UNIVERSITY OF CALIFORNIA
SANTA CRUZ

TABLETOP ROLE-PLAYING GAME DESIGN THROUGH A PATTERN LANGUAGE SOFTWARE MODEL

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTATIONAL MEDIA

by

Alexander Mayben

September 2020

The Thesis of Alexander Mayben
is approved:

Professor Noah Wardrip-Fruin, Chair

Professor Michael Mateas

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

Tabletop Role-Playing Game Design

Through a Pattern Language Software Model

by

Alexander Mayben

Currently, barriers to access are high for new or inexperienced designers in the fields of tabletop role-playing game (TTRPG) design. TTRPGs remain a relatively unexplored design space, to the detriment of designers and system homebrewers– those who wish to modify existing TTRPG systems, rather than write new ones. A useful solution to this issue is to compose a pattern language representative of these systems. A pattern language provides a useful blueprint for designers of complex design systems, simplifying the TTRPG design process. In this thesis, we create a conceptual language which presents a simple-to-understand method of developing a TTRPG's core elements alongside a set of pre-defined options and goals. This pattern language will then be simulated by an application developed within the JavaScript engine Blockly, as both a demonstration and as a tool with which designers can write systems. This tool, which we have named Assemble, will demonstrate how designers could benefit from utilizing pattern languages in computer-aided design contexts for TTRPG systems.

To my grandfather,

William Harmon Mayben Jr.

(*1920-2002*)

For my education.

To my grandmother,

Patricia Bowing Fournier

(*1927-2020*)

For my childhood.

Acknowledgments

I first want to thank Jason Morningstar and his game *Fiasco*, which inspired this work. I also thank Meguey Baker and Vincent Baker for *Powered by the Apocalypse*, as well as Sage LaTorra and Adam Koebel for *Dungeon World*, the system which introduced me to *Powered by the Apocalypse*.

Much thanks to my faculty advisors Michael Mateas and Noah Wardrip-Fruin, as well as Elizabeth Swensen and Katherine Isbister, for their patience and guidance.

Further appreciation goes out to my lab peers Elizabeth Oliver, Devi Acharya, Max Kreminski, Isaac Karth, Melanie Dickinson, Rehaf Al Jammaz, Stacey Mason, and Shi Johnson-Bey, and to my colleagues Tamara Duplantis, Henry Zhou, and Celeste Jewett.

This project would not have happened without Argent Eliot Reinhard and Benjamin Rowland, whose assistance in validating this thesis' pattern language was indispensable.

I also thank Benjamin Smithers, Andrew Miller, Tom O'Rourke, Adam Chan, Nick Uhlig, Kevin Zhang, Mason Sipple, and Thomas McMaster for supporting my interest in tabletop games– role-playing or otherwise– throughout my time in university.

I thank my dear friends Sebastian Timpe, Owen Chin, Anthony Boileau, Michael Adam Mahar, Dustin Matheny, Logan McBride, and Manuel Palomares, for their ongoing emotional support and encouragement through academic and personal hardship.

Finally, to my mother Mary, my father Bill, my brother Andrew, my aunt Janet, my aunt Rhea, and my cousin Darcy: I would not be where I am today without your bottomless love and support. Thank you all so much.

Chapter 1

Introduction

Tabletop role-playing games (TTRPGs) are defined as a class of interactive narrative-based games with a heavy focus on improvisational storytelling, imagination, and, commonly, elements of random chance, such as dice rolls. TTRPGs are usually driven by a group of players who assume the roles of fictional characters, and are mediated and facilitated by a game master (GM). The GM plans out the game's "campaign"– its session-to-session story– by building upon the game's world, driving conflict and action in the game's narrative, and evaluating player decision-making over the course of play to accommodate and ensure interactivity.[23] To encourage certain approaches in how stories are told, TTRPGs form structured systems that guide player interaction and determine the tone and content of a role-playing campaign's story and setting. Each TTRPG system, such as *Dungeons & Dragons 5th Edition*, *FATE*, and *Apocalypse World*, conveys these structures through one or more reference guides, detailing the rules of play, the world in which the game is set, and the kinds of roles which players are able to assume within that world.[6][12][16]

For many GMs in the TTRPG community, existing role-playing systems do not always suit the needs of the stories they wish to tell. This phenomenon most notably occurs from a failure by the rules or accompanying setting of a role-playing system to match the GM's standards or expectations. As every campaign is different, with any given role-playing group bearing a unique style of story-crafting, there may often be circumstances for which a pre-established rule structure would be insufficient for accommodating certain story situations or attitudes toward play. Yet to an individual GM, designing an entirely new system can be a massive undertaking; in most circumstances, "homebrewing"– tweaking existing systems to fit the GM's or players' needs– is a much more forgiving alternative. Homebrew is a GM's reinterpretation, change, or addition of mechanics within an existing TTRPG system, thus re-purposing said system to apply to the specific needs of a given campaign. Homebrew can take many forms, ranging from minor, innocuous reinterpretations of certain rules, to the addition of entirely new mechanical interactions or story approaches. For instance, a GM may want to devise rules for handling a vehicle in a system where foot-based travel is usually implied, or plan a campaign set in the modern age using an explicitly high-fantasy system originally written for a medieval setting. Homebrew-minded approaches are indispensable for capturing these kinds of interactions, without the need for the GM to change to a different, potentially less-preferable system, or write a new system from scratch.

While TTRPGs have gained popularity in recent years, designing new role-playing systems– and to a lesser degree, homebrew– remains challenging and unintuitive for many newcomers. Few comprehensive texts on tabletop role-playing system design exist, and the field of TTRPGs shares a fault with the broader field of game design: there are insufficient tools or strategies to on-board designers who are not already extensively trained in game design.[9] There also exists no design ontology for TTRPG systems within current research. The question of how a system is designed resides at the core of what motivates homebrewers, and so developing an effective design ontology would help not only those developing systems of their own, but also those who are more interested in adapting existing systems for homebrew purposes. Additionally, while the field of homebrew typically requires less effort on the part of the designer compared to authoring full systems, the game design considerations made by homebrewers are similar to those of designers of role-playing systems in general. Expanding access to design tools and literature for GMs and TTRPG designers could therefore encourage further innovation within TTRPG design and ultimately aid future game design research.

Given that TTRPG systems occupy an abstract design space when compared to other practices, it is worthwhile to consider a solution that allows the process of system design to be tactile, easy to follow, and structurally comprehensive. Pattern languages present a worthwhile solution to help simplify the process of both homebrew and general system design. A pattern language effectively generalizes existing systems by breaking them down into an interactive network of generic design patterns. In research, pattern languages function not as a taxonomy, but a partonomy, organizing a system and its parts into a structured hierarchy to further understand their relationships and function. The elements of a pattern language are interrelated; patterns can require the satisfaction of related patterns, refer to other patterns that are dependent on their completion, or do both.[4] Pattern languages work jointly as partonomies and practical aids, describing the composition of an effectively-designed exemplar system while providing a useful road-map for designers to author new systems. They can also provide a unified understanding of design processes. Thus, producing a well-validated pattern language of TTRPG systems would have positive implications for expanding upon their design's theory and practice.

