The PAWN System: How Procedurally Adaptive Webbed Narratives Create Stories

Steven T. Bordelon

University of New Orleans, stbordel@uno.edu

Follow this and additional works at: https://scholarworks.uno.edu/td

Part of the Other Computer Sciences Commons

Recommended Citation
https://scholarworks.uno.edu/td/3169

This Thesis is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.
The PAWN System:
How Procedurally Adaptive Webbed Narratives Create Stories

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Bachelor of Science
in
Computer Science

by

Steven Bordelon

May, 2024
Contents

List of Figures iv

List of Tables v

Abstract vi

1 Introduction 1

2 Related Work 4
  2.1 LLM-Generated Narratives ............................................ 4
  2.2 Branching Narratives ................................................... 5
  2.3 Emergent Narrative ..................................................... 6
  2.4 Event-Driven Procedural Narrative Systems ......................... 7

3 PAWN System 9
  3.1 Game Background ....................................................... 9
  3.2 Actors ........................................................................ 14
    3.2.1 Actor Creation ...................................................... 14
    3.2.2 Actor ................................................................. 17
    3.2.3 Potential Conversations ........................................... 18
    3.2.4 Actor Database ..................................................... 20
  3.3 Occurrences .................................................................. 20
    3.3.1 The Occurrence as an Object ..................................... 21
    3.3.2 The Occurrence Spawner .......................................... 22
    3.3.3 The Occurrence Handler .......................................... 23
    3.3.4 Occurrence Formation Example ................................. 25
  3.4 Predicates ...................................................................... 26
    3.4.1 The Predicate as an Object ....................................... 27
    3.4.2 The Predicate Handler ............................................. 28
  3.5 Narrative Patterns ....................................................... 29
    3.5.1 The Narrative Pattern as an Object .............................. 30
    3.5.2 Spawning Narrative Patterns .................................... 30
    3.5.3 The Narrative Pattern Handler ................................... 31
    3.5.4 Finding Patterns .................................................... 31
  3.6 WebWeaver ................................................................. 33
    3.6.1 Updating Crew Logic ............................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.2</td>
<td>Updating Crew Conversations</td>
<td>34</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Launching Conversations</td>
<td>36</td>
</tr>
<tr>
<td>3.6.4</td>
<td>In Conversations</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>Results</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Procedure</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Participants</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Results</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Other Miscellaneous Notes</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Vita</td>
<td>44</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 AI Dungeon: Snapshot of A Game Session ........................................ 5  
2.2 The Walking Dead: Branching Storylines ....................................... 6  
2.3 Dwarf Fortress: A Dwarf and His Loyal Dog .................................. 7  
2.4 Nemesis: The Poet ........................................................................ 8  

3.1 A PAWN System Overview .............................................................. 10  
3.2 FTL: Faster Than Light ................................................................ 11  
3.3 Dialogue in FTL ............................................................................ 12  
3.4 WTP: Walk the Plank ..................................................................... 13  
3.5 Dialogue in WTP ........................................................................... 14  
3.6 The Player’s Ship and Their Crew .................................................. 15  
3.7 Actor Creation and Storage ............................................................. 16  
3.8 Occurrence Creation ...................................................................... 21  
3.9 Predicate Spawning Pipeline .......................................................... 26  

4.1 Results: "I found the gameplay hard to understand." ...................... 45  
4.2 Results: "I was confused by the crew member dialogues." ............... 45  
4.3 Results: "I derived enjoyment from this gameplay demo." ............... 45  
4.4 Results: "I was attached to my crew members." .............................. 46  
4.5 Results: "I enjoyed my crew member’s dialogue." .......................... 46  
4.6 Results: "I found the gameplay hard to understand." ...................... 46  
4.7 Results: "I felt my crew member’s dialogue were responsive to gameplay events." 47
List of Tables

1.1 PAWN System Goals ..................................................... 3
Abstract

This thesis describes the design, implementation, and testing of a novel procedural narrative system called the Procedurally Adaptive Webbed Narrative (PAWN) system. PAWN procedurally generates characters and, responding to choices made by the player, produces more responsive characters and relationships involving the player and these narrative agents. Initially, this thesis discusses other interactive narrative types that exist, such as emergent or event-driven narratives, along with their strengths and weaknesses. It then examines each aspect of PAWN, starting with initial actor generation, then moving to the capturing of game events and translating them into logical objects called Occurrences. These Occurrences are then parsed into Predicates, which are more character-focused relational objects. These Predicates are continually queried for specific Narrative Patterns, and, if found, introduce more specific and interesting Predicates into the PAWN’s current Predicate list. Furthermore, the dialogue selection and tailoring process derived from the most interesting of these Predicates is explained. Conducting A/B tests with PAWN and the control narrative system (NONPAWN) in the same game base revealed unanimous user preference for PAWN, with users judging PAWN characters to be more responsive and interesting than their NONPAWN alternatives.
Chapter 1

Introduction

In interactive media and gaming, responsiveness to player action and input is an important aspect of the general gameplay experience, if not the most important aspect.

The very basic concept of a 'game' is typically one of give and take, the ability of the player to express agency in some way and then noticing and responding to the effects caused by their actions[3]. Game feel, engagement, player expression, all of these aspects of games rely on a very basic foundation of either the player taking some action or the game itself giving some action to the player to then respond to.

As examined in Rigby and Ryan’s work relating to self-determination in “The Motivational Pull of Video Games: A Self-Determination Theory Approach”, a sense of autonomy and choice is intrinsic to the human condition of feeling satisfied, and this is especially true of video games[8]. When players feel ‘listened to’, when the actions they take in games are ones that they feel like they have chosen themselves and then specific reactive outcomes result from these player choices, players have their yearning for autonomy met and will likely be satisfied with their experience.

For example, the game *The Stanley Parable* puts the player in the shoes of a seemingly mundane office worker named Stanley in an equally mundane office space[9]. But upon choosing to make Stanley perform any typical action - or atypical action - in that office, a narration expands on the player’s choices in a very specific manner; that is if the player chooses to stay in their beginning cubicle for an unusually long time, the narration will note this and wonder what exactly Stanley is doing. This narration provides gratification to the player, letting them know that their action that they have chosen from a broad list of possibilities has been thought of, planned for, and noticed, and the world is now changed for them having committed those actions.

Making a game responsive can prove to be quite difficult, however.

In attempting to account for player choice, keeping track of player decisions and reacting to them with any degree of satisfying specificity can be an arduous task. Should the developers choose to respond in too broad a manner to player action, for attempting to save on development time and cost, they run the risk of not being specific enough. Instead of feeling listened to, players could instead not only feel that their actions didn’t matter in the context of the game, but remember that they are engaging in a game in the first place, taking them out of the experience and derailing player engagement.
Conversely, should the developers attempt to maximize specificity and responsiveness to player actions so that the player always feels listened to and that their choices always matter, they run the risk of a scaling problem. Especially with potential for branching actions, developers can quickly run out of time and resources when trying to write, program, and produce every viable branch of every possible path that the player could choose.

Solving this problem of balancing game responsiveness with developing scale is enormously important, and if a team of developers manages to successfully balance it, their project will be appreciated by the player base for its satisfying responsiveness, while also being of a scale to actually be completed as a project.

An example of this is *Fallout: New Vegas*. New Vegas is largely hailed as one of the greatest video games of all time, not only for its world, story, and gameplay, but primarily in its responsiveness to the player’s actions. The factions the player interacts with, the morality of the actions the player decides, their dialogue choices picked - all of these are satisfyingly reflected in the world state in each task, quest, and story.

Once this delicate balance of responsiveness is found, the reception of a game by its intended audience will likely be much warmer, and the likelihood of the team to complete the game will be much greater.

A world that has perfected this balance of responsiveness will be one of engaging games that respond intelligently to player action, and make players feel listened to and thought of. These players will increase engagement and consumption of these games that they feel they are actualized by, and in turn will spend more of their money and time on those products, further of which will reward the developers and companies that prized this responsiveness in the first place.

Many different developer teams and released games have attempted to solve this task of game responsiveness already and do so in different ways. Games like the branching narrative game *The Walking Dead*, the emergent gameplay classic *Dwarf Fortress*, event-driven procedural story generation games like the *Shadow of Mordor* series, and AI text generation games like *AI Dungeon* all have incredible narrative strengths. These are analyzed in the related works section further into this paper.

However, while the above systems and design philosophies are intricate, impressive, and compelling, they each lack in certain desired aspects. No one system manages to satisfy all the below desired characteristics in a narrative architecture.

My solution to attain a system that addresses all the above points is the Procedurally Adaptive Webbed Narrative system, or PAWN for short.

PAWN is a narrative system for interactive media that intelligently selects and tailors dialogue from prewritten conversation catalogues depending on its generated characters and player agency. Using this intelligent selection of customized dialogue for in-game agent dialogue, it seeks to maximise responsiveness to the game world and player choice by creating impactful dialogue for its generated characters, so that players are more interested and engaged in these agents and therefore the game as a whole.

PAWN tracks specific user input via hard-coded responses to gameplay events, and then translates these events into predicate-logic form objects called Occurrences. Later, derived from these Occurrences are corresponding Predicate objects, that describe character or plot relational information derived from these very set and rigid Occurrences. For added complexity, Narrative Pattern objects then parse these Predicates, and introduce new Predicates
Goal Description
Prioritizes responsiveness to player actions. This system should prioritize providing dialogue and game events that reflect player choice and seem fitting as natural events in the game world.
Explained in-depth for further examination. The system should be publicly available for examination and discussion, facilitating iteration and recreation to expand and improve it.
Based on pre-written dialogue. Despite procedurally selecting dialogues and modifying them to reflect character aspects, dialogue should still be human authored and not completely generated by AI.
Scalable specificity of dialogue entree choice. The dialogue selection of the system should prioritize specificity, selecting the most reflective dialogue of the player situation while still offering more general options when specific ones are absent.

Table 1.1: PAWN System Goals

Based on each Narrative Pattern’s own set prerequisite Predicate objects and the Narrative Pattern’s resulting new Predicates to add. And finally, once prompted to, the system delivers prewritten dialogue exchanges depending on the specific predicates that are existing at the current play time and state.

The PAWN system decides what dialogues to select to react to each predicate via a ranked interest ladder system, where the more specific and interesting a dialogue is judged to be (either at playtime or hard-coded beforehand), the more likely it is to be chosen to be displayed to the player. And finally, in situations where the developers want a more rigid and traditional specific personality for a character in the game world, the Spirit system is implemented, which is an ability for a character’s dialogue to come from a separate pool of responses written specifically for them, as opposed to a larger pool of varying identity genericness.

In the related works section, I’ll discuss other similar papers and systems that attempt to respond to the problem of responsiveness that PAWN attempts to address, and why PAWN is different enough to warrant its own existence as a separate system.

