 7). Baltimore: Association for Computing Machinery.

Montero Reyno, E., & Á. Carsí Cubel, J. (2008). Model-Driven Game Development: 2D Platform Game Prototyping. *GAMEON*, (pp. 5-7). Valencia.

Pellens, B., De Troyer, O., & Kleinermann, F. (2008). CoDePA: a conceptual design pattern approach to model behavior for X3D worlds. *Proceedings of the 13th international symposium on 3D web technology* (pp. 91-99). Los Angeles: Association for Computing Machinery.

Rajlich, V. (2006). Changing the Paradigm of Software Engineering. *Commun. ACM*, 67-70.

Ramsin, R., & Paige, R. F. (2008). Process-Centered Review of Object Oriented Software Development Methodologies. *ACM Comput. Surv.*, Article 3.

Roedavan, R., Agus, P., Korio Utoro, R., & Putri Sujana, A. (2020). Zetcil: Game Mechanic Framework for Unity Game Engine. *International Journal on Artificial Intelligence Tools*, 96-105.

Ruparelia, N. B. (2010). Software Development Lifecycle Models. *SIGSOFT Softw. Eng. Notes*, 8-13.

Sarinho, V. T., & Apolinário, A. L. (2009). A Generative Programming Approach for Game Development. *VIII Brazilian Symposium on Games and Digital Entertainment* (pp. 83-92). Rio de Janeiro: IEEE.

Scrum Alliance. (n.d.). *Your Quick Guide to All Things Scrum*. Retrieved from Scrum Alliance: https://resources.scrumalliance.org/Article/quick-guide-things-scrum

Singh, A. (2019, December 6). *What Is Rapid Application Development (RAD)?* Retrieved from Capterra: https://blog.capterra.com/what-is-rapid-application-development/#:~:text=Rapid%20Application%20Development%20(RAD)%20is,strict%20planning%20and%20requirements%20recording.

State of Agile. (2014, August 28). *Why Agile?* Retrieved from State of Agile Version One: https://web.archive.org/web/20140828012224/http://stateofagile.versionone.com/why-agile/

Sutherland, J., & Schwaber, K. (2020, November). *The 2020 Scrum Guide*. Retrieved from Scrum Guides: https://scrumguides.org/scrum-guide.html

Takeuchi, H., & Nonaka, I. (1986, January). *The New New Product Development Game*. Retrieved from Harvard Business Review: https://hbr.org/1986/01/the-new-new-product-development-game

Thompson, J. M., Heimdahl, M. P., & Miller, S. P. (1999). Specification-Based Prototyping for Embedded Systems. *SIGSOFT Softw. Eng. Notes*, 163-179.

Thompson, T., Ebner, M., Schaul, T., Levine, J., Lucas, S., & Togelius, J. (2013). Towards a Video Game Description Language. *Dagstuhl Follow-ups*, 85.

Tutorials Point. (n.d.). *SDLC - Software Prototype Model*. Retrieved from Tutorials Point: https://www.tutorialspoint.com/sdlc/sdlc_software_prototyping.htm

Van Rozen, R. (2020). Languages of Games and Play: A Systematic Mapping Study. *ACM Computing Surveys*, Article 123.

W. Ambler, S. (n.d.). *Examning the Agile Manifesto*. Retrieved from Ambysoft: http://www.ambysoft.com/essays/agileManifesto.html

Walker, J. (2018). Class diagram. *Design-UML 08/02/2018.* Cambridge. Retrieved from https://jjrwalker.github.io/general/UMLdesign/

Xie, M., Shen, M., rong, G., & Shao, D. (2012). Empirical Studies of Embedded Software Development Using Agile Methods: a Systematic Review. *Proceedings of the 2nd international workshop on Evidential assessment of software technologies* (pp. 21-26). Lund: Association for Computing Machinery.

Zhu, M., & Inge Wang, A. (2019). Model-driven Game Development: A Literature Review. *ACM Computing Survey*, Article 123.

Zook, A., & Riedl, M. O. (2014). Generating and Adapting Game Mechanics.

10. Appendices

Appendix A – An Analysis of the Gameplay Mechanics Present in *Hades* and *The Binding of Isaac*

The Binding of Isaac

- Gameplay Framework
  - Opening Cutscene
  - Character Selection
- Game Systems
  - Level Progression
    - Rooms
    - Door Locking/Unlocking
    - Procedural Generation
  - Thing Spawning
    - Enemy Spawning
    - Heart spawning from enemy death
    - Item Spawning
    - Block Spawning
  - Mini-Map
  - Stats
  - Shop
    - Buy items in game
- Gameplay
  - Health
    - Hearts
    - Shields
  - Simple Collection Inventory
    - Coins
    - Bombs
    - keys
    - Shop
  - Movement
    - Top Down
  - Attack
    - Projectile
    - Bomb placement + explosion
  - Perks
    - Active
      - (single controllable hovering persistent projectile)
      - (change attack type to poison)
      - (Change projectile pattern)
  - Abilities
    - Unique Collectables
      - Petrify Enemies
  - Pickups
    - Attack Modifiers (Perks)
    - Stat Boosters
      - HP Increase
      - Stat Booster
      - Shields Giver
    - Collectables on ground
      - Hearts
      - Coins
      - Keys
  - AI
    - Enemy AI
      - Movement
      - Attacking
        - Melee attack
        - Projectile attack patterns
        - Beam Attack
        - Bomb placement
    - Boss AI
      - Movement
      - Attacking
        - (All of Enemy AI + specifics)
    - Companion AI
      - Movement
  - Environment Blocks
    - Destructible Items
      - Rocks
      - Pots
      - Turds
      - Fires
    - Impassable Items
      - Rock Walls
    - Hazards
      - Spikes
    - Chests
      - Normal
      - Golden Turds
    - Doors

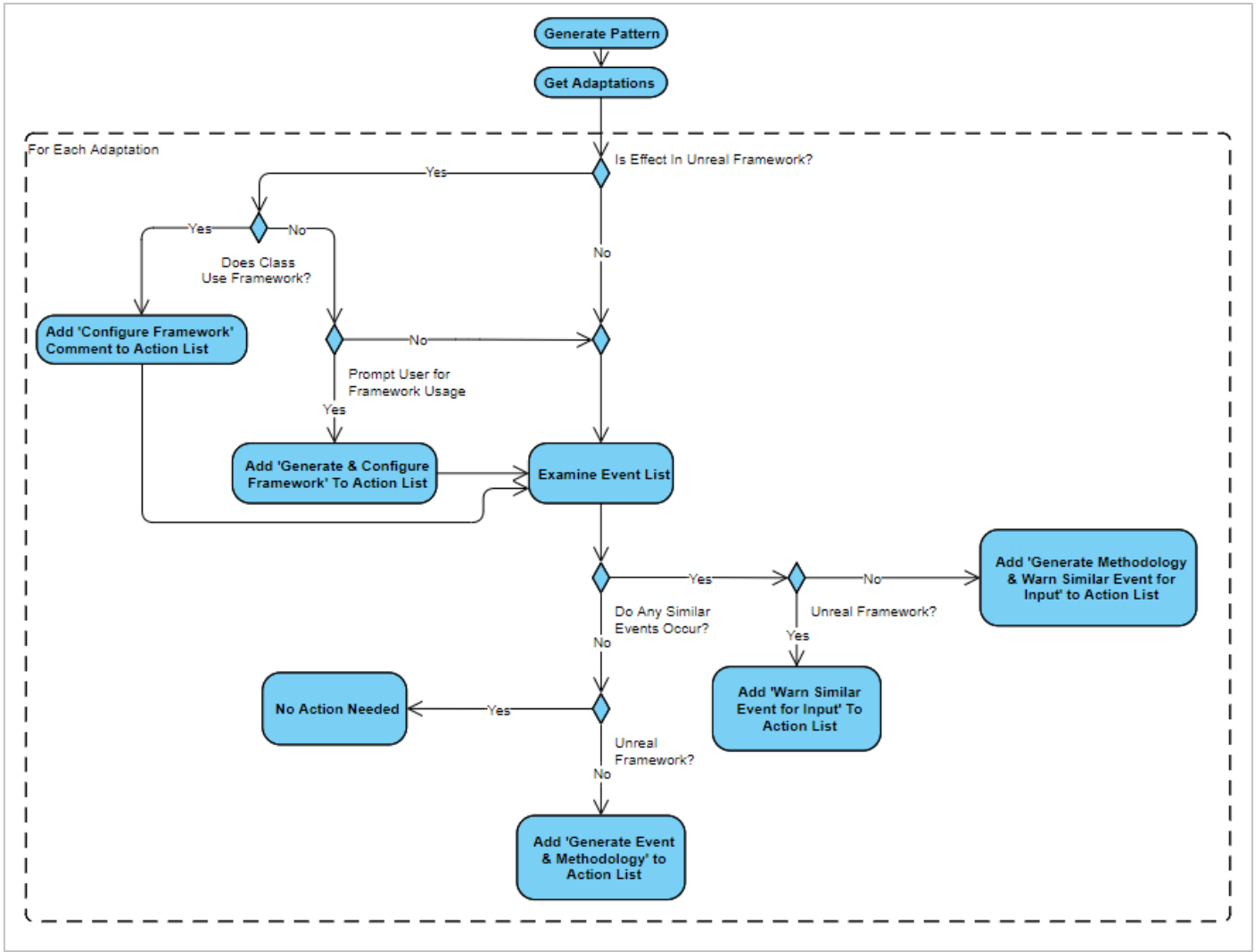
Hades

- Gameplay Framework
  - Opening Cutscene
- Game Systems
  - Dialogue Barks
  - Level Progression
    - Rooms
    - Door Locking/Unlocking
  - Thing Spawning
    - Enemy Spawning
    - Dialogue Thing Spawning
    - Coin Spawn on Enemy Death
    - Pickup Spawning
    - Reward Spawn on Door Open
  - Stats
  - Shop
    - Buy stats between games
    - Buy Weapons between games
    - Buy Items in Game