For a pattern language to be practical, it also must be accessible to those who are closely involved in design. Accomplishing this requires access to design software that effectively incorporates pattern languages and allows users to create full systems from those languages. Accordingly, software also can ease the process by which design concepts are transmitted from person to person while a system is in the process of being drafted. This can especially be useful for homebrew purposes, as homebrews themselves have been known to inherit or adapt concepts between different systems, and there is a broad tradition in the homebrew community of sharing others' design modifications.[1][2]

This thesis will demonstrate a solution to accommodating design choices made by TTRPG designers and homebrewers by describing a pattern language representing an example class of TTRPGs, thereby introducing several meaningful TTRPG design patterns. Furthermore, this thesis will detail a software application which implements the pattern language in a manner that can aid the process of TTRPG design and homebrew. The digital interface which this thesis specifies will present a workflow which streamlines TTRPG design, while at the same time producing TTRPG systems that satisfy the expected criteria of their category. This application should appeal to people without significant design experience who wish to develop a process for TTRPG design, as well as intermediate-level GMs who are interested in including new mechanics into their stories and campaigns.

Chapter 2

Related Work

2.1 On Pattern Languages

Pattern languages were first introduced in the 1977 book *A Pattern Language* by Christopher Alexander et al. The work presents a recontextualization of design systems into patterns, conceived as a hypertextual structure informed by computational reasoning. Through this hypertext, design elements are referenced by or connected with each other, thereby comprising a structured, practical, and pedagogical system with interlocking and reflexive elements[4]. Though the ideas outlined in *A Pattern Language* were originally conceived for architecture, the pattern language concept has been expanded to other fields, such as information science and human-computer interaction, to cover a wide range of various useful applications. In "Pattern language and HCI", Yue Pan and Erik Stolterman outline some of these applications in detail, including ethnographic research, interface design, and software engineering. Furthermore, Pan and Stolterman argue that pattern languages are more effective for design practice than patterns alone, facilitating design cohesion as a part of a larger whole, rather than developing each pattern in an ontological vacuum[18].

Researchers and game designers have also adopted pattern languages within their own approaches toward understanding games and storytelling systems. One key example is Staffan Bjork and Jussi Holopainen's extrapolation of Alexander's ideas to game design contexts, in their book *Patterns in Game Design*[7]. The Game Ontology Project is yet another work concerned with the development of a partonomy for the purpose of better understanding design in games, citing industry demand as a key driver of interest in developing such a language[22]. Pattern language-based ontologies have also become increasingly used as mechanisms for understanding intention-based design in the live-action role-playing (LARP) community, as demonstrated by descriptive systems such as Morningstar and Li's "Pattern Language for LARP Design"[17] and experiments such as Mikołaj Wicher et al.'s "LARP Design Cards"[21]. One important thing to note here is that game design and LARP hold different general attitudes towards balancing storytelling and causal system interactions. While game design occupies a more rules-oriented space that often prioritizes the act of play over story content, LARP usually tends to be motivated significantly more by story than mechanics. By comparison, TTRPGs can be viewed as occupying a spectrum between these two interests, with different role-playing systems varying upon the extent to which they may prioritize one over the other.

2.2 On Tabletop Role-Playing Game Design

Beyond pattern languages, researchers have made other observations on how interactive storytelling and TTRPGs could be conceptualized. Robin Laws, author of the *Over the Edge* role-playing system, offers one such concept in his book *Hamlet's Hit Points*. Here Laws introduces a representation of plot, which he terms "Beat Analysis", to break down stories into their constituent "beats" to better understand the mechanics of narrative causality. Laws then indicates how this treatment of storytelling can apply to both existing narratives (using the titular *Hamlet* as one example) and interactive contexts as with TTRPGs. In the latter case, Laws specifically focuses on the GM's organizational process, offering the Beat Analysis system as an option for GMs to plan individual role-playing sessions and campaigns.[15]. In his Ph.D. dissertation, Aaron Reed contextualizes TTRPGs as "storygames". Designing a storygame, Reed writes, is heavily dependent upon the game's facilitation of one or more of four key activities: generation, negotiation, administration, and what he refers to as "storywrighting"; the process by which ideas are "combined and shaped into an ongoing, coherent, and compelling story". According to Reed, storygames occur at the midpoint of a "simulative spectrum" between world simulation and storytelling, and a "performative spectrum" between authorship and improvisation. Reed also characterizes the game master role as unique to storygames in particular, imposing a level of mediation between simulation and performativity.[19].

Vincent Baker, co-creator of *Apocalypse World* and *Powered by the Apocalypse*, has also shared some of his philosophy on TTRPG design through his blog on the Lumpley Games website, in which he describes the design motivations behind his work. Baker writes that for a TTRPG to be designed effectively, it should start with an outline, and have that outline be repeatedly iterated upon, with revisions and optimizations made between each iteration. He also indicates how *Powered by the Apocalypse* has benefited TTRPG designers in presenting a useful outline to serve as a starting point for this iteration process. Because of this, Baker argues, *Powered by the Apocalypse* should be viewed not as a strict category of TTRPGs but as a general TTRPG design approach[5]. Baker's notion of establishing an iterable preliminary system outline provides a promising indication of how deploying an effective pattern language might serve an important design role.

Chapter 3

Methods

3.1 Developing a Pattern Language

Given that a holistic pattern language does not yet exist for TTRPGs, it would be imprudent for the pattern language we define to attempt to encompass all TTRPG systems without any basis to start from. Instead, this language was based on a well-defined subset of tabletop role-playing systems, laying a foundation upon which a more comprehensive pattern language can be developed in the future. The *Powered by the Apocalypse* set of systems, originally conceived by Meguey Baker and Vincent Baker, is therefore a more reasonable template to work from within the scope of this thesis. *Powered by the Apocalypse* systems are characterized by their definition of player actions within the system (known as "Moves"), the use of modular, class-specific character sheets ("playbooks"), and the determination of situational outcomes via two six-sided dice and a simple set of parameters (a roll of 10 or higher being a success, 7 to

9 a partial success, 7 and above counting as a hit, and a roll of 6 or lower being a miss). *Powered by the Apocalypse* systems tend to be light on rules, well-defined regarding intended player action, and tailored for quick and easy setup. Together, these simple, easily-specifiable qualities make *Powered by the Apocalypse* systems more amenable to the process of composing a pattern language.

To be effective, a pattern language should define a cohesive, yet nonrestrictive process for writing material for a system, standardize that material to a set of constraints, and specify a logical connection of patterns that, when used to assemble a completed system or homebrew, will help foster robust, insightful, and impactful play experiences. Starting with *Apocalypse World*, the exemplar system of *Powered by the Apocalypse*, we established the initial outline for the pattern language through empirical observation, and then rigorous comparison against a variety of *Powered by the Apocalypse* systems, amending the pattern language as necessary until we were satisfied that it sufficiently covered a broad sample set. This process began with the author examining the contents of two additional systems, *Monsterhearts 2*[3] and *Dungeon World*[14], before collaborating with two undergraduate students to test and revise the prototype pattern language against *Apocalypse World* and three additional systems: *Big Bad World*[8], *Bluebeard's Bride*[13], and *Monster of the Week*[20], which was accomplished via a shared spreadsheet. Once validation was completed, the author proceeded to the next phase of the project.