Following related works, I’ll discuss each specific aspect of PAWN, including how it forms agents called Actors in its system, how it captures game input and translates it into Occurrence objects, how it derives the more relational Predicate objects from these Occurrences, how PAWN leverages Narrative Pattern objects into producing new and satisfying Predicates from already existing Predicates pattern, and how the system selects and tailors pre-written dialogue entries based on these existing Predicates.

In the results section, I will elaborate on the testing conducted with the PAWN system and the results gleaned from said testing. When ten testers were given two separate versions of a work-in-progress video game, one with generic context-less dialogue snippets given to players via character dialogue and the other instead using the PAWN system for dictating
character responses, with randomized testing order and no signals as to which implemented my system, all ten testers identified the PAWN system as the more enjoyable, more responsive experience.

Chapter 2

Related Work

Many narrative systems already exist to serve game narrative purposes. In this section, I will examine related systems and elaborate on why PAWN is needed or different from these popular narrative solutions.

2.1 LLM-Generated Narratives

Normal game narratives are pre-written by human authors, but with modern artificial intelligence and machine learning systems, this no longer is the only way to produce legible text for game narratives.

Large Language Models (LLMs) are incredibly sophisticated learning algorithms trained on large corpora of text data. Chat-GPT 3, a model now out-dated by newer versions of 3.5 and 4, was trained on a corpus sizing greater than 450 billion tokens[2]. These models, once trained on appropriate text data, can generate staggeringly human-like natural language responses at runtime, based on user input.

This powerful, responsive text generation has been leveraged into producing responsive narratives in video games. As discussed by Hua and Raley[7], AI Dungeon uses the LLM model Chat-GPT 2 to create stories and guide the user along a text-adventure, with the game’s entire dialogue being generated at runtime in response to user input.

In terms of responsiveness and agency, AI Dungeon using LLMs in this way is completely unmatched in any other method of narrative structure for games. Player choice is limited only to what they can type into a console, and the next step of their adventure is always created by the LLM immediately after the user pressing enter. However, this approach has its limitations.

LLM-driven games like AI-Dungeon still suffer from the same problems many LLM model applications suffer from. Examples of this include the inability for longer-term memory and plot structure planning, and occasionally suffering from hallucinations, which is a habit of LLMs to sometimes completely devolve into nonsense babbling that can completely derail the experience.
Since PAWN consists of modified but pre-written dialogue entries to be shown to the player, it will never be as directly responsive to player input. However, it also has the ability to ‘remember’ and track relevant characteristics of characters, and never hallucinate.

2.2 Branching Narratives

Branching narrative systems are extremely popular for delivering high quality, focused, engaging narratives in video games.

These narratives are planned and thought-out stories designed to change in response to certain choices given to the player. The story changes at very specific, pre-created story junctions designed and created by the game’s developers ahead of time, and depending on the player’s input to that choice, the story will go down a specific path.

This narrative approach has many positives. In comparison to more linear narratives, players typically express more agency and other positive attributes towards games that give them branching narrative options. Participants in the examination of Fridlund and Gustafsson found that, in comparison to more typical linear narratives, branching narratives allowed more player input and were considered by players more interesting and less “boring” [5]. This structure does have drawbacks, however.
Players can only experience stories or make decisions about what has already been created and accounted for by developers. This can lead to a sort of choice bottleneck in narrative creation, wherein the number of choices the player is presented with is smaller than the regular amount of choices in most narrative games, as each choice must then be accounted for and change the player’s game experience in some way.

As well, not as much of the game usually changes between playthroughs as other generative narrative systems. Since the player is selecting between choice A and B and C with little other variance, and each choice leads to a predictable game state A or B or C, repeated game experiences can be viewed as too similar compared to other, more player responsive narrative approaches.

As a final negative, this approach can lead to players discovering that the actions that they view as leading down different story paths and producing future consequences actually sometimes change nothing at all. This illusion of choice can be incredibly frustrating to players, leading them to feel betrayed and wonder why the choice was created in the first place.

2.3 Emergent Narrative

Emergent Narrative systems like Dwarf Fortress craft complex and action-responsive narratives from complex simulations, but ask more imagination from the player for the actions to be perceived as narratives.

As discussed thoroughly by the game’s own developer Tarn Adams, Dwarf Fortress engages numerous complex and intertwining character systems to add complexity to the game experience[1]. From generating scars and wounds for characters, to according to Adams the

Figure 2.2: The Walking Dead: Branching Storylines
more well-received idea of generating each dwarf’s likes and dislikes of things in the simulated world around them, giving these characters endearing personality traits goes a long way with players to adding complexity and quality to the game experience.

These dwarf agents exist in the world and semi-autonomously perform actions related to their environment, player commands, and their own internal variables. When interacting with the world or other agents in the simulated world, these agents perform sequences of actions that can be viewed as interesting or entertaining, and responsive to each other. However, these sequences of actions must be construed as a narrative by the player, and are not explicit.

For example, a dwarf is killed while exploring outside of its fortress with its pet dog. The dog proceeds to wait where the dwarf was killed, assuming its master will return. The recently-perished dwarf indeed does return, but as an angry ghost, and returns to his home fortress and haunts the still-living inhabitants. The dog, still attached to its master, returns with him and joins the assault on the inhabitants.

![Figure 2.3: Dwarf Fortress: A Dwarf and His Loyal Dog](image)

Recollected back, this can be viewed as amazingly intricate behavior, displaying social bonds connecting and echoing through death in an emergent game world. In reality, to see this play out in your own game would be less dramatic than this retelling. The player would need to make these connections themselves without much given to them in terms of direct dialogue or referential actions. They would need to imagine that this dog loved his master beyond death and what emotions that might entail, and would need to imagine how this ghastly dwarf felt haunting his old residence, because no dialogue is ever spoken reflecting any of these in-game actions.

### 2.4 Event-Driven Procedural Narrative Systems

Event driven procedural narrative systems derive plot and narrative points from both simulated game logic and player choices, generating unique and new stories at runtime while maintaining a polished, handcrafted feel to the characters and interactions they undergo with the player.
The premier commercial example of this type of narrative system is the NEMESIS system found in the *Shadow of Mordor* series of games, created by Monolith Productions and published by Warner Bros. Examined thoroughly by Parosu, Hage, and Magnusson, these games implement a complex event-driven procedural narrative system that focuses on developing interesting and evolving player-NPC (Non-Player Character) relationships via conflict[6].

The main enemy of the game, orcs, battle with the player in the game’s third-person combat system. Some of these orcs, however, are elevated to be Nemeses, either through the action of killing the player in combat or by need of the system for a Nemesis for narrative purposes.

While the hordes of enemy orcs the player defeats throughout the game are largely unremarkable, a select few orcs can become evolved into a more complex and crafty foe - a Nemesis. These orcs are special gameplay-wise in their increased battle difficulty, but are most interesting in their newly-given identity.

These Nemeses are taken from an expansive pool of procedural character archetypes, with pre-written character histories that match the character and how they were established into the player’s experience. For example, one nemesis type is called ”The Poet”. While his real name is procedurally generated each run, he is always dressed resembling Shakespeare, and his dialogue always rhymes.

Every Nemesis, including the Poet has a large, pre-created library of dialogue created for selective usage depending on whatever game events are linking the player and the Nemesis. For every game event the player can involve with the Poet, dialogues are written, voiced, animated, and compiled ahead of time, and then depending on what events are in the recent context of the player and their nemesis, those specific dialogues are played.

For example, should the poet be losing the encounter, there is a chance that his action is chosen for him to flee. He will announce his retreating in a lyrical manner, and then should the player not be able to stop him, he will successfully flee and the battle will conclude.
The system adds increasing complexity, however, in its chaining together of player and nemesis interactions in influencing the Nemesis’s dialogue. For example, should the Nemesis have previously fled from the player, as the Poet did above, the next time they encounter each other, the Poet may announce that last time was the only time he’ll ever flee the player, and that today the player would be the one fleeing. This referential history to previous encounters helps tremendously with establishing players in these procedural character’s identities, and in forming lasting relationships with these NPCs.

However, there are several drawbacks for systems like Nemesis. For one, this system was created using internal and complex architectures that are not publicly available or explained. Beyond that, it is actually patented, so those that attempt to understand, improve, or iterate on this system risk exposure to lawsuits by the Shadow of Mordor series developer.

Beyond lawsuits, the sheer amount of character design, 3d modeling, script writing, animating, voice-over work, and programming needed to create each Nemesis in the system is immense, in both financial and labor terms. These costs are so high, that likely all but the most prepared, well-financed teams of experienced developers would prefer to use some other system of narrative.

Now having examined these popular but partially limited narrative systems, I will discuss my own system proposal to cover areas that I view these other systems as lacking.

Chapter 3
PAWN System

The PAWN system is outlined in this section. I will first describe a custom created game base for the PAWN system to be tested in, then expand on every asset of the system itself, then examine the 'lifetime' of a hypothetical PAWN narrative snippet.

3.1 Game Background

Needed for testing the PAWN System is a basic gaming framework that it can be implemented into for testing and for launching narratives from. As well, for testing, this gaming framework needs to have the ability to exist as a playable experience without the PAWN System implemented, and also have the ability for the PAWN System when active to seamlessly be incorporated into the game to present to the player as a cohesive whole.

When deciding on what gameplay framework to select, many factors were considered.
The playable base of the framework must be small-scale enough for implementation and sophisticated usage and testing by myself while under time constraints of graduate school and work.

The framework’s narrative must be text based in presentation, since the alternatives of voice acting, 3D modeling, and animation are all too expensive for me to produce in terms of time and resources.

It must provide numerous interesting and varied game play events for dialogue entries to be spawned from.

And ideally, the game base is fun to play on its own, regardless of any narrative system later integrated into it.

Considering all of these factors, I decided to create a framework game base heavily inspired by the 2012 indie hit “FTL: Faster Than Light” (referred to going forward as FTL for brevity).

FTL is a 2D space-faring roguelike PC game. Released in September 2012 by indie developer studio Subset Games, FTL consists of the player controlling the crew of a spaceship, eluding capture by a pursuing rebel fleet. The player must guide their chosen ship through numerous space sectors, each with a bevy of events and battles all generated in unique arrangement for each playthrough in typical roguelike tradition. While battling pirates and rebel ships, players can also upgrade their ship, buy new weapons and systems from shops, and bring aboard new crew members. If during combat, the player ship is destroyed or all the player crew is lost, then the game ends and the player is returned to the initial menus of the game, to select a new ship and crew and try again.
For the player to progress, they must travel across eight galactic sectors, filled with randomly placed beacons, which act as the game’s levels. From the beacon at which they currently reside, the player uses fuel to travel to a neighboring beacon, which then begins a usually random level encounter. These level encounters can contain prompts to fight rogue pirates, help innocent civilian ships or colonies, or trade with merchants.