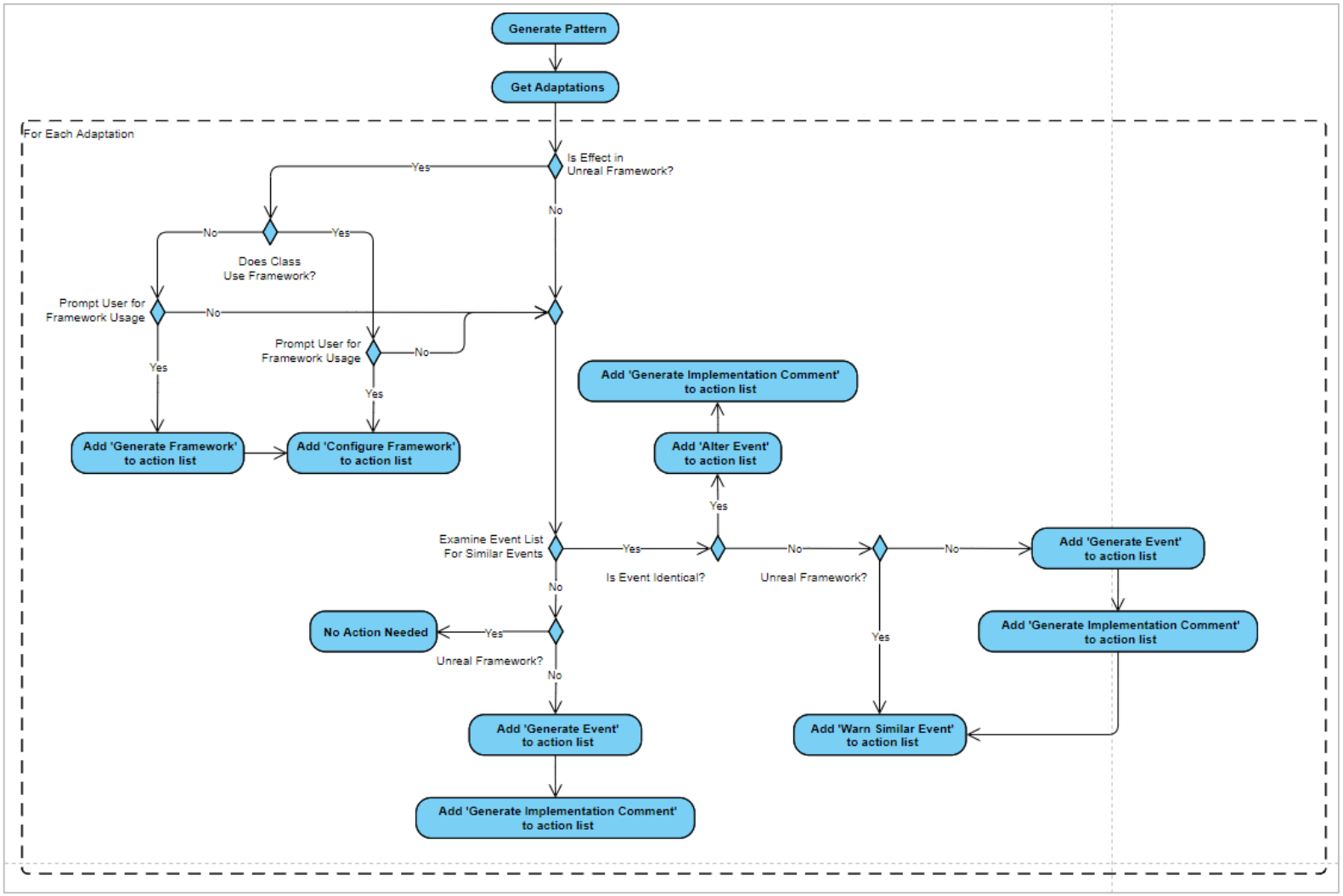
- Gameplay
  - Health
    - Health Bar
  - Simple Collection Inventory
    - Ammo
    - Gems
    - Coins
  - Movement
    - Isometric
  - Attack
    - Melee
      - Stab
      - Knockback
      - Wall slam *(From Knockback)*
    - Ranged
      - Single Arrow Charge-up
      - Arrow Pattern
    - Cast *(ranged attack, requires ammo)*
    - Special Attack *(Depends on Weapon)*
  - Abilities
    - Dash
    - Blade Rift *(AoE)*
  - Perks
    - Active
      - Increase Attack/Cast/Dash Damage
      - Give Additional Attack to Attack/Dash/Cast
      - Give Status Curse to Attack/On-Damage
    - Passive
      - Restore Health Between Rooms
      - Second Wind
  - Dialogue
    - Perk Selector
  - Pickups
    - Coins
    - Gems
    - Ammo
    - Health
    - Keys
  - AI
    - Passive AI
      - Movement
    - Enemy AI
      - Activation
      - Health
        - Health Bar
      - Movement
      - Attack
        - Melee
        - Projectile Shoot
        - Bomb lob
        - Dash
  - Environment Blocks
    - Destructible
      - Pots
      - Walls
    - Doors
    - ‘Chests’
    - Hazards
      - Spikes
      - Turrets
      - Explosive ‘Barrels’

Appendix B – The Full Evolution of the Action List Deduction Process

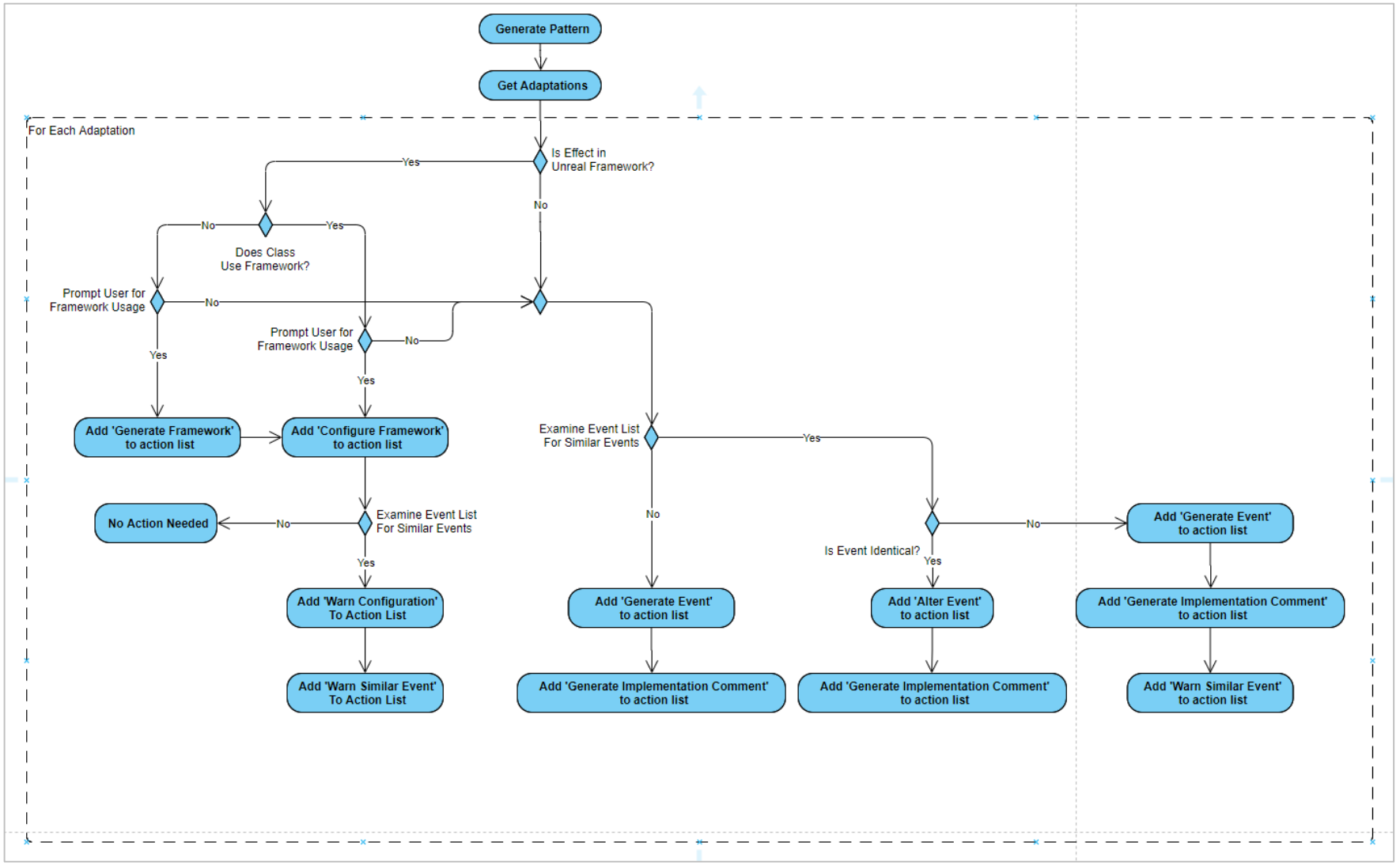
Version 1 – 4th February



Version 2 – 10th February



Final Version (3) – 16th February



Appendix C – Code Implementation of the Action List Deduction Process

```
void AdaptationManager::GenerateActionList()
{
	UE_LOG(LogAdaptationManager, Log, TEXT("Generate Action List:"));

	// Load Event Conflicts and Effect Conflicts DT
	EventConflictsDT = Cast<UDataTable>(StaticLoadObject(UDataTable::StaticClass(), NULL, TEXT("DataTable'/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'")));
	EffectConflictsDT = Cast<UDataTable>(StaticLoadObject(UDataTable::StaticClass(), NULL, TEXT("DataTable'/AdvancedClassWizard/DT_EffectConflicts.DT_EffectConflicts'")));

	// Make sure the action list is empty.
	ActionList.Empty();

	for (const FAdaptation Adaptation : GenerationFileObject->GetAdaptations())
	{
		// I need to process unreal framework generations individually, but I can process the non framework generations together.
		// So I need to create two arrays here.
		// ^ The first has all the individual framework generations
		// ^ The second isn't an array, it's just the rest of the effects
		// I can then run the framework generations through that process, and the non-frameworks through the other.

		// Calculate Individual Framework-Generations needed, and filter adaptation to structure generations
		const TArray<FAdaptationEffectPair> FrameworkGenerations = ExtractFrameworkGenerations(Adaptation);
		const TArray<FAdaptationEventPair> StructureGenerations = ExtractStructureGenerations(Adaptation, FrameworkGenerations);

		for (const FAdaptationEffectPair& FrameworkGeneration : FrameworkGenerations)
		{
			const FString Effect_RelevantClass = GenerationUtils::ExtractTopLevelClass(FrameworkGeneration.Effect);
			const FString Effect_Event = GenerationUtils::ExtractTopLevelEvent(FrameworkGeneration.Effect);
			const FString Effect_Framework = GenerationUtils::GetFrameworkDependency(EffectsDT, Effect_Event);

			// Add 'Generate Framework' Action if framework does not already exist.
			const FFrameworkDecision* Decision = FrameworkDecisionMap.Find(FrameworkGeneration);
			if (Decision && !Decision->bFrameworkAlreadyPresent)
			{
				AddActionListItem_GenerateFramework(Effect_RelevantClass, Effect_Framework);
			}

			// Add 'Configure Framework' Action
			AddActionListItem_ConfigureFramework(Effect_RelevantClass, Effect_Framework, Effect_Event);

			// Check for conflicting Mechanics
			const TArray<TPair<FAdaptation, float>> ConflictingMechanics = CheckConflicts(FrameworkGeneration, Effect_RelevantClass);
			UE_LOG(LogAdaptationManager, Log, TEXT("Print Conflicts for Framework Generation: %s, %s: %s"), *FrameworkGeneration.Adaptation.ToString(), *FrameworkGeneration.Effect,
*Effect_RelevantClass);
			PrintConflicts(ConflictingMechanics);

			// Add warning comment action list item generations
			for(const TPair<FAdaptation, float>& Conflict : ConflictingMechanics)
			{
				if (Conflict.Value <= FLT_EPSILON)
					continue;

				// Add warn configuration to action list for each conflicting mechanic found
				AddActionListItem_WarnConfiguration(Effect_RelevantClass, Effect_Framework, Conflict.Key, Conflict.Value, FrameworkGeneration.Adaptation);

				// Add warn similar event to action list for each event in the conflicting mechanic
				for(const FString& Event : Conflict.Key.Events)
				{
```

```
                        AddActionListItem_WarnSimilarEvent(Effect_RelevantClass, Event, Conflict.Key.GetEffects(), Conflict.Value, FrameworkGeneration.Adaptation);
                }
        }
}

// Run structure generation
for (const FAdaptationEventPair& StructureGeneration : StructureGenerations)
{
        const FString& Event_RelevantClass = GenerationUtils::ExtractTopLevelClass(StructureGeneration.Event);

        // Get conflicting events
        const TArray<TPair<FAdaptationEventPair, float>> OtherMechanics = CheckConflicts(StructureGeneration, GenerationUtils::ExtractTopLevelClass(StructureGeneration.Event));

        // Find the highest conflict value to find out what we need to generate.
        const TPair<FAdaptationEventPair, float>& OtherMechanic = [&]()
        {
                int HighestValueIndex = 0;
                float HighestValue = 0.f;
                int TempIndex = 0;
                for (const auto& Other : OtherMechanics)
                {
                        if (Other.Value > HighestValue)
                        {
                                HighestValue = Other.Value;
                                HighestValueIndex = TempIndex;
                        }
                        ++TempIndex;
                }
                return OtherMechanics[HighestValueIndex];
        }();

                        // Deal with the generation appropriately.
                        if (OtherMechanic.Value <= FLT_EPSILON)
                        {
                                // Add Generate Event to action list
                                AddActionListItem_GenerateEvent(Event_RelevantClass, StructureGeneration.Event);

                                // Add Generate Implementation Comment to action list
                                AddActionListItem_ImplementationComment(Event_RelevantClass, StructureGeneration.Event, StructureGeneration.Adaptation.GetEffects());
                        }
                        else
                        {
                                if (OtherMechanic.Value >= 100.f - FLT_EPSILON)
                                {
                                        // Add Alter Event to action list
                                        AddActionListItem_AlterEvent(Event_RelevantClass, OtherMechanic.Key.Event, OtherMechanic.Key.Adaptation.GetEffects());

                                        // Add Generate Implementation Comment to action list
                                        AddActionListItem_ImplementationComment(Event_RelevantClass, OtherMechanic.Key.Event, StructureGeneration.Adaptation.GetEffects());
                                }
                                else
                                {
                                        // Add Generate Event to action list
                                        AddActionListItem_GenerateEvent(Event_RelevantClass, StructureGeneration.Event);

                                        // Add Generation Implementation Comment to action list
                                        AddActionListItem_ImplementationComment(Event_RelevantClass, StructureGeneration.Event, StructureGeneration.Adaptation.GetEffects());

                                        // Add Warn Similar Event to action list.
                                        AddActionListItem_WarnSimilarEvent(Event_RelevantClass, OtherMechanic.Key.Event, OtherMechanic.Key.Adaptation.GetEffects(), OtherMechanic.Value,
StructureGeneration.Adaptation);
                                }
```

```
                    }

                }
            }
            PrintActionList(ActionList);
}
```

