The *Unlimited Rulebook*
Architecting the
Economy Mechanics of Games

Wilson Kazuo Mizutani

Thesis Submitted
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
for the
Doctorate Degree in Computer Science

Program:
Computer Science

Advisor:
Prof. Fabio Kon

This research was supported by FAPESP
under grant 2017/18359-6

São Paulo, September 2021

The *Unlimited Rulebook*

Architecting the Economy Mechanics of Games

Acknowledgements

My academic journey through the world of game development officially began in November 2009, when some college friends and I decided to make a game together. That team went on to become USPGameDev, a student special interest group at the University of São Paulo. I love to tell everyone I find in the street how many of us now work on *real game companies* all over the world. Except for me, that is. I spent the last decade learning, programming, teaching, organizing events, participating in game jams, researching, and dedicating myself to many, many other smaller activities involving USPGameDev. Even though I always admired my friends that now actively take part in the game industry, it seems something kept drawing me to that place. The place I helped build where anyone could go to start their own journey in the game development world.

This thesis was the way I found to materialize everything I lived these past 12 years. It is my tribute to everyone I met during this time, for every single one of you taught me something that made its way into the process of making this work. That said, not everything found a place among the pages of this final manuscript. These are dark times and I grossly overestimated my willpower to properly tie all the loose ends. Instead, I shamefully offer these simple words of gratitude to *all of you*.

There are a few names I do want to shine the spotlight on. First of all, Vinícius, one of the fellow founders of USPGameDev, who agreed to co-author a journal article with me. Also my eternal *Age of Empires* mentor — may your villagers never meet the unforgiving tusks of the wild boar! Next, I would like to offer my eternal gratitude to those who joined me in my shenanigans to put the *Unlimited Rulebook* to the test: Rica, Zé, and Heitor. Heitor also taught me the joy of writing and convinced me to finally create something of my own. Thank you three for always indulging me. Speaking of indulging, I am forever indebted to the four patient souls that agreed to the interviews needed for my research (and sorry I cannot put your names here!). Professor Paulo and fellow researchers Higor and Leonardo showed me the light when the time came to design the more formal steps of my research methodology. And, of course, I thank my advisor, professor Fabio, for making sure I had all the reasons to be proud of my work. During the worst times of the last two years, the ones I could always count on were my partner, Maki, and my brother, Akira. It is thanks to both of you that I had the safest and most welcoming home to shelter in while the storm raged outside.

Last, there will always be a special place in my memories for the Institute of Mathematics and Statistics from the University of São Paulo. The professors, the staff, and the hundreds of colleagues I met there give me the unshakeable certainty that it was worth it seeing this work through.

Abstract

The creative process of developing digital games alternates between modifying the game and playing it to evaluate the produced experience. However, the more technical effort is required to change the game, the longer it takes to evaluate it and the more expensive it becomes to change it. Our research focuses on a part of the creative process in games that is particularly prone to expensive technical changes: economy mechanics with self-amending rules. To minimize the cost of the changes they involve, we propose the *Unlimited Rulebook*, a reference architecture that guides developers when designing their game systems. We use a consolidated systematic process that traces each design decision back to the information sources that support it then evaluates the resulting reference architecture both qualitatively and empirically. The results show that, for the appropriate game genres and feature sets, the *Unlimited Rulebook* successfully avoids expensive retroactive changes by relying on extensibility, data-driven design, adaptive object models, and emulated predicate dispatching.

Keywords: computer games, software architecture, reference architecture, design patterns, game mechanics.

Resumo

O processo criativo de se desenvolver um jogo digital alterna entre modificar o jogo e jogá-lo para avaliar a experiência resultante. No entanto, quanto mais trabalho técnico é necesário para mudar o jogo, mais se demora para fazer essa avaliação e mais caro fica fazer mudanças. Esta pesquisa se concentra em uma parte do processo criativo em jogos cujo esforço técnico em realizar mudanças é bastante custoso: mecânicas de economia com regras de auto-alteração. Para minimizar o custo das mudanças que essas mecânicas envolvem, nós propomos o *Unlimited Rulebook*, uma arquitetura de referencia que guia desenvolvedores ao projetar seus sistemas de jogos. Usamos um processo sistemático e consolidado que associa cada decisão de projeto à fonte de informação que levou a ela e, depois, avalia a arquitetura de referência resultante tanto qualitativamente quanto empiricamente. Os resultados mostram que, para gêneros de jogos e conjuntos de funcionalidades condizentes, o *Unlimited Rulebook* reduz mudanças retroativas caras com sucesso, valendo-se de extensibilidade, *data-driven design*, *adaptive object models* e *predicate dispatching*.

Palavras-chave: jogos de computador, arquitetura de software, arquitetura de referência, padrões de projeto, mecânicas em jogos.

Contents

Chapter 1

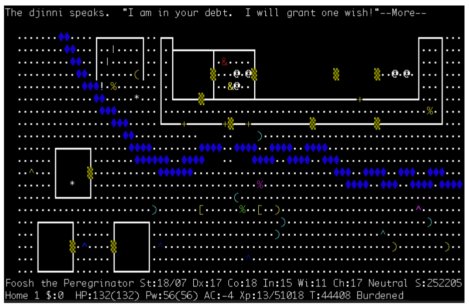
Introduction

"For me, good design means that when I make a change, it's as if the entire program was crafted in anticipation of it."

Nystrom (2014)

NetHack is a dungeon-crawling role-playing game so old its original "graphics" were rendered through ASCII characters on console terminals (DevTeam, 1987) (see Figure 1.1). Yet it is one of many surprisingly complex titles from the *rogue-like* family of games[1]. To illustrate this complexity and, more importantly, the development cost it carries, let us talk about one of the most infamous creatures in *NetHack*: the cockatrice.

Cockatrices (and chickatrices) "resemble roosters with a reptilian tail and bat wings" (NetHack Wiki, 2019). They are a kind of monster players must fight in *NetHack*. However, they are particularly complex and dangerous because merely touching them (or their dead bodies) instantly petrifies any living being. That means the players die and have to start a new character all over again. Because of this lethality, cockatrices provide some of the most frustrating and entertaining[2] experiences in the game. Yet, as programmers and software architects, let us consider how that might work inside the code (which is written in `C`). There is no specific action "touch this" in *NetHack*. Touching is merely an implicit part of many other actions, such as striking with a weapon, picking an item up from the floor, or even just walking over the same tile as another entity. Whenever the developers created the cockatrice, they had to add a check for the possibility of touching a petrifying object in every single one of



```
The djinni speaks.  "I am in your debt.  I will grant one wish!"--More--
```

```
Foosh the Peregrinator St:18/07 Dx:17 Co:18 In:15 Wi:11 Ch:17 Neutral S:252205
Home 1 $:0  HP:132(132) Pw:56(56) AC:-4 Xp:13/51018 T:44408 Burdened
```

Figure 1.1: Screen capture from *Nethack* (DevTeam, 1987). By default, it has an ASCII-based user interface, though graphical frontend alternatives also exist nowadays.

[1]The term *rogue-like* refers to role-playing games about exploration, combat, and resource management where death is permanent and the world is always procedurally different every time you play. The name itself comes from the game of the same name, *Rogue* (Toy *et al.*, 1980), and has seen an increase in popularity over the last decade, with many games borrowing different aspects from the format.

[2]Players can wield cockatrice bodies if they wear gloves and swing them at *other* monsters, effectively weaponizing its petrifying properties at great risk of tripping over in the next set of stairs and becoming a victim themselves.

these cases.

Though we cannot affirm if that is exactly how it went, by investigating the source code of *NetHack*[3], we can assert that they use a macro called `touch_petrifies` to verify whether a monster or monster corpse comes from a cockatrice, and it is present in fifty-eight (58) different lines of code across twenty-four (24) different source files. That means it is possible that adding this single creature to the game required reading, understanding, and modifying dozens if not hundreds of lines of code (if we consider that programmers have to understand the context of each added line). Besides, if the programmers wanted to change the macro to also work on items, for instance, that would require changing all the fifty-eight lines of code as well as other collateral consequences in each case, such as adding new parameters to functions. Similarly, we can imagine that whenever they add an in-game action that might implicitly involve touching something, they have to remember the possibility that something might be a cockatrice *or* a cockatrice body.

The point here is **not** that things like *NetHack*'s cockatrices were a bad idea. On the contrary, they are amazing. More games should implement this kind of surprising behavior. However, how can we add such cross-cutting features into a game without spending more time making sure it works than actually experimenting with new fun ideas? This thesis is not specifically about *NetHack* nor its cockatrices but about creating games with similarly complex and entertaining features by making the most out of the time we spend working on them. To do so, we must clarify what these "complex and entertaining features" are and understand how they interact with the development process of games. Then, once we know how the dynamic works, we propose our solution.

1.1 Research Problem

Our starting motivation is that we want to make fun games. There are, in fact, many aspects of games that make them fun but only one aspect that produces amusing experiences *and* is unique to the games: **gameplay**. Though this exclusivity follows from the term itself, in reality, there is no consensus for the exact definition of gameplay. Adams and Dormans (2012) define it as "the challenges that a game poses to a player and the actions the player can perform in the game", for instance, while other authors simply take its meaning for granted. We discuss this further throughout Chapter 2, especially Section 2.1, but, for now, based on Adams and Dormans' definition we see that gameplay is something interactive (it relates to how users interact with the system) and engaging (the users willfully face challenges for their own amusement).

The problem here is identifying whether a given instance of gameplay is engaging and why. There are many approaches and techniques, but the most assured way of knowing is by *playing the game* and assessing it yourself or through analysis of other people playing because experiencing gameplay is subjective (Schell, 2020, chapter 28). This means that game creators not only need to apply a world of knowledge when designing a game, but they also need to do it over and over after each gameplay change they make. **It is an iterative creative process** (Schell, 2020, chapter 8).

We want the creative process of designing gameplay to be as smooth as possible so that all

[3]github.com/NetHack/NetHack, last accessed February 26th, 2021.

our effort goes into tackling this already challenging and subjective domain and because this is a great part of what aggregates value to games. If possible, we want the creative team working full-time on making the game fun. That, nonetheless, imposes many challenges at different levels of the game development process. **In particular, it is the role of the software architect to solve a great part of these challenges.**

1.1.1 Creative Process Pipeline

The iterative design process in games is like any other design process: designers determine the problem at hand, find a solution, evaluate its success, then either go back if it is not enough or move on to the next problem otherwise. In this continuous cycle, every feedback counts because designers cannot test their game forever, and testing it has an implicit cost. That is, in digital games, like any software, to pick a solution we have to put it into the system — only then can we effectively play the new gameplay content. Once we have done that, we need to evaluate it and tweak it until we are satisfied or give up on that particular approach. All the while the game system keeps changing. In theory, this means the creative process of games continuously stumbles into the *technical process* of writing software with every iteration cycle[4]. The development goes from creative to technical and technical gets in the way of creation, because either the designers must interrupt their workflow or they have to curb that idea to circumvent the technical limitations.

In practice, there are many ways to avoid the creative process from stumbling into technical limitations like requiring programmers to write code for a new feature. The **reuse** principle is the most straightforward: make new features using software that already exists. This can be done by allowing features to be added through data instead of code (Rabin, 2000), where data is created, modified, and exported through editing applications then loaded into the game at runtime. Even more, some games can *dynamic load the data while running*, sometimes even from an editor embedded into the game itself, further reducing the feedback latency of the creative process. **These approaches, however, are not trivial: they require careful forethought of the *game architecture* so that it supports the constantly unstable feature set of games under development.**

1.1.2 Software Evolution Costs

Even then, software reuse is not always possible. When you must add entirely new types of features into a game (e.g., *NetHack*'s cockatrices), (re)writing code is unavoidable. In that case, the less code programmers read and change, the faster the creative process goes back to full speed. Minimizing the size of code intervention may not always be the best long-term solution but it is the ideal scenario of this already expensive case where reuse is not possible. Minimal changes are possible when all we need is to change how an isolated case works or when we add a new case

[4]Throughout this thesis, we rely on the dichotomy between these so-called creative and technical processes (along with creative and technical teams) simply as a didactic tool to highlight the role of software architecture in the development of games. In practice, this division is blurry at best. Developers may work on both “creative” and “technical” aspects of a game, especially in smaller studios. At the same time, there is creativity in programming and technical formality in design.

that pre-existing code already supports, for instance. Again, **the structure of a codebase can only afford this with premeditated effort — also involving its architectural design**. Though at the beginning of writing the game code this is easier, the more the project grows, the harder it is to accommodate new code without a conscious design effort. In software engineering, the quality of software architectures that allows new features to be added with minimal or no changes to previous code is known as **extensibility**. That is, making software that is as ready as possible for new, unpredictable features.

1.1.3 Worst Case Scenario

Why was neither reusability nor extensibility possible for *NetHack*'s cockatrices, where dozens of lines of code had to be inspected and changed? Because there is a worse case among the worst cases when the code simply cannot be ready for a new feature. This happens because some types of features are so different from what the game did so far that it requires a widespread revision of old code — like adding an if-case to all pre-existing features where the player is implicitly touching something in *NetHack*. If it comes to that, programmers ideally need this to be as easy as possible, i.e., if they have to interact with multiple parts of the code, they will work faster if we can reduce those parts as much as possible and have very clear guidelines of what parts must change. We want **flexible** code that keeps its complexity in check. That said, there are no exact approaches to achieve flexibility like with reusability and extensibility, though there are good practices and patterns that help in specific situations.