Critically, these levels are displayed to the player through simple dialogue menus. As opposed to images and renders of speaking characters or complicated environments, simple text describes anything more complicated than the procedurally generated pixel-art cosmic backgrounds. Any established characters important to narrative are limited to one beacon, and are rarely developed enough to even be given a name (very rarely, for the few number of quests in the game related to unlocking new ships, some alien characters are given names, but this is exceedingly rare. In the whole game this occurs only a handful of times).

This dialogue display system for conveying narrative to the player is ideal for PAWN integration. Needing nothing more than different dialogue to produce new levels fits this project’s scope well, and serves as an excellent testing ground for PAWN integration.

Also as described briefly above, a core gameplay component in FTL is the management of the player’s ship’s crew. Each run, the player’s ship is populated by up to eight crew members that can be ordered to man a specific system (this manning act provides bonuses to aspects of the player ship). Beyond manning systems, crew can repair a damaged system, fight off intruders, and put out fires raging in the player’s ship. These crew each have a defined species (human or one of several different alien species types), appearance, and name, which is changeable by the player at any time.
These crew provide bonding opportunities for the player through experience levels. As a crew member is used more, a limited number of tasks when repeated by that unit raises its adeptness at that specific task. For example, a crew member that frequently is tasked by the player with manning the shields system becomes ‘better’ at manning that system, improving the gameplay bonus that crew provides when manning it.

Crew take damage from battle, whether from projectiles and bombs fired from enemy ships, fighting fires, suffocating from lack of oxygen, or throwing down with enemy crew members in hand to hand combat. Crew can normally be healed with a working medbay ship system, but should the crew member’s health reach zero, that crew will permanently die if not saved by additional ship systems.

This ability of crew members to grow into a role, as well as to be renamed by the player, and finally the ability to die once their health is reduced enough, combines together in effect to punch above its weight when it comes to endearing these characters to the player despite their relatively simple programmatic complexity. In the online FTL community it is very common for players to mourn crew that have served them faithfully and fallen in the line of duty.

Finally, while the last thing I’d call FTL is simple or easy to implement as a recreation, compared to recreating other potential games with established and proven successful formulas as a testing base, there were no clear second choices in my view for alternative frameworks, especially considering the ease of level creation limited to text and a solidified avenue for the PAWN System to express itself through (the crew members).

So, with FTL having been selected as the choice for recreation, over the past several years I have (painstakingly) recreated each system and mechanic of the game to act as a playable testing base for this thesis project.
I have named it “WTP: Walk the Plank” (referred to as WTP going forward for brevity), taking direct inspiration from FTL for the name as well as differing it enough to not infringe on FTL’s trademark. WTP is nautical themed instead of space themed, which is what the titular ‘Walk the Plank’ is inspired by. Created in the Unity game engine in the C# programming language, development has been underway from mid-2019 until present day 2024, with public release intended but not yet definitive.

As a close recreation, WTP contains virtually every element explained in detail above, including battle, crew, ships, and player directing of their crew.

WTP contains a level and dialogue system identical to FTL, with players traversing between encounters by selecting neighboring map nodes and interacting with these levels through dialogue menus.

WTP also contains crew that mirror the FTL crew in their ability for players to grow attached to them and their ability to permanently die from taking too much damage.

Purposefully absent from the WTP crew system however is the ability for the player to name their crew members. This is an intentional design choice as a part of the PAWN System, as one of the large areas for improvement in the base FTL formula is the lack of personality of the crew members. Characters in WTP are treated as real characters as opposed to completely controllable puppets, and as one small reflection of this, you cannot repeatedly change their name. Instead, their names are generated as a part of the character’s internal history. This is part of an initiative on behalf of WTP to make them feel like real characters instead of game puppets, and is one of the largest goals of PAWN.
3.2 Actors

In the PAWN System, Actors are programmatic objects representing narrative agents that exist and create stories in a narrative web. Each existing character (also called units) in the player world necessitates the creation, storage, and assignment of an Actor object to it.

3.2.1 Actor Creation

In the Unity game engine WTP is created from, everything in the game world is a GameObject. A GameObject is simply a collection of scripts, which themselves are just bundles of computer instructions. These scripts control the GameObjects, which control everything about the game itself.

In WTP, every unit in the game is a GameObject, but since they exist in the PAWN System, every unit in the game is assigned an Actor object component script. This Actor object acts as the mechanism through which this plain game-play puppet is shown to the player as an intelligent, real, responsive generated character. All the personality that can be taken from a unit in this game will be derived from the Actor logic object stored in the unit itself. These Actor objects are created from the ActorCreator script, and nowhere else.

Firstly, a call from somewhere else in the WTP framework initiates the process of Actor creation. Usually, this happens as a result of the game beginning, necessitating player crew generation. These calls will pass in an ActorSpawnInformation object.
These are objects representing initial spawn information for Actors. Critical personal character information needed in the Actor creation process is usually decided beforehand and contained in these objects, and they are referenced constantly throughout the Actor creation pipeline. However, should any of these aspects be missing from the initial ActorSpawnInformation call, ActorCreator will treat this as a request to be randomly generated at runtime, and will do so.

Using whatever information is contained in the given ActorSpawnInformation object, the ActorCreator first spawns the empty Actor shell and stores the ActorSpawnInformation in it for later reference.

Using this Actor reference, it then furnishes the Actor with personality information, starting with translating the ActorSpawnInformation order into an ActorStats object.

ActorStats are interior objects stored in each Actor generated during Actor creation and used for fetching personal, procedural variables about this specific character at runtime. The unique, random, personal qualities that make an Actor who they are in the game are stored in these objects. Such characteristics as ID, Race, Gender, and more are contained in this object.

But initially, this ActorStats object is empty. It is then furnished with information either already decided from ActorSpawnInformation, or randomly decided.

Before the more front-facing and personal variables like name, race, and gender are decided, the Actor’s ActorStats is given a unique ID. This is an absolutely unique string ID in the form of a numerical number. No other created Actors will ever share this ID, and it will never be reused in the same play through for other actors, even if this Actor is destroyed and never used again.
This ID is used in two main ways. First, an Actor’s ID is the identifier used for showing an Actor’s participation for any of the PAWN system’s relational objects. A Predicate consisting of “becomesCrewmates(10,15)” reflects the process of two crew mates becoming teammates, those crew mates having IDs of 10 and 15 respectively. Second, ID’s are instrumental in fetching Actors from relevant Actor lists and dictionaries in ActorDatabase, which is elaborated on later in this chapter.

After the Actor’s ID is stored in its ActorStats object, its other needed qualities are either translated from ActorSpawnInformation or randomly decided and put into ActorStats.

First among these qualities is race. An Actor’s race variable is an enumeration of type RACE that reflects an actor’s race (meant in the fantasy sense, not in the ethnographic sense). In the fantasy nautical world of WTP, numerous races roam the seas, and each Actor’s specific race is tracked via an initially established enumeration variable called race. There are 7 exclusive race possibilities: human, elf, orc, dwarf, cuttlefish, insectoid, and frog-person.

An Actor’s race is a large determinant of their character makeup and how they will react generically to events that occur in the game. As well, an Actor’s race influences their name and appearance during gameplay. For example, an elf’s name is generated from pools of first and last names that differ drastically from orc naming pools. After the initial assignments, races cannot be changed during the course of the game.

Another quality is Spirit. Spirit is a unique and important mechanic in the PAWN system. A Spirit for an Actor is an optional parameter string that represents what ‘Spirit’ this Actor will be given once spawned. If left null, no Spirit will be given to the created Actor object. If this string is anything besides null, ActorCreator will assign the Actor object the inputted string as a Spirit. How this string will be used to affect the character is described in more detail in a later section.

A third important quality is Gender, a string variable that influences their initial first and last name generation, as well as the Actor’s pronoun usage throughout the game.
Gender is decided randomly, unless specified differently in the originating ActorSpawn-Information order. In FTL, gender is only visually used in human units, as they are the only sexually dimorphic race included in the game. WTP is largely the same, as humans are still the only race or species that appear different visually based on gender (though gender still affects other unit aspects, such as pronoun usage and name generation).

Gender is represented as a string as opposed to a boolean despite only male and female genders being currently implemented because further expansion of the system is possible in the concept of multiple or non-conventional genders, not only in humans but also in some of the more biologically alien species such as the frogpeople or insectoid races.

Next, the Actor’s name is generated according to the information previously decided, like name, gender, and race. This generating process considers the passed-in Actor’s gender and, based on that, it grabs a random value from the relevant string arrays corresponding to the actor’s gender and race. If the Actor is male, the first and last names grabbed are from male-[race of actor]-[first or last name] array. Those names are then assigned to the Actor’s ActorStats variables of firstName and lastName, to be used later for conversations at runtime and for displaying to the player in CharacterDisplayModules. For example, a female orc would randomly select a first name from the list ‘female-orc-firstName’ and a last name then from ‘orc-lastNames’. Some generated names include ‘Radnok Grimlaw’ for a male orc, or ‘Ceria Starlight’ for a female elf.

A final decided aspect of ActorStats is Crew. Crew is a boolean, and is used primarily for later sorting by ActorDatabase. Essentially, this variable determines whether or not this Actor is intended to ever be a part of the player’s retinue, or in enemy ships and encounters. In the case given above of beginning game Actor generation, every Actor here would have a Crew value of true. Some Actors have Crew values of false, usually in the generated crew of enemy ships. As of now, these Actors have no method to express their personality, and can never join the player crew.

To conclude the process, the finally created Actor is passed to two separate places: it is returned to whatever method prompted its creation for further usage, and is given to ActorDatabase, for sorting and storage.

The above-described process is the usual method of Actor creation, but there does exist an exception to this. Rarely, ‘empty’ Actors are needed. Sometimes, relational logic is launched that only truly involves one actor, but a second is needed for reference and/or context. For example, an instance of the Occurrence ‘hurtBy(tom, cannonBall)’ is possible only if ‘cannonBall’ is created and stored as an Actor. However, ‘cannonBall’ should not be treated as a real Actor for most situations, so an Actor object with virtually no information with race EMPTY is created and sent to ActorDatabase.

### 3.2.2 Actor

An Actor object itself has many different components to its makeup. Below are some of the most important variables needed for explanation.

All Actors have a Spirit object variable. Spirits are simple string objects used for identifying unique Actor identities ‘put into’ Spirit-less actors. Mostly, Spirits are just string call-signs used for fetching unique and specific responses and dialogues for the Actor that contains them.
Spirits are given to Actors either from spawn or in specific ‘evolution’ circumstances, depending on what that Spirit requires as precondition(s) to form into an existing Actor. An example could be something as simple as a character almost dying from combat. If this event is recognized and decided to be used for generating a Spirit, a new Spirit object with a name variable of “NearDeathExperience” would be generated and stored in this Actor’s Spirit variable. Going forward, when this Actor’s ID is used for any type of Predicate or narrative logic, instead of using more generic responses based on the actor’s race or just generic dialogue tags, their Spirit is instead used to call up potential conversations to be displayed to the player.