Appendix D – Action List Item Explanations

Generate Framework

Format:

GENERATE IN [class] FRAMEWORK [dependency_name]

Explanation:

Add the *UNREAL ENGINE Framework Component* specified by *dependency_name* to the class specified by *class* as a member variable and add a default initialization for it in the class' constructor.

Configure Framework

Format:

COMMENT IN [class] CONFIGURE [dependency_name] IMPLEMENTATION [effect]

Explanation:

Add a comment in the class specified by *class*' constructor instructing the user to alter the configuration of the *UNREAL ENGINE Framework Component* specified by *dependency_name* to enable the effect specified by *effect*. The exact comment to be generated will be queried from the *Unreal_Framework_Comment* column of the *Effects* database (Appendix E).

Warn Configuration

Format:

COMMENT IN [class] CONFIGURE [dependency_name] WARN [conflict_adaptation]

CONFLICT [conflict_percentage] [new_adaptation]

Explanation:

Add a comment under the configuration of the *UNREAL ENGINE Framework Component* specified by *dependency_name* in the class specified by *class'* constructor, informing the user that there is a *conflict_percentage* chance of a conflict between an existing adaptation *conflict_adaptation* and the adaptation currently being generated *new_adaptation*. Conflicts are determined from the *EventConflicts* database (Appendix F).

Warn Similar Event

Format:

COMMENT IN [class] EVENT [event] WARN [effect] CONFLICT [conflict_percentage]

[new_adaptation]

Explanation:

Add a comment in the function specified by *event* in the class specified by *class* informing the user that there is a *conflict_percentage* chance of a conflict between the existing effect *effect* and the adaptation currently being generated *new_adaptation*. Conflicts are determined from the *EventConflicts* database (Appendix F).

Generate Event

Format:

GENERATE IN [class] EVENT [event]

Explanation:

Add the declaration and definition for the event *event* inside the header and source file of the class *class.* If *event* is a *Delegate Event*[9], this will also include binding the generated event to said *Delegate*.

Generate Implementation Comment

Format:

COMMENT IN [class] EVENT [event] IMPLEMENTATION [effects]

Explanation:

Add a comment in the function *event* within the source file for *class* instructing the user how to go about creating the logic for each effect specified by *effects*. The exact comment to be generated will be queried from the *Comment* column of the *Effects* database (Appendix E).

[9] See Appendix G for an analysis of the events and delegates of *Unreal Engine*

Alter Event

Format:

COMMENT IN [class] EVENT [event] ALTER [effects]

Explanation:

Add a comment in the function *event* within the source file for *class* instructing the user that the function must be altered to create the logic for each effect specified by *effects*. Comments for how to go about creating each effect will also be generated – Queried from the *Comment* column of the *Effects* database (Appendix E).

Appendix E – The Effects Database Used for the Sample Implementation

	Row Name	Unreal Framework Dependency	Unreal Framework Comment	Comment
1	SetMovementDirection			Changing the movement direction of an entity will always mean modifying the velocity of said entity. If a UMovementComponent derivative is being used, then there will likely be a method to directly change its velocity e.g.: NavMovementComponent::RequestDirectMove Some derivatives, such as ProjectileMovementComponent, may not have such a method, so will need their Velocity value setting directly. If a derivative is not being used, then the same principle of directly setting a velocity value can be used.
2	Move			To 'Move' is simply to change position with respect to time. The most common implementation of move will be with a specific derivative of UMovementComponent, however a simple modification of location every frame can be used to 'move' a component. (velocity)
3	Ricochet	ProjectileMovementComponent	Ricochet movement is provided in UProjectileMovementComponent. You can configure this with bShouldBounce, and other associated properties.	Ricochet is a colloquial term for reflection. Given a surface, and an object with velocity leading into the suface: The angle between the surface normal and the velocity can be used as the reflection angle, allowing a calculation of the object's velocity after reflection.
4	PhysicsMovement	ProjectileMovementComponent	ProjectileMovementComponent allows for physics like movement. Configure this with bSimulationEnabled and other associated properties.	Any component that has collision can be configured to use Unreal's physics engine. It will first need its physical properties e.g. Mass, Drag configured. Physics Movement can be enabled and disabled by setting bSimulatePhysics on that component to true and false, respectively. Physics Movement will also apply to any children of the component if they are set to relative location and/or rotation.
5	AddImpulse			Any component (derivative of UPrimitiveComponent) can have an impulse applied to it with: UPrimitiveComponent::AddImpulse. This will likely be used in combination with Physics Movement to give objects movement. Add Impulse can also be handled by some UMovementComponent derivatives, such as: CharacterMovementComponent::AddImpulse In general, the movement component methods should be used over the component ones. If implementing this manually, Newtonian equations (F = MA). Impulse equations (dP = F * dT), and Momentum Equations (P = MV) can be used to calculate a new velocity for an object
6	Spawn			Most spawning is done through the 'UWorld': UWorld::SpawnActor However, some more specific spawning i.e. ParticleEmitters is handled by helper libraries e.g.: UGameplayStatics::SpawnEmitterAtLocation or UAIBlueprintHelperLibrary::SpawnAIFromClass
7	Destroy			'Destroying' an object is handled by Unreal Engine's underlying systems (assuming this object inherits from UObject). Typically, the larger classes will have their own destruction methods: AActor::DestroyActor() UActorComponent::DestroyComponent() If you're unsure on which destruction method to use, consult your class's hierarchy in the Unreal Documentation to find the 'closest' 'Destroy'-like method.
8	RadialDamage			Applying radial damage is one the cases handled by Unreal's GameplayStatics Damage System. Use UGameplayStatics::ApplyRadialDamage(...)
9	StartTimerLoop			Each 'UWorld' in Unreal has a TimerManager, which has to be used to set and manipulate timers. Accessing the TimerManager can be done through a variety of class specific functions, e.g: AActor::GetWorldTimerManager Alternatively, get the current world (GetWorld()), and then use: UWorld::GetTimerManager FTimerManager::SetTimer can then be used to set a looping timer.
10	StopTimerLoop			Each 'UWorld' in Unreal has a TimerManager, which has to be used to set and manipulate timers. Accessing the TimerManager can be done through a variety of class specific functions, e.g: AActor::GetWorldTimerManager Alternatively, get the current world (GetWorld()), and then use: UWorld::GetTimerManager FTimerManager::SetTimer can then be used to stop a looping timer (given a handle produced when the timer was created)
11	StartTimer			Each 'UWorld' in Unreal has a TimerManager, which has to be used to set and manipulate timers. Accessing the TimerManager can be done through a variety of class specific functions, e.g: AActor::GetWorldTimerManager Alternatively, get the current world (GetWorld()), and then use: UWorld::GetTimerManager FTimerManager::SetTimer can then be used to set a timer.

Appendix F – The Events Conflict Database Used for the Sample Implementation

	Row Name	Conflicts
1	OnBeginOverlap	("OnBeginOverlap" = 100.000000, "Tick" = 25.000000)
2	Tick	("Tick" = 100.000000, "OnBeginOverlap" = 25.000000)
3	Pressed	("Pressed" = 100.000000)
4	Released	("Released" = 100.000000)
5	Spawned	("Spawned" = 100.000000)

Appendix G – Code Generation from Specification File Deep Dive Diary Excerpt

```
{
    "classes":
    [
        {
            "name":"Projectile",
            "parent":"Actor",
            "elements":
            [
                {
                    "inputs":["Tick"],"effects":["Move(Forward)"]
                },
                {
                    "inputs":["Collision(Any)"],"effects":["Destroy(This)","Spawn(Explosion_Particle)","Damage(Any)"]
                }
            ]
        },
        {
            "name":"ThrowProjectileComponent",
            "parent":"ActorComponent",
            "elements":
            [
                {
                    "inputs":["Press(Fire)","Delay(0.25)"],"effects":["Spawn(Projectile)"]
                }
            ]
        }
    ]
}
```

Where would *I* start to build this?

> Take *Tick* and *Move(Forward)*, I should first look to the Unreal framework. This combination implies *ProjectileMovementComponent*, so instead of progressing further into a generation, I can use that.
>
> > This implies the creation of the *ProjectileMovementComponent* member, and its configuration in the construction script
> >
> > This *also* comes with dependencies. For instance, the *ProjectileMovementComponent* required an *UpdatedComponent*, therefore I must create a *Root SceneComponent* to be updated.
> >
> > > Again, this means I have to add that as a member, and configure it in the construction script, so that it is the root component.
>
> This is an example of an *Adaptation Pattern*, or a *Sub-Level Pattern*. An existing implementation of the set of *Atomic Actions: Tick & Move(Forward).*
>
> > It might be argued that the assumption of desire for the *ProjectileMovementComponent* is incorrect, and leads to a *specialized* pattern that is *inflexible* and *difficult to extend*. In rebuttal, I would say that not just for the *ThrowProjectile* pattern, but for the concept of *Tick & Move(Forward)*, there is no alternative interpretation, as for other *Adaptation Patterns*, there are *different* terminologies. The aim of the tool is to guide the user towards the correct one, as their mistake of choosing *ProjectileMovementComponent* in code is analogous with the in-tool situation.
>
> Now, if I was to generate this code from scratch, the first thing would be to collect a list of member variables.
>
> > The first is obvious, the *ProjectileMovementComponent*. This comes with a dependency of a *Root SceneComponent*, so I will place the *PMC* on a stack, followed by the *SC*. This is because the dependency indicates a specific order in which things must be initialized.
> >
> > I will then add the *SC* to the header file, switch to the source file, and add the *SC Construction Code*. I can then pop the stack, and perform the same process with the *PMC*, with the now existing reference to the dependency.
> >
> > > I now have a problem, because the *actual* projectile code uses a *SphereComponent* as its root, with a *StaticMeshComponent* attached to that. This would imply that for the *PMC* to update the correct component, it would need to know which one. It does not, however, as anything other than root component updating is non-standard usage, therefore I can simply use *GetRootComponent* for the *PMC* configuration. This also distils the dependencies of the *PMC* into simply requiring a valid root component. This leads me towards the notion that the *PMC* pattern's only requirement for *adaptation* is that it is configured after any root component setup.
> > >
> > > While it is unlikely that any adaptation involving this will *not* have an already configured root component, I can place a relatively simple requirement on that adaptation that it must be the last component configured. As a side note, I could also add the condition that *any* derivative of *UActorComponent* but not *USceneComponent* must be configured after the *SC*s.
>
> The end result from this thought process is that the adaptation *Tick & Move(Forward)* can be directly translated to an addition of a *PMC* that updates the root component.