In games in particular, when it comes to adding new gameplay features, we have identified a particularly challenging source of "worst-case scenarios". Like with the cockatrice example, this case happens when games have **self-amending mechanics**: gameplay features that treat other gameplay features as first-class values. Features that specify things such as "whenever features with property A happen, do B", "the value of X counts as double its amount for features of type Y", "features with property A cannot happen while features of property B are in place", among many, many other possibilities. **When games have self-amending mechanics, their software architecture becomes a challenge for the creative process**.

Moreover, we have also identified in what kinds of games self-amending mechanics are predominant. They are games where simulating an **internal economy** is one of the (if not the most) important types of gameplay in its design, as opposed to the simulation of physics and progression (for discussion on the definitions we use, see Section 2.1.4). This includes, as expected, role-playing games, but also strategy games of many kinds — from real-time strategy worldwide warfare games to turn-based, 5-minute-play card games — sandbox games, interactive fiction, and a variety of simulation-based games.

Thus, **this research focuses on games that favor economy mechanics**, since they are a more general superset of self-amending mechanics, and aims **to help teams that depend on a fast and continuous creative process** to add value to their products. We understand that, as computer scientists and software engineers, one of the key parts of meeting the requirements of this problem is appropriately designing the software architecture of the game.

1.1.4 Knowledge Reuse

The next step is, given a certain game with its particular economy mechanics and creative process, determining *how* we design a good architecture for it. The first question should be whether there are any architectures or frameworks that already solve our problem and how we can reuse them. Game development is known, for instance, to rely on game engines since they offer reuse and, thus, reduced development costs. However, most game engines do not solve our problem entirely, especially concerning economy mechanics, because games are too unique in this aspect. General-purpose engines (like *Unity3D*[5]) leave much of the work of game-specific needs to developers of the game itself and genre-specific engines (like *RPGMaker*[6]) are, of course, not usable in all cases. We want a solution that supports as many games as possible but also takes into consideration any engines and tools they are using. Even more, we would like a solution that provides architects with a *standard* for what a working architecture for this problem looks like.

For that, instead of proposing a software reuse approach through engines and frameworks, we believe that *knowledge reuse* is more effective, since it contemplates the uniqueness of each game, both in terms of gameplay and creative process. Results from one of our previous studies support this approach since it showed that "[game development] researchers favor reduced development complexity, but often tailor their solutions to specific games or genres", concluding that "a valuable avenue for future research in the field is the generalization of architectural solutions around specific types of mechanics" (Mizutani *et al.*, 2021).

This approach includes, for instance, researching *design patterns* (Gamma *et al.*, 1995) and *architectural patterns* (Buschman *et al.*, 1996, pages 1–8). In particular, we propose, based on the many individual solutions game developers have used over the decades, a *reference architecture* (Bass *et al.*, 2003; Nakagawa *et al.*, 2011) for games focused on economy mechanics, which is a more comprehensive and systematic approach. It has the added benefit of providing a *reference model* to help specify the requirements of economy mechanics and provides a standard through which architects may evaluate how their architecture supports the creative process of economy mechanics.

1.1.5 Problem Statement

In digital games focused on economy mechanics and, in particular, self-amending mechanics, the software architecture directly impacts the productivity of the creative process. Nevertheless, the lack of a generic and reusable implementation leaves architects to design for themselves without a means to consistently assess the quality of their work. There is no formalized, structured knowledge to serve as a standard such as a reference architecture.

1.2 Proposal: the *Unlimited Rulebook*

In this Ph.D. thesis, we propose *Unlimited Rulebook*, a reference architecture that formalizes how individual game architectures support the creative process behind their economy mechanics, even

[5]unity.com (last accessed January 14th, 2021)
[6]rpgmakerweb.com (last accessed February 23rd, 2021)

when they have self-amending mechanics. Using a pre-existing methodology for designing reference architectures (Nakagawa *et al.*, 2014), we investigate multiple information sources to build a model of how the domain of economy mechanics works and what its architectural requirements are, then design and evaluate the *Unlimited Rulebook*. Details of our methodology are further explained in Chapter 3.

1.2.1 Objectives

Though our central proposal is the *Unlimited Rulebook* architecture, our research as a larger process involves a series of objectives:

1. To determine when economy mechanics can be easily extended and when they require deep architectural changes.
2. To formalize how software architecture improves the creative process of economy mechanics.
3. To collect, design, and evaluate solutions that guide architects in games with economy mechanics.

That is, understanding the problem domain and the implications of the many approaches that developers have used so far is just as important a part of our work as the *Unlimited Rulebook* itself. Besides its practical purposes, this reference architecture serves as an embodiment of the knowledge accumulated in this research. At the same time, this thesis works as comprehensive documentation of the fundamentals, rationale, and validation behind the *Unlimited Rulebook*.

1.2.2 Contributions

Besides the *Unlimited Rulebook* architecture itself, our research brings the following contributions:

- An extensive study of the relations between game development processes, economy mechanics, and software architecture.
- An extension to the ProSA-RA method of designing reference architectures.
- A quasi-experiment design for evaluating the *Unlimited Rulebook* architecture.
- Findings from interviewed game developers about the role of software architecture in economy mechanics.
- A reference model for representing the economy mechanics simulation of particular games.
- Two proofs-of-concept, one being a complete game prototype, that provide reference implementations for the *Unlimited Rulebook* architecture.
- A new course on game programming for the Computer Science Graduate and Post-Graduate programs at the University of São Paulo.

1.3 Text Organization

After this introductory chapter, we follow with an extensive review of the literature, in its many forms, regarding game development and software architecture in Chapter 2. This includes related work from other research initiatives in the same or neighboring fields. Chapter 3 details our research methodology, including how we use different kinds of expert knowledge as a basis for the design of the *Unlimited Rulebook* architecture. We present and discuss each of our information sources. Following that methodology, Chapter 4 lists the exact requirements the *Unlimited Rulebook* needs to meet to fulfill the objectives of our research. This chapter elaborates on the many kinds of economy mechanics and architectural challenges extracted from our information sources. Next, in Chapter 5, we compile all these findings into a reference model and present the *Unlimited Rulebook* itself, explaining the causal relations among the requirements, the model, and the actual design of the architecture. Chapter 6 evaluates our proposal in two different ways: a quasi-experiment and a series of proofs-of-concept, showing how it achieves its objectives and, thus, how it solves our research problem including the costs and benefits involved. Finally, Chapter 7 concludes this thesis with a discussion of the achievements of the *Unlimited Rulebook* architecture and lists future work this research field would benefit from.

Chapter 2

Literature Review

"Seemingly inconsequential decisions about data, representation, algorithms, tools, vocabulary and methodology will trickle upward, shaping the final gameplay. Similarly, all desired user experience must bottom out, somewhere, in code."

Hunicke *et al.* (2004)

Developing games is a particularly multidisciplinary software enterprise. Before we discuss how to architect economy mechanics, we must first better understand the context that surrounds them and how their challenges differ or align with other types of architectural enterprises. This chapter has the goal of revising the fundamental concepts from the literature (Section 2.1) and how other fields of research tackle the same or similar issues as the ones we do (Section 2.2).

2.1 Fundamental Concepts

Economy mechanics are but one of many aspects of game development. Yet, they do not exist in isolation. They interact with other types of mechanics and many — if not all — other parts of a game system. Games rely on a remarkably wide spectrum of features ranging from real-time data exchange via network connections to fine-tuned artificial intelligence algorithms. This section is an attempt to summarize how all these pieces of the puzzle fit together, so we can properly discuss the role of economy mechanics in the implementation of a game. Then, we can explain why, when, and how its architecture impacts the development process as a whole.

2.1.1 Digital Games

Digital games compose a specific type of software system. There are many ways to categorize games regarding other types of systems or game formats (e.g., board games). Though these definitions always bear blurry limits, they are nevertheless useful, if not necessary, to determine the scope and limitations of our research.

Compared to games in general, digital games are games that require a digital system (such as a computer or game console) to run a piece of software that users interact with to experience the game (Schell, 2020, chapter 3). This both limits and expands the possibilities for what digital games can do compared to other types of games. Computers are capable of such fast computations that digital games can simulate virtual worlds many orders of magnitude more complex than what a human brain could in games where that is the main means of adjudication (e.g., board games). However, the set of possible events inside a digital game must somehow arise from the predetermined code and data that execute it. For instance, if a player tries to walk past the limits of the virtual world, they are most likely to find nothing but an eerily empty space. On the other hand, table-top role-playing games, for instance, have no such limitation: the player known as the “game master” can always come up with a new piece of the virtual world to present to the other players whenever they go beyond the expected boundaries.

When we consider games as a particular type of software system, we might classify them based on their purpose. Where many systems consist of tools to automate our daily tasks, such as a spreadsheet processor, game systems are usually developed to provide entertainment. As computers, smartphones, and digital technology, in general, became more and more widespread, many forms of digital entertainment have come to be. Yet we can still tell game software apart from a movie and television show streaming service, which is also a software system designed for entertainment. The key difference, we argue, is that the game software *itself* provides a fundamental part of the entertainment, as opposed to the media content it gives access to. That is, while a video player is fun because of the movies I watch on it and a social media app is fun because of what I see other people posting on it, a digital game is fun in great part *because of how it runs as a program* — or, more technically, because of its features. In fact, Murphy-Hill *et al.* (2014) shows that creativity is more valued among game developers than among developers for other types of software systems, suggesting that fun can rise from the game code itself. Of course, there are exceptions to this from both sides. Visual novels, for instance, are games almost entirely focused on content instead of features (usually). At the same time, the growing practice of gamification brings every other kind of entertainment system ever closer to games, by definition. More and more features are added to these systems, so users have fun with them, increasing their engagement and, therefore, consumption.

Both these loose attempts at defining digital games bring us to an actual and more promptly useful definition proposed by Gregory in his book, *Game Engine Architecture*. According to this author, game systems are **soft real-time interactive agent-based computer simulations** (Gregory, 2019, page 9). He goes on to explain the role of each part of this rather long, composed term but, for now, let us focus on the core aspect of Gregory’s definition: that a digital game boils down to a *computer simulation*. A simulation allows us to “represent a source system via a less complex system” (Bogost, 2006) and, in games, that source system has the peculiarity of being often fictional and/or abstract (hence the *soft* part of the definition, as opposed to *hard simulations*). The games in the *Pokémon* series (Game Freak, 1996–2021), for instance, simulate a fictional world where fantastical and charming creatures can be tamed and deployed in competitive matches[1] (Figure 2.1).

Much of the appeal a *Pokémon* game has revolves around its promise to bring its fictional world to life through its gameplay — the promise that anyone can catch and raise their own

[1]Since its inception over twenty years go, this conveniently chosen example — *Pokémon* — has gone on to become a widely famous multimedia franchise across the world. That is, there are many ways people experience the fictional world of *Pokémon* (the source system) aside from games. In fact, as the generations pass, they are more likely to experience them this way *before* playing a *Pokémon* game, if ever.

Pokémon and become the very best like no one ever was. This fictional world (source system) is translated into the *virtual* world (less complex system) simulated by the game. That is, the basic feature of the game — its simulation of the "Pokémon world" — is, by itself, inherently fun — which is how we link Gregory's definition to our discussion over what makes a digital game different from other entertainment systems. The virtual worlds which come to life through the simulation code inside a digital game are a key part of what makes games, games.

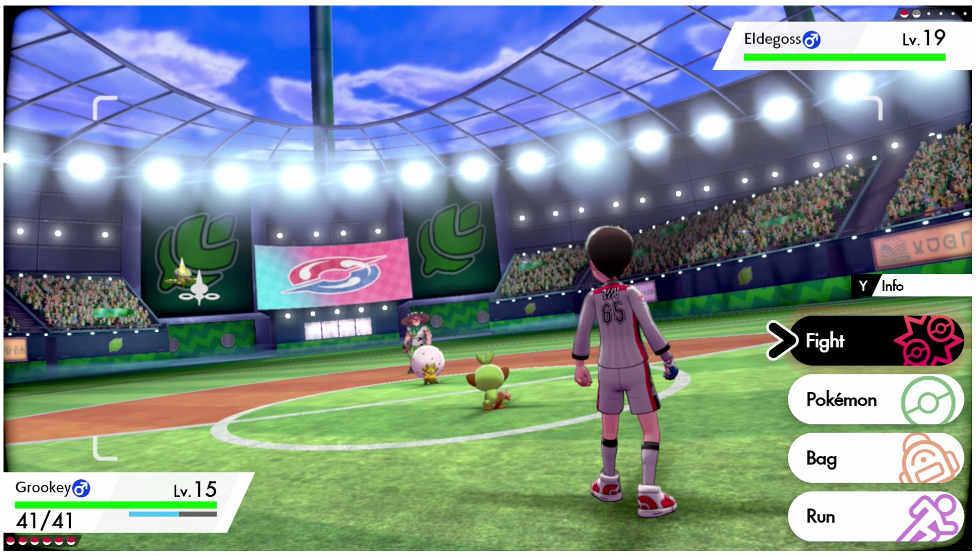


Figure 2.1: Screen capture from *Pokémon Sword & Shield* (Game Freak, 2019). Pokémon games are not only about the aesthetics of the world and its creatures, it is very much about what you can do in that world: capture, train, battle, raise, trade, among many other possibilities. Pokémon has considerably complex simulations under the hood and that is a part of why gamers continue to buy new titles.