The point of this Spirit dynamic is to establish a flexible system for using concrete, pre-written dialogue and conversations with in-game characters in a way that reflects their game history while still being able to use generic dialogues when nothing notable has yet happened to the Actor to trigger an evolution into a Spirit. Using this system, some characters can use a smattering of more generic dialogue, written for and usable by any characters that exist or are a specific race, and others can have more hand-crafted personalities emerge, like the 'Morose’ Spirit used in the initial testing of WTP (this Spirit makes the Actor respond to each situation miserably).

Actors have a connection to their Walker. Walker in this context is simply a term for the game representation object for the Actor in the world. This Walker is a reference to an existing instance of Walker.cs, a script that controls the in-game body of Actors. Walker components for an Actor are only assigned if the Actor currently exists in the world and is not in storage.

If active in the world and a part of the player’s crew, Actors also contain a reference to a CrewDisplayModule. This is the Actor’s informational portal display to the player, that allows them to see information about the Actor, such as their current health, sprite image, generated name, and, when the Actor has a conversation ready to display, contains an interactable button to launch that conversation.

The last notable aspect of Actor is NextConversation. This is a holding variable for the next PotentialConversation object that this Actor is ready to initiate. This is the potential conversation judged to be most suitable from its batch-mates derived from Predicates involving this crew member. All of this is expanded upon more below.

### 3.2.3 Potential Conversations

Actors also have a list of PotentialConversation objects. This is a list full of objects that this Actor can call upon when prompted to display a conversation to the player. And as it is named, these conversations are potential, meaning that not every conversation in this list will be played or even seriously considered.

When an Actor responds to new logical Predicates involving itself, a part of that response process is collecting all possible dialogues relating the Actor to this new logic that it could respond with, and then storing references to all of those responsive conversations in this list. The conversations are each stored as PotentialConversation objects.

PotentialConversation objects are simply bundles of information relevant to an Actor launching a potential conversation.
PotentialConversation objects include the variable DialogueName, a string reference and the most critical aspect of a PotentialConversation. This string represents the title of the specific dialogue that exists in the dialogue database to be played should this PotentialConversation be selected to be ‘said’ by the actor.

PotentialConversation objects also include references to all the instigating logic that spawned them, such as the creator Predicate and originating encounter (both expanded on later).

The relevant Conversation object is also stored in PotentialConversation. This is a reference giving the PotentialConversation the specific conversation entry object that it refers to. Instead of a name of the dialogue like DialogueName, this is a reference to the specific conversational object titled DialogueName’s value. It is useful for retrieving the below variables, like InterestLevel, ExpiryNumber, and FastPass.

InterestLevel is a PotentialConversation integer originating from the pre-written conversation entry. This integer reflects how interesting this conversation is judged to be on a numeric scale. This is used when later sifting through many of the Actor’s PotentialConversations, and typically the PotentialConversation with the highest interest level is selected to be relayed to the player. Most of the time, the more specific the conversation’s prerequisites are, the higher the assigned interest level. For example, Predicates that spawn from Narrative Patterns (patterns of existing Predicates that introduce new Predicates) are almost always higher interest level than Predicates that spawn directly from game events, and the greater amount of prerequisite Predicates that a Narrative Pattern requires, the higher the interest its resulting Predicates typically are.

ExpiryNumber is a PotentialConversation integer value which is also predetermined, just as InterestLevel is, from the pre-written conversation entry. This number reflects how many encounters the player passes that a PotentialConversation can still exist, remain in ‘memory’ so to speak, before it is culled for being too old and outdated. This is useful for removing PotentialConversation objects that, while they were suitable to be shown to the player at a certain point, would be too strange to play going forward as the events that conversation would reflect on would be too far in the past. An example of an outdated PotentialConversation could be one of two crew members remarking on how close the battle was, and how scary the enemy ship was, but the referenced conflict was several encounters ago, leading to confusion from the player as the most recent battle might have been remarkably easy, or possibly the player might not have even battled in the time it took for this conversation to be selected.

The FastPass boolean is also sourced from the conversation database side of the system, and stored in each PotentialConversation. Should FastPass be marked true, when this PotentialConversation is formed and submitted and checked for usage later by the WebWeaver component (the main script in the PAWN system that orchestrates all of these different parts), it will interrupt current game processes and level flows and launch this conversation as soon as possible. This functionality is critical for events that need to happen immediately, as opposed to reflections after the level (usually combat) has subsided. For example, very rarely, when the player ship is about to lose its last points of health, a FastPass PotentialConversation called ‘sacrifice’ can be launched that, instead of being processed after the level which would result in the player’s loss, will interrupt the battle with a dialogue that estab-
lishes that one of the player’s more loyal crew members are sacrificing themselves to blow up the enemy ship from the inside, saving the player and their remaining crew members. Without FastPass as a feature, dialogue events like this could not occur.

The integer batchID is assigned on PotentialConversation creation. For each encounter that launches bunches of PotentialConversation objects, each PotentialConversation launched from the same encounter has the same batchID. Then, when a PotentialConversation is selected for usage, all other PotentialConversation objects that share the same batchID are removed from possible selection lists for that crew member. This is done to not overly bloat the potentialConversation lists.

### 3.2.4 Actor Database

The script ActorDatabase acts as a central repository and reference for Actors in WTP. While the actual creation of Actors is handled in ActorCreator, upon creation, ActorDatabase stores and sorts Actors into different lists and dictionaries for usage from different facets of the PAWN System.

Immediately after every Actor is created, it is given to ActorDatabase for sorting into a variety of useful lists. An Actor can reside in several lists at a time, though some are exclusionary to others. For example, every Actor is placed into the AllActors list, and placed into the highly useful ActorDict with their ID as their key value, but only active Actors, those that are currently in the game attached to a Walker unit, are placed into the ActiveActors list. Actors are sorted based on whether or not they are crew members, their active status, their life state, and more.

These lists are used for the variety of fetch methods contained in ActorDatabase. Actors can be quickly retrieved via a reference to their ID using ActorDict, or found via being the mystery missing part of a relational Predicate ("find me actor ? in the Predicate hitsWithHammer(? , Jerry)"); Actors can even be found by any other variable state if they exist, though this retrieval method would be very slow and involve much iterating through large Actor lists.

Near the tail-end of the initial Actor acceptance by ActorDatabase, it also calls the methods to link these Units to their assigned in-game Walker units. When being sorted into lists, if the Actor has an assigned Walker reference, here ActorDatabase actually links these two to each other.

Now that Actors are described in-depth and can be usefully conversed about, I will explain the more logic-oriented side of PAWN, starting with Occurrence objects.

### 3.3 Occurrences

An Occurrence in the context of the PAWN System is a plain matter-of-fact record of a specific gameplay happening, typically including at least one actor, but usually two. An example of an Occurrence could be thought of in the form of “becomesCrewmatesWith(A, B)”, with “becomesCrewmatesWith” being the Occurrence type, and “A” and “B” being stand-in variables that will be replaced by the participating Actors when the Occurrence is launched at runtime.
When a game event occurs that needs to be reacted to, whatever game method handles the logic for that event also calls to a static listener script called OccurenceLauncher to create an Occurrence to track this game event.

The Occurrence is created by OccurenceLauncher using information given to it by whatever initiated the call in the first place. These calls are varied and happen from all over the project’s script space.

The type of Occurrence is hard-coded into the initial call, but the involved Actors will vary each time an Occurrence is created, assuming the instigating Actors themselves change.

Occurrences are vital to the PAWN System in many ways. As mentioned above, Occurrences are the life blood of tracking what events happened and what Actors they involved. Without Occurrences, there would be no way to reference game events to the player in any real way outside of hard-coding boolean flags for specific events. As well, Occurrences are used for the creation of Predicates, the more relational logical objects used in the PAWN System. A series of Predicates are linked to each Occurrence, and once that Occurrence has been created as an event in the game, eventually it is looked at to be harvested. This harvesting process forms Predicates from each Occurrence, so the main way Predicates are formed in the game is through harvesting Occurrences. Without Occurrences, there are no Predicates, and no relationships between characters, and the entire PAWN end goal of specific, unique conversations that breathe personality into these characters and make the player feel listened to is missed.

### 3.3.1 The Occurrence as an Object

The Occurrence class is actually quite simple for how vital a function it serves in the greater ecosystem of the PAWN System.

It contains the two string variables ‘from’ and ‘to’. These strings capture the involved Actor ID’s of who or whatever was involved in the encounter. These are used for, among other things, transferring relational data about who was involved in the encounter to relevant needed references for the Predicates spawned from this encounter. As well, it serves as a reference and record about who did what, or what happened to whom, etc.
Occurrences also contain the string descriptor ’type’. Type is a unique identifier for what this Occurrence actually is. An Occurrence object representing the perishing of a unit would be of type “dies”. An Occurrence object representing a fire burning a unit and inflicting harm upon them would be of type “isHurtByFire”. Type acts as a unique ID for each Occurrence, so an Occurrence type is never repeated unless the event itself is repeated.

Occurrences also contain a list of ’Potential Predicates’. This is a list of custom objects of type “PredNamesAndOddsOfSpawning”. Each of these objects represents a potential Predicate to spawn from this specific Occurrence. Each PredNamesAndOddsOfSpawning object has two parts: ’name’ is a string descriptor used for finding the real Predicate this object is linking the Occurrence to, and the integer ’odds’. Odds is an integer that controls the likelihood of the Predicate of type ’name’ to spawn once this Occurrence is harvested. Whatever Odds is, will be compared to a random value out of 100. If the random value is less than Odds, the Predicate of type name will be spawned. Otherwise this PredNameAndOddsOfSpawning object will be ignored and not launch a Predicate of type ’name’, even though it’s being harvested for its Predicates. Though, should ’odds’ be zero or less, that Predicate will always succeed this odds check. Most of PAWN’s Occurrence’s Potential Predicates are this way, but the optional odds functionality is useful and used enough to justify its existence.

3.3.2 The Occurrence Spawner

OccurrenceSpawner is the front-facing intermediary between the large, varied collection of places around the entire project that can call Occurrences, and the definitive Occurrence spawning methods contained inside OccurrenceHandler.

Throughout the project, hard-coded references to spawn Occurrences are embedded in various gameplay methods. Once done performing their game tasks, these methods contact other specific methods contained here to create Occurrences with specific types reflecting the originating game play event. Unlike the unchanging event types from these methods, Actor involvement is constantly changing depending on which Actors were involved in the instigating gameplay event.

After receiving these varied method calls to spawn specific Occurrences, OccurrenceSpawner sorts these calls to either of two specific OccurrenceLaunching methods: Spawn and SpawnOncePerEncounter.