3.2 Producing a System Design Tool

Once the pattern language was developed, it was translated into a software paradigm where it could then be used to dynamically compose new systems within a visual interface. This software application needed to be capable of fluidly manipulating system elements and loading content produced externally. For this purpose, we used Blockly, a JavaScript engine developed by Google.[10] Based on the Scratch visual programming language used to help teach coding, Blockly's open-source access and verbose documentation made it especially useful for development outside its original core purpose. Blockly's dynamic compartmentalization of information is especially helpful for simplifying complex processes into an understandable visual vocabulary. The approach of assembling a hierarchical block tree to complete a functional work product is more than suitable for a pattern language-based use case, as pattern languages themselves are hierarchical. Therefore the technical implementation of the pattern language in Blockly could provide for a visual, comprehensive, and descriptive demonstration of the TTRPG design process, making design more intuitive and understandable.

The Blockly application was first developed on the block-by-block level via Google's Blockly Developer Tools engine[11], with each block representing an individual pattern of the pattern language. Once each pattern was translated into a corresponding block, those blocks were converted into both XML and JavaScript code and placed into the Blockly engine. The blocks' JavaScript code was then developed further over a four-month period to allow for interaction between different sets of blocks within the

Blockly workspace, including functions such as populating menus with blocks' content, packing certain data into JSON objects stored within the blocks' XML, and specifying dynamic inputs within the block structure. As the blocks were developed in JavaScript, quality assurance was conducted via an HTML file displaying them, which was edited and maintained alongside the blocks' internal code. Upon completion of the JavaScript portion of the software, the application was given the name Assemble, and the application's HTML portion was further developed into a complete website, receiving features such as export and import functions for Assemble workspaces, a list of three example systems ported to Assemble, and an informational sidebar explaining Assemble's function and purpose.

Chapter 4

Results

4.1 Pattern Language

This section will describe the pattern language developed over the course of the Assemble project. This pattern language lays out a network of discrete patterns which produce a valid *Powered by the Apocalypse* TTRPG system as output. In this language, we say that a *system* is comprised of three patterns: *themes*, *setting*, and *mechanics*.

4.1.1 Theme

Themes are the simplest of the three patterns used to form a system. A *theme* is a short description of a principle or tonal affect that drives the design and play of a system. It characterizes what makes a system interesting, or what that system is generally "about". Some examples of themes may include genre (i.e. the post-

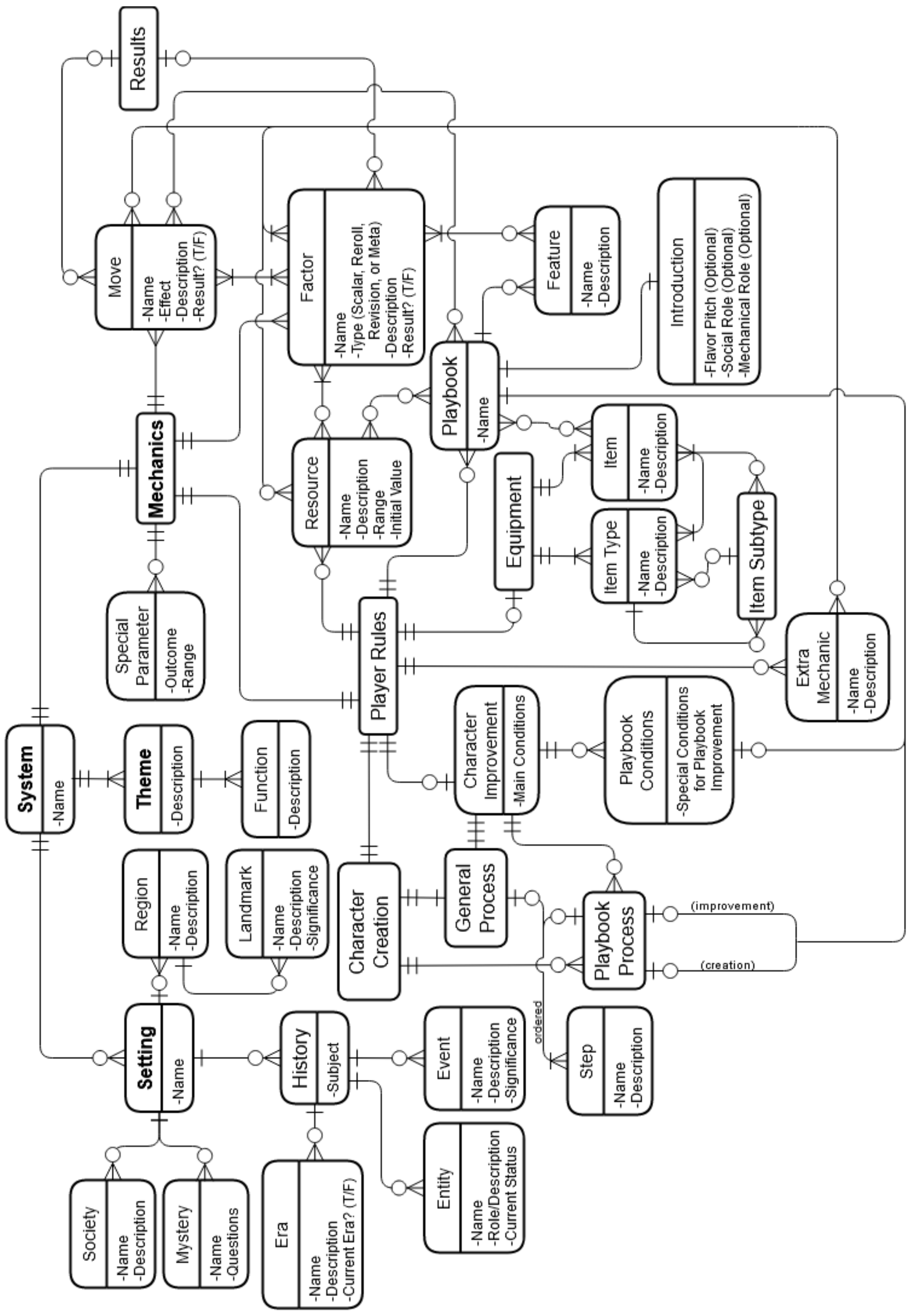


Figure 4.1: An entity relationship diagram of the pattern language.

apocalyptic framing of *Apocalypse World*), an approach that puts an interesting spin on role-playing (i.e. modular storytelling), or a compelling prompt that introduces the general premise of the system (i.e. players control fantasy monsters emerging from a modern-day nuclear event).