In particular, the promise of being a *Pokémon* master can only be fulfilled because the game allows the player to actively engage with its virtual world. After all, as Gregory pointed out, games are *interactive* computer simulations (Gregory, 2019, page 9). User interactivity is what pushes digital games beyond the bytes processed in their simulation and into the brain of users. That is how the software becomes the means through which the games are experienced — as the first attempt at an informal definition at the beginning of this section highlighted. Digital games function as games in its broader sense because the player can take part in its simulation through interaction with an interface. This part of digital games has two complementing technical requirements. First, the player must be able to *perceive* the current state of the simulation — usually through graphics and sound — so they can make informed decisions about what they want to do next. Second, whatever it is they want to do, there must be a way to input that information into the game system. That is done through one or more peripheral input devices (keyboards, controllers, touch screens, etc.), which are processed by the game and translated into state changes inside the simulated virtual world. Additionally, the advances in augmented reality and virtual reality in recent years take game immersion many steps further (Viana and Nakamura, 2014).

An important part of what makes user interaction engaging in games is that it happens in *real-time* (another part of Gregory's definition). By rendering an up-to-date visual representation of the simulation state 30, 60, or even more times per second, the player has the illusion of a living world behind the screen. When that image is properly matched to music, sound effects, and voice-overs, the game further bridges the gap between its virtual world and the user's experience (Karen Collins, 2008; Mizutani, 2017; Yoshikawa, 2018). Every input or output delay dampens the momentum of the player, causing from minor yet distracting inconveniences to major unfair outcomes, particularly in multiplayer on-line games. The need for performance in this field has also always been a strong motivator in the progress of both hardware and software. Graphics cards, greatly responsible for allowing games to improve graphics while keeping a steady rate of Frames Per Second (FPS), are a good example of this, especially since their use in data science and other fields of computer science continues to increase.

We have mentioned a few times so far the notion that the simulation inside a game has a *state*. The use of this term assumes that game systems operate as Finite State Machines (FSM): there is a finite (but usually immensely large) set of possible states a game simulation can be in (e.g., every possible coordinate the player avatar can be in) and a (likely even larger) set of possible state transitions (e.g., the player can change coordinates by walking or jumping). Much of our research cares about how to add new possible states and state transitions to a game with minimum programming effort. Part of the reason why this is challenging is that, as Gregory put it, games are *agent-based* simulations (Gregory, 2019, page 9). This means that a significant part of its state is divided into the individual states of independent "agents" that inhabit the virtual world of the simulation — in this work, we call them **game entities** or simply entities. What an entity represents in the game varies widely, from the self-evident avatars of virtual characters to invisible geometries and event timers. Their key characteristic is that their state is highly dynamic and usually independent of other entities or static state (e.g., the shapes composing the virtual world) but they *do* interact with each other. For instance, the procedures that cause an entity representing a car to move inside the simulation are responsible for constantly changing its position state but might take into consideration whether that car is colliding with other cars or static obstacles in the virtual world. The potentially vast set of possible interactions between entities in the game is, thus, one of the main concerns of our research. This brief discussion also hints at the important separation between simulation state and simulation *behavior*, which is the part of the game system responsible for causing the state transitions in the state machine of its simulation.

Summarizing this section, we have explored the definitions of digital games as:

1. games that are experienced through the interaction between players and a software system;
2. media- and feature-based entertainment systems; and
3. soft real-time interactive agent-based computer simulations.

We focused on this third definition, proposed by Gregory (2019). Its more technical nature helps us better understand what digital games do in terms of software implementation. At the same time, we keep the other two in mind since they highlight the purpose behind the implementation of a game. Now that we know what kind of system we want to study, we can discuss how people develop them.

2.1.2 Game Development Process

Digital games have long moved from university projects (Landsteiner, 2015) to a full-fledged industry. Studios may develop a game over a month with ten or fewer people, or they may take the greater part of a decade (Kahney, 2006) and spend hundreds of millions of dollars (McLaughlin, 2013). However, professionals still struggle to consistently meet their goals (Schreier, 2020).

Our research investigates the relationship between software architecture, economy mechanics, and the creative process in games. These are all aspects of the greater structure that is the development process behind game-making, where you see the effects of poor architectural decisions consuming resources (time, budget, people, etc.) from the production pipeline (Fowler, 2019). By understanding the technical requirements of this process, we make more informed decisions on the architecture of a game and determine how it deals with the simulation of economy mechanics. We must especially acknowledge where the architecture is more likely to hold back the pipeline.

Creative vs. Technical Processes

As a form of digital entertainment, games require both creative and technical effort (Nordmark, 2012, page 4). At the same time, as highlighted in Section 2.1.1, *the game simulation itself is a source of fun*, which suggests a fundamental intersection between the creative process of designing how a game works and the technical process of producing the software that embodies that design. In other words, developers often cannot introduce a fun feature or piece of content into a game without changing the game itself — its source code, its dependencies, its invariants, and, eventually, its architecture — but the process for making a game fun and the process for making a game work are different. Although not mutually exclusive, that difference implies an increased cost in the overall development process.

On the one hand, the subjective nature of the creative process benefits most from feedback cycles (Schell, 2020, Chapter 8). Whenever developers make a change to the game, they simply cannot predict for certain whether that change will have a desirable effect or not. The best way to make sure is to play it. At first, that can be the developers themselves and the rest of the team but, in later stages of development, even that is not enough: they need *other people* to play their game. This is called *playtesting* (Schell, 2020, Chapter 27). For a digital game, assessing the produced experience requires the software to be in a minimally playable state. The kind of change you make in the game may or may not compromise its stability[2]. When it does, developers cannot test it until the technical effort is put into implementing the necessary adjustments. As an additional note, the study by Politowski *et al.* (2020) shows, from an analysis of 200 game industry *post-mortems*, that design issues are the most common problem in game development.

On the other hand, the extensive precision of the technical process is expensive and part of the second and fifth most common problems in game development (Politowski *et al.*, 2020). The more a software system grows, the harder it is to change it, for a variety of reasons. If the involved parts of the code are *coupled* (ISO, 2017), changing one likely requires changing others that directly or indirectly depend on it. New features need to be integrated into the existing code

[2]In fact, understanding what architectures support what kinds of changes is one of the main aspects of this research.

at multiple entry points. Duplicated code affected by the change incurs duplicated effort. At the same time, preventing these and other issues with forethought is hard because there might be cases that were not considered or that were never needed in the first place.

Hence, while the creative process is exploratory, the technical process is conservative. Yet, they must interact with and balance each other. In their survey on software architectures and the creative process in games, Wang and Nordmark conclude that "the creative team can affect the software architecture [. . .] by adding in-game functionality" but also "has to some degree adjust their game play ideas to existing software architecture based on a cost/benefit analysis" (Wang and Nordmark, 2015). This shows that, despite the creative-technical dichotomy not being exclusive to games, they have the particularity that *the implementation itself is part of the creative process* since gameplay features and content may be part of the simulation itself, leaving both kinds of effort at odds with each other.

To understand where and under what circumstances these intersections characterize a bottleneck in the development process, we must elaborate on the actual technologies, tools, and assets involved in the development of a game.

Production Pipeline

It is easier to explain how the production pipeline of making games is by starting from the final product then tracking backward the steps that produced it. Upon installation on the end-user's machine, a game application consists usually of three parts: **an executable, the libraries it depends on, and a database with all the data the game needs to run**. After being executed, games further access more data on the computer to persist the players' profile and progress. The development pipeline for a game is responsible for streamlining the production of the first three parts of the software product. Figure 2.2 illustrates a simplified and generalized version of this pipeline and the rest of this section briefly explains the role of each node and step in that pipeline.

The bottom part of Figure 2.2 shows **the game executable**. It is what the end-user runs to effectively play the game. As with any other program, the development team writes its source code and deploys it to the target platforms. The game source code and its resulting executable are referred to as the **runtime component** of the game (Gregory, 2019, page 38). In particular, our focus is on the **runtime architecture** of games. Library code is often part of the runtime component too, but it can also be part of the **tool-side components**, which we describe next.

It is possible to make a game entirely out of code, especially if you rely heavily on *procedurally generated content* (Grey, 2017). However, in practice, developers still need to load **game data** at runtime to varying degrees. Some argue that the more you can build as data instead of code, the better (Rabin, 2000). Some important types of game data include **assets, configuration files, and scripts** (Gregory, 2019, page 59). In the context of game development, **assets** refer to media such as textures, 3D models, music, etc. The tools used to produce game data and the pipeline they compose are the tool-side components of the game development process and have architectural needs of their own, though that is outside the scope of this work.

Given the complexity of game data, developers might need dedicated editing tools to produce and maintain some if not all of it, as represented by the `editor.x64` artifact in Figure 2.2. Such

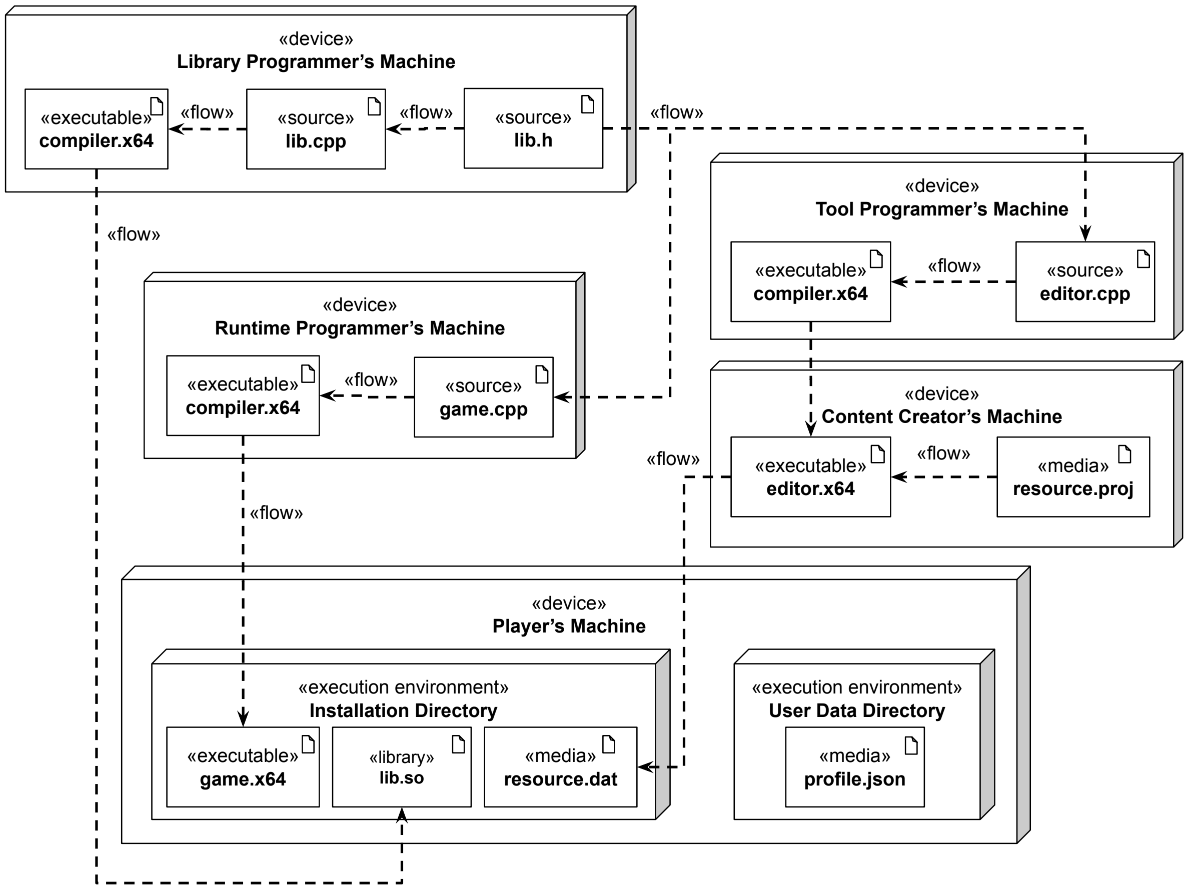


Figure 2.2: UML Diagram for a simplified and generic production pipeline in game development. In this diagram, we assume the game is written in `C++` and other format extensions are merely illustrative. Each node represents a *category* of possible implementations. For instance, the `Library Programmer's Machine` could be an in-house computer in the game studio or a third-party team's machine. It could also be used to implement a media manipulation library (e.g., reads and write textures), a reusable feature library (e.g., physics), or even a full game engine.

instruments, called **Digital Content Creation (DCC) tools** (Gregory, 2019, page 59), usually provide a **graphical user interface (GUI)** — an editing application — to manipulate data. They can be developed in-house by a game studio or be provided by third-party developers, both cases represented as the `Tool Programmer's Machine` node in Figure 2.2. In-house tools are more commonly used for studio- and game-specific formats, such as the world, character, and item data fed into the game simulation. Third-party tools are more common when it comes to more common-use data formats. For instance, artists could use *Blender*[3] to produce 3D models and export them using an industry-standard format such as `gltf`[4].