A general purpose spawning method, Spawn is used when the game event launching the Occurrence is relatively ordinary, has no potential for quickly repeating, and has no special considerations. It simply passes the information gotten from the method call to OccurrenceHandler’s main Occurrence spawning method.

Example of this: when a run starts, Actors selected to be a part of the player’s crew are established as crew members with each other via a looping “becomesCrewmatesWith” Occurrence call. When done, every unit of the player’s crew will have an Occurrence linking it to be crew with each other unit of the crew.

SpawnOncePerEncounter is a method that functions similarly to the above Spawn method, but with one significant change. Before calling OccurrenceHandler’s SpawnOccurrence method with the given information, it uses a lambda function to quickly check if in the Occurrence-Handler’s current level history there already exists an Occurrence matching this call’s ’type’,
from’ Actor, and ‘to’ Actor exactly. If such a matching Occurrence does exist, then the spawning of this Occurrence is aborted. If no matching Occurrence is found, this method is functionally identical to the original Spawn method above.

This method is critical for limiting the amount of Occurrences that could happen frequently each level. For instance, “isHurtByCannonball” is a type of Occurrence that will happen many times each encounter if simply using Spawn, since cannonballs are the most common method of damaging units in the game. But ten identical Occurrences describing the Actor Tom being hit by a cannonball would be useless. Even more dangerous would be “isHurtByFire”, a method that gradually reduces the health of units in real time by decrementing their float values’ health every tenth of a second. If ten times a second identical Occurrences were being spawned for one, two sometimes four or more units taking four to ten seconds to put out fires - the amount of repetitive and useless Occurrences created and stored adds up quickly. This method sidesteps these issues by quickly checking if these Occurrences have already been established, and if so avoiding their duplication.

While this script contains other smaller functions, its main job is established as written above: it collects calls to spawn Occurrence objects from throughout the WTP framework, organizes them, limits those that are needlessly repetitive, and then passes along the correct orders to the next part of this document, the OccurrenceHandler.

### 3.3.3 The Occurrence Handler

The OccurrenceHandler script is the main receptacle and acting vessel for any actions and organizing that is needed in relation to Occurrences. At a high level, this component receives commands to create new Occurrence objects from throughout the WTP framework, creates them, fills them with information, and stores and sorts them into separate lists for archival and active usage uses.

On game initialization, this script spawns and forms a reference Occurrence dictionary. This dictionary is filled from a JSON file called OccurrenceReference. For every Occurrence used in PAWN, this JSON file holds the name and possible Predicates that can be spawned from them. They are stored in the following format:

```json
Name: NameOfThisOccurrence,
ResultingPredicates: [
{
Name: NameOfPredicate,
Odds: 50
}
], ...
```

Values for each Occurrence are taken from each object in this JSON file, and stored in this reference dictionary with their NameOfThisOccurrence value being used as the key and a list of the PredNamesAndOddsOfSpawning objects in the JSON’s ResultingPredicates list used as the dictionary slot’s value.
Now that our reference dictionary of Occurrences is established, we are able to accept calls for new Occurrences and create them.

SpawnOccurrence is the main method for this class, and does exactly what its name implies; it acts as the main and truly only method that creates objects of type Occurrence. This is the method that pre-processing methods in OccurrenceSpawner call.

This method takes in three string parameters: ‘name’, ‘from’, and ‘to’. These parameters act as the matching name for the Occurrence to be spawned, and ‘from’ and ‘to’ are ID references to the Actors involved in the Occurrence.

An example of these parameters could be a name value of “tricks”, ‘from’ of 11 (matching the ID of Actor ‘Tom’), and ‘to’ of 22 (matching the ID of Actor ‘Jerry’), for a final form of “tricks(11,22)”.

Using these parameters, SpawnOccurrence finds a list of JSON PredNamesAndOddsOfSpawning objects via using ‘name’ as a key to query the dictionary we established above. It uses the returned list of PredNamesAndOddsOfSpawning objects as the final piece to actually generate the new Occurrence.

This newly-created Occurrence object will have: ‘name’ as what type the Occurrence is, ‘to’ and ‘from’ as ID references to the Actors involved in this Occurrence, and the Predicates list taken from the PredNamesAndOddsOfSpawning objects as the resulting Predicates to be checked for spawning when this Occurrence is harvested (discussed later).

Once spawned, this new Occurrence is stored in a variety of different lists. Initially it is placed into the list ‘unharvestedOccurrences’ to be harvested for its Predicates later, then placed into allOccurrences for any needed specific referencing/querying/debug displaying, and then finally into currentLevelReport so that there is a record that this Occurrence was created at this time with these other Occurrences.

AllOccurrences is simply a reference list of every created Occurrence that has existed in this playthrough. It is mainly useful for debugging, so during runtime should this list grow too large and computationally taxing, it can be emptied at will.

A LevelReport object holds information relevant to each level of the game the player has either already played or is playing currently. The way that it tracks information about these levels is simply by storing references to each Occurrence that spawns in that level. This in turn reflects the factual game events that happened on this level and what Actors they involved. This is useful for any type of specific callbacks that author’s using the PAWN system would like to do. The CurrentLevelReport mentioned above specifically is the LevelReport object representing the current level the player is experiencing. As Occurrences of the current level are spawned, they are additionally stored in this report’s Occurrences variable. A new LevelReport object is spawned and assigned for every new destination the player travels to, and later sealed and stored into a large list when the player moves to a new encounter.

UnharvestedOccurrences is a temporary sorting list for Occurrences to reside in. This list is where Occurrence objects wait, until the harvesting process is undergone. As mentioned previously, harvesting is the term given to describe the process of taking each Occurrence that has yet to be examined in unharvestedPredicates for spawning its Predicates, and then deriving the potential Predicates linked to each of those Occurrences.

An example of the harvesting process:
1. Resulting from a game event, Occurrence “tricks(11,22)” is created and stored in the unharvestedOccurrences list, allOccurrences list, and currentLevelReport.Occurrences list by the SpawnOccurrence() method.

2. Upon the end of a level, or a sudden event that wants to update the logic of the PAWN System immediately, the method HarvestOccurrences() is called from an outside script.

3. The “tricks(11,22)” Occurrence is looped through by HarvestOccurrences() since it is in the unharvestedOccurrences list, and each PredNamesAndOddsOfSpawning object in its Predicates list is sent to PredicateHandler.cs for potential spawning based on the odds included in that PredNamesAndOddsOfSpawning object.

4. Later, when certain display menus are activated to observe which Occurrences are in what lists, “tricks(11,22)” is accessed via the allOccurrences list and shown to players.

5. When the level “tricks(11,22)” was spawned in concludes, the currentLevelReport is sealed with a reference to it inside, to be used later in specific situations.

### 3.3.4 Occurrence Formation Example

Here is an example of an Occurrence’s life, from instigating event to storage.

- The enemy ship launches a cannonball at the player ship.
- Upon contact with the ship, the cannonball explodes, dealing damage to the system in the targeted ship room.
- The player then sends specific actors, named Tom and Jerry, to repair the system after it took damage from this cannonball.
- Upon starting to repair the room, embedded in the method of being repaired, the system recognizes that this should launch an Occurrence.
- The system being repaired calls OccurrenceSpawner.SpawnOccurrenceOncePerEncounter, giving it the name “repairedSystemTogether”, ID of Actor Tom, and ID of Actor Jerry.
- OccurrenceSpawner.SpawnOccurrenceOncePerEncounter checks if this Encounter has already been made this encounter, and finding it has not, sends this information to OccurrenceHandler.SpawnOccurrence
- OccurrenceHandler.SpawnOccurrence takes the type and involved Actors of this Occurrence-to-be, and spawns a new empty Occurrence object, and furnishes this object with the type given to it as parameters, as well as the involved Actors.
- The newly created Occurrence is placed into the UnharvestedOccurrence list, to be harvested later for its Predicates.

After being formed, these Occurrences are used for creating similar but different logical objects called Predicates.
3.4 Predicates

Predicates are the lifeblood of the PAWN System. Predicates are simply pieces of Predicate logic describing some type of relation between two Actors in the PAWN System.

Whereas Occurrences are intended to be a reflection of matter-of-fact chronicle of events, Predicates are more geared towards describing Actor relationships or reactions to those matter-of-fact events that happen in the game.

For example, while an Occurrence might be of type “hitByCannonball”, Predicates derived from it might include types like “wasHitByCannonball”, “scarredByCannonball”, “scaredOfCannonballs”, etc. Sometimes Predicates still sound matter-of-record, but mostly they are intended and used to describe personal feelings of Actors to the game events occurring around them in the game and to each other.

Predicates are harvested from existing and unharvested Occurrence objects, therefore every Predicate has a direct ancestral Occurrence that links their creation to some specific game event.

Predicates can also be created via specific prerequisite patterns of Predicates already being in the mix when the main Predicate list is analyzed. When these patterns are recognized, new Predicates are introduced that are related to the prerequisite predicates. These patterns are called Narrative Patterns, and will be explained more in-depth later in the document, but for now just accept that Predicates can also be spawned from these objects.

Predicates are stored in JSON for reference for runtime creation in the following syntax:
PredicateName( [Main Actor ID], [Secondary Actor ID])

'PredicateName' signifies what type of Predicate this is, and acts as the main descriptor of it. '[Main Actor ID]' is the ID of the Actor which is the primary instigator of the predicate. This could also be thought of as the “From” or “sender” of the action of the predicate. Finally, '[Secondary Actor ID]' is the ID of the Actor that is the recipient of the Predicate being initiated or caused by the first Actor. This can also be thought of as the “To” or “target” of the relational predicate.

Predicates are used for determining which dialogues are to be played when for which Actors. All Predicates have an attached interest level integer, so when the game requires a dialogue to be displayed to the player, the potential Predicates are queried for those deemed most interesting to be played and that has an acceptable linked conversation to be played. Once the most interesting conversation is found via the most interesting predicate, it is displayed to the player as coming from the relevant primary Actor of that predicate.

3.4.1 The Predicate as an Object

The Predicate class is the object class for the relational Predicates used in the PAWN System. They typically, but not always, take the following form.

whatHappened( IDofWhoDidIt, IDOfWhoItHappenedTo )

Type is the main descriptor variable of a predicate. This string describes what is the actual idea that this Predicate is representing. It is the initial word in the predicate’s logical form, in the above example ‘whatHappened’.

As a stylistic choice to visually differentiate them from Occurrences, types for Predicates are always in the past tense, as opposed to the occurrence’s present tense,

From and To are vital string variables contained in each Predicate that hold the IDs of which Actors are involved in the relation.

‘From’ contains a string matching the ID of the Actor ‘committing’ this relation, referred to going forward as the primary Actor. In the above example, it is ‘IDofWhoDidIt’.

‘To’ contains a string matching the ID of the Actor ‘receiving’ this relation, which will be referred to going forward as the secondary Actor. In the above example, it is 'ID-OfWhoItHappenedTo'.