Any given theme should have one or more *functions.* A function is a means by which a given theme is embodied within the design and play of the system. For instance, *Monsterhearts 2*'s theme of "adolescents who are secretly monsters" satisfies two specific functions: exploring the disorientation inherent in adolescence through the lens of monstrosity, and using monstrosity as an allegory to explore the other "monstrous" qualities of teenagers, such as gender expression, high school social politics, and developmental angst. Functions also can have a more direct design role, with *Apocalypse World*'s post-apocalyptic theme having functions of a scarcity in resources that drives character motivation and providing an ominous sense of the unknown which haunts both play and story.

4.1.2 Setting

A *setting* is a world in which the system's story is set. A system can have more than one setting; this is true in *Monsterhearts 2*, where these individual settings take the form of "Small Towns". A system can also simply have an undefined setting, leaving worldbuilding up to the GM if the system's general focus is on mechanics instead.

A setting can be comprised of up to four patterns: *society*, *region*, *mystery*, and *history.* A *society* is a description of a group of people that lives within the setting, of which there can be none or many. A *region* is a defined geographical area within the world with unique characteristics. A region can also have a number of *landmarks*, or notable locations and features within that region that may bear some significance.

A *history* is an account of past events, both recent and distant, that are significant to the story of a particular setting. As there may be different historical accounts depending on the subject, or even the narrator (i.e. the *Rashomon* effect), a setting can have more than one history. The history pattern bears three patterns of its own: *era*, *entity*, and *event*. An *era* is a period in time with pertinence to the story; it both establishes how far back the history's defined plot originates, as well as what technological period the present is established in. For instance, given that *Bluebeard's Bride* is set in the world of fairy-tales, its main history will have one "medieval" era. As *Apocalypse World* is post-apocalyptic, its main history has *two* defined eras: one before the apocalypse, and one after. An *entity* is a person or group with significance within the system's narrative, and an *event* is a relevant past occurrence.

4.1.3 Mechanics

The system's *mechanics* describe the rules and procedure of the system's play, and references four patterns: *factor*, *move*, *special parameter*, and *player rules.*

For a *Powered by the Apocalypse* system, *moves* are quintessential. The system's moves define various classes of player interaction within the story. Each move will have a name, a story effect, a description of how the move is executed through system interactions, and a set of factors. A *factor* is a metric that has some impact on the outcome of a given mechanic, with factors uniting to form the building blocks which drive the resolution of moves and other mechanics. A factor can also occur as a move result; if a move is successful (or unsuccessful, depending on the move), it may also cause one or more additional factors that could have a future mechanical effect. Most moves also depend on dice rolls to be resolved, and to this end *Powered by the Apocalypse* uses a set of pre-defined parameters (See Section 3.1) to determine the outcome of roll-based moves. If these parameters prove to be insufficient, the system can also define a number of *special parameters* to specify additional types of outcomes.

We can see one example of how moves use factors with *Monster of the Week*'s "manipulate someone" move. Note that this move uses an "advanced success", signifying a roll of 12 or above, as a special parameter.

> Once you have given them a reason, tell them what you want them to do and roll +Charm.
>
> *For a normal person*[1]:
>
> On a 10+, then they'll do it for the reason you gave them. If you asked too much, they'll tell you the minimum it would take for them to do it (or if there's no way they'd do it).
>
> On a 7-9, they'll do it, but only if you do something for them right now to show that you mean it. If you asked too much, they'll tell you what, if anything, it would take for them to do it.
>
> Advanced: On a 12+ not only do they do what you want right now, they also become your ally for the rest of the mystery (or, if you do enough for them, permanently).
>
> *For another hunter*[2]:
>
> On a 10+, if they do what you ask they mark experience and get +1 forward.
>
> On a 7-9, they mark experience if they do what you ask.

[1]A non-player character (NPC).

[2]A player character.

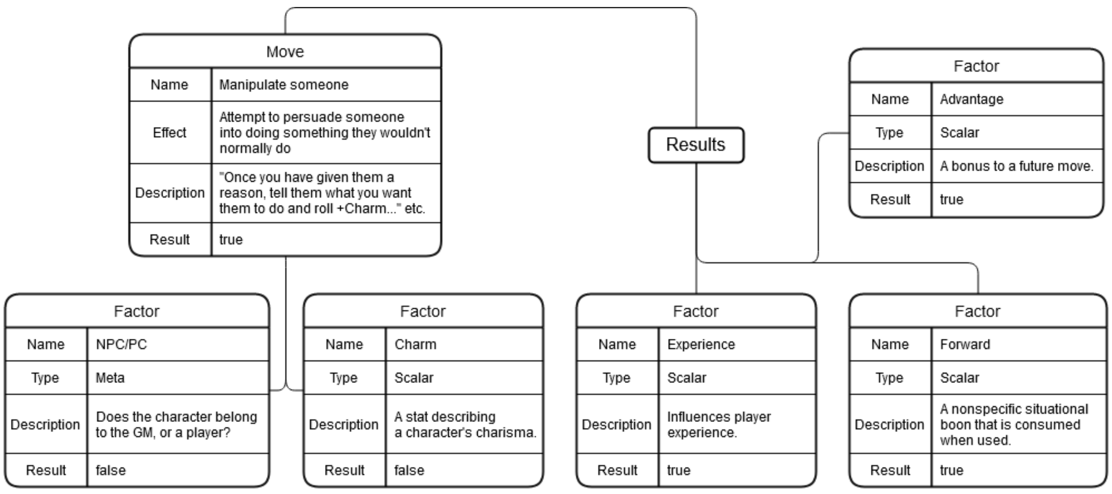

Move
Name
Manipulate someone
Effect
Attempt to persuade someone into doing something they wouldn't normally do
Description
"Once you have given them a reason, tell them what you want them to do and roll +Charm..." etc.
Result
true
Results
Factor
Name
Advantage
Type
Scalar
Description
A bonus to a future move.
Result
true
Factor
Name
NPC/PC
Type
Meta
Description
Does the character belong to the GM, or a player?
Result
false
Factor
Name
Charm
Type
Scalar
Description
A stat describing a character's charisma.
Result
false
Factor
Name
Experience
Type
Scalar
Description
Influences player experience.
Result
true
Factor
Name
Forward
Type
Scalar
Description
A nonspecific situational boon that is consumed when used.
Result
true

Figure 4.2: A diagram of *Monster of the Week*'s "manipulate someone" move as implemented in the pattern language.

> On a miss, it's up to that hunter to decide how badly you offend or annoy them. They mark experience if they decide not to do what you asked. Monsters and minions cannot normally be manipulated.
>
> Advanced: On a 12+ they must act under pressure to resist your request. If they do what you ask, they mark one experience and take +1 ongoing while doing what you asked.[20]

Starting from the beginning of the move, we examine what mechanical interactions occur as the move is executed. The move starts with a Charm roll, so its first factor is the player's Charm stat. Next, the outcome will depend on whether or not the character is played by the GM or another player, thus we might describe "NPC or PC" as the next factor. The next result depends on the parameter under which the outcome falls; in our pattern language this is assumed for all moves unless stated otherwise, so there is no need to list this as a factor. Since there are no other conditions for this move's resolution, we must then determine if there is any mechanical result from the move beyond its story impact. In this case, this does not occur for the NPC portion of the move, but it does for the player character portion. If the move is a success, the move's target gets experience and forward. If it's a partial success, they just get experience. If it's a miss, nothing happens. If the move ends in an advanced success, they get experience and a +1 advantage to all moves while the manipulated action is being carried out. Therefore, we can say that this move has the results of "experience", "forward", and "advantage", constituting the full set of factors that could arise from the move.