Similarly, programmers can reuse routines for loading standard data formats through **programming libraries**. In the `gltf` example, they could use the *Assimp*[5] library to load data in that format. That is why libraries can be both part of the runtime and tool-side architectures. DCC tools usually work on top of their own media formats and allow the creator to export their

[3]blender.org (last accessed January 11th, 2021)
[4]khronos.org/gltf (last accessed January 11th, 2021)
[5]assimp.org (last accessed January 11th, 2021)

work in more compatible formats, which must then be imported by the runtime game software using the libraries when needed.

Some DCC tools, however, have formats tied to very specific loading and execution mechanisms, so they come with both an editor for the media format they manipulate and a runtime library (also used by the editor) that knows how to import and load that media into the game. This kind of technology, which is present in both the tool side and the runtime component of games to support specific loading mechanisms, is also known as **middleware**. That is, their libraries are part of both "architectural sides", providing a bridge between them. For instance, FMOD Studio[6] and VORPAL (Mizutani, 2017) are audio middleware tools for games: the sound designer can edit sound events in them, then the programmers load and play those events using provided libraries to produce the same playback at runtime that the sound designer created on the tool side.

There can be a lot of different kinds of technology involved in the development of a game. We will call the **production pipeline** the overall process of creating media, controlling versions, converting them into compatible formats, building binaries, and packing everything together to distribute the game. One of the roles of the production pipeline is to allow an efficient workflow for the creative team to iterate over the game design and content. Automation of production steps, integration between technologies, reuse of software and formats, dynamic data loading, and other debugging utilities are among the practices that improve the production pipeline of game development (Wang and Nordmark, 2015). In particular, some of these can (only) be achieved through the adequate design of the runtime game architecture.

Game Engines

Loading standard data formats is not the only case for code reuse in games. Recurring features, such as physics simulation, can be promptly supplied by libraries such as *Bullet*[7] or *Havok*[8]. As with DCC tools, programming libraries used in games can be third-party or developed in-house.

However, there is much more in common between game implementations than just the programming libraries they use. Most games need to render graphics, process user input, provide graphical user interfaces, read from and write to the file system, connect over a network, etc. To capitalize on this common requirement, the industry developed many **game frameworks** with general-purpose infrastructural code capable of hosting a wide variety of games (Gregory, 2019, page 11). With them, for each new game, developers write only (or, at least, mostly) code and data that is unique to their game. Architectural patterns like the *Game Loop* or *State*, discussed in Section 2.2.2, are already laid out, for instance. These game frameworks are also known as **game engines**.

As Gregory (2019, page 12) puts it, game engines consist of "software that is extensible and can be used as the foundation for many different games without major modification" and "arguably a *data-driven architecture* is what differentiates a game engine from a piece of software that is a game but not an engine" (emphasis theirs). This distinction suggests that engines

[6]fmod.com (last accessed January 15th, 2021)
[7]pybullet.org (last accessed January 11th, 2021)
[8]havok.com (last accessed January 11th, 2021)

work similarly to a music or video player: you feed it data in the appropriate format, and it plays the content you gave it. That is what Gregory meant by "data-driven architecture" and other practitioners support this notion (Rabin, 2000). In practice, however, not all frameworks considered to be game engines follow this approach. For instance, *Godot*[9] and *Unity3D* are very data-driven engines while *LÖVE*[10] is more akin to a conventional white box software framework (Johnson, 1997).

In the end, game engines have become a very diverse field, with designs, implementations, and commercialization approaches that meet widely different requirements. Some engines include their own world (and other engine-specific) editors, while, on the other hand, world editors are practically always tied to a specific engine[11]. This is a particularly relevant phenomenon since it suggests that simulation data formats are harder to reuse, likely because each game has very different requirements for them and because their architecture is more coupled to them. Another important aspect of world editors is that, in some engines, they are built on top of the engine itself (e.g., *Godot*), in which case they probably share some or all data formats used in their runtime components and tool-side applications (Gregory, 2019, pages 65–67).

Development Models

A full-fledged production pipeline, along with the eventual in-house game engine, takes a considerable amount of effort and time to achieve. The steps that lead to a complete and functional pipeline depend on the development model used because it determines the priorities of development. That said, the creative aspect of game development makes the development process usually be characterized by three phases: preproduction, production, and maintenance (Aleem *et al.*, 2017). The flow between these phases is not necessarily linear or discrete.

Preproduction is experimental and exploratory. In this phase, the creative team tries to shape the game's design while the technical team tests what technologies best fit the team's needs. During this phase, the team might develop prototypes (also called minimally viable products or MVPs) and demonstration products (demos) so they can better assert their options and capture funds from investors and publishers. These artifacts might be completely thrown away once the game has well-defined goals.

That is when production starts, though it can be very hard to know when is the best moment to leave preproduction. The production phase is about building the game itself now that the team knows what they want the game to be and that it is viable to do so. Since the goals are less likely to shift mid-production, this is the moment when it is usually worth paying the costs of building a proper production pipeline. After all, the creative team still has a lot of creative work to do, and they need that technical friction out of the way.

In the past, games would only finish the production phase when they were considered complete and ready for consumption. Nowadays, as a game becomes minimally playable, companies start bringing in players for the so-called *early access*. This allows the company to iterate more explicitly given the concrete feedback from real clients — a practice that is sometimes seen in

[9] godotengine.org (last accessed January 14th, 2021)
[10] love2d.org (last accessed 14th, January 2021)
[11] There are exceptions, like, for instance, *Tiled*, found at mapeditor.org (last accessed January 15th, 2021)

a bad light because players are effectively working as a quality-assurance source for free (if not actually charged) and because some games are published in an early-access state but sold at full price. This is an example of how the business model seeps into the development process.

Once a game is complete, its production phase ends and its maintenance phase begins. During this phase, developers work mostly on fixing bugs, improving performance, and providing minor updates that they deemed unnecessary for the full release. Sometimes, however, developers might start work on a major update for the game, which can be sold separately in the form of an expansion or downloadable content (DLC). Each of these new updates requires a development cycle of its own — preproduction, production, and maintenance. This is another example of how the business model affects the development process. *Path of Exile* (Grinding Gear Games, 2013–2021), an online action role-playing game, releases major updates three to four times a year without any cost for the players, while *Faeria* (Abrakam Entertainment S.A., 2016–2020) charged for expansion packs released once a year, for instance. When this kind of content update cycle is more frequent, the role of the production pipeline in the creative process is even greater than usual.

Though the three-phase cycle might bear many similarities to cascade processes, it isn't the only way to use the three development phases. Especially indie and hobby developers are more likely to employ drastically shorter development cycles and "release early and often" as per agile principles. Of course, these kinds of developments lend to cruder games that become polished and more complete over time. For examples, *Factorio* (Wube Software, 2016–2021) released in early access in 2016 but was only considered complete with an update in 2020 (Team, 2020) and *Dungeon Crawl Stone Soup* has been in "early access" since 2006 and is in active development up to this day, despite its thousands of players[12]. In these cases, the technical and creative processes work tighter together, because the architecture of the game is growing at the same rate that its creative content is. We speculate this makes it harder to predict requirements and, thus, brings its own set of challenges to architects.

Games also have a very particular niche when it comes to developer-player interaction. Some games support *modifications* (mods): they allow the end-user to add to and change the game's media and data files to further develop their experience. Games like *Factorio* have a vast community of mod developers (modders) and this is believed to extend the lifetime of games, increasing loyalty and likeliness of buying future products from the same company (Lee *et al.*, 2020). Making games with support for mods is very challenging, however, since their architecture must be designed for embedding content and behavior through a great amount of external data. Moreover, "moddable" games strive to make the development of mods in their games accessible to non-programmers more interested in creative rather than technical content. Modding also provides valuable insight into the architectures of othewise closed-source games (Scacchi, 2017).

2.1.3 Software Architecture in Games

The subject of software architecture in games is vast. Scacchi and Cooper argue that "the architectural design of games, and how to make trade-offs therein, remains an open challenge in

[12]Based on the community size at reddit.com/r/dcss (last accessed February 8th, 2021)

[computer games software engineering]" (Scacchi and Cooper, 2015, page 271). Gregory has extensively written about it in *Game Engine Architecture* (2019) and we highly recommend that book as a more complete reference. Another important work in this field is *Game Programming Patterns* by Nystrom (2014). We refer to these books many times in this section and even throughout the thesis. Here we only provide a brief and superficial introduction to the concepts we believe are necessary to discuss our research in later chapters.

Software Architecture

There are many definitions for software architecture. According to Bass *et al.*, the "software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" (Bass *et al.*, 2003, section 2.1). In other words, this definition considers architecture to be a *quality* any software system possesses regarding some aspects of how it is organized. Shaw and Garlan (Shaw and Garlan, 1996), on the other hand, define software architecture in terms of its role in software design, software documentation, and software specification. That is, they consider it a discipline where you solve structural issues of software systems, building a (preferably documented) body of knowledge that describes the structural solutions chosen and how and why they fulfill the requirements of the developed system. This way, Shaw and Garlan's definition makes it clear that software architectures possess *intent* and depend on *communication* to be acknowledged.

One way or another, both these and other definitions often indicate that software architecture regards the "elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns" (Shaw and Garlan, 1996, page 1). There is a shared notion that software systems can be divided, at least conceptually, into parts with well-defined roles. At the same time, there are usually different ways through which this division is done, even when applied to the same system. These are different *views* we switch from and to to communicate and evaluate how a single architecture is characterized according to different perspectives. For instance, a view could consider how different components communicate over a network, while another view might elaborate on how the architecture achieves compatibility with different operating systems using a common interface.

Though software architecture usually applies to individual systems, some architectural solutions can be reused across different applications. The scope and type of solution reused varies. The classic *design patterns* (Gamma *et al.*, 1995), for instance, are more or less local solutions to runtime object-oriented code structure, usually covering how a handful of system elements interact with each other. An architectural pattern like the *Model-View-Controller* (Buschman *et al.*, 1996; Krasner *et al.*, 1988), on the other hand, involves the system-wide separation of elements and how control and data flow among them.

Ultimately, the purpose of studying and expanding the field of software architecture is to engineer software structures that are "cost-effective solutions to practical problems by applying scientific knowledge" (Shaw and Garlan, 1996, page 6). The actual costs and effectiveness criteria depend on the system domain and set of structural issues under consideration.

Putting together all these aspects of software architecture, we can now determine how this discipline intersects with game development. To discuss and propose solutions in this field, we must establish:

1. what structural *qualities* a game system has;
2. what goals define the *intent* behind the structure of a game system;
3. how to describe the structure of a game system through *elements*, *views*, and the *relations* between them;
4. what structures are commonly *reused* across different game systems and why.

The remaining of this section briefly presents some of the most common and fundamental concepts that address these questions. The discussion here is superficial and focuses on the architectural knowledge of games that is most pertinent to the creative process of designing gameplay.

The *Game Loop*

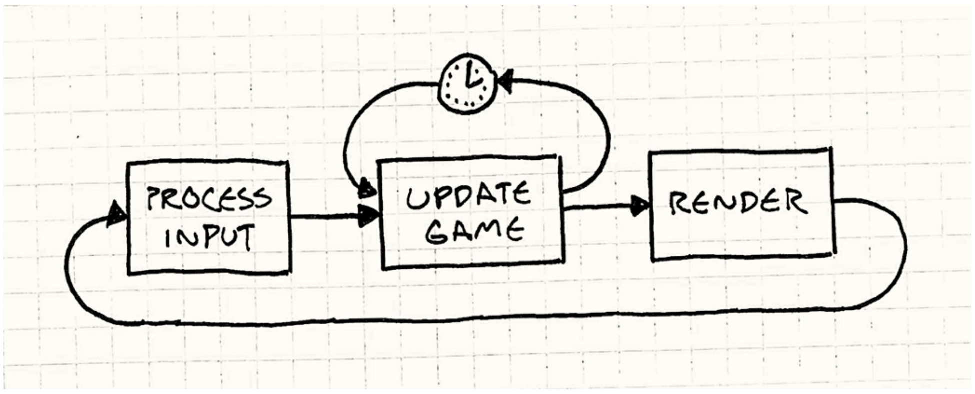


Figure 2.3: An informal flowchart illustrating how a standard *Game Loop* pattern looks, extraced from Nystrom's book (2014, Chapter 9).

As we have seen in Section 2.1.1, game systems are real-time interactive simulations. This basic definition already determines a series of requirements the structure of a generic game must follow. First, there are two recognizable parts of the system: user interaction and simulation. Second, a temporal simulation means the game possesses a sequence of states that progresses as time passes. That time must match the real-time interaction so that user input is processed as soon as possible and game output is kept up-to-date with the simulation state with a latency usually in the order of no more than several milliseconds. This synchronization is the link between simulation and interaction and the structure of the game system has to balance execution time between them.

Nystrom claims this concern is so ubiquitous that almost every game reuses the same architectural pattern to address it: the *Game Loop* (Nystrom, 2014, Chapter 9). Zamith *et al.* reinforce this notion by asserting that "game loops are of central importance in game development" (Zamith *et al.*, 2016) and Gregory further details its role in synchronizing the many real-time processing elements of a game (Gregory, 2019, pages 525–527). Essentially, the *Game Loop* pattern determines that a game system should have a high-level loop that alternates between advancing the simulation time and exchanging input and output with the user. In rough terms, the loop (see also Figure 2.3):

1. reads input data from peripheral devices and process them into user commands;
2. updates the simulation state according to how much time passed since the last iteration and according to the user commands issued;
3. processes the relevant data in the simulation to expose in the form of image, sound, and other output media available; then
4. unless the user requested to close the program, goes back to the first step.