Below is a full example Predicate.

crewmatesWith(11, 22)

In this example, ‘crewmatesWith’ is the Predicate type, ‘11’ is the ID of the primary Actor, and ‘22’ is the ID of the secondary Actor.

Besides these relational variables, Predicates contain multiple values useful for its later potential selection into conversations.

InterestLevel is an integer value taken directly from the JSON Predicate list and remains static for the entire Predicate’s existence. This integer is used for evaluating how engaging this Predicate would be to use for a dialogue, in comparison to other potential Predicates to derive conversations from.
Usually, the more specific, obscure, or rare a Predicate is, the higher its interest level will be. As well, Predicates spawned from NarrativePatterns are typically given higher interest levels than their counterparts which are spawned from normal game play occurrences.

These interest levels are NOT calculated at runtime. In the original JSON text, as a part of their formulation, Predicates are hard-coded with an interest level and these remain unchanged through their lifetime.

Age is another integer used to track how old a Predicate is. Upon Predicate creation, the instigating encounter number is stored in this variable, to give the ability for comparative reference later to see how long it has been since this Predicate was derived from its source and created as a real object.

This variable is used primarily for NarrativePattern processing, specifically the NarrativePattern’s optional time constraints. When checking if a Predicate is too old to use as a precondition Predicate for a NarrativePattern to activate, this age is checked.

Note that Age is NOT relational by definition. It does not give how many encounters have been played through since its creation, but just the number of the encounter it was spawned under.

Predicates also contain a method called Evaluate. This method essentially matches given relational parameters, usually Actor IDs, and then compares these given values to its own. If they match, it returns true to the method caller, and false otherwise.

This is notable in the evaluation of Narrative Patterns over all existing game Predicates, and again will be expanded upon later.

A small note still needs to be made, however. Very occasionally, there are some Predicates that do not typically involve two real Actors. For example, dies(Tom, ? ). This Predicate does not track the killer as an Actor, it is simply launched to mark Tom dying. In these cases, the secondary Actor is listed as a wildcard, *. * signifies that that specific Actor is empty, and can register as activating for anyone.

### 3.4.2 The Predicate Handler

Similar to OccurrenceHandler, PredicateHandler acts as the main handler and storage of any Predicates involved in the PAWN System. It can be thought of as functioning similarly to how OccurrenceHandler does, but for Predicates.

First, PredicateHandler establishes a series of dictionaries and lists later used in spawning Predicate objects. The two main lists are PredicateMegaDictionary, and the simply titled Predicates. PredicateMegaDictionary is a reference dictionary initialized at the beginning of the game, and populated from a Predicates JSON file supplied before compilation. It furnishes the dictionary with string keys of Predicate names, and values of created Predicate objects matching those names. Predicates is simply a list of all the spawned Predicates for each run, derived typically from harvesting Occurrences but sometimes from Narrative Patterns introducing new, specific Predicates into the list.

After establishing these lists, this Handler occasionally is prompted to harvest Occurrences contained in OccurrenceHandler and transform them into Predicates.
It iterates through each Occurrence in unharvestedOccurrences, and from them checks for each of their PredNamesAndOddsOfSpawning objects. It looks to these objects, and runs odds checks on each of them according to their supplied odds to see if a new Predicate of matching type to the PredNamesAndOddsOfSpawning object should be created and introduced into the Predicates list.

For example, in the Occurrence hitsWithHammer(Tom, Jerry), this might contain a PredNamesAndOddsOfSpawning object of type scaredOfHammers and odds of 30, meaning that if the odds check succeeds and rolls a number lower than 30 out of 100, the predicate of type scaredOfHammers would be cloned from PredicateMegaDictionary.

Finally, once created, this new Predicate is added into the list Predicates for future checking and reference.

The only other way to spawn new Predicate objects compared to the above is through Narrative Pattern Predicate spawning. When Narrative Patterns wish to introduce new Predicates, they do so basically in the same manner, only they completely avert the odds checking process.

Predicates are then used to decide character dialogues to display to the player via the WebWeaver component, but before that, the final logical piece of PAWN must be explained: the Narrative Pattern.

### 3.5 Narrative Patterns

Narrative Patterns are a crucial part of the PAWN System. They serve the purpose of adding specificity and complexity to the existing Predicate list that exists in PredicateHandler, and adding even greater and more interesting Predicates depending on the Predicates that already exist in that list.

Narrative Patterns can be basically thought of as Predicate recipes. They require certain Predicates to exist to activate, and once they activate, they spawn and introduce new Predicates into the existing Predicates list depending on the already existing Predicates.

Each Narrative Pattern has two broad parts:

- Prerequisite Predicates are Predicates that are required to exist in the manner stipulated in the Narrative Pattern exactly for this Narrative Pattern to activate.
- Payload Predicates are the Predicates spawned and introduced into the existing Predicates list with the Actors used in the prerequisite Predicates.

Narrative Patterns can be conceptually thought of as follows:

**Required Predicates:** hitWithHammer(A, B) and hitWithHammer(A, C)

**Payload Predicates:** bondOverBeingHitWithHammer(B, C)

A, B, and C each represent Actors in Predicates. Critically, these Actors must remain consistent throughout the paired requisites for the Narrative Pattern to activate.

For example, if the Predicates list contains the following Predicates:

hitWithHammer(Tom, Jerry), hitWithHammer(Tom, Alex)
Their existence would activate the above Narrative Pattern, since Tom, acting as Actor A, is in each of the Predicates in the correct order. However, instead of those two Predicates, if below was the case:

\texttt{hitWithHammer(Tom, Jerry), hitWithHammer(Alex, Tom)}

The above Narrative Pattern would NOT activate. The order of the the Actors is different, so it would not introduce new Predicates into the main Predicates list. It would be read via symbols as the following:

\texttt{hitWithHammer(A, B) \text{ AND } hitWithHammer(C, A)}

Since the symbols are out of order, the Narrative Pattern does not recognize the pattern as activated, even though Predicates exist of the matching required type with the same involved Actors.

### 3.5.1 The Narrative Pattern as an Object

The Narrative Pattern object is the representation of recognizable patterns of Predicates that should produce more Predicates.

Name is the first and main variable of the Narrative Pattern object. It is simply what this Narrative Pattern is referred to, and acts as the NP’s ID of sorts, in lists and sequences. Each Narrative Pattern name is unique, so that it can be distinguished from others in a master Narrative Pattern list. In the classical Narrative pattern form, it is the first word in the sequence.

Narrative Patterns also contain two main lists: prerequisites and payloads. Each of these contains objects of type NP-Predicate. These NP-Predicates store detailed information on what information a matching existing Predicate needs to have, and if an existing Predicate matches, specifically in the prerequisites list, then the effects list is translated into new real Predicate objects and added to the player’s Predicates list.

Such detailed information inside of NP-Predicates includes relevant time constraints and Actor information, both of which will be elaborated on later when discussing how these objects are formed at game initialization.

Narrative Pattern objects also contain other useful values, such as integers for tracking the total number of unique Actors involved in their prerequisites, the number of times each Narrative Pattern has been triggered, and a counter variable representing the last level encounter number this Narrative Pattern was triggered.

### 3.5.2 Spawning Narrative Patterns

Narrative patterns start their existence in a JSON file NarrativePatternJSON, and the parsing process of transforming these objects into real Narrative Pattern objects are described in this section.

They are structured in JSON as follows.
"name": "crewmateKilledByCannon",
"precons": "becameCrewmatesWith(a,b)[1:].killedByCannon(a,a*)",
"effects": "mournsCrewmateKilledByCannon(b,a)"

This JSON is read into existence in the following way. A new, empty Narrative Pattern object is created first. The value of this empty Narrative Pattern’s name is assigned to be the JSON object’s name value, in this case 'crewmateKilledByCannon'. The 'precons' and 'effects' strings are parsed in a specific, exhaustive manner, but to summarize it, '.' characters represent where one NP-Predicate ends and another begins and '[]' represent conditional time constraints that are expanded on later.

3.5.3 The Narrative Pattern Handler

NarrativePatternHandler is the main script and series of methods dedicated to storing, recognizing, and activating Narrative Patterns when their conditions are met.

The initial and most critical variable in this Handler is a list of NarrativePattern objects. It is filled with NarrativePattern objects created from the JSON list initializer on the start of the game. This is the main Narrative Pattern reference list used for determining whether or not there are any identifiable patterns in the existing player Predicate list.

This Handler also tracks a record of all Narrative Pattern activations in a list called alreadyUsedNPs. This list is mainly used in a way such that, since Predicates aren’t removed upon the activation of a Narrative Pattern, the same Narrative Patterns are not repeatedly activated every time Narrative Patterns are called to loop through the Predicate list for analysis.

On its first frame of existence, this Handler populates its relevant initial required Narrative Pattern reference lists in the method described above in the 'Reading NPs Into Existence' section. After storing a reference of each Narrative Pattern object, it then waits for it to be called from WebWeaver to parse through existing Predicates to find Patterns.

3.5.4 Finding Patterns

After that, FindNarrativePatterns is called. This method is a large and critical piece of the PAWN system.

The main process left to be described involving Narrative Patterns is how these objects are actually used in examining Predicates to find patterns. Here I will describe this process.

The instigating method for this process is called whenever the logical foundations of PAWN (Occurrences and Predicates) have just been updated, and we must see if any new Narrative Patterns are now able to be activated. There would be no point to call this method if we don’t suspect that the Occurrences and Predicates have not changed, as this is completely reliant on sifting through those lists to produce any Narrative Patterns.

This method has many different loops running layered into each other. First, it loops through each Narrative Pattern in our Narrative Patterns reference list, which has an example of every Narrative Pattern possible. Then, for each of these Narrative Pattern objects, a combination of three Actors is chosen for the Narrative Pattern to use for its evaluation.
Every possible combination of Actors is checked for each Narrative Pattern type, including order, since symbolic placement changes the meaning of Narrative Patterns and the Predicates they produce.

EvaluateNarrativePattern(Tom, Jerry, Alex)

The above example will be called to check for evaluation, as well as the below example.

EvaluateNarrativePattern(Alex, Jerry, Tom)

Both are done even though they contain the same Actors, since their order is different.

This evaluation method takes three Actors for the specified Narrative Pattern, and queries the existing Predicates list if any Predicates exist that match the conditions of each of the Narrative Pattern’s prerequisite Predicates.

It first evaluates the time constraints of each prerequisite Predicate, indicated by bracketed number in the original pattern formulation. These constraints can be either relative or absolute.

Absolute time constraints mean that, besides the specific Predicate existing in the main Predicates list with correct Actor order, it must also have existed a specified number of levels before registering as true. This acts as a way to add time between when something occurs, and when a Predicate is added because of it.