Appendix H – Testing Excerpts from Development Diary

ApplyRadialDamage

```
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) ALTER Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

This works fine, and the generation file is up to standard

ClickDestroy

```
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(RightMouseButton)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(RightMouseButton) IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

This is correct, however these two side by side highlight that the events for generation have their class specifier, whereas the existing ones don't. There's potential to remove this inconsistency, however it's likely that the action list implementations can handle this themselves.

ExplodeEveryXSeconds

```
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'StartTimerLoop' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: Spawn
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Spawn' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'RadialDamage' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Spawned,Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ Check Effect: Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Spawned' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Explode,Projectile.Spawn(Explosion_Particle) AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Spawn(Explosion_Particle)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Explode' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Spawned
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Spawned IMPLEMENTATION Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Explode
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Explode IMPLEMENTATION Projectile.Spawn(Explosion_Particle) AND Projectile.RadialDamage
```

This generation has highlighted that I don't have entries for some events and effects in their datatables. I should add these now, quickly.

Now hold on. This has likely happened in the other two, and I just haven't noticed.

Go back over them, and update the datatables as you go as well.

ApplyRadialDamage

```
LogAdaptationManager: Effect Top Level Event: Destroy
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Destroy' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'RadialDamage' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.OnBeginOverlap(Any),Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Destroy
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) ALTER Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

Destroy & *RadialDamage* are not in the effects table yet. I also need to add them to the effect conflicts table.

```
LogAdaptationManager: Effect Top Level Event: Destroy
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.OnBeginOverlap(Any),Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Destroy
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) ALTER Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

Destroy & RadialDamage are now in both the *Effects & EffectConflicts* datatables; and generation shows no warnings.

ClickDestroy

```
LogAdaptationManager: Effect Top Level Event: Destroy
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: ThrowProjectileComponent.Pressed(RightMouseButton),Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Destroy
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Pressed' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(RightMouseButton)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(RightMouseButton) IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

I need to add *Pressed* to the event conflicts datatable.

```
LogAdaptationManager: Effect Top Level Event: Destroy
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: ThrowProjectileComponent.Pressed(RightMouseButton),Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Destroy
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(RightMouseButton)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(RightMouseButton) IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

With this added, the action list is still as expected, and there are no invalid rows accessed.

ExplodeEveryXSeconds

```
LogAdaptationManager: Effect Top Level Event: StartTimerLoop
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'StartTimerLoop' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: Spawn
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Spawn' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Spawned,Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ Check Effect: Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Spawned' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Explode,Projectile.Spawn(Explosion_Particle) AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Spawn(Explosion_Particle)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Explode' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Spawned
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Spawned IMPLEMENTATION Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Explode
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Explode IMPLEMENTATION Projectile.Spawn(Explosion_Particle) AND Projectile.RadialDamage
```

I need to add *StartTimerLoop & Spawn* to both effects datatables, and *Spawned* to the conflicts datatable. I don't need to add *Explode*, as this is a custom user defined event, which the *GENERATE* action will handle correctly.

```
LogAdaptationManager: Effect Top Level Event: StartTimerLoop
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: Spawn
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Spawned,Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ Check Effect: Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Explode,Projectile.Spawn(Explosion_Particle) AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Spawn(Explosion_Particle)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Explode' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Spawned
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Spawned IMPLEMENTATION Projectile.StartTimerLoop(Projectile.Explode)
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Explode
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Explode IMPLEMENTATION Projectile.Spawn(Explosion_Particle) AND Projectile.RadialDamage
```

There are now no unexpected warnings, and the action list is still as expected.

TrackNextTarget

```
LogAdaptationManager: Effect Top Level Event: SetMovementDirection
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'SetMovementDirection' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.OnBeginOverlap(Pawn),Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ Check Effect: Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) ALTER Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) IMPLEMENTATION Projectile.SetMovementDirection(GetNearestPawn)
```

I need to add *SetMovementDirection* to both effects datatables. This particular generation presents a problem only easily solvable with a recursive descent parser, which would allow effects passed as parameters to be fully verified every step of the way. I should talk about this both in terms of limitations *and* future work.

```
LogAdaptationManager: Effect Top Level Event: SetMovementDirection
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.OnBeginOverlap(Pawn),Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ Check Effect: Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) ALTER Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) IMPLEMENTATION Projectile.SetMovementDirection(GetNearestPawn)
```

When generating an alter event, the conflicting effects should be used, not the generated effects. This also affects the ricochet test case, which I didn't catch.

```
LogAdaptationManager: Effect Top Level Event: SetMovementDirection
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.OnBeginOverlap(Pawn),Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ Check Effect: Projectile.SetMovementDirection(GetNearestPawn)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) ALTER Destroy(This) AND Damage
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) IMPLEMENTATION Projectile.SetMovementDirection(GetNearestPawn)
```

With this change, the action list is now generated as expected.

Ricochet

```
LogAdaptationManager: Effect Top Level Event: Ricochet
LogAdaptationManager: ^ is in Unreal Framework? true
LogAdaptationManager: Effect Top Level Class: Projectile
LogAdaptationManager: ^ uses dependency: Projectile: ProjectileMovementComponent
LogAdaptationManager: PopupClosed: False
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.OnBeginOverlap(Wall OR Floor),Projectile.Ricochet
LogAdaptationManager: ^ Check Effect: Projectile.Ricochet
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) ALTER Destroy(This) AND Damage
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) IMPLEMENTATION Projectile.Ricochet
```

Choosing '*No*' when asked for usage of the *ProjectileMovementComponent* framework now generates as expected as well.

Grenade

```
LogAdaptationManager: Effect Top Level Event: PhysicsMovement
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'PhysicsMovement' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: AddImpulse
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'AddImpulse' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: StartTimer
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'StartTimer' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: Destroy
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Tick,Projectile.PhysicsMovement
LogAdaptationManager: ^ Check Effect: Projectile.PhysicsMovement
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Spawned,Projectile.AddImpulse AND Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ Check Effect: Projectile.AddImpulse
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.ExplodeGrenade,Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Destroy
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'ExplodeGrenade' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Tick ALTER Move(Forward)
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Tick IMPLEMENTATION Projectile.PhysicsMovement
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Spawned
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Spawned IMPLEMENTATION Projectile.AddImpulse AND Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.ExplodeGrenade
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.ExplodeGrenade IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

I need to add *PhysicsMovement, AddImpulse & StartTimer* to both effects datatables. I do not need to add *ExplodeGrenade* to the events datatables, as it is another user defined event.

```
LogAdaptationManager: Effect Top Level Event: PhysicsMovement
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: AddImpulse
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: StartTimer
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: Destroy
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Tick,Projectile.PhysicsMovement
LogAdaptationManager: ^ Check Effect: Projectile.PhysicsMovement
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Spawned,Projectile.AddImpulse AND Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ Check Effect: Projectile.AddImpulse
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.ExplodeGrenade,Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Destroy
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'ExplodeGrenade' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Tick ALTER Move(Forward)
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Tick IMPLEMENTATION Projectile.PhysicsMovement
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Spawned
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Spawned IMPLEMENTATION Projectile.AddImpulse AND Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.ExplodeGrenade
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.ExplodeGrenade IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

With these additions, there are no unexpected warnings. The generation is unexpected however – physics movement can be applied using the projectile movement component, therefore it needs an Unreal dependency.

```
LogAdaptationManager: Effect Top Level Event: PhysicsMovement
LogAdaptationManager: ^ is in Unreal Framework? true
LogAdaptationManager: Effect Top Level Class: Projectile
LogAdaptationManager: ^ uses dependency: Projectile: ProjectileMovementComponent
LogAdaptationManager: Effect Top Level Event: AddImpulse
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: StartTimer
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: Destroy
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: RadialDamage
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: PopupClosed: True
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Tick,Projectile.PhysicsMovement
LogAdaptationManager: ^ Check Effect: Projectile.PhysicsMovement
LogAdaptationManager: ^ ^ Wants Framework: True
LogAdaptationManager: Print Conflicts for Framework Generation: Projectile.Tick,Projectile.PhysicsMovement, Projectile.PhysicsMovement: Projectile
LogAdaptationManager: Print Conflicts:
LogAdaptationManager: ^ Tick,Move(Forward): 100.000000
LogAdaptationManager: ^ OnBeginOverlap(SphereComponent,Any),Destroy(This) AND Damage: 25.000000
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.Spawned,Projectile.AddImpulse AND Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ Check Effect: Projectile.AddImpulse
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: Extract Framework Generations for Adaptation: Projectile.ExplodeGrenade,Projectile.Destroy AND Projectile.RadialDamage
LogAdaptationManager: ^ Check Effect: Projectile.Destroy
LogAdaptationManager: ^ ^ Wants Framework: False
LogAdaptationManager: ^ Check Effect: Projectile.RadialDamage
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'ExplodeGrenade' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile CONFIGURE ProjectileMovementComponent IMPLEMENTATION PhysicsMovement
LogAdaptationManager: ^ COMMENT IN Projectile CONFIGURE ProjectileMovementComponent WARN Tick,Move(Forward) CONFLICT 100.000000 Projectile.Tick,Projectile.PhysicsMovement
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Tick WARN Move(Forward) CONFLICT 100.000000 Projectile.Tick,Projectile.PhysicsMovement
LogAdaptationManager: ^ COMMENT IN Projectile CONFIGURE ProjectileMovementComponent WARN OnBeginOverlap(SphereComponent,Any),Destroy(This) AND Damage CONFLICT 25.000000 Projectile.Tick,Projectile.PhysicsMovement
LogAdaptationManager: ^ COMMENT IN Projectile EVENT OnBeginOverlap(SphereComponent,Any) WARN Destroy(This) AND Damage CONFLICT 25.000000 Projectile.Tick,Projectile.PhysicsMovement
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Spawned
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Spawned IMPLEMENTATION Projectile.AddImpulse AND Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.ExplodeGrenade
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.ExplodeGrenade IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

This change, and the acceptance of using the framework, generates the expected action list, albeit with one additional similar event warning that I had not picked up on when doing the process manually. Talking about this 'thought process automation' could be a highly useful comment in the evaluation section.