A complete iteration of these steps is usually called a *frame*, referring to the rendered image frame on the screen. The *frame rate* or *frames per second* (FPS) measure indicates how many frames a game produces every second.

The main consequence of this pattern regarding the architecture of games is that most kinds of runtime processing are done in *discrete steps* every frame instead of being computed in full at once. For instance, if an AI algorithm calculates a path for a game entity to follow, it has to advance through that path in small steps spread across probably hundreds of frames. Notably, this means that each part of the game must be able to *remember* where it stopped the previous frame, implying some form of *local state* that is kept across game loops. The pattern also suggests how the general control flow of the game system works: most code is reached by a series of routine calls that trace back to the *Game Loop*.

Interaction Contexts

Like most interactive applications, what you see and what you can input into the system at any given time depends on the context of that interaction. For instance, clicking a certain position on the screen has very different results if you are on a menu or in the mid of playing a match of a first-person shooter game. That is, there usually are different *screens* a user sees when playing a game, each with its own set of input and outputs, and the player can navigate across these screens. Some game engines support this by using solutions similar to (or exactly like) the *State* pattern (Gamma *et al.*, 1995), commonly named *scenes*. The term, however, is also associated with the different stages players must progress through to finish a game, so different scenes do not necessarily imply entirely different user interactions — sometimes it is just the content that changes.

The use of the *State* pattern allows programmers to write user interaction code for a specific context without having to consider the interactions that exist in other contexts (see Figure 2.4). The interaction contexts become decoupled because of the additional level of indirection. One of the consequences of this is that state changes are a commonplace entry point for creating and removing runtime elements in a game. Some engines load entire scenes from data files, which list all elements the game should load to compose that scene, while elements that belong to a previous scene are likely no longer needed and may be unloaded from memory.

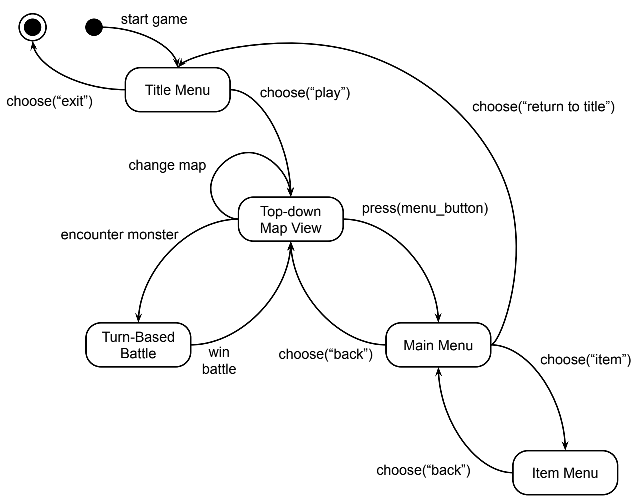


Figure 2.4: An example of a finite state machine for the interaction contexts of a hypothetical classic role-playing game like *Final Fantasy* (Square Enix, 1987–2020) or *Pokémon* (Game Freak, 1996–2021). This UML diagram we created shows how the user's input and the game output interact differently depending on the context. The *State* pattern (Gamma *et al.*, 1995) is used in these situations to allow implementing how the current game state responds to each interaction independently of one another.

Layered Architectures

Games are multimedia applications, relying on several lower-level technologies, from typical file system operations to carefully aligned data streaming to video cards. It also requires many math, geometry, and data structure algorithms, used across the entire code base. On top of these essential features, more complex ones are developed, such as physics simulation integrated with animation processing, all the way up to the *Game Loop*. Gregory states that the architecture of game engines organizes these elements into layers, where normally "upper layers depend on lower layers, but not vice versa" (Gregory, 2019, page 38) (see also Figure 2.5).

When designing elements of game architecture and their relations, it is always important to understand where they fit into the layered perspective of the architecture. Consciously considering the dependencies between elements helps prevent or, at least, prepare for changes that propagate through different parts of the architecture. For instance, by defining reusable interfaces to abstract lower-level implementations, games decouple most, if not all, of their systems from the specifics of operational system libraries, network protocols, data structure implementations, etc.

Subsystems

While the layered view gives us a "vertical" structure to think about game code, when you consider the many elements of a game architecture together with the *Game Loop* pattern, you can also find a "horizontal" way of dividing the elements of the game system. As we explained, the *Game Loop* follows a series of steps, more or less alternating between interaction and simulation updates. In practice, interaction and simulation are further divided into major elements of a game architecture, each responsible for processing a core aspect of the system. Gregory calls these major elements the *subsystems* of a game (Gregory, 2019, pages 526–527). For instance,

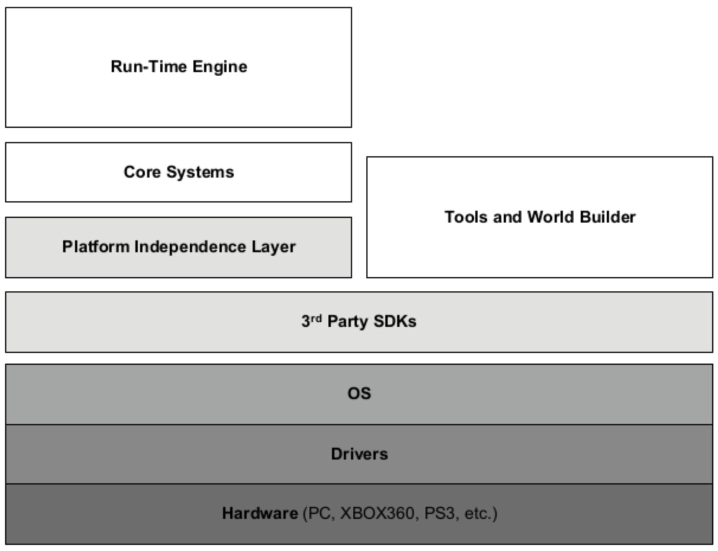


Figure 2.5: A simplified view of a possible layered structure used for both the runtime and tool-side architecture of a game, extracted from Gregory's *Game Engine Architecture* (2019, page 65). In this image, Gregory illustrates how some games and DCC tools share common features through the reuse of programming libraries, called third-party software development kits (SDKs) in the image. Among other benefits, parts like the *Platform Independence Layer* allow, as the name implies, implementing games and game engines that are cross-platform.

the graphical output could be divided into an animation subsystem, which updates the positional data of graphical objects, and a rendering subsystem, which draws the graphical objects onto the correct position of the screen.

Essentially, the subsystems of a game are how it divides its real-time processing responsibilities, be them physics, input mapping, network synchronization, AI decisions, etc. They are the elements that are directly serviced by the *Game Loop*, passing high-level control flow among themselves. For any part of the game that is developed, architects must consider under what subsystem it belongs and/or how subsystems access it when needed. The works of West (2018) and Plummer (2004) have very clear examples of the division of a game architecture into subsystems (see Figure 2.6).

Interaction vs. Simulation

We have been discussing game systems as interactive simulations since Section 2.1.1. However, in practice, this division is not explicit. Subsystems, for instance, do not always clearly fit into either simulation or interaction exclusively. An animation subsystem, especially in action games with real-time combat, effectively processes data that is part of both the simulation and the user interaction. If a 3D character model has an animation where it swings a giant hammer capable of destroying rocks, such an action would have both a simulation effect (breaking rocks hit by the shape of the hammer during its motion) and a user interaction perception (the avatar will visibly move according to the animation). Another blurry case regards user interface elements that have real-time effects, requiring a simulation of their own. A simple example could be a button that emits particles when the user hovers the cursor over it since the movement of the

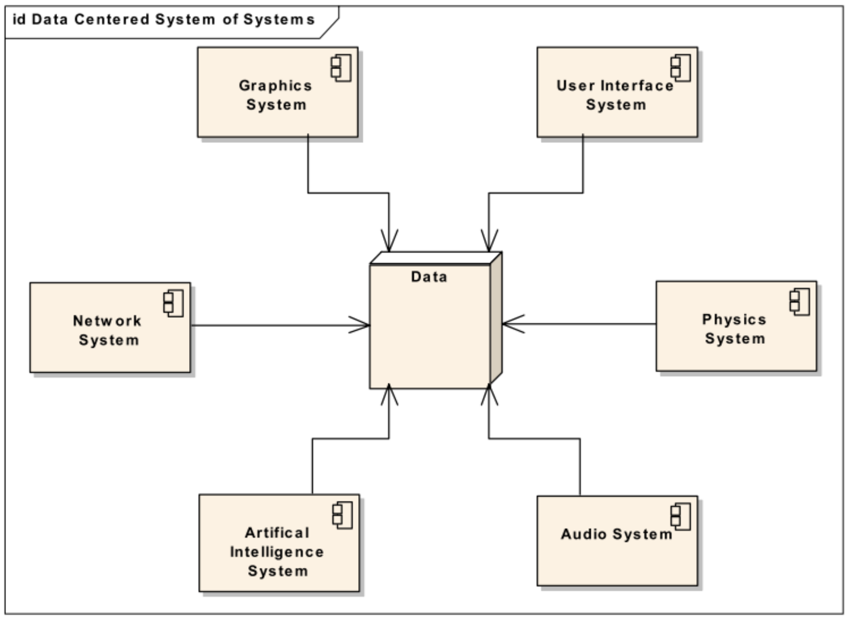


Figure 2.6: One of the UML diagrams from the game architecture proposed by Plummer in his master's dissertation (2004, page 51). His *Data-Centered System of Systems* design explicitly separates the structure of a game system into subsystems that apply specific operations over the general state of the game. While not all games use the data-centered approach, dividing responsibilities into subsystems is a common practice.

particles requires physical simulation every frame. The button, however, is not usually a part of the virtual world, conceptually speaking (it is not *diegetic*), so you might not consider it part of the simulation[13]. Though discerning features that relate to the in-game narrative from those that exist only for the player's eyes and ears (extra-diegetic features) is useful, it is important to acknowledge that both involve some form of simulation.

Because of this, rather than using the interaction-simulation dichotomy to separate subsystems, as an initial consideration might suggest, it is more useful to associate them with how data flows through a game system (see Figure 2.7). Simulation elements of a game both read and write data that belongs to the runtime memory of the game, like the positions of avatars, the colors of interface elements, the current score of the player, etc. Interaction elements either read from or write to data that is outside the game application: inputs events from a gamepad, packets from the network, pixel data to the screen, audio samples to speakers, etc. In this sense, all subsystems simulate something to a certain degree (and hence the reason why they need real-time servicing from the *Game Loop*) but only some of them directly interact with the user.

The important architectural consequence of this discussion is that most if not all subsystems of a game operate on a part of the internal game state, even if they deal mostly with interacting with the user. How that data is organized and, more notably, *shared* across subsystems is a fundamental part of designing a game architecture.

[13]The user interfaces of games like *Dead Space* (EA Redwood Shores, 2008) and *Nier: Automata* (PlatinumGames, 2017) are explicitly part of the diegesis and, thus, demonstrate how that assumption is weaker than it seems.

[14]Icons used are licensed under *Creative Commons BY 3.0* and were obtained from game-icons.net (last accessed May 13th, 2021).

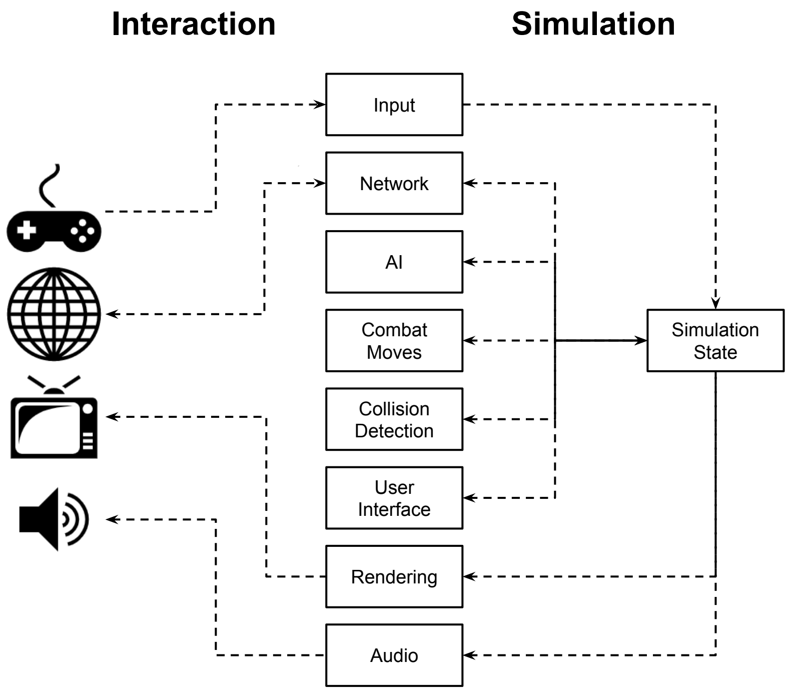
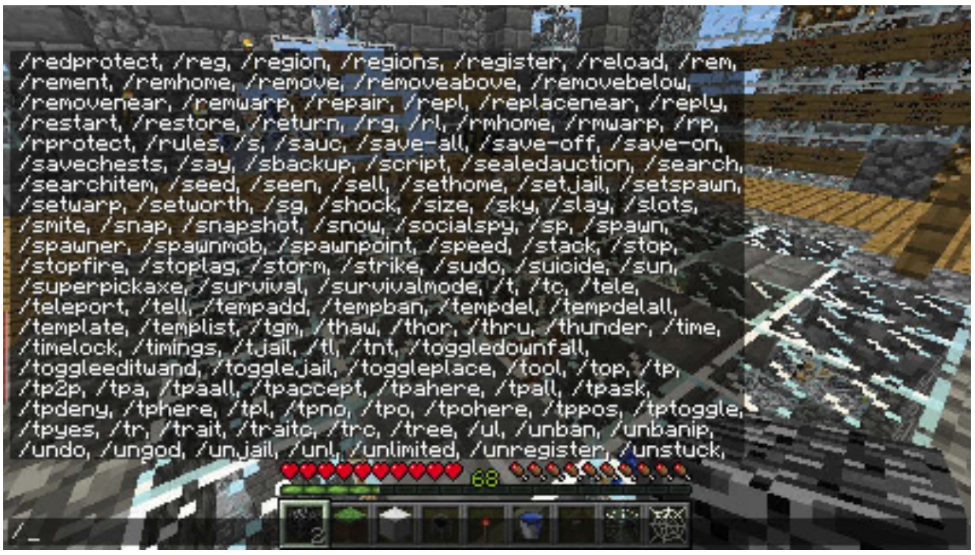