An example of this could be a prerequisite Predicate of ‘friendDies’ with an absolute time constraint of 5 levels, so that when 5 levels pass, a payload predicate of type ‘reminisceAboutOldFriend’ is launched. Age is needed for some Predicates.

Relative time constraints function similarly, but instead of being relative to the global player ‘time’, they relate to the time of other prerequisite Predicates in the Narrative Pattern. For example, in the Narrative Pattern ‘repeatedlyHurt’, the prerequisite Predicates are ‘hurt[1:]’ and ‘hurt’, meaning that two separate Predicates of type ‘hurt’ must exist, but one must have spawned at least 1 level before the other for this Pattern to Evaluate to being ready to activate.

After evaluating time constraints, it then simply evaluates the symbolic form of each prerequisite Predicate to see if the pattern could be successfully applied to the Actors it has been given. If, when assigned to symbols and checked for matching reference, this combination of Actors fit the Narrative Pattern’s ‘recipe’, it is registered as fulfilling the requirements of the Narrative Pattern and continues in the method.

At this point, if the method is continuing, its Actors are matching the required prerequisites Predicate symbols and any time constraints are non-issues, so the only thing to still do is to return true, except for one quick task.

Once Narrative Patterns are officially created, as mentioned previously, a record of them is created via NarrativePatternArchive objects and stored away. Here, the Narrative Pattern about to be made is checked for already existing in the NPA list. If the Narrative Pattern exists already in the NarrativePatternArchive list, and it has the same Actors in the same order, the method is aborted. Otherwise, if the Narrative Pattern has passed this final check, it is ready for spawning.

32
If all the above conditions are met, then the Narrative Pattern activates. It takes in the involved actors in its found prerequisite Predicates, and spawns its payload Predicates, replacing the symbols in their logical formulas with the Actors involved in the prerequisite Predicates.

Take for example again the Narrative Pattern of crewmateKilledByCannon below.

```
"name": "crewmateKilledByCannon",
    "precons": "becameCrewmatesWith(a,b)[1:].killedByCannon(a,a*)",
    "effects": "mournsCrewmateKilledByCannon(b,a)"
```

If this Narrative Pattern is satisfied by the below existing Predicates:

Predicate 1: becameCrewmatesWith(Tom, Jerry)
Predicate 2: killedByCannon(Tom,*)

Then it activates, and launches these payload Predicates into the main Predicates list.

Payload Predicate: mournsCrewmateKilledByCannon(Jerry, Tom)

### 3.6 WebWeaver

So now we have established and understood all logical components of our PAWN system, including Occurrences, Predicates, Narrative Patterns, and all their respective scripts and handlers and means of spawning. But what is still needed is something to Orchestrate all the collective pieces, and force them all to function together to actually produce the dynamic procedural dialogue we are striving for. This is done via a script called WebWeaver.

WebWeaver is the jack-of-all-trades script that acts as the main hub of and instigating force of the PAWN System. This script is responsible for the high level commanding of different parts of the PAWN System, and taking each of their disparate pieces and combining them into the end goal of translating gameplay events all the way into triggering dialogue relevant to those gameplay events.

#### 3.6.1 Updating Crew Logic

WebWeaver contains variable references to all the relevant static and non-static components of the PAWN System, like PredicateHandler, OccurrenceHandler, and ActorDatabase.

The first main task WebWeaver handles is the updating of the logical components of the PAWN system. UpdateCrewLogic is the front-facing public method for when the Actor logic (Occurrences, Predicates, Narrative Patterns) needs to be refreshed. This method does two main tasks: it updates the main Predicates list in PredicateHandler by harvesting unharvested Occurrences, and then parses this updated Predicate list for new patterns via Narrative Pattern objects.

It first prompts the OccurrenceHandler to harvest its Occurrences into their potential Predicates. Once these Occurrences are harvested and the Predicates list in PredicateHandler is full of new Predicate objects, UpdateCrewLogic then prompts NarrativePatternHandler
to find new Narrative Patterns. This process then scans the Predicates list, including our newly introduced Predicates, and checks for any new Predicates to introduce based on the list of NarrativePattern objects to compare to.

When both these sections of the method have concluded, the Predicates list in PredicateHandler is up to date and reflecting the current game state as much as can be reflected, and our logic is officially ‘updated’.

### 3.6.2 Updating Crew Conversations

Next, WebWeaver calls UpdateCrewPotentialConversations; this method does much. Initially, it scans through each of the Actors in the active crew list in ActorDatabase, and then updates each Actor’s conversations depending on current existing Predicates involving them.

For an Actor being filled with conversations, first WebWeaver finds every Predicate applying to this specific Actor. For each that exists, that Predicate is then analyzed along with the involved Actors for any potential conversations that are catalogued and written.

But simply finding every possible conversation reflecting this situation is not enough, as there exist many different conversations for most of WTP’s gameplay events, all from different perspectives and levels of vagueness.

For example, a Predicate describing a fight involving an Orc and an Elf should not play a conversation meant for a duel between two humans. It should either find a specific conversation, or reduce the specified requirements to one more generic, but never choose a contextually-wrong dialogue.

It finds the most appropriate dialogues for the situation by following the ladder of specificity, a naming convention system that finds dialogues based on how specific they are to the instigating Predicate and its involved Actors.

This is that priority list, from most to least specific.

- **Spirit x Spirit <= MOST SPECIFIC**
- **Spirit x Race**
- **Spirit x Generic**
- **Race x Spirit**
- **Generic x Spirit**
- **Race x Race**
- **Race x Generic**
- **Generic x Race**
- **Generic x Generic <= LEAST SPECIFIC**

Each entry of the above list is read in the form of ‘Primary Actor Descriptor’ x ‘Secondary Actor Descriptor’. Each of these words refers to the following:

- **Spirit** label means this Actor has a Spirit component to it that has a written conversation component relevant to this Predicate. For example, for a Predicate wasHurt, this involved Actor not only has a Spirit, but has a dialogue entry written in this spirit to reflect the Predicate wasHurt.
The Race label reflects that the Actor’s race has a written conversation component relevant to this Predicate. So, should the instigating Predicate be wasHurt and the Actor have a race of type Orc, for this Actor’s race to be used there must exist a dialogue entry reflecting an Orc’s reaction to the Predicate ‘wasHurt’.

The Generic label means that a generic written conversation component relevant to this Predicate exists. The least responsive and specific type of reaction that exists in the system, this is the least valuable of each label.

By ordering conversation selection like this, the most specific possible existing written dialogues are selected, while still having the ability to display conversations if no real specificity has yet been written for this encounter.

After identifying each of these conversation types, every single conversation found is added to the Actor for storage in their Potential Conversations list. This list holds all the possible conversations that this Actor could then use for interacting with the player, from all of their previous experiences. These Potential Conversation objects will reside in this Actor list, until they are chosen for displaying or filtered out.

Conversations are filtered as a part of the WebWeaver selection process. Sometimes, by this point each Actor will have several seemingly valid conversations ready in their Potential Conversations list, but in actuality those conversations would be out of place to display to the player.

For example, if this exact conversation has already been played before to the player and is not specified to be repeatable, or if the conversation is deemed ‘too old’, in that it reflects events that happened too long ago in the game relative to where the player is now, then that conversation should not be in contention for displaying to the player, even though this conversation involved Predicates and Actors relevant to the player and might have previously been judged to be very specific and responsive. This process of eliminating seemingly acceptable PotentialConversations from the Actor’s PotentialConversations list is referred to as culling.

So after all possible conversations have been gathered to the Actor, and the list then purged for undesirable conversations, it is then able to be allowed to converse.

To converse, the Actor undergoes several different steps.

First, it must pass a ’chattiness check’. This is a simple odds check, meant to introduce some randomness to which crew members talk to the player, and how often they talk. If passed, this Actor then sets its most ‘interesting’ conversation.

Inside each Potential Conversation object is an inherent interest level integer, given to it by its instigating Predicate. The Potential Conversation with the highest of these interest levels is assigned as the Actor’s NextConversation value. This is the Potential Conversation object that has been selected to be played to the player, and as soon as the player selects the Actor’s dialogue button, it will start.

While basically ready for player conversation, some additional cleanup is employed right after NextConversation is assigned.

Every Potential Conversation object in PotentialConversations that reflects the same situation as the most interesting conversation selected for NextConversation is collected and purged from the Actor. This is done so that multiple conversations reflecting the same
situation are not conveyed to the player from the same Actor. For example, should the Actor be mourning a friend, the Actor shouldn’t convey this multiple times to the player with lessening degrees of specificity. To avoid this, these additional conversations are removed.

After culling, this Actor is finally ready to converse with the player. For each Actor that has been selected to talk and that has a NextConversation reference, that character’s display module is enabled for interaction by the player, and signals to the player that this Actor is ready to converse.

3.6.3 Launching Conversations

Should the player select the Actor’s activated conversation button, then the conversation initiating process is officially launched.

Between the player’s clicking of this button and the appearance of the dialogue, PAWN first assigns relevant runtime conversation variables, like ‘first’ and ‘second’ Actor. These are the two Actors involved in the Predicate that spawned this conversation, and are used below for fetching personal variables for customizing the dialogue text.

After this assignment of dialogue variables, then the conversation finally starts, and plays to the player. When this conversation finishes, WebWeaver wipes the Actor’s nextConversation variable, so that nextConversation can reflect the Actor’s real state of not having a viewable conversation anymore.

3.6.4 In Conversations

In the process of displaying conversations to the player, each specific pre-written line is filtered for relevant insertions of special character sequences. These sequences contain information for words or parts of words that should be changed depending on the stored Actor values in the previously assigned ‘first’ and ‘second’.

For example, the below line is taken from a dialogue reflecting a crew mate consoling another after taking damage from a cannon weapon.

‘Whew, you ok there [2.firstName]?’ [1.firstName] said over the smoking hole the cannonball had struck where [2.firstName] had been standing just a moment before.

‘[1.firstName], it’ll take more than that to kill me.’ [2.he] laughed.

Any sequence of repeated characters included in brackets is signaled to be filtered out of the conversation and replaced using regex with variables according to what the brackets actually contain.

The characters ‘1’ and ‘2’ act as replacements for the main and secondary Actor in the exchange, and the trailing descriptors attached to these characters via a ‘.’ character are for signaling what to replace the bracket-capped snippets with.

For example, the snippet ‘[2.firstName]’ would be parsed at runtime by first identifying ‘2’, which means that this snippet refers to the secondary Actor in the exchange. Then, it attempts to return the string variable ‘firstName’ on the secondary Actor object in this exchange.
So, if the Actors of Tom and Jerry were the primary and secondary Actors respectively chosen for the above exchange, the dialogue showed to the player after parsing according to their specific variables would be something like the below snippet:

‘Whew, you ok there Jerry?’ Tom said over the smoking hole the cannonball had struck where Jerry had been standing just a moment before.

‘Tom, it’ll take more than that to kill me.’ he laughed.