```
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Tick ALTER Move(Forward)
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Tick IMPLEMENTATION Projectile.PhysicsMovement
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.Spawned
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.Spawned IMPLEMENTATION Projectile.AddImpulse AND Projectile.StartTimer(Projectile.ExplodeGrenade)
LogAdaptationManager: ^ GENERATE IN Projectile EVENT Projectile.ExplodeGrenade
LogAdaptationManager: ^ COMMENT IN Projectile EVENT Projectile.ExplodeGrenade IMPLEMENTATION Projectile.Destroy AND Projectile.RadialDamage
```

Choosing not to use the framework also gives a reasonable action list, however a human programmer would recognize an adaptation of tick, physicsmovement to be a configuration issue. This could be better handled by the system.

AutoFire

```
LogAdaptationManager: Effect Top Level Event: StartTimerLoop
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Effect Top Level Event: StopTimerLoop
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'StopTimerLoop' not in DataTable '/AdvancedClassWizard/DT_Effects.DT_Effects'.
LogAdaptationManager: ^ is in Unreal Framework? false
LogAdaptationManager: Generate Action List:
LogAdaptationManager: Extract Framework Generations for Adaptation: ThrowProjectileComponent.Press(Fire),ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ Check Effect: ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Press' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Extract Framework Generations for Adaptation: ThrowProjectileComponent.Release(Fire),ThrowProjectileComponent.StopTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ Check Effect: ThrowProjectileComponent.StopTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ ^ Wants Framework: False
LogDataTable: Warning: UDataTable::FindRow : '' requested row 'Release' not in DataTable '/AdvancedClassWizard/DT_Conflicts.DT_Conflicts'.
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Press(Fire)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Press(Fire) IMPLEMENTATION ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Release(Fire)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Release(Fire) IMPLEMENTATION ThrowProjectileComponent.StopTimerLoop(Spawn(Projectile))
```

I need to add *StopTimerLoop* to the effects datatables. I need to rename *Press* to *Pressed*, and *Release* to *Released*, adding the latter to the conflicts datatable.

```
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(Fire)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Pressed(Fire) IMPLEMENTATION ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT Pressed(Fire) WARN Spawn(Projectile) AND Cooldown CONFLICT 50.000000 ThrowProjectileComponent.Pressed(Fire),ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Released(Fire)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Released(Fire) IMPLEMENTATION ThrowProjectileComponent.StopTimerLoop(Spawn(Projectile))
```

With these changes, the conflict checker doesn't seem to be recognizing *Pressed(Fire)* as a conflict. This is because I set the conflict value to 50 for Pressed.

```
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT Pressed(Fire) ALTER Spawn(Projectile) AND Cooldown
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT Pressed(Fire) IMPLEMENTATION ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Released(Fire)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Released(Fire) IMPLEMENTATION ThrowProjectileComponent.StopTimerLoop(Spawn(Projectile))
```

With these changes, the generated action list is as expected, however my initial conflict value for Pressed -> Pressed as 50 is because there can be multiple pressed events on a class, whereas others, such as spawned, can be only one. This distinction lies with whether the event is handled as a dispatcher or an overridden event, which my system should, but doesn't take into account right now.

Appendix I – Completed PLESI Form

UNIVERSITY OF HUDDERSFIELD
SCHOOL OF COMPUTING AND ENGINEERING

PROJECT ETHICAL REVIEW FORM

Applicable for all research, masters and undergraduate projects

Project Title:	Towards an Application of Adaptive Programming to Gameplay Mechanics
Student:	Joshua Pritchard
Course/Programme:	N421: Bsc (Hons) Computing Science with Games Programming SW
Department:	Computing and Engineering
Supervisor:	Dr. Minsi Chen
Project Start Date:	1st January 2021

ETHICAL REVIEW CHECKLIST

	Yes	No
1. Are there problems with any participant's right to remain anonymous?	☐	☒
2. Could a conflict of interest arise between a collaborating partner or funding source and the potential outcomes of the research, e.g. due to the need for confidentiality?	☐	☒
3. Will financial inducements be offered?	☐	☒
4. Will deception of participants be necessary during the research?	☐	☒
5. Does the research involve experimentation on any of the following?		
(i) animals?	☐	☒
(ii) animal tissues?	☐	☒
(iii)human tissues (including blood, fluid, skin, cell lines)?	☐	☒
6. Does the research involve participants who may be particularly vulnerable, e.g. children or adults with severe learning disabilities?	☐	☒
7. Could the research induce psychological stress or anxiety for the participants beyond that encountered in normal life?	☐	☒
8. Is it likely that the research will put any of the following at risk:		
(i) living creatures?	☐	☒
(ii) stakeholders (disregarding health and safety, which is covered by Q9)?	☐	☒
(iii)the environment?	☐	☒
(iv)the economy?	☐	☒
9. Having completed a health and safety risk assessment form and taken all reasonable practicable steps to minimise risk from the hazards identified, are the residual risks acceptable (Please attach a risk assessment form – available at the end of this document)	☒	☐

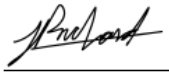
STATEMENT OF ETHICAL ISSUES AND ACTIONS

If the answer to any of the questions above is yes, or there are any other ethical issues that arise that are not covered by the checklist, then please give a summary of the ethical issues and the action that will be taken to address these in the box below. If you believe there to be no ethical issues, please enter "NONE".

NONE

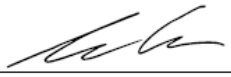
STATEMENT BY THE STUDENT

I believe that the information I have given in this form on ethical issues is correct.

Signature:  Date: 20/04/2021

AFFIRMATION BY THE SUPERVISOR

I have read this Ethical Review Checklist and I can confirm that, to the best of my understanding, the information presented by the student is correct and appropriate to allow an informed judgement on whether further ethical approval is required.

Signature:  Date: 20/04/2021

SUPERVISOR RECOMMENDATION ON THE PROJECT'S ETHICAL STATUS

Having satisfied myself of the accuracy of the project ethical statement, I believe that the appropriate action is:

The project proceeds in its present form	X
The project proposal needs further assessment by an Ethical Review Panel. The Supervisor will pass the form to the Ethical Review Panel Leader for consideration.	

RETENTION OF THIS FORM

- The Supervisor must retain a copy of this form until the project report/dissertation is produced.
- The student must include a copy of the form as an appendix in the report/dissertation.

OUTCOME OF THE ETHICAL REVIEW PANEL PROCESS, WHERE REQUIRED

	Tick One
1. Approved. The ethical issues have been adequately addressed and the project may commence.	☐
2. Approved subject to minor amendments. The required amendments are stated in the box below. The project may proceed once the form has been amended in line with the requirements and signed by the Supervisor in the box imediately below to confirm this.	☐

I confirm, as Supervisor, that the amendments required have been made:

Signature: _____ Date: _____

3. Resubmit. The areas requiring further action are stated in the box below. The project may not proceed until the form has been resubmitted and approved.	☐
4. Reject. The reasons why it will not be possible to address the ethical issues adequately are stated in the box below.	☐

For any of the outcomes 2, 3 or 4 above, please provide a statement in the box below.

AFFIRMATION BY THE REVIEW PANEL LEADER

I approve the decision reached above by the review panel members:

Signature: _____ Date: _____

THE UNIVERSITY OF HUDDERSFIELD: RISK ANALYSIS & MANAGEMENT

ACTIVITY: Software Development			Name: Joshua Pritchard	
LOCATION: Private Residence			Date:20/04/2021	Review Date:
Hazard(s) Identified	Details of Risk(s)	People at Risk	Risk management measures	Other comments
None				

Appendix J – Generated ThrowProjectile Pattern Code

Projectile.h

```
#pragma once

#pragma region Includes

#include "CoreMinimal.h"

#include "GameFramework/Actor.h"

#include "Projectile.generated.h"

#pragma endregion

UCLASS()
class AProjectile : public AActor
{
	GENERATED_BODY()

#pragma region Constructors

public:

	/** Default Constructor */
	AProjectile();

#pragma endregion

#pragma region Initialization

protected:

	/** Called when game starts - Called when actor spawned */
	virtual void BeginPlay() override;

#pragma endregion
```

```
#pragma region Update

public:

	/** Called every frame */
	virtual void Tick(float DeltaTime) override;

#pragma endregion

#pragma region Components

public:

	/** Sphere Component for Collision */
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
		class USphereComponent* SphereComponent;

	/** Static Mesh for Visual Representation */
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
		class UStaticMeshComponent* StaticMesh;

	/** Movement Component */
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
		class UProjectileMovementComponent* ProjectileMovementComponent;

#pragma endregion

#pragma region Effects

public:

	/** Particle used for collision */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Damage")
		class UParticleSystem* ExplosionEffect;

#pragma endregion

#pragma region Damage

public:
```

```
	/** Damage Type dealt by projectile */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Damage")
		TSubclassOf<class UDamageType> DamageType;

	/** Damage amount dealt by projectile */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Damage")
		float Damage;

protected:

	/** Called when actor destroyed */
	virtual void Destroyed() override;

#pragma endregion

#pragma region Projectile

protected:

	/** On Impact - Apply Damage to hit actor - Destroy self */
	UFUNCTION(Category = "Projectile")
		void OnProjectileImpact(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector
NormalImpulse, const FHitResult& Hit);

#pragma endregion

};
```

Projectile.cpp

```
#pragma region Includes

#include "Projectile.h"

#include "Components/SphereComponent.h"
#include "Components/StaticMeshComponent.h"

#include "GameFramework/ProjectileMovementComponent.h"
#include "GameFramework/DamageType.h"

#include "Particles/ParticleSystem.h"

#include "Kismet/GameplayStatics.h"
#include "UObject/ConstructorHelpers.h"

#pragma endregion

#pragma region Constructors

AProjectile::AProjectile()
{
	// Set actor to tick
	PrimaryActorTick.bCanEverTick = true;

	// Enable Replication
	this->bReplicates = true;

	// Sphere Component Initialization - Used for Collision - Used for RootComponent
	SphereComponent = CreateDefaultSubobject<USphereComponent>(TEXT("RootComponent"));
	SphereComponent->InitSphereRadius(40.f);
	SphereComponent->SetCollisionProfileName(TEXT("BlockAllDynamic"));
	RootComponent = SphereComponent;

	// Register SphereComponent Collision
	SphereComponent->OnComponentHit.AddDynamic(this, &AProjectile::OnProjectileImpact);
```

```
	// Static Mesh Initialization - Used for Visual Representation
	StaticMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));
	StaticMesh->SetupAttachment(RootComponent);
	static ConstructorHelpers::FObjectFinder<UStaticMesh>
DefaultMesh(TEXT("StaticMesh'/Game/StarterContent/Shapes/Shape_Sphere.Shape_Sphere'"));
	if (DefaultMesh.Succeeded()) {
		StaticMesh->SetStaticMesh(DefaultMesh.Object);
		StaticMesh->SetRelativeLocation({ 0.0f, 0.0f, -40.f });
		StaticMesh->SetRelativeScale3D({ 0.75f, 0.75f, 0.75f });
	}

	// Explosion Effect Initialization - Used when Colliding
	static ConstructorHelpers::FObjectFinder<UParticleSystem>
DefaultExplosionEffect(TEXT("ParticleSystem'/Game/StarterContent/Particles/P_Explosion.P_Explosion'"));
	if (DefaultExplosionEffect.Succeeded()) {
		ExplosionEffect = DefaultExplosionEffect.Object;
	}

	// Projectile Movement Component Initialization - Used for Movement
	ProjectileMovementComponent = CreateDefaultSubobject<UProjectileMovementComponent>(TEXT("ProjectileMovement"));
	ProjectileMovementComponent->SetUpdatedComponent(SphereComponent);
	ProjectileMovementComponent->InitialSpeed = 1500.f;
	ProjectileMovementComponent->MaxSpeed = 1500.f;
	ProjectileMovementComponent->bRotationFollowsVelocity = true;
	ProjectileMovementComponent->ProjectileGravityScale = 0.0f;

	// Damage Initialization - Type - Amount
	DamageType = UDamageType::StaticClass();
	Damage = 10.0f;

}

#pragma endregion

#pragma region Initialization

void AProjectile::BeginPlay()
{
	Super::BeginPlay();
```

```
}

#pragma endregion

#pragma region Update

void AProjectile::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
}

#pragma endregion

#pragma region Damage

void AProjectile::Destroyed()
{
	// Spawn Explosion Effect at Actor Location
	const FVector SpawnLocation = GetActorLocation();
	UGameplayStatics::SpawnEmitterAtLocation(this, ExplosionEffect, SpawnLocation, FRotator::ZeroRotator, true,
EPSCPoolMethod::AutoRelease);
}