The *player rules* pattern governs system interactions pertaining to player characters. This pattern references six others: *resource*, *equipment*, *playbook*, *character creation*, *character improvement*, and *extra mechanic.*

A *resource* is any stored value that a player tracks in their personal notes. This might include health, energy, or other such values. Player stats (such as the Hot stat in *Apocalypse World* or the Dark stat in *Monsterhearts 2*) also count as resources. A resource usually tends to have an implied range of possible values, as well as an initial value indicating the amount of a given resource that a player begins play with. If this default value varies, it can be clarified further within the *character creation* pattern.

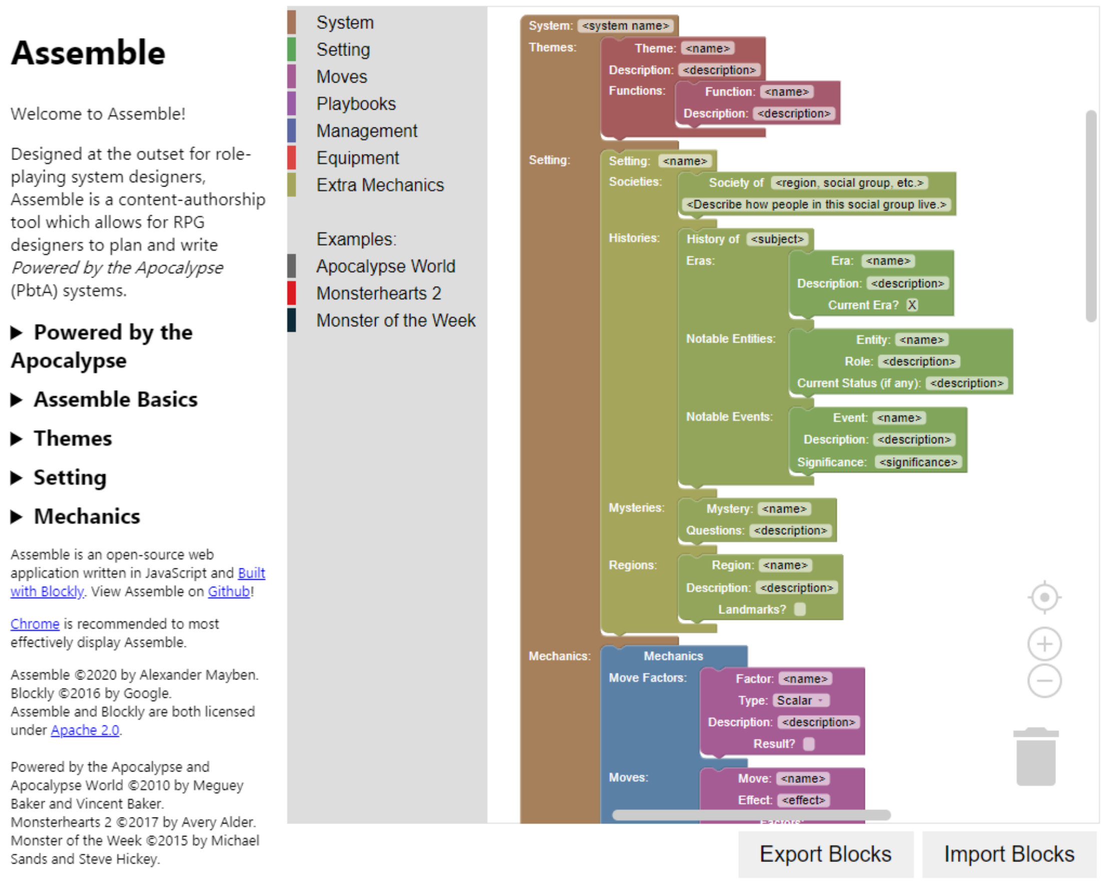
The *equipment* is the pattern which describes pre-defined items that players can acquire. Equipment references two patterns: *item type* and *item.* An *item type* is a category of items with defined properties in the system. Any item type can also have a number of subtypes defined; these subtypes are themselves identical to item types, and can also have subtypes. An *item* is any tool or object that can be acquired in the system, and must have at least one distinguishing item type associated with it. An item can also have additional types and subtypes.

A *playbook* is a character class in the system, fulfilling a particular role in play. Playbooks can have their own unique moves and resources, and might have a number of items to choose from at the start of play. Beyond these, a playbook has two unique patterns: *introduction* and *feature*. A playbook's *introduction* summarizes what makes it unique among other playbooks. The introduction can include a flavor pitch tonally describing the playbook from the standpoint of the system's setting, a description of the playbook's social role (e.g. the intended personality of the playbook's character), and/or a description of its mechanical role (e.g. what abilities the playbook may have within the context of the game itself). A *feature* is a mechanic that affects play outside of the context of a move. Features specify a number of factors that might affect a variety of moves or play situations for that playbook, while not necessarily corresponding to a specific action. The distinction between moves and features in this case is intentionally vague, allowing the designer to decide for themself whether a mechanic corresponds to a class of player action, or is unsuited for being designated as a move *per se*. This also encourages the designer to consider the benefits of providing a playbook with traits beyond the framework of moves.

The *character creation* and *character improvement* patterns are closely similar in their function, as both describe processes by which playbooks are maintained over the course of play. Character creation specifies the process of generating a new playbook from scratch, while character improvement describes character advancement, or "leveling up". Each pattern lays out a *general process* followed by all classes, demarcated by an ordered list of *step*s describing each individual action to be taken. These patterns can also define special processes for playbooks, signified with the *playbook process* pattern. Character improvement also requires conditions to be met for improvement to occur; this can also be specific to certain playbooks, in which case the *playbook conditions* pattern is used.

The last pattern described by the pattern language is the *extra mechanic* pattern. This pattern allows for additional rules to be established that affect the course of play. Extra mechanics can also include a set of associated factors, moves, or resources.

4.2 Assemble


Assemble

Welcome to Assemble!

Designed at the outset for role-playing system designers, Assemble is a content-authorship tool which allows for RPG designers to plan and write *Powered by the Apocalypse* (PbtA) systems.

▶ **Powered by the Apocalypse**

▶ **Assemble Basics**

▶ **Themes**

▶ **Setting**

▶ **Mechanics**

Assemble is an open-source web application written in JavaScript and Built with Blockly. View Assemble on Github!

Chrome is recommended to most effectively display Assemble.

Assemble ©2020 by Alexander Mayben.
Blockly ©2016 by Google.
Assemble and Blockly are both licensed under Apache 2.0.

Powered by the Apocalypse and Apocalypse World ©2010 by Meguey Baker and Vincent Baker.
Monsterhearts 2 ©2017 by Avery Alder.
Monster of the Week ©2015 by Michael Sands and Steve Hickey.