Figure 2.7: An example[14] of how to discern interaction features from simulation features in a hypothetical game. In this case, the diagram was inspired by fighting games, where physics and graphics must precisely match each other. All subsystems read data from and/or write data to the simulation state and these operations are all part of the simulation. Some subsystems additionally exchange data with lower-level drivers to reach the appropriate hardware pieces and, thus, the end-user. These are interaction features. Note how the "User Interface" subsystem, which you might expect had interaction features, in this case only communicates with the simulation state. That means only the "Rendering" subsystem sends geometry and texture data to the video card, while the "User Interface", during its frame slice, processes and prepares the part of this data that is under its responsibility. That is just how this example was designed but serves to demonstrate subsystems are not always clearly divided between interaction and simulation.

Development Tools

Finally, one basic aspect of game architecture we have to consider is how it interoperates with or provides development tools to a production team. As we discussed in Section 2.1.2, making a game involves a great amount of content, thus requiring proper tools to productively author this content. This might involve considering the architecture of the production pipeline as a whole, where the runtime game system and all other tool-side applications are the elements whose relations we must determine. Despite not being the focus of this research, we must be aware that these relations affect and are affected by the internal architecture of the game. Sometimes, as mentioned before, these tools might even be built on top of the game engine itself.

For instance, when starting a scene, some games load their data from the disk. If that data is already formatted the way the game expects it to be laid out in memory, loading is not only trivial but fast. However, it means that tool-size applications that manipulate that data must duplicate code to store it the same way the game does, convert it back and from a format of its preference, or reuse parts of the game code (creating a direct dependency). If the production



```
/redprotect, /reg, /region, /regions, /register, /reload, /rem,
/rement, /remhome, /remove, /removeabove, /removebelow,
/removenear, /remwarp, /repair, /repl, /replacenear, /reply,
/restart, /restore, /return, /rg, /rl, /rmhome, /rmwarp, /rp,
/rprotect, /rules, /s, /sauc, /save-all, /save-off, /save-on,
/savechests, /say, /sbackup, /script, /sealedauction, /search,
/searchitem, /seed, /seen, /sell, /sethome, /setjail, /setspawn,
/setwarp, /setworth, /sg, /shock, /size, /sky, /slay, /slots,
/smite, /snap, /snapshot, /snow, /socialspy, /sp, /spawn,
/spawner, /spawnmob, /spawnpoint, /speed, /stack, /stop,
/stopfire, /stoplag, /storm, /strike, /sudo, /suicide, /sun,
/superpickaxe, /survival, /survivalmode, /t, /tc, /tele,
/teleport, /tell, /tempadd, /tempban, /tempdel, /tempdelall,
/template, /templist, /tgm, /thaw, /thor, /thru, /thunder, /time,
/timelock, /timings, /tjail, /tl, /tnt, /toggledownfall,
/toggleeditwand, /togglejail, /toggleplace, /tool, /top, /tp,
/tp2p, /tpa, /tpaall, /tpaccept, /tpahere, /tpall, /tpask,
/tpdeny, /tphere, /tpl, /tpno, /tpo, /tpohere, /tppos, /tptoggle,
/tpyes, /tr, /trait, /traitc, /trc, /tree, /ul, /unban, /unbanip,
/undo, /ungod, /unjail, /unl, /unlimited, /unregister, /unstuck,
/_
```

Figure 2.8: In-game command-line interface of *Minecraft* (Mojang Studios, 2011), where the user can type commands that inspect and change the game state for debugging and other purposes.

pipeline instead relies on a separate format for scene data, there is less coupling and repeated code, but loading data at runtime is slower and more complex.

While DCC tools help produce content for games, they are also an additional source for invalid data and other incompatibility errors. For this and other reasons, an important class of elements a game architecture needs is a debugging toolset. Simpler debugging methods such as printing to a console terminal are poorly fit for real-time applications such as games. Visual inspection, tolerant error handling, and embedded command lines (see Figure 2.8) are more expensive but might save hours and days worth of programmers' time trying to figure out the origin of an inconsistent bug. One of the biggest impacts this has on the game architecture is that it relies on the simulation state being robust to unexpected behavior. For instance, the architecture might dictate that movement processing is only performed by a single subsystem and that it processes all the movement necessary for that frame in a routine call, with no other parts of the simulation state changing concurrently. This narrows down possible bug sources and prevents the introduction of inconsistent states by other parts of the *Game Loop*.

2.1.4 Game Mechanics

"Game mechanics" is a very common yet loose term used by the game community. It vaguely refers to parts of the design of a game related to *gameplay*, that is, how playing the game works. For that reason, creating, evaluating, and shaping game mechanics is usually the job of a game designer. However, as we have seen, the creative work behind a game is often tied to what its technical aspects allow creators to do. That includes, among other things, the runtime architecture of games.

From what we have gathered in the literature, there are mainly two lines of thought regarding what game mechanics are, *from the perspective of game design*. Some authors define game mechanics as the *affordances* a game provides to whoever (or whatever, in the case of AIs) interacts with it (Dubbelman, 2016; Järvinen, 2008; Osborn *et al.*, 2017; Sicart, 2008). For instance, Järvinen considers game mechanics as a "means to guide the player into particular behavior by constraining the space of possible plans to attain goals" (Järvinen, 2008, page 254). Other authors propose that game mechanics are essentially the rules determining the valid game states and state transitions that might occur, *more or less* like a mathematical model of the simulation (Adams and Dormans, 2012; Hunicke *et al.*, 2004; Larsen and Schoenau-Fog, 2016; Schell, 2020). Hunicke *et al.* (2004), for example, propose that mechanics "describe the particular components of the game, at the level of data representation and algorithms".

Simply put, these two veins of game design studies implicitly disagree on whether game mechanics revolve around player interaction or not. Now, looking at them *from the perspective of software architecture in games*, we can more clearly see the implications of each. The first definition (game mechanics as player affordances) maps to both simulation and interaction features since they consider how the user intervenes in the game state but, at the same time, it disregards all simulation features that might change the game state without input from the player. The second definition (game mechanics as the rules binding the game state), we argue, has a practically one-to-one correspondence to the simulation features of a game, where the game state is kept and changed according to how the simulation is supposed to work (see Figure 2.9 for an example of how these definitions apply).

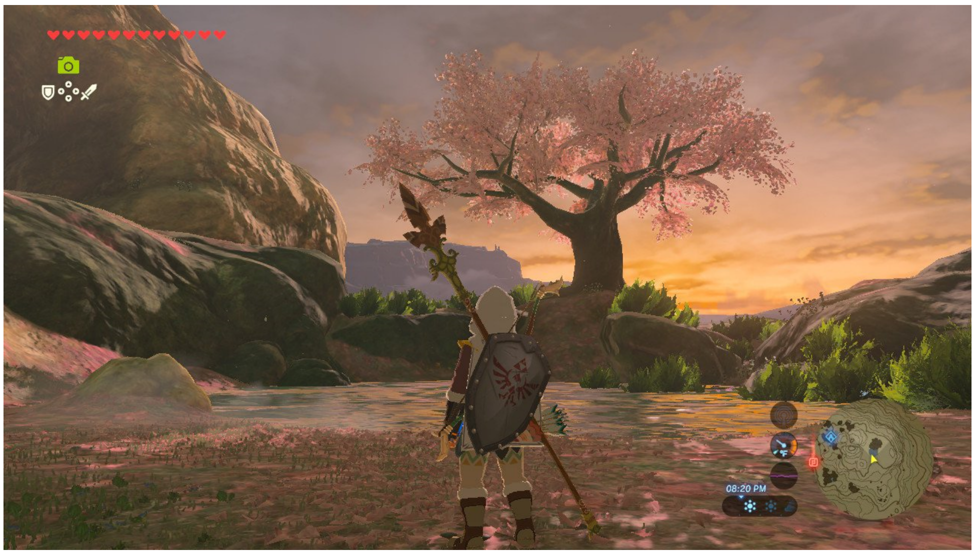


Figure 2.9: Screen capture from *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017). In this game, some creatures only appear during the day and others, only during the night. Depending on the definition of game mechanics you use, these behaviors may not be considered game mechanics because they lack direct user interaction. In the definition we chose, they are, since they comprise rules for how to simulate the game world.

From the definitions above, we concluded that the second one has a more practical use for software architecture research. It aligns our terminology, allowing us to discuss game mechanics as the simulation features that implement the rules that control the virtual world of a game. As such, we propose the following definition for use in the remainder of this text:

Game mechanics comprise:

- the set of valid states of the virtual world simulated by a game,
- the set of possible initial and end states for the game, and
- the rules that dictate changes between states.

Since this definition is based on the studies from the field of game design, it inherits some convenient benefits defended by its authors. Hunicke *et al.* (2004) argue that designers cannot *directly* manipulate the experience players have when interacting with a game and that they can only really manipulate its mechanics[15]. That said, by understanding the relation between mechanics and the produced experience, designers can make informed decisions on what works best for their games. Similiary, Schell (2020, pages 51–59) proposes the *elemental tetrad*, a concept that divides games into four major elements: aesthetics, mechanics, story, and technology. In this view, as software architects, we contribute to the technology of a game but each of the elements "powerfully influences each of the others" (Schell, 2020, page 55). This way we know that the "features" our software must implement support one or more of aesthetics, mechanics, and story. By mapping mechanics to simulation rules, we have a much clearer picture of what parts of the game system are responsible for the mechanics.

There are, nevertheless, many types of game mechanics and, again, we must borrow from the field of game design before adapting to the concerns of software architecture. Adams and Dormans (2012, pages 6–8) propose five different types of game mechanics: physics, economy, progression, tactical maneuvering, and social interaction. Schell (2020, chapter 12), on the other hand, proposes seven: space, time, objects, actions, rules, skill, and chance. When we try to look at these categories as programmers, we see that not all of them will be a part of the code. Some of them are *emergent behavior* — or *dynamics* in Hunicke *et al.*'s (2004) terminology — which means they "exist" on a strategical or phenomenological level. For instance, prioritizing your queen in a game of chess is not something you would program into the code, but is an active part of the game design and players are aware of it. This "mechanic" (outside the definition we adopted) would likely fall into Adams and Dormans' "tactical maneuvering" or Schell's "action" categories.

By filtering these categories by what is possible to implement in code and then removing any redundancies, we find a more useful categorization of mechanics when it comes to designing the architecture of a game system. Since our main concern is managing software evolution costs, that is the criterion we use for setting different mechanics apart from each other. In this regard, we believe that Adams and Dormans' categories are easier to adapt since some of Schell's categories are too broad (e.g., "objects" and "actions"). Thus, in our research, we found that **the main categories of game mechanics we must consider are physics, progression, and economy**, while tactical maneuvering and social interaction are either emergent behaviors or

[15]In this case, Hunicke *et al.* are likely disconsidering design activities such as interface design and character design, which is one of the limitations of their work.

are already covered by one of the three groups we proposed.

Physics mechanics determine how the positions and shapes of entities interact with the space around them inside a game simulation, including other entities. These interactions are not necessarily realistic. In a game of tic-tac-toe, physics dictates there are only nine possible spaces to occupy and only a single mark (cross or circle) may exist in a given space at any time, for instance. This category covers Schell's "space" mechanics mainly, but all of his other categories intersect with physics one way or another. In terms of software architecture evolution, physics mechanics can be very expensive but their requirements are usually not likely to change abruptly. This makes particular implementations very reusable, even across different games, since many of them aim to simulate more or less realistic or convincing physics, enabling the use of dedicated third-party libraries. In other words, unless a game is specifically trying to implement unusual physics, the software architecture of physics mechanics, either third-party or in-house, might be expensive but is relatively stable once developed and validated.