#pragma endregion

#pragma region Projectile

void AProjectile::OnProjectileImpact(UPrimitiveComponent * HitComponent, AActor * OtherActor, UPrimitiveComponent * OtherComp,
FVector NormalImpulse, const FHitResult & Hit)
{
	// Apply damage to Other Actor - Destroy Self
	if (OtherActor) {
		UGameplayStatics::ApplyPointDamage(OtherActor, Damage, NormalImpulse, Hit, GetInstigatorController(), this,
DamageType);
	}
	Destroy();
}

#pragma endregion
```

ThrowProjectileComponent.h

```
#pragma once

#pragma region Includes

#include "CoreMinimal.h"

#include "Components/ActorComponent.h"

#include "ThrowProjectileComponent.generated.h"

#pragma endregion


UCLASS( Blueprintable, ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class UThrowProjectileComponent : public UActorComponent
{
	GENERATED_BODY()

#pragma region Construction

public:

	/** Default Constructor */
	UThrowProjectileComponent();

#pragma endregion

#pragma region Initialization

protected:

	/** Called when play starts */
	virtual void BeginPlay() override;

#pragma endregion
```

```
#pragma region Update

public:

	/** Called every frame */
	virtual void TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction) override;

#pragma endregion

#pragma region References

protected:

	/** Pointer to owning player pawn */
	UPROPERTY(BlueprintReadWrite, Category = "References")
		class APawn* PlayerPawn_Reference = nullptr;

#pragma endregion

#pragma region Shoot

protected:

	/** Projectile type to fire */
	UPROPERTY(EditDefaultsOnly, Category = "Shoot")
		TSubclassOf<class AProjectile> ProjectileClass;

	/** Delay between shots - seconds - controls fire rate - prevents overflow of guaranteed replications */
	UPROPERTY(EditDefaultsOnly, Category = "Shoot")
		float FireRate;

	/** Is Character firing weapon */
	bool bIsFiringWeapon;

protected:

	/** Begin Weapon Firing */
	UFUNCTION(BlueprintCallable, Category = "Shoot")
		void StartFire();
```

```
	/** Stop Weapon Firing - Allow StartFire to be called */
	UFUNCTION(BlueprintCallable, Category = "Shoot")
		void StopFire();

	/** Server - Spawn Projectiles */
	UFUNCTION(BlueprintCallable, Category = "Shoot")
		void HandleFire();

protected:

	/** Provides m_FireRate delay for firing */
	FTimerHandle FiringTimer;

#pragma endregion


};
```

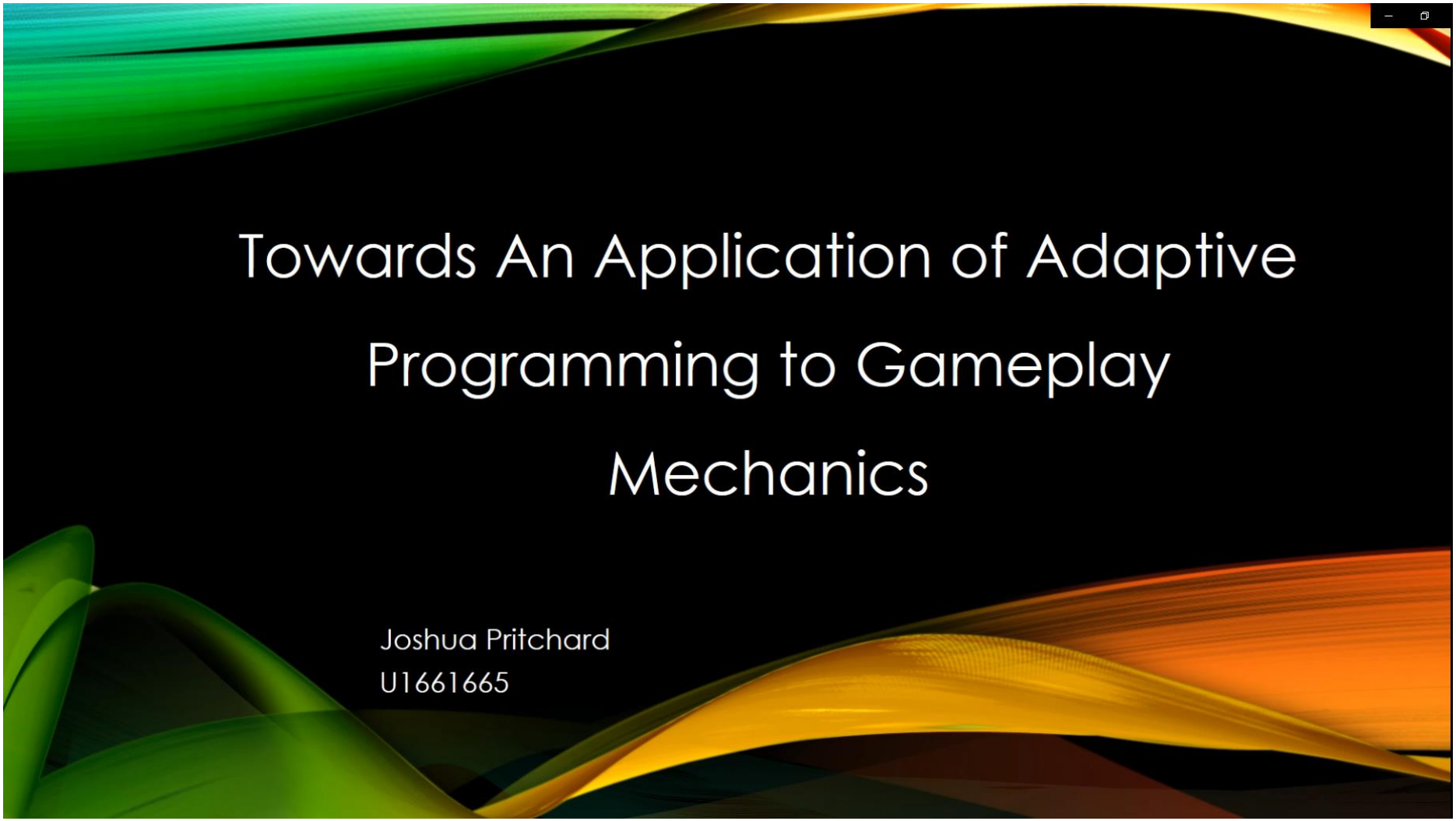
ThrowProjectileComponent.cpp

```
#pragma region Includes

#include "ThrowProjectileComponent.h"

#include "Projectile.h"

#pragma endregion

#pragma region Construction

UThrowProjectileComponent::UThrowProjectileComponent()
{
	PrimaryComponentTick.bCanEverTick = true;

	// Initialize Projectile Type
	ProjectileClass = AProjectile::StaticClass();

	// Initialize Fire Rate
	FireRate = 0.25f;
	bIsFiringWeapon = false;
}

#pragma endregion

#pragma region Initialization

void UThrowProjectileComponent::BeginPlay()
{
	Super::BeginPlay();

	// Store reference to owning player
	PlayerPawn_Reference = Cast<APawn>(GetOwner());
	check(PlayerPawn_Reference);

	// Register Fire Input
	PlayerPawn_Reference->InputComponent->BindAction("Fire", IE_Pressed, this, &UThrowProjectileComponent::StartFire);
```

```
}

#pragma endregion

#pragma region Update

void UThrowProjectileComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
	Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
}

#pragma endregion

#pragma region Shoot

void UThrowProjectileComponent::StartFire()
{
	// If not firing weapon - Start FireRateTimer - Spawn Projectile
	if (!bIsFiringWeapon) {
		bIsFiringWeapon = true;
		UWorld* World = GetWorld();
		World->GetTimerManager().SetTimer(FiringTimer, this, &UThrowProjectileComponent::StopFire, FireRate, false);
		HandleFire();
	}
}

void UThrowProjectileComponent::StopFire()
{
	bIsFiringWeapon = false;
}

void UThrowProjectileComponent::HandleFire()
{

	// Spawn Projectile in front of player
	const FVector SpawnLocation = PlayerPawn_Reference->GetActorLocation() + (PlayerPawn_Reference->GetControlRotation().Vector()
* 100.0f) + (PlayerPawn_Reference->GetActorUpVector() * 50.0f);
	const FRotator SpawnRotation = PlayerPawn_Reference->GetControlRotation();

	FActorSpawnParameters SpawnParameters;
```

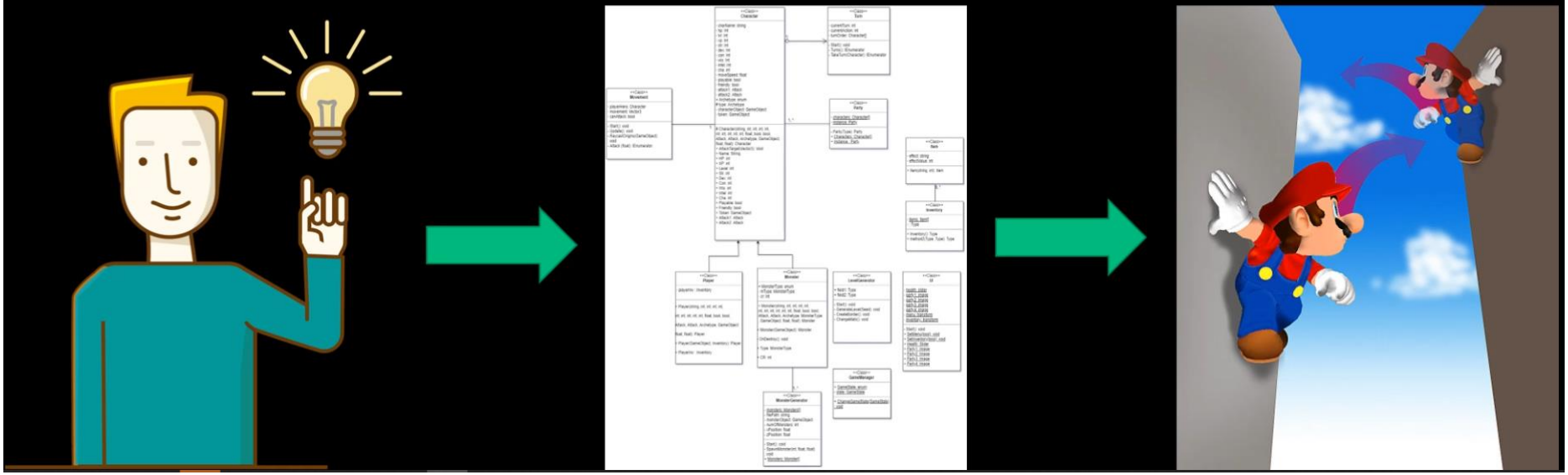
```
        SpawnParameters.Instigator = PlayerPawn_Reference->GetInstigator();
        SpawnParameters.Owner = PlayerPawn_Reference;

        AProjectile* SpawnedProjectile = GetWorld()->SpawnActor<AProjectile>(SpawnLocation, SpawnRotation, SpawnParameters);
}