System
Setting
Moves
Playbooks
Management
Equipment
Extra Mechanics

Examples:
Apocalypse World
Monsterhearts 2
Monster of the Week

System: <system name>
Themes: Theme: <name>
Description: <description>
Functions: Function: <name>
Description: <description>
Setting: Setting: <name>
Societies: Society of <region, social group, etc.>
<Describe how people in this social group live.>
Histories: History of <subject>
Eras: Era: <name>
Description: <description>
Current Era? X
Notable Entities: Entity: <name>
Role: <description>
Current Status (if any): <description>
Notable Events: Event: <name>
Description: <description>
Significance: <significance>
Mysteries: Mystery: <name>
Questions: <description>
Regions: Region: <name>
Description: <description>
Landmarks?
Mechanics: Mechanics
Move Factors: Factor: <name>
Type: Scalar
Description: <description>
Result?
Moves: Move: <name>
Effect: <effect>

Export Blocks
Import Blocks

Figure 4.3: The Assemble interface, as viewed in Google Chrome.

Assemble is a computer-aided design application written in JavaScript that translates the aforementioned pattern language into an interface. This interface allows designers to organize and develop system parts through a comprehensive visual vocabulary that aids understanding and composition of *Powered by the Apocalypse* systems. Assemble was developed using the open-source engine Blockly.

Assemble's editor consists of a central tree of blocks that simulates the pattern language of Section 4.1, with each block signifying a pattern. This tree can be navigated with the scroll wheel of the mouse or either of the two scroll bars. Blocks can be added into the workspace from the *toolbox* (the gray toolbar on the left side of the editor), or deleted by dragging and dropping them into the trash bin icon on the bottom right.

The toolbox divides the blocks of the pattern language into seven main categories: **System** contains the theme, function, and player rules blocks, **Setting** contains all patterns related to the system's setting, **Moves** contains factor, move, and parameter blocks, **Playbooks** contains playbook, introduction, feature, and playbook move blocks, **Management** contains character creation, character improvement, playbook process, step, condition, and resource blocks, **Equipment** contains equipment, item type, and item blocks, and **Extra Mechanics** contains extra mechanics blocks. The toolbox also lists example blocks from three existing systems entered into Assemble: *Apocalypse World*, *Monsterhearts 2*, and *Monster of the Week*.

Below the workspace are two buttons that allow the user to import and export Assemble projects. "Export Blocks" saves the current workspace to the user's hard drive as an XML file, while "Import Blocks" loads a previously-saved XML file back into Assemble. Not only does this allow for Assemble's users to write system content over multiple design sessions, but it allows for pre-existing system material to be shared and edited, encouraging homebrew-minded use cases.

On the far left side of the page is a sidebar that describes how to interact with the editor, summarizing *Powered by the Apocalypse* and the pattern language.

4.2.1 Blockly Development Observations

For the pattern language to be translated into Blockly, a unique block needed to be coded to represent almost every pattern within it. Blocks within the Blockly engine are defined via an initialization function, which specifies the fields, inputs, connections, and validators used by the block. We can examine this further within the history pattern's code, which is reproduced in Figure 4.4.

The function begins with the declaration of all named inputs and fields, in order. Each input is a grouping of fields that are shown on a subsequent row of the block. Assemble uses two types of inputs: dummy inputs, which only store fields specific to the given block, and statement inputs, which can specify both block-specific fields and connections to other blocks. Together, both provide for full coverage of the necessary operations of a pattern language: dummy inputs specify the data within a pattern's own fields, whereas statement inputs allow for patterns to easily reference sub-patterns. In this example, the history pattern has one dummy input containing the history's subject, and three statement inputs. Each of these inputs references a corresponding sub-pattern: era, entity, and event.

Once inputs have been declared, the block then specifies how it can connect to blocks on the same hierarchical level in the block tree. This specifies whether a pattern acts on its own or utilizes multiple instances of the same pattern to accomplish its purpose. In the case of the *history* pattern, our pattern language specifies that multiple history blocks can be referenced by a given setting. Thus, we specify "next" statements,

```
Blockly.Blocks['history'] = {
  init: function() {
    this.appendDummyInput()
        .setAlign(Blockly.ALIGN_CENTRE)
        .appendField("History of ")
        .appendField(new Blockly.FieldTextInput("<subject>"), "name");
    this.appendStatementInput("era")
        .setCheck("era")
        .appendField("Eras: ");
    this.appendStatementInput("entity")
        .setCheck("entity")
        .appendField("Notable Entities: ");
    this.appendStatementInput("event")
        .setCheck("event")
        .appendField("Notable Events:");
    this.setInputsInline(false);
    this.setPreviousStatement(true, "history");
    this.setNextStatement(true, "history");
    this.setColour(75);
    this.setTooltip("What happened or is happening in the world that is
       relevant to the players?");
    this.setHelpUrl("");
  }
};
```

Figure 4.4: The history pattern as coded within the Blockly engine.

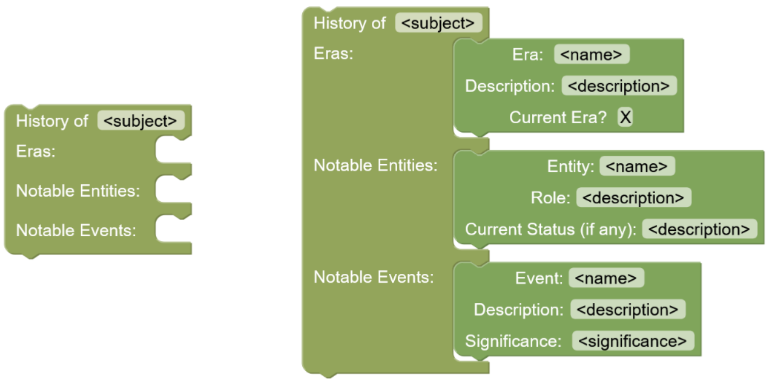


Figure 4.5: Left: The block generated by the code in Figure 4.3. Right: With sub-pattern blocks (era, entity, and event) connected.

allowing multiple history blocks to be connected together.

For some blocks, especially those encompassed by the *mechanics* pattern, we might specify dynamic inputs, which affect the shape of the block depending on how the user interacts with it. Many of these dynamic inputs are simple, only requiring the use of a simple validator function based on the content of a particular field. Several other blocks require the use of memory to track their inputs, which must be handled by a mutator function. Mutators store a string of information within a block's XML code to be re-accessed when the block is reloaded within a page. These functions facilitate communication between the block's JavaScript code and the markup language of the browser in which Assemble is viewed, tracking the number of dynamic inputs that the block contains. Given the variety of data types that can be saved as JSON (JavaScript Object Notation), this functionality is well-suited for JavaScript to store and read information. However, mutators mandate the use of a helper function to process the XML's stored information, requiring additional effort on the development end.