This bracket-snippet system of regex replacement allows for in-depth dialogue tailoring to each conversation’s specific characters.

Chapter 4

Results

4.1 Procedure

For years, I have developed the aforementioned video game WTP: Walk the Plank. Inside this incomplete game play demo exists two exclusionary systems of crew member dialogue choice.

One system, referred to going forward as NONPAWN, produced crew member dialogues after each encounter in WTP by picking from a random pool of generic dialogue choices that would broadly fit any situation the player was in. For example, after a battle a crew member would sometimes say, ”I love sailing, the sea has no match!” or ”Ugh, more hard tack for dinner, eh?” . The point of these dialogues was to be short and generic enough to apply to any possible situation the player and their crew experienced. The NONPAWN system is the intended control of the testing experience.

The other narrative system is the also aforementioned PAWN system, that produces responsive dialogue depending on occurred gameplay events.

For each test, the structure of the testing session was as follows:

1. The tester filled out a short survey, describing their demographic information and their past history with video games.

2. The tester played one of the two above systems for narrative for anywhere for 15-25 minutes. Which system the participant played was randomized.
3. The tester filled out a short form describing their experience playing the previous snippet.

4. The tester played whatever narrative system they didn’t already play.

5. The tester filled out an identical form inquiring about their experience playing the second system.

The gameplay base of WTP that participants tested was scaled down in such a manner to facilitate easier short-scope testing. Among these changes were that players were limited to picking only one ship type, and had a limited selection of crew members to choose from. Specifically, they were limited to two crew members, one of type human, and one of type Orc. This was done to reduce needed writing for the demos themselves, as with seven unique race types, many different dialogue combinations would be needed for this early pilot testing. Instead, the demo potential crew types were limited to two different race types, orcs and humans.

Also, embedded in one of these human characters was the Spirit ‘Morose’. This was a Spirit whose dialogue entries were morose and sad over anything that happened, sometimes in a humorous manner.

4.2 Participants

Ten game play testers followed the testing procedure listed above. Their demographic data is as follows: all ten participants were male. All ten self-described as white ethnicity. One participant lives in Prarieville, Louisiana, USA, while the other nine participants live in Hull, United Kingdom. Participant ages ranged from 21-32, with all but 2 participants being in their 20s. Participant education level varied from completed high school to completed bachelors degree. Participant relationship status range from single to married.

Besides pure demographic data, participants were also queried upon their past history with video games. Only one participant self-described as not playing video games very much, while most (7) identified as playing some games, 2-10 hours per week. In this category of game time, most of the games these participants played were social and competitive online games, such as Apex: Legends and Call of Duty: Warzone. They occasionally played single player games, but these were not their typical fare.

Two participants identified as heavy gamers, spending more than ten hours weekly playing games. These testers identified a wide range of games in their play history, including the social and online games listed above, purely singleplayer story games, such as Cyberpunk 2077, and fortunately, both had extensive playtime in FTL: Faster than Light. This is relevant since that is the game that the demo for WTP is based on, and the game base in which they would be testing the PAWN system.
4.3 Results

Testers were given a series of questions with the following prompt: "Please rate the following statements on a scale of 1 - 7 based on how strongly you agree with them, 1 being completely disagree and 7 being completely agree." Results can be viewed in figures 4.1-4.7 in the end of this chapter, with tester’s first initial and age combined for their anonymous ID.

- Nine of ten participants ranked the PAWN system as being ‘more responsive’ to gameplay events.
- Nine of ten participants ranked the PAWN system as having the more enjoyable dialogue
- Participants found the first gameplay section more difficult than the second, regardless of which order they experienced the gameplay sections in
- Every participant felt either as equally or more attached to their crew members when experiencing the PAWN system
- When asked if they felt the crew dialogue choices were odd or confusing, only one participant ranked PAWN as the more confusing, with most identifying it as not confusing at all
- Participants who had less to no previous gameplay experience found the game moderately frustrating at first, while those with more previous gaming history found the overall experience less to not frustrating at all

4.4 Discussion

A successful batch of testing results would be those that support the idea of the PAWN system being more responsive to player choice and gameplay. If, during PAWN testing, players felt that their actions in game were reflected in their crew’s dialogue, especially more so than the control NONPAWN dialogue narrative system, then this system will have been proven to produce a more responsive and engaging experience.

The results of the testing and the questionnaires fortunately confirm this. When asked the question,

"How do you feel about the following statement on a scale of 1 to 7, 1 being extremely disagree and 7 being extremely agree: I felt my crew’s dialogue was responsive to gameplay events."

Testers replied with a 5.7 average, meaning that as a whole they agreed with the statement. The median response was 6.0, which almost tends it toward the average player saying they extremely agree with the prospect of PAWN feeling responsive to the events of their playthrough.

One participant noted he thought the characters were "really funny, in a good way". He enjoyed that the dialogues of the crew responded to the stations they were tasked to man by the player, for example. He especially loved the dialogue from one crew member manning the
weapons station, whose dialogue was about being hard of hearing due to constantly being surrounded by cannons. "Because it makes sense, you know? She would struggle to hear if she’s next to exploding cannons for hours,” he shared informally after the test.

Another participant noted he “felt bad” that he allowed to have a crew member be hurt repeatedly, but only after the crew member himself launched a dialogue asking the captain to go easy on him (launched from the Narrative Pattern payload Predicate of type repeatedlyHurt ). This is very clearly a uniquely PAWN interaction, as the dialogue was applied to the crew member specifically from the player’s allowance of them taking damage, and then this in turn made the player feel emotions that changed their playstyle and future dynamic and relationship with that character.

Testers also identified the responsiveness of units in the NONPAWN control as noticeably worse than the PAWN equivalent, with an average agreement rating of 3.6 and a median of 3.5 to the same question when referring to the NONPAWN control gameplay demo.

4.5 Other Miscellaneous Notes

By far, the crowd favorite character was whichever human crew member had the 'Morose' spirit. This is especially interesting because most of the dialogue writing for this spirit was misallocated, in the sense that much of its breadth was established but it typically lacked response depth. For example, 'Morose' had responses for it to respond to any type of damage source hurting it, but most of the test runs were brief enough that the 'Morose' crew member was only hurt once, if at all. Therefore, only one or two instances each run of this character being responsively sad was enough to establish it as the favorite amongst any participant that had a favorite, which supports the idea of specificity and feeling handcrafted of dialogue being more interesting to the player. As well, the biggest and most reliable laugh of every play test was 'Morose' questioning to themselves, "Oh gods, why am I still doing this to myself?" when the player began each run (Launched from the beginRun() occurrence).

Character’s procedurally-generated names were not remembered at all, despite being displayed to the player and being generated uniquely for every run. I believe this had to do with the limited amount of time the players spent interacting with each crew member. While 10-25 minutes can sound like a lot, typically that time was more spent on becoming familiar with the gameplay and controls as opposed to jumping right in with the characters. Players still had favorites, but referred to them as ‘the orc’ or ‘the one that manned the shields’.

There were some unexpected bugs that occurred during testing. Occasionally, dialogue entries would activate and show multiple times despite built-in guards against this type of happening. Players were instructed to skip these additional dialogues and continue their test. This occurred in both PAWN and NONPAWN versions of the test.

Additionally, a more severe bug occurred occasionally, in which aiming the player ship’s weapons during combat would stop responding to player input, and malfunction. In these cases, once occurred, it would cause an unrecoverable error, and the play session would be forced to end. Fortunately, this happened rarely (only twice) and occurred when that particular session was already far enough along to provide useful feedback.
Despite these bugs, both experienced and non-experienced users judged the game to be great fun.

Testers enjoyed the NONPAWN system more than anticipated, despite the context-less and bland dialogue it presented to users. When questioned why, some users enjoyed it since it was typically shorter, snappier, and occasionally unintentionally humorous. For example, a crew member proclaiming, "I love sailing!" after an intense battle against pirates usually entertained users via the contrast.

Chapter 5

Conclusion

In this document I have introduced the PAWN system.

Characters for the system are generated into Actor objects, in which personal aspects are generated and stored inside. These objects can have optional Spirits, which force that character to act and respond in a specific and consistent way, establishing the Actor with a more hand-crafted feeling to their dialogue.

Through the capturing of game events, PAWN logs corresponding logical objects called Occurrences that reflect the game event they were spawned from and those characters involved in that event.

These Occurrences are parsed through, and derived from them are Predicates, relational objects that represent character responses and connections formed through these Occurrence events.

More varied and interesting Predicates are introduced via Narrative Patterns, objects that activate through a process in which patterns of are Predicates checked for and, if found, produce new Predicates into the existing Predicate pool.

Finally, when prompted, the existing Predicates judged to be most interesting are taken and chosen to launch dialogues about. These dialogues are displayed to the user based on the most specific written dialogue reflecting the chosen Predicate and its involved Actors, and is customized to its involved Actor’s generated personal qualities using regex substitution techniques at runtime.

Unlike other narrative systems, PAWN generates characters and responds to events procedurally, in a natural reaction to player participating in-game events, with a scalability of specificity in dialogue and ease of use for smaller developer teams in dialogue choice (but by design not limiting for larger dialogue entry amounts). As well, it is not patented and, if you are reading this paper, is explained in-depth for understanding and replication by other users.
The results in the testing of PAWN were ideal; every tested participant responded that they either enjoyed the PAWN version of an A/B test as much if not more than a context-less control dialogue system in the same gameplay testing environment.

Developing the PAWN system answered a long-winded but looming issue for game developers: what is a framework that can be used to have procedurally generated responses by characters in a game, that is flexible in its selection of dialogue choice, that has a feeling of direct responsiveness to player actions from its selected dialogue, while still presenting them with quality, handwritten dialogue, in a system that isn’t patented and is explained clearly for future iteration and improvement.
References


Vita

The author was born in New Orleans, LA in 1997. Raised with a love of narrative and artificial-intelligence, he joined the University of New Orleans to attain his undergraduate degree in Computer Science and to pursue projects fusing these two passions together. He later joined the Computer Science Masters program to pursue a Master of Science and Expert Certification in Machine Learning and AI. In 2021, he joined Dr. Ben Samuel’s Light Lab as a Graduate Research Assistant, researching the application of Virtual Reality (VR) technology to the classroom and worked in conjunction with Dr. Matthew Tarr and Top Right Corner to develop VR training labs for the Chemistry Department.
Figure 4.1: Results: "I found the gameplay hard to understand."

"I was confused by the crew member dialogues."

"I derived enjoyment from this gameplay demo."

Figure 4.3: Results: "I derived enjoyment from this gameplay demo."
Figure 4.4: Results: "I was attached to my crew members."

Figure 4.5: Results: "I enjoyed my crew member's dialogue."

Figure 4.6: Results: "I found the gameplay hard to understand."
Figure 4.7: Results: "I felt my crew member’s dialogue were responsive to gameplay events."