#pragma endregion
```

Appendix K – Presentation Slides


Towards An Application of Adaptive Programming to Gameplay Mechanics
Joshua Pritchard
U1661665

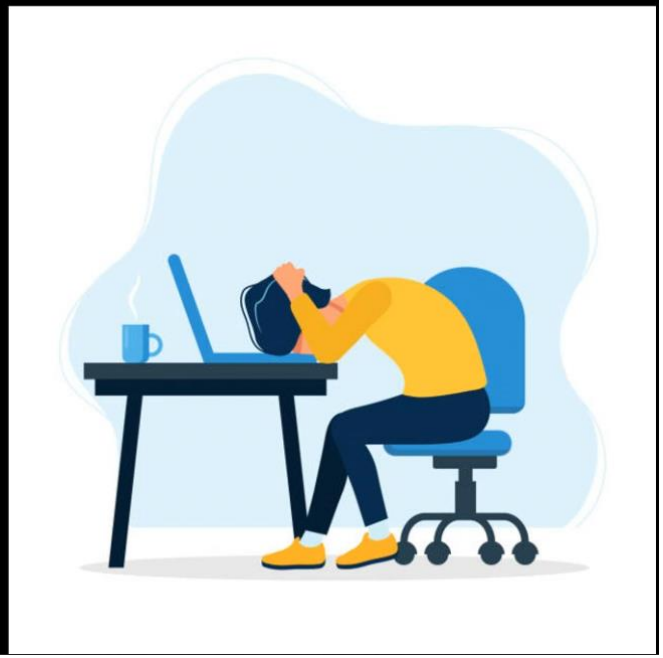
GAMEPLAY PROGRAMMING

- What is it?



GAMEPLAY PROGRAMMING

- What is it?
- Why is it important?



GAMEPLAY PROGRAMMING

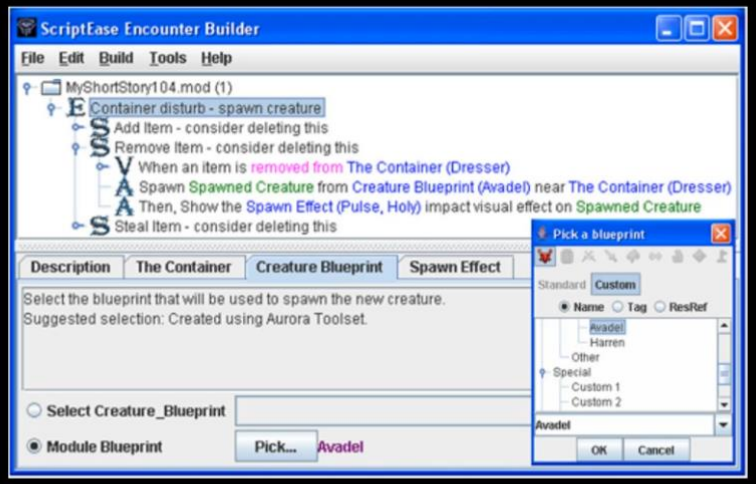
- What is it?
- Why is it important?
- Why does this happen?

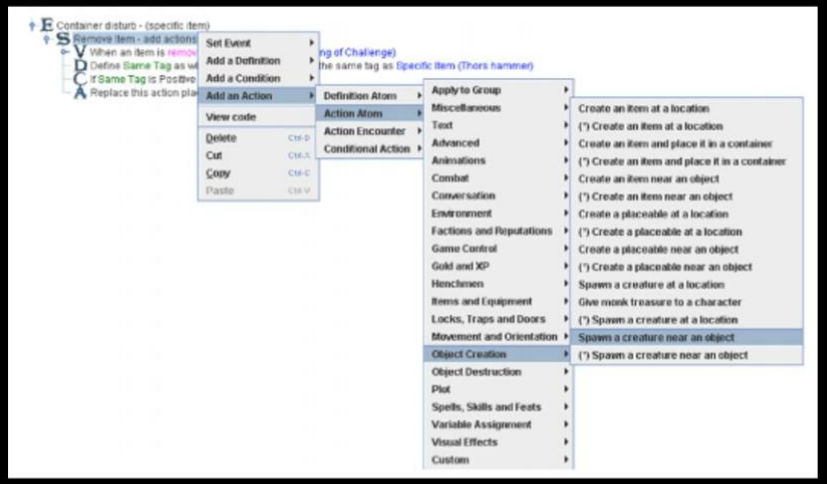
GAMEPLAY PROGRAMMING

- What is it?
- Why is it important?
- Why does this happen?
- Why has no-one solved this yet?

SCRIPTEASE

- What is it?

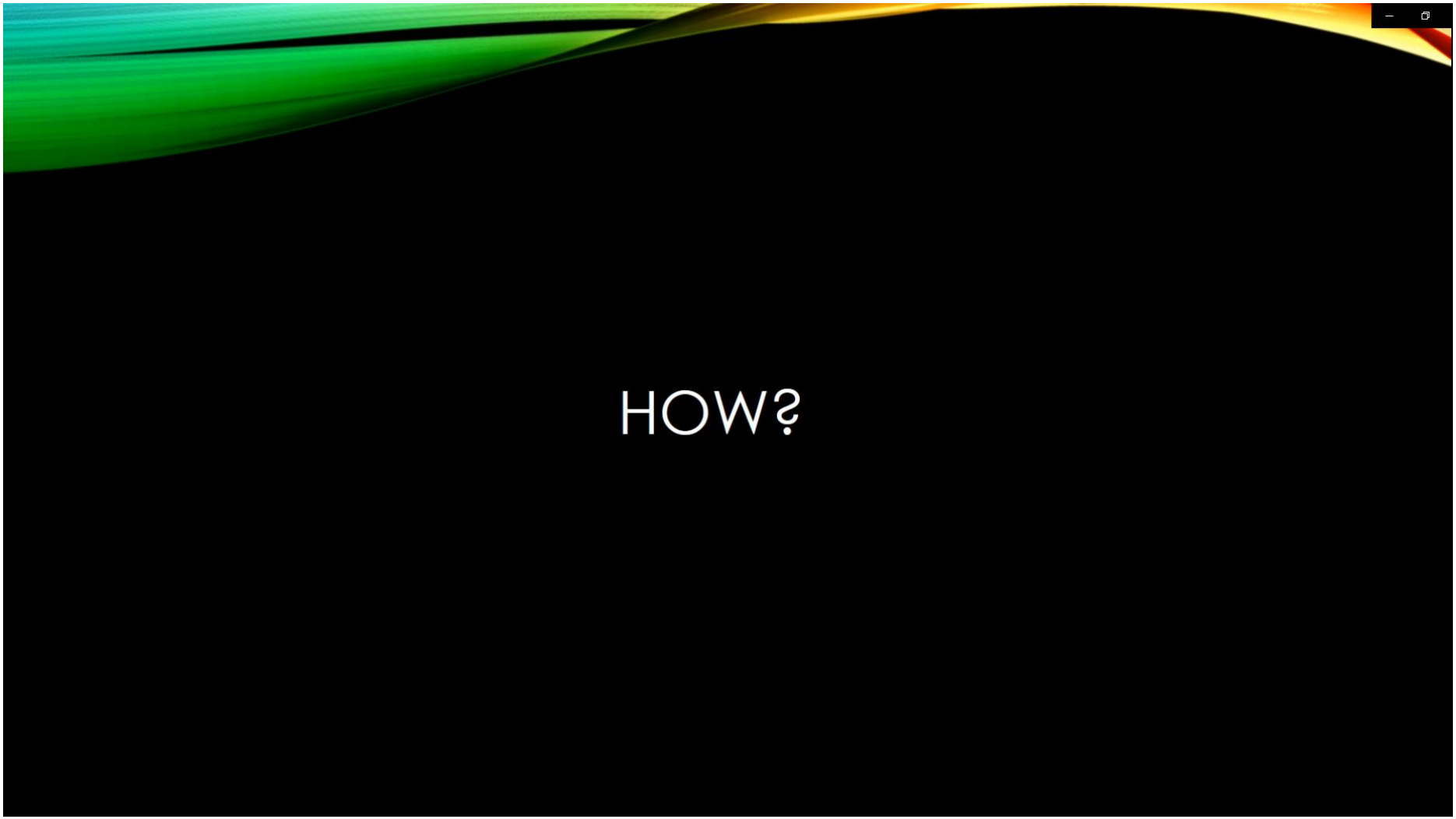


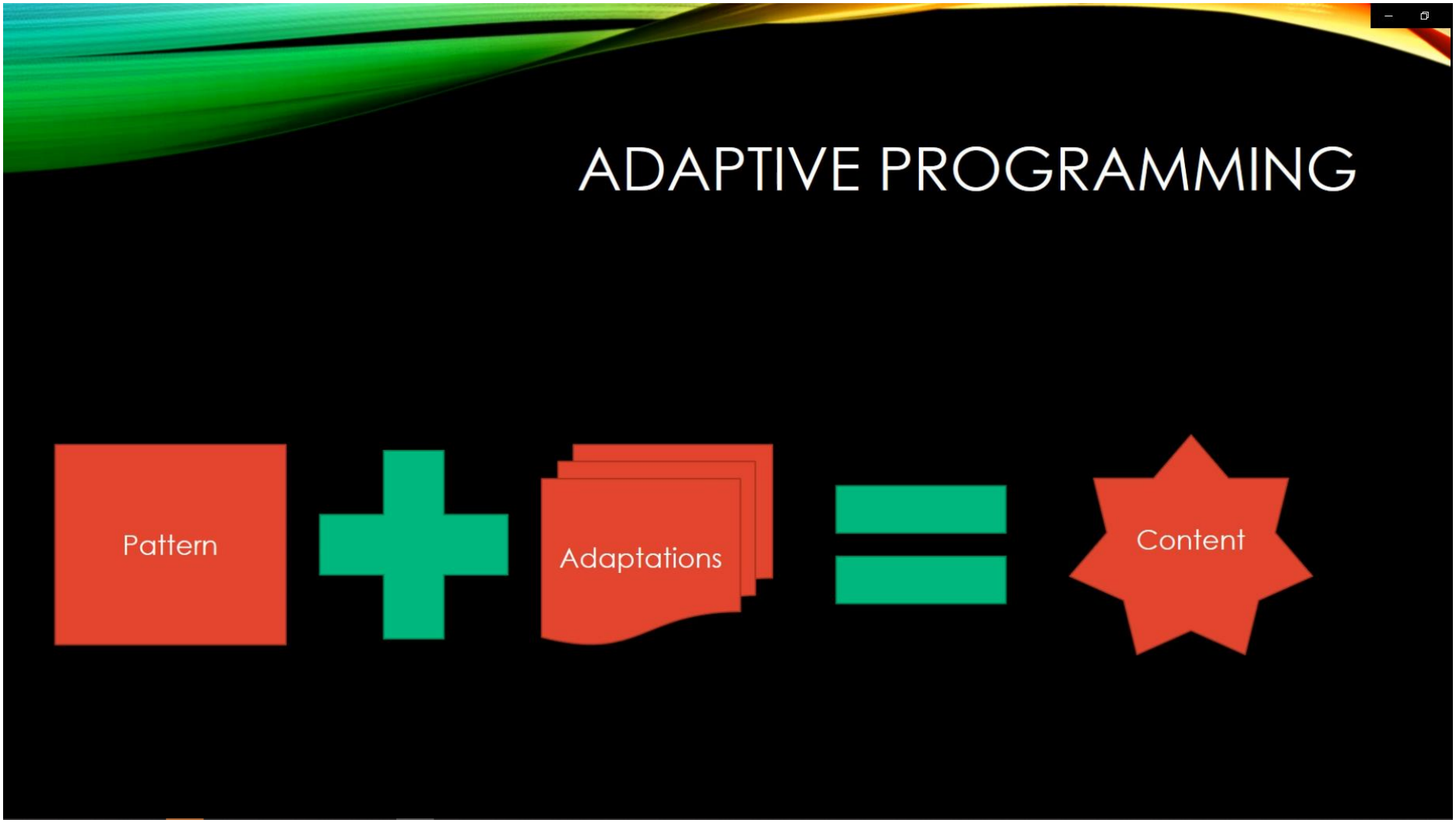


SCRIPTEASE

- What is it?
- Why is it so great?


SCRIPTEASE
• What is it?
• Why is it so great?
• Why is this relevant?


HOW?


ADAPTIVE PROGRAMMING
Pattern
Adaptations
Content

ADAPTIVE PROGRAMMING

- Step 1: Patterns

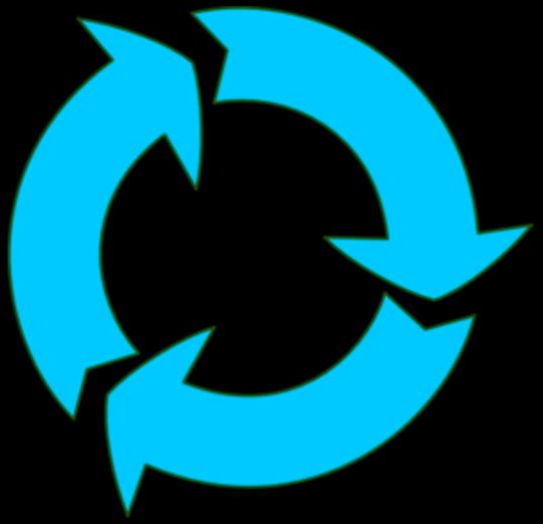
ADAPTIVE PROGRAMMING

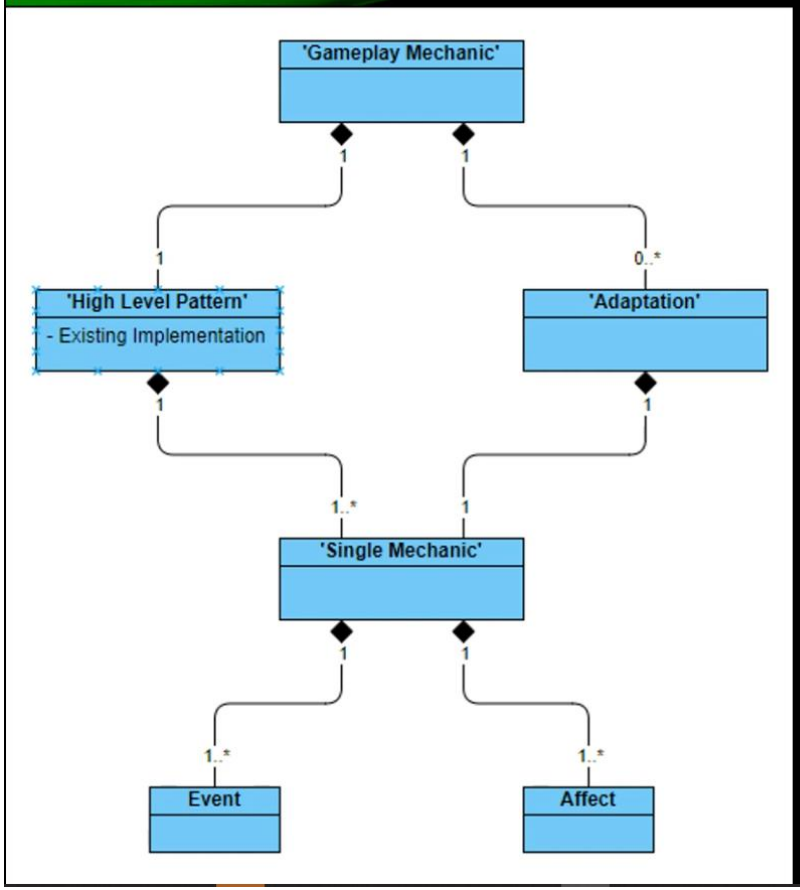
- Step 1: Patterns
- Step 2: Adaptations

≠

ADAPTIVE PROGRAMMING

- Step 1: Patterns
- Step 2: Adaptations
- Step 3: Automated Generation & Adaptation System





GAMEPLAY MECHANICS

- Ontology

SMALL SIDE NOTE - PATTERNS