Progression mechanics determine what are and how the player gets to the end states of the game simulation. That includes win and loss conditions but also the overall structure of gameplay when it comes to bringing the player closer to those conditions. In action-adventure games, for example, it is common to divide the game into stages the player must overcome in sequence, each with its theme and an overall difficulty curve. This stage structure is part of the progression mechanics. Some games have nonlinear progression by offering a finite, predetermined set of paths for the players to follow. Sometimes, games rely on other types of mechanics to give the player more freedom despite this predetermined progression, by establishing goals and leaving them to navigate the simulated virtual world in search of a way forward. For instance, in *Factorio* (Wube Software, 2016–2021), the main objective is to spend resources to advance through a research tree (nonlinear progression) but obtaining those resources depends on the factory simulation mechanics of the game, which involves both physics and economy.

The predetermined nature of progression requires manually establishing when advancement occurs and what branches exist at the player's disposal. The main architectural challenge of that is providing a streamlined method of adding new sequencing or branching points into the game. Since this kind of progression amounts to providing the player with an explicitly discrete number of choices, it is often possible to use data-driven design to feed these choices as data to the game. For instance, in a visual novel engine, it might be enough to allow story writers to provide a list of choices after a certain dialogue, each pointing to a dialogue that continues the corresponding branch of the narrative. If the rules for determining what choices are available and how they affect progression are more dynamic, some form of scripting support might be necessary, like checking if the player has spoken with this or that character before the present choice.

Economy mechanics are the rules that simulate quantifiable entities and aspects of entities of the virtual world, as opposed to the more vectorial and geometric nature of physics mechanics. The most obvious example is simulating how much virtual money a player has and the transactions they can do with it. However, economy applies to much more than just money: army troops, material stocks, combat statistics, cards and dice, crafting recipes, and even character emotional states are but a few examples of possible economy mechanics. One of the main characteristics of economy mechanics is that they can simulate *any* source system, while physics mechanics across games normally represent the same source system (real-world physics) one way or another[16]. Designers can even use economy mechanics to represent physics without using physics mechanics. For instance, in *Magic: the Gathering* (Wizards of the Coast, 1993), a card game, players use creature cards to attack their opponents but may be blocked by their opponents' creatures instead. However, creatures with a special trait called "flying" cannot be blocked except by other creatures with the same trait. This simple exception rule simulates the advantage that flying entities have over ground-bound ones. At the same time, this is not part of physics mechanics, so we consider them economy mechanics — after all, creatures in *Magic: the Gathering* are a resource, and the different traits they carry are what determines their "value" in combat.

Because of this, economy mechanics require widely different implementations in each game, reducing the opportunity for reuse. That is, the lack of a fixed source system to simulate makes their requirements change completely from case to case. This is one of the main issues why we decided to focus on this particular type of mechanics when addressing software architecture in games. There is, however, a subset of economy-centered games that are particularly hard to design software for.

Peter Suber (1982) invented a pen-and-paper game called *Nomic* where "changing the rules is a move". The game had a set of rules describing how players can change, through a voting system, the very same rules while they play, with the initial objective of rolling dice to accumulate a certain amount of points (which is usually the first rule to suffer an amendment). We consider most of the mechanics of *Nomic* to be economy mechanics, mainly through a process of exclusion but also because of how "*Nomic*-like" mechanics appear in other games. Though it would be probably impossible to completely implement *Nomic* as software, many digital games bear **self-amending mechanics** like *Nomic*. Even with a more limited form of self-amendment, such games still pose the challenge of implementing a simulation with rules that are dynamically changed by other rules and events inside that simulation. The clearest examples of games like these are collecting card games such as *Magic: the Gathering* (Wizards of the Coast, 1993), *Hearthstone* (Blizzard Entertainment, 2014), *Faeria* (Abrakam Entertainment S.A., 2016–2020), and *Shadowverse* (Cygames, 2016). In these games, the general rules are constantly overridden by specific rules in the cards. This is often known as the "golden rule". Cards may change win and loss conditions, skip or repeat turns, and even change the rules in other cards (see, for instance, Figure 2.10). Designing an architecture that allows the creative team to not only come up with an endless variety of entertaining card ideas but to do so within the time and budget constraint of the project, is one of the many things we hope to achieve through the *Unlimited Rulebook*.

There are other types of games with self-amending economy mechanics. In general, genres[17] that are closely related to board games, like 4X games[18], are more likely to have rich economies and rule-bending elements. An example is *Sid Meyer's Civilization V* (Firaxis Games, 2010).

[16]In this sense, one way to phrase our definition of economy mechanics is "anything that is not either physics or progression". This suggests that, in terms of types of source systems being simulated, physics and progression are simply particular types of mechanics and the remaining possible mechanics are simply too diverse and unfocused to group into further categories. This is an interesting discussion but out of scope in this research, where we are specifically concerned with what types of mechanics are most challenging to support in the creative process of games through the application of software architecture knowledge.

[17]It is important to note that game genres are subjective and, thus, have no clear definitions. We use them here merely for illustrative purposes.

[18]4X stands for *eXplore, eXpand, eXploit, eXterminate*, a subgenre of strategy games.

Although with a now distant origin in table-top role-playing games, digital role-playing games are another common source of self-amending economy mechanics. A very illustrative example is *Path of Exile* (Grinding Gear Games, 2013–2021), with not only a shockingly complex combat economy but also four major updates releasing every year, always introducing *even more* rule-changing mechanics. Inside the role-playing genre, *rogue-likes* and *rogue-lites* are a particularly reliable source of self-amending economy mechanics, since they usually offer a variety of game entity types greater than typical games and favor complex interactions like in the cockatrice example from Chapter 1.

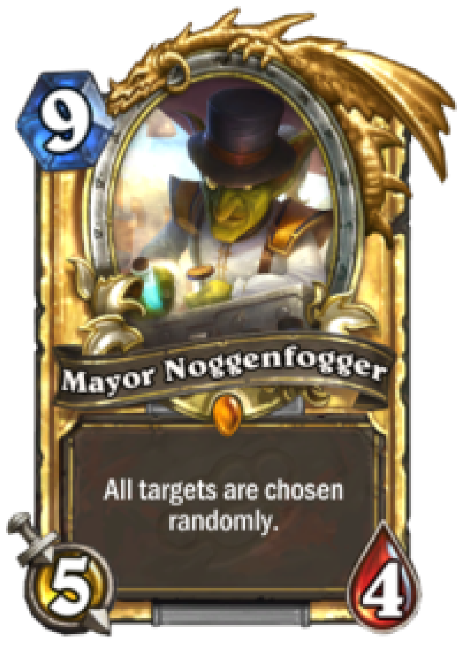


Figure 2.10: The *Mayor Noggenfogger* card from the game *Hearthstone* (Blizzard Entertainment, 2014). While this card is in play, the rules for selecting targets, which is one of the most basic interactions in the game, are replaced by a randomizing effect. When the designers came up with the idea of this card, how much code did they have to change to allow such a feature?

To summarize, game mechanics are the rules behind the simulation of a game and can be divided into physics, progression, and economy mechanics. Among these three types, progression is predetermined while physics and economy represent more complex systems through the simulation. Between physics and economy, physics simulates a single source system while economy simulates any source system. Physics is vectorial, geometric, and requires the notion of space, while economy is usually scalar. Economy has the particular case of self-amending mechanics, which change simulation rules at runtime. This research is about the role of software architecture in the creative process of games when they have economy mechanics in constant development, especially where the subset of self-amending mechanics is involved.

2.2 Related Work

The problem of addressing the creative process of games through the discipline of software architecture (and software engineering in general) has many approaches. We chose to design a reference architecture given the particular requirements of our research but there are other reference architectures with distinct yet similar purposes, as well as entirely different approaches, which we very briefly describe here.

2.2.1 Reference Architectures

According to Bass *et al.*, a “reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them”, where the “reference model is a division of functionality together with data flow between the pieces” (Bass *et al.*, 2003). By this definition, a reference architecture is more or less like a function from the space of possible reference models to the space of the effective architectures a system can have. Nakagawa *et al.*, after reviewing several different definitions, conclude that reference architectures “encompasses the knowledge about how to design concrete architectures of systems of a given application domain” (Nakagawa *et al.*, 2011). They add that “it must address the business rules, architectural styles, [. . .] best practices of software development, [. . .] and the software elements that support development of systems for that domain”. This definition details some of the main types of knowledge involved in a reference architecture, giving us clear artifacts used to turn a reference model into the architecture of a digital game.

I believe it is safe to assume Gregory’s book (2019) very much defines a reference architecture for games. He does not claim so and clarifies that the architecture he describes is strongly based on the one developed at *Naughty Dog* (where he works as the lead programmer), which focuses on high-budget, realistic 3D action-adventure first- and third-person games. Since the book formalizes the author’s knowledge and experience into an abstract architecture that captures the similarities between many game implementations, it can be used as a reference architecture and is probably the most comprehensive one available in the literature.

In his thesis, Plummer proposes a general-purpose, flexible and extensible reference architecture, though he does not call it so (Plummer, 2004). His research intends to “design at a higher level of abstraction than the design of game engine” and a “distinction should be made between an architecture and a fully fleshed out system design” (Plummer, 2004, page 6), which makes us believe that his work is, indeed, an actual reference architecture. His design used a *Data-Centered System of Systems* architectural style, where the game state is kept in a central data repository (like in the *Blackboard* pattern, see Section 2.2.2 below) while simulation and interaction behavior is delegated to a number of independent systems — which we have been calling subsystems so far. In this architecture, the subsystems are strictly decoupled from each other and the central data repository, so that the game can add new subsystems or replace current ones with new ones. Plummer’s proposal seems to agree with Gregory’s (2019) subsystem-based division of engine architectures and data-centered design defended by West (2018) where the entire game state should be kept in a single structure, but the main difference in both cases is Plummer’s emphasis on decoupling the software parts.

Folmer proposes a reference architecture for games with an interestingly different purpose (Folmer, 2007). His objective was to make a cost-benefit analysis of using *Commercial off the Shelf* (COTS) third-party technologies to develop games and, for that, he proposed a reference architecture “to provide a common vocabulary with which to discuss different game implementations and commonalities between those game implementations”. The architecture itself follows a layered design that resembles a very simplified version of Gregory’s design (2019, page 39).

Pinhanez’s work, on the other hand, proposes a reference architecture with a narrower scope than the ones we discussed so far (Pinhanez, 2000). The *story-character-device* (SCD) was designed by Pinhanez to support story-driven interactive spaces. It serves as an example of how we can design reference architectures for more specific domains, benefitting from the specific requirements to provide a more streamlined workflow for the creative process.

In a similar vein, Sarinho *et al.* (2018) provide what they call a *subdomain game architecture* aimed at multiplatform quiz games. They use a feature-based approach and focus on software variability, with their proposal being based on the *Model-View-Controller* pattern. The authors make the very interesting claim that "the idea of an one-size-fits-all game architecture can be misleading, being necessary to built reference game architectures for target (sub)domains". This reinforces that proposing reference architectures for narrower domains in games might satisfy requirements that general-purpose designs cannot.

As we have seen, there are many reference architectures for games in the literature. Some provide solutions that support a broad spectrum of game archetypes, while others focus on more specific genres, formats, and target platforms. Each of them also has different goals in their design: enabling extensibility, supporting more complex narratives, providing a standard to compare available solutions, meeting the requirements of a specific domain, or even just illustrating what the state of the practice is. We can see that our proposal differs in its own way: we focus on games with self-amending economy mechanics because of the challenge they bring to teams with an active and continuous creative process.

2.2.2 Architectural Patterns

Like design patterns, *architectural patterns* (Buschman *et al.*, 1996) are idiomatic solutions to recurrent design problems and goals but with a broader scope inside a software system instead. This does not mean they are mutually exclusive; design patterns like *Composite* (Gamma *et al.*, 1995) can be used as architectural patterns (like in the *Godot* engine). At the same time, while architectural patterns affect most if not entire system architectures, they usually solve one particular requirement in the design, as opposed to how reference architectures describe the entire structure of the system, likely using one or more architectural patterns and design patterns to guide the implementation of its parts. Here we superficially review some architectural patterns that are related to our research because they have the same or similar goals in one or more aspects. In Section 5.3 we revisit them to discuss their actual use in our problem domain.

High-Level Dependency Patterns

Games sometimes have dozens of subsystems and other internal and external modules, all sharing data and control flow of the application. Determining what parts depend on each other as well as how coupled those dependencies are is a crucial part of the architecture design since changes that propagate through dependencies are expensive.

A recurrent pattern we saw in Section 2.2.1 and presented back in Section 2.1.3 was the ***layered systems*** pattern (Buschman *et al.*, 1996; Gregory, 2019). It establishes a hierarchy based on reusability and platform coupling: the more an element is used across the game and the more it is coupled to the platform it runs on, the lower the layer it should be in. This helps determine what components directly affect each other and enables cross-platform support.

A common pattern among interactive applications is the ***Model-View-Controller*** pattern (Buschman *et al.*, 1996; Krasner *et al.*, 1988; Olsson *et al.*, 2015). It separates "view" code (related to graphical user interface elements) from "model" code (related to domain-specific logic), while both are mediated by a higher-level component called the "controller". Depending on how the model and the view are decoupled, you can change one without affecting the other. The process for adding and revising features becomes more structured as well — you have a clearer picture, based on the nature of the feature, what parts of the system it should be implemented on. As discussed in Section 2.1.3, games sometimes have relatively coupled requirements for graphical and game-specific elements, which sometimes makes it harder to apply the *Model-View-Controller* pattern. Nevertheless, separating the model from the view *where possible* is still beneficial to the system design (Olsson *et al.*, 2015).