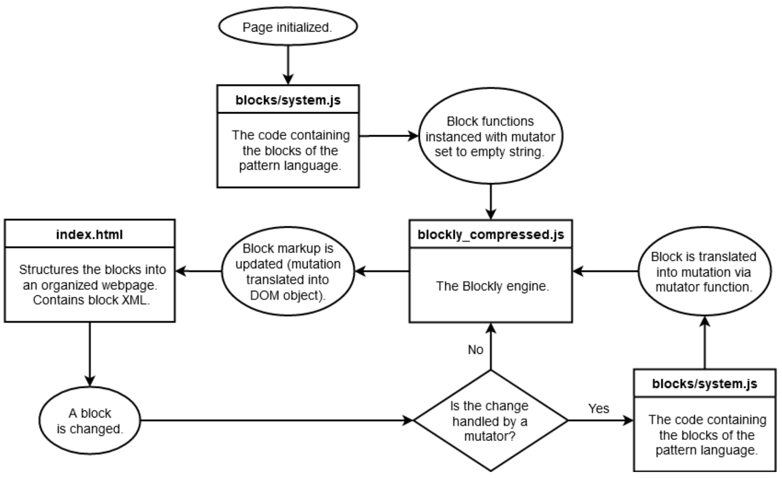
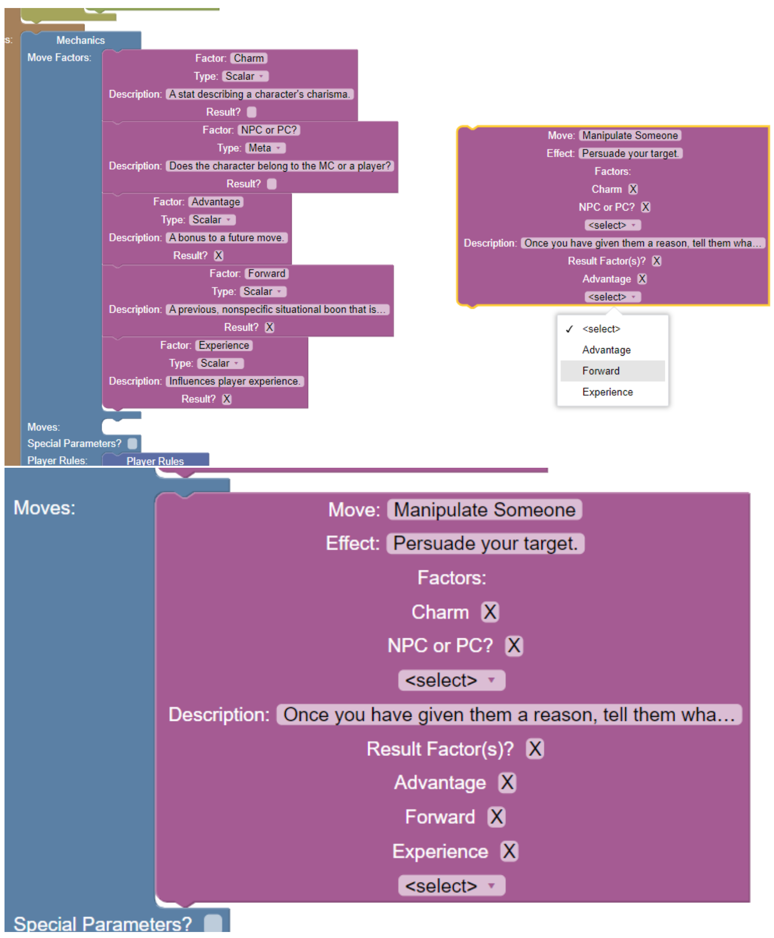

Page initialized.
blocks/system.js
The code containing the blocks of the pattern language.
Block functions instanced with mutator set to empty string.
blockly_compressed.js
The Blockly engine.
Block is translated into mutation via mutator function.
Block markup is updated (mutation translated into DOM object).
index.html
Structures the blocks into an organized webpage. Contains block XML.
A block is changed.
Is the change handled by a mutator?
No
Yes
blocks/system.js
The code containing the blocks of the pattern language.

Figure 4.6: A flowchart summarizing how blocks interact with mutators.

Since mutators are effective at storing dynamic information, they are quintessential for allowing patterns to correspond with one another. Mutators and their helper functions serve numerous purposes within Assemble's code, such as specifying the types and subtypes that a given item pattern can use, or listing the available starting items which a playbook has to choose from. Another example of this can be seen in move blocks. Before building a move, we must first build its factor blocks and slot them into the system. Once we begin filling in the move itself, we can then select its factors from a drop-down list that populates with the factor blocks in the tree. There are two lists in this case, one for regular factors and a second for results. For each factor added to a list, the mutator function stores the factor's name into a JSON object maintaining both lists as arrays. This object is then stored as a string within the move block's XML data. The block's helper function then receives that string, re-parses it into two arrays, and uses those arrays to update the block's inputs.


Mechanics
Move Factors:
Factor: Charm
Type: Scalar
Description: A stat describing a character's charisma.
Result?
Factor: NPC or PC?
Type: Meta
Description: Does the character belong to the MC or a player?
Result?
Factor: Advantage
Type: Scalar
Description: A bonus to a future move.
Result? X
Factor: Forward
Type: Scalar
Description: A previous, nonspecific situational boon that is...
Result? X
Factor: Experience
Type: Scalar
Description: Influences player experience.
Result? X
Moves:
Special Parameters?
Player Rules: Player Rules

Move: Manipulate Someone
Effect: Persuade your target.
Factors:
Charm X
NPC or PC? X
<select>
Description: Once you have given them a reason, tell them wha...
Result Factor(s)? X
Advantage X
<select>

✓ <select>
Advantage
Forward
Experience

Moves:
Move: Manipulate Someone
Effect: Persuade your target.
Factors:
Charm X
NPC or PC? X
<select>
Description: Once you have given them a reason, tell them wha...
Result Factor(s)? X
Advantage X
Forward X
Experience X
<select>
Special Parameters?

Figure 4.7: Top: A move block being written, beside the list of factors it uses to generate options for its drop-down lists. Bottom: The finished block slotted into place within Assemble's tree.

Though many of Blockly's features make the engine well-suited to composing pattern languages, there are some hurdles that hinder the engine's usefulness with regards to pattern language development. The first is optimization; when portions of block data rely on reading lists of other blocks to generate content, the functions which use that data tend to draw more heavily upon system resources when there are greater numbers of blocks. Large volumes of blocks within the workspace or toolbox often reduce performance. An engine that is more optimized for larger numbers of blocks, especially blocks with dynamic components, would thus be more effective in aiding the design process. Another hurdle lies in Blockly's rendering of text fields. As of writing, text-wrapping functionality does not exist in Blockly for text-based block fields; this results in strings longer than 50 characters being cut off by an ellipsis within the editor. The inability to view an entire string of design text in a block's position makes it difficult to read the pattern's information at a glance, hampering productivity. Finally, the nature of the Blockly engine simply makes programming nonstandard functionality both challenging and unintuitive. Many features which would be useful for simulating pattern languages are simply not implemented at the engine level, requiring a large number of helper functions to be directly coded, tested, and debugged. This unfortunately limits Blockly's access to developers with intermediate-level or greater experience in writing JavaScript, and significantly impacted the time required to develop Assemble. While in many ways the engine is structurally and visually well-suited to pattern language visualization, Blockly currently lacks functionality that could significantly ease the process of writing pattern languages in the engine.