```
{
    "classes":
    [
        {
            "name":"Projectile",
            "parent":"Actor",
            "components":
            [
                {
                    "name":"SphereComponent",
                    "parent":"SphereComponent",
                    "components":
                    [
                        {
                            "name":"StaticMesh",
                            "parent":"StaticMeshComponent"
                        }
                    ]
                },
                {
                    "name":"ProjectileMovementComponent",
                    "parent":"ProjectileMovementComponent"
                }
            ],
            "mechanics":
            [
                {
                    "events":["Projectile.Tick"],"effects":["SphereComponent.Move(Forward)"]
                },
                {
                    "events":["SphereComponent.OnBeginOverlap(Any)"],"effects":["Projectile.Destroy","Any.Damage"]
                },
                {
                    "events":["Projectile.Destroyed"],"effects":["Projectile.Spawn(Explosion_Particle)"]
                }
            ]
        },
        {
            "name":"ShootProjectileManager",
            "parent":"ActorComponent",
            "mechanics":
            [
                {
                    "events":["ShootProjectileManager.Pressed(Fire)"],"effects":["ShootProjectileManager.Spawn(Projectile)","ShootProjectileManager.Cooldown"]
                }
            ]
        }
    ]
}
```

- Specification Files

DEMONSTRATION

- Step 1: Specify a Pattern

```
{
    "pattern":"ThrowProjectile",

}
```

DEMONSTRATION

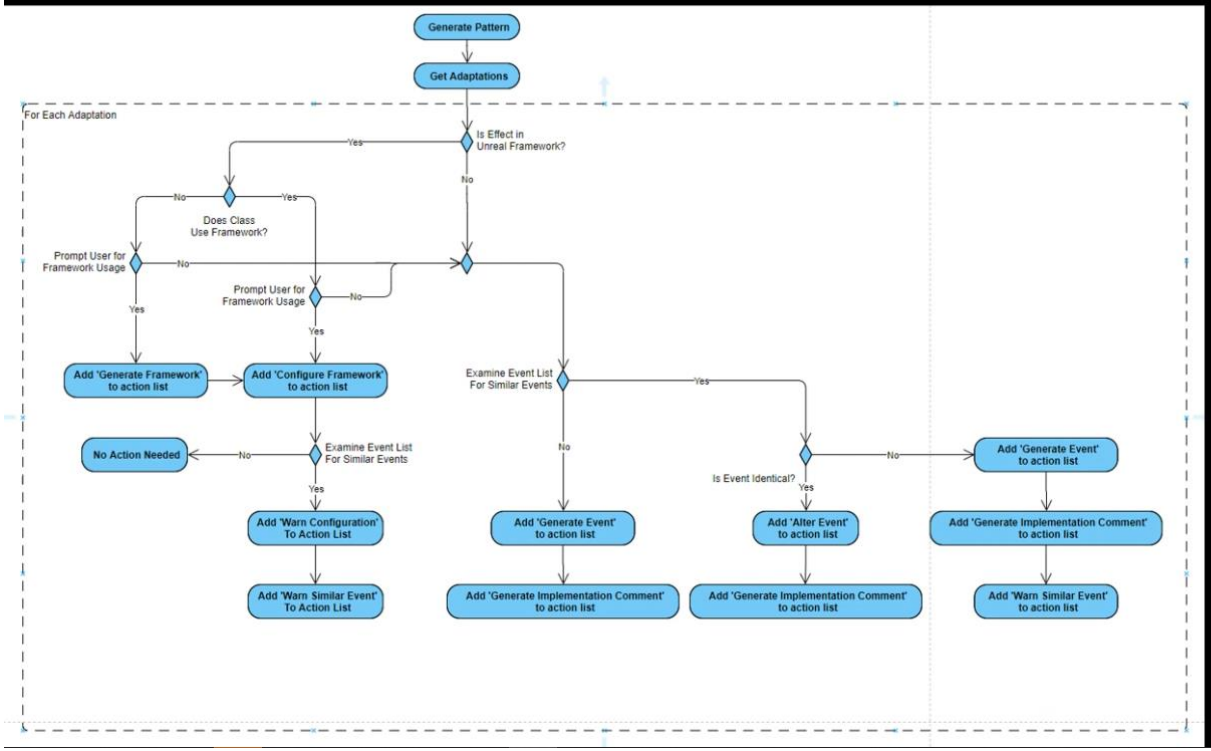
- Step 1: Specify a Pattern
- Step 2: Specify Adaptations

```
{
    "pattern":"ThrowProjectile",
    "adaptations":
    [
        {"events":["ThrowProjectileComponent.Pressed(Fire)"],"effects":["ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))"]},
        {"events":["ThrowProjectileComponent.Released(Fire)"],"effects":["ThrowProjectileComponent.StopTimerLoop(Spawn(Projectile))"]}
    ]
}
```

DEMONSTRATION

- Step 1: Specify a Pattern
- Step 2: Specify Adaptations
- Step 3: Tell the computer to do the hard stuff.

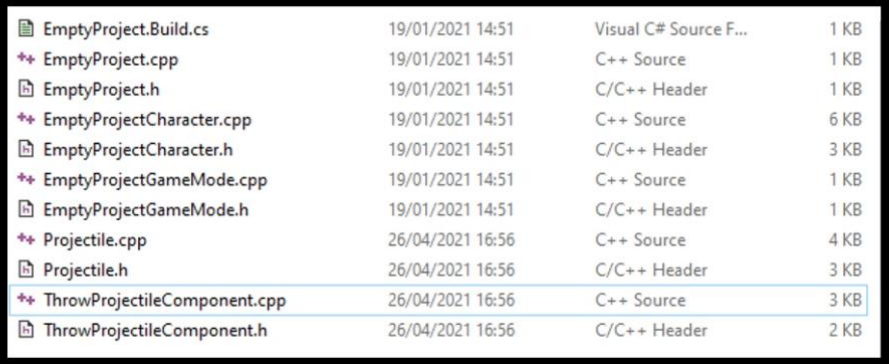
DEMONSTRATION



- Step 3.1: Figure out Intent

DEMONSTRATION

- Step 3.1.1: Generate Pattern Code



EmptyProject.Build.cs	19/01/2021 14:51	Visual C# Source F...	1 KB
EmptyProject.cpp	19/01/2021 14:51	C++ Source	1 KB
EmptyProject.h	19/01/2021 14:51	C/C++ Header	1 KB
EmptyProjectCharacter.cpp	19/01/2021 14:51	C++ Source	6 KB
EmptyProjectCharacter.h	19/01/2021 14:51	C/C++ Header	3 KB
EmptyProjectGameMode.cpp	19/01/2021 14:51	C++ Source	1 KB
EmptyProjectGameMode.h	19/01/2021 14:51	C/C++ Header	1 KB
Projectile.cpp	26/04/2021 16:56	C++ Source	4 KB
Projectile.h	26/04/2021 16:56	C/C++ Header	3 KB
ThrowProjectileComponent.cpp	26/04/2021 16:56	C++ Source	3 KB
ThrowProjectileComponent.h	26/04/2021 16:56	C/C++ Header	2 KB

DEMONSTRATION

- Step 3.1: Figure out Intent

```
LogAdaptationManager: Print Action List:
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT Pressed(Fire) ALTER Spawn(Projectile) AND Cooldown
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT Pressed(Fire) IMPLEMENTATION ThrowProjectileComponent.StartTimerLoop(Spawn(Projectile))
LogAdaptationManager: ^ GENERATE IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Released(Fire)
LogAdaptationManager: ^ COMMENT IN ThrowProjectileComponent EVENT ThrowProjectileComponent.Released(Fire) IMPLEMENTATION ThrowProjectileComponent.StopTimerLoop(Spawn(Projectile))
```

EVALUATION

- Pros
- Cons
- Where next?