Chapter 5

Conclusion

In this thesis we examine how to improve design strategy for tabletop role-playing games and evaluate how a simplified design process can aid top-level system design, homebrew design, and ultimately play. Simultaneously, we recognize that there is a lack of research toward developing design ontologies for tabletop role-playing games. This work responds to this need by introducing a pattern language which covers a well-known class of tabletop role-playing games, as a theoretical basis for a future, more holistic design ontology. Here we also describe Assemble, a digital interface that simulates said pattern language, as a demonstration of how pattern languages can contribute to a streamlined and intuitive design process for tabletop role-playing design. We also note aspects of Blockly, the engine in which Assemble is implemented, that are both amenable to (and less suited for) the simulation of pattern languages. This evaluation also posits some considerations for how software can most effectively ease enablement of general-purpose pattern languages through computer-aided design resources.

Due to both the scope and engine in which the application was developed, the focus of this thesis had to be adapted. Furthermore, as a result of a relatively small work team and the conditions present in the academic environment in which this research was conducted (in the midst of both a graduate student worker strike and a global pandemic), production of the pattern language and Assemble was somewhat hampered. While the language itself was developed based on the contents of six different *Powered by the Apocalypse* systems, the validation process was only completed for approximately four, with a fifth only being validated later in the software design stage. Additional time and resources would have assisted in allowing more systems to be validated in this manner, which would have likely made the pattern language more rigid as a result. On a broader level, the scope limitations inherent to a thesis of this nature also inhibited a pattern language applicable beyond a specific set of role-playing games from being developed.

On the software side, Blockly, the engine used by Assemble, has many affordances that ease the process of development for pattern language simulation, but unfortunately lacks other key features that would have substantially reduced the timeline necessary to implement the pattern language. While the simplistic way in which the application handles visualization and data linkage provided many benefits for organizing a complex pattern language, the lack of certain predefined methods for pattern reference and the requirement of major time investment to sufficiently implement the previously designed pattern language proved to be significant obstacles for speedy completion of the software. Runtime performance of the final application was also less than ideal, as larger block trees constructed in Assemble unfortunately suffer from deficiencies in speed and high resource demand, both inherent within the Blockly engine.

Limitations aside, this thesis effectively developed a pattern language that responds affirmatively to *Powered by the Apocalypse* systems and lays forward a potential framework for a future TTRPG language to be produced. Furthermore, the Assemble program shows an novel and useful means of examining system design, presenting the process in a format that is relatively intuitive and understandable for designers when compared with conventional methods. The ease and simplicity by which Assemble can be used makes the application useful for both high and low level system designers, including homebrewers. As a result, both the pattern language and Assemble represent a conceptual breakthrough in easing the TTRPG system design process, providing a worthy blueprint for future innovation in TTRPG design ontology.

Bibliography

[1] Homebrew campaign settings. RPGNet Wiki, `https://wiki.rpg.net/index.php/Homebrew_Campaign_Settings`, accessed 2020.

[2] r/unearthedarcana. `https://www.reddit.com/r/UnearthedArcana/`, accessed 2020. Subreddit.

[3] Avery Alder. *Monsterhearts 2.* Buried Without Ceremony, 2017.

[4] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, 1977.

[5] D. Vincent Baker. Powered by the Apocalypse, part 1. Lumpley Games, `https://lumpley.games/2019/12/30/powered-by-the-apocalypse-part-1/`, December 2019.

[6] Meguey Baker and D. Vincent Baker. *Apocalypse World.* Lumpley Games, 2010.

[7] Staffan Bjork and Jussi Holopainen. *Patterns in Game Design.* Charles River Media, Hingham, Massachusetts, 2004.

[8] Nathan Black and Sean Nittner. *Big Bad World*. Big Bad Con, `https://www.bigbadcon.com/big-bad-world/`, 2016.

[9] Elin Carstensdottir, Erica Kleinman, and Magy Seif El-Nasr. Player interaction in narrative games: Structure and narrative progression mechanics. In *The Fourteenth International Conference on the Foundations of Digital Games (FDG '19)*. Association for Computing Machinery, August 2019.

[10] Neil Fraser, Quynh Neutron, Ellen Spertus, and Mark Friedman. Blockly. Google Developers, `https://developers.google.com/blockly`, 2012.

[11] Neil Fraser, Quynh Neutron, Ellen Spertus, and Mark Friedman. Blockly Developer Tools. Blockly Demo, `https://blockly-demo.appspot.com/static/demos/blockfactory/index.html`, accessed 2020.

[12] Fred Hicks and Rob Donoghue. *FATE Core System.* Evil Hat Productions, 2013.

[13] Marissa Kelly, Sarah Richardson, and Whitney Beltrán. *Bluebeard's Bride.* Magpie Games, 2017.

[14] Adam Koebel and Sage LaTorra. *Dungeon World.* Sage Kobold, 2015.

[15] Robin D. Laws. *Hamlet's Hit Points.* Gameplaywright Press, Roseville, Minnesota, 2010.

[16] Mike Mearls and Jeremy Crawford. *Dungeons & Dragons: Player's Handbook.* Wizards of the Coast, Renton, Washington, 2014.

[17] Jason Morningstar and J Li. Pattern language for LARP design. `http://www.larppatterns.org/`, May 2016. Paper downloaded from ”main document” button.

[18] Yue Pan and Eric Stolterman. Pattern language and HCI. In *Proceedings of the 2013 ACM Conference on Human Factors in Computing Systems (CHI 2013): Changing Perspectives*. Association for Computing Machinery - Special Interest Group on Computer–Human Interaction (ACM SIGCHI), April 2013.

[19] Aaron A. Reed. *Changeful Tales: Design-Driven Approaches Toward More Expressive Storygames*. PhD thesis, University of California, Santa Cruz, June 2017.

[20] Michael Sands and Steve Hickey. *Monster of the Week*. Evil Hat Productions, 2015.

[21] Mikołaj Wicher, Agnieszka Kisiel, and Marcin Słowikowski. Larp Design Cards. Nordic Larp, `https://cdn.nordiclarp.org/wp-content/uploads/2018/05/Larp-Design-Cards-PDF.pdf`, May 2018.

[22] José P. Zagal, Michael Mateas, Clara Fernández-Vara, Brian Hochhalter, and Nolan Lichti. Towards an Ontological Language for Game Analysis. In *Proceedings of DiGRA 2005 Conference: Changing Views – Worlds in Play*. Digital Games Research Association (DiGRA), June 2005.

[23] Csenge Virág Zalka. Adventures in the classroom: Creating traditional story-based role-playing games for the high school curriculum. *Storytelling, Self, Society*, 12(2):173–206, Fall 2016.

Appendix

Links

- Pattern language validation spreadsheet: `https://docs.google.com/spreadsheets/d/1P2GpTZyKnU2-WIhK3DCZRbxRujiWSvs8Ge7eYKy6Crc/edit?usp=sharing`
- Assemble's GitHub repository: `https://github.com/amayben/assemble`