A design pattern related to the *Model-View-Controller* in games is the *State* pattern we discussed in Section 2.1.3. It provides a way of "switching" controllers, making it easier to separate user interactions according to context. The combined use of these patterns is important in determining control flow and entry points to game features.

So far we saw patterns that design solutions for how different game subsystems and components depend on each other and how control flow passes through them. However, they do not specify how *data* is shared among these software elements. Some developers argue that, given the complexity and unpredictability of game features, it is often pointless to encapsulate data (West, 2018). Instead, they propose the use of *data-oriented design* or *data-centered design*[19], a general approach where runtime game data is centralized and passed along (in full or in parts) or globally shared across all subsystems. In turn, subsystems and many other components that implement features become stateless, "communicating" through the shared global state. The *Blackboard* pattern (Buschman *et al.*, 1996, pages 71–95) is one possible implementation design, as in Plummer's (2004) reference architecture (presented back in Section 2.2.1).

Object Model Patterns

As discussed in Section 2.1.1, the runtime state of the simulation inside a game is usually divided into smaller parts, i.e., the game entities. This division makes it easier to add behavior to the simulation because you do not have to consider the entire state all the time. It also aligns with how the creative process produces content because designers can think in terms of adding and managing small and simple parts of the game instead of understanding it as a single complex system. Game architectures have many ways of designing structures that represent entities. Gregory (2019, page 1043) calls this the *runtime object model* of a game engine and there are architectural patterns known for implementing it.

While simpler games, especially ones with little variation of entity types, can implement their object model with a conventional object-oriented approach, using inheritance for denoting the hierarchy between types, this is often not expressive enough for games. Instead, practitioners have a preference for composition-based approaches (Gregory, 2019; Nystrom, 2014). In a very informal manner, the current "industry standard" in this regard is to use the *Entity-Component-System* pattern, as exemplified by the *Unity3D* engine. This architectural pattern has two parts. First, every entity has very little data and behavior by itself, relying on "component" or "property" objects to give them data and behavior when attached. Second, these components are usually implemented after different domains of the game simulation (e.g., collision, shape, movement, damage, etc.) so that you may process all components of the same type, regardless of the entity

[19]Not to be confused with *data-driven design*, which relates to how content is loaded into the game and not how data inside it is organized and shared.

that owns them, in the same subsystem, improving data and code locality for cache optimization. Not all implementations use this second part, though, thus using only the *Entity-Component* part of the pattern.

A different way of using composition to allow a wide variation of entity types is using the classic *Composite* pattern (Gamma *et al.*, 1995). The *Godot* engine is an example of this, though most engines use this to a certain degree, even *Unity3D*. In this approach, the state of the game simulation is represented by a tree where each node is an entity by itself, allowing nested entities as children of other entities. This allows using inheritance to differentiate node types without limiting the variety of effective entity types obtained through composition. There is an extra benefit because “scene trees” like these align with how most graphics pipelines work in games: the affine transformations that position, rotate, and scale rendered entities are composed according to the parent-child relationships, making it more straightforward to add entities as parts of other entities. For instance, the position of a hat entity is relative to its parent’s, a character entity position. The *Composite* pattern also works well with the *Decorator* pattern (Gamma *et al.*, 1995), allowing even more combinations of entity behaviors.

Both the *Entity-Component-System* and the *Composite* patterns, however, have one shortcoming: defining new components or node types requires writing code, which means the creative team has to interrupt their work when an entity behavior they envisioned is not currently possible in the game. This is not usually a problem because third-party game engines already have hundreds of component and node types available and usually provide a way to easily script new types. However, not all games use third-party engines and, even then, there are games with specific features that the engine cannot foresee.

There is one approach that generalizes object models proposed by Yoder and Johnson called the *Adaptive Object-Model* pattern (Yoder and Johnson, 2002). This pattern allows creating entirely new object types, relationships, and behaviors at runtime and can be applied with considerable variability to fit the particular needs of a system. The pattern is composed of smaller patterns that can be combined to provide the adaptability desired; these patterns include the *Type Object*, the *Property* (which is similar to the idea behind the *Entity-Component-System* pattern), the *Type Square*, the *Strategy*, the *Interpreter* (and accompanying *Composite*), among a few others. The *Adaptive Object-Model* pattern also supports data-driven design as it allows runtime object types to be loaded from persistent data. The main downside to this approach is the cost of initial development and future maintainability since the system becomes considerably complex.

Dynamic Behavior Patterns

The simulation inside games is not only composed of entities and their states (i.e., simulation data) but also the *simulation behaviors* that impose changes in their states. The most straightforward way of adding behavior to a system is writing methods and routines that implement that behavior, maybe using design patterns to solve design issues. However, this would still require a technical intervention upon the creative process of game development. That is why game developers look for ways to allow *dynamic* behavior: behavior that is dynamically loaded into the game simulation at runtime.

The *Adaptive Object-Model* pattern we just discussed includes solutions to this problem. It essentially proposes using an *Interpreter* pattern, with trees implementing each behavior being loaded from a database. In this case, the creative team could even have a DCC tool to create these behaviors. Nystrom (2014, chapter 11) proposes a different approach using a simple *byte-code virtual machine* since it is simpler to create behaviors for in some cases. Of course, if we assume writing simple code is accessible to the creative team, the game could simply embed an actual scripting language to determine simulation behaviors. That is what many games and game engines do[20].

These approaches provide ways to add new simulation behaviors to a game but they do not help add *entry points* for those behaviors. For instance, a combat designer could create a new spell behavior in a role-playing game, but how can they specify that such behavior should only occur when a character enters a certain magical circle? Using code, a simple solution to decoupling the invoker of a routine from the routine itself is using the *Observer* pattern (Gamma *et al.*, 1995; Nystrom, 2014) — this way, connecting a new behavior to existing triggers is a matter of registering an observer. This is how the ***Event-Based System*** architectural style works (Shaw and Garlan, 1996, pages 23–24). The only remaining step to support the creative team is allowing them to connect, *from outside the source code*, the triggering events to the triggered behavior, like the *Godot* engine does with its signal system.

In our research, we deal with games that have self-amending rules and these games have an extra level of complexity in their simulation behavior. The behavior assigned to an in-game event might completely change according to the state of the involved entities. In other words, the simulation *rules* change dynamically because there are *rules for changing rules*. Plotkin proposes *rule-based programming* for games like these (Andrew Plotkin, 2009), which is more akin to a *predicate dispatching* mechanism (Ernst *et al.*, 1998) than an actual *rule-based system*, as the name otherwise suggests.

In this paradigm, every routine in a program has multiple implementations (like in polymorphism) but each is tied to a predicate, not a type. Invoking a routine causes only implementations with valid predicates to be executed and a conflict resolution mechanism is necessary to determine precedence and composition between implementations. Each predicate-implementation pair is called a *rule*. This way, the game simulation can have multiple behaviors assigned to the same in-game event and only the right ones will happen according to the game state. Predicate dispatching is more of a programming language feature, not a pattern, but we consider that emulating it is a recurrent solution to the design problem of having routines that change themselves at runtime.

2.2.3 Software Product Lines

A similar and closely related approach to reference architectures is **software product lines** (SPLs) and their **product line architectures**. SPLs consist of "a set of re-usable assets that includes a base architecture and the common, perhaps tailorable, elements that populate it" (Bass *et al.*, 2003, section 14.1). Once developed and deployed, an SPL can produce new software

[20]See, for instance, a list of games that use `Lua` as an embedded scripting language at en.wikipedia.org/wiki/Category:Lua_(programming_language)-scripted_video_games (last accessed February 19th, 2021).

for its targeted domain at an increased speed by relying on the reuse of highly compatible components thanks to its underlying product line architecture. Nakagawa *et al.* (2011) discuss the differences and relationship between SPLs and reference architectures, concluding that SPLs and product line architectures focus on even narrower domains than reference architectures and are strongly based around the commonalities and variabilities of systems in that domain, while reference architectures are based on knowledge reuse. However, Nakagawa *et al.* propose that product line architectures may use reference architectures as a basis.

The most notable work in this area is Furtado's dissertation (Furtado, 2012). His work proposes an extensive approach to game development where software engineers develop specific SPLs for game (sub)domains using **domain-specific languages** (DSLs), generators, and a product line architecture. His *Domain-Specific Game Development* proposal also includes the use of *Model-Driven Development*, the next subject in this literature review.

2.2.4 Model-Driven Development

Model-Driven Software Development, or simply *Model-Driven Development* (MDD) for the purpose of this thesis, is a methodology that "focuses on the models rather than the code", where these models are formal so that they "can be transformed into the software automatically" and they are "created with a modeling language at a higher abstraction level than the programming language" (Zhu, 2014, page 16). When you compare this definition to Bass *et al.*'s definition of reference architecture as the mapping between a reference model and a concrete architecture, it is easy to see a resemblance. The two main differences are that MDD *literarily transforms a model into a program* and that such a program is a full implementation, not only its architecture. Like with SPLs, however, we believe MDD works best when the software domain is narrower than what we study in our research — economy mechanics in games. That said, the models used in the MDD approach to game development are useful references because part of the *Unlimited Rulebook* is the reference model it operates on.

Zhu (2014), in his doctoral thesis, proposes the *Game World Graph* as a model for classifying game architectures according to how they distribute (or not) the game state, e.g., over a network. This model divides the simulation of games into three separate "worlds": the global world, which contains the definite and complete state; the local world, which contains a limited version of the world as it concerns a particular player; and the perceptible world, that is exactly the world the player perceives through the screen, speakers, etc. This model is interesting because we can draw some relations to our simulation-interaction model and, in particular, the MVC pattern. Besides, with self-amending economy mechanics, what is perceivable or not is often an important part of the rules, like in card games where the perceptible world of players is limited to the cards on their hands and those lying face-up on the table.

There are other MDD studies of interest. Llansó *et al.* propose using a variation of MDD where the model is used to validate databases of game entities that use the *Entity-Component-System* pattern (Llansó *et al.*, 2011). This retains the expressivity of using composition over inheritance, but benefits from the robustness of the formal constraints of inheritance and other ontological relations. Sarinho and Apolinário propose a more traditional, general-purpose MDD approach using generative programming called the *GameSystem, DecisionSupport, SceneView* (GDS), and the language their model is written in, *Game Specification Language* (GSL) (Sarinho *et al.*, 2018). Williams *et al.* combine MDD with evolutionary algorithms to automate the production of game characters in a fighting game (Williams *et al.*, 2011).

2.2.5 Frameworks and Game Engines

Despite the many approaches discussed in this section, the ideal solution that maximizes reuse so that there is no need for technical intervention in the creative process of games is to have all the technology already implemented before making the game. Although the development conditions are rarely that convenient, that is the purpose behind using third-party game engines or frameworks.

In this sense, the most popular game engines are general-purpose frameworks with a *what-you-see-is-what-you-get* (WYSIWYG) world editor where game creators place entities and assign them behaviors, preferably without having to write code, though scripting and visual programming is widely accepted. That is, these engines make heavy use of data-driven design, as Gregory suggests (2019, page 12). We have already mentioned *Unity3D* and *Godot* as engines in this category, but other equally important references are *Unreal Engine*[21], *Cryengine*[22], *Source Engine*[23], *Defold*[24], among many others. These general-purpose engines help with the most common features games need (e.g., graphics rendering, input handling, physics, etc.) but still leave a lot for developers to fill in exactly because the engine cannot make too many assumptions about the developed applications. The asset store and active community help mitigate that.

There are also data-driven engines with WYSIWYG world editors that focus on specific game genres and formats. A notable example is the *RPGMaker* series of engines, where creators have a completely streamlined workflow for making role-playing games that resemble classics from the *Dragon Quest* (Square Enix, 1986–2020) and *Final Fantasy* (Square Enix, 1987–2020) series. Since the game simulation rules follow a relatively consistent standard, *RPGMaker* requires practically no programming unless the game deviates from the norm. This is particularly interesting for our research because role-playing games are economy-centered games, sometimes with self-amending mechanics. However, our goal is not limited to role-playing games, so solutions like *RPGMaker* are not general enough for our purpose.

Another game engine that is also focused on a format that allows it to provide a leaner workflow is *Ren'Py*[25], a visual novel engine, though it does not have (and would not make much sense to have) a world editor. The last notable example of a game engine in this style is *Inform7*[26], an interactive fiction engine for text-adventure games mainly. It uses rule-based programming (Andrew Plotkin, 2009) through its English-based language, which makes it the engine with the greatest support for self-amending mechanics we found so far.

There are many other game engines. The ones we discussed here are those we found most relevant. That could be because of their popularity, well-defined workflows, or specific support

[21] unrealengine.com (last accessed February 23rd, 2021)
[22] cryengine.com, last accessed February 23rd, 2021.
[23] developer.valvesoftware.com (last accessed February 23rd, 2021)
[24] defold.com (last accessed February 23rd, 2021)
[25] renpy.org (last accessed February 23rd, 2021)
[26] inform7.com (last accessed February 23rd, 2021)

for self-amending economy mechanics one way or another. Their design, features, and implementation (when publicly available) are used as references throughout this research.

This chapter reviewed all the literature concepts we consider essential to discussing our research, as well as similar approaches to similar problems. We highlighted the gaps in game development we want to fill with the *Unlimited Rulebook* reference architecture. Next, before we can finally explain the architecture itself, we need to establish the methodology we used to analyze, design, and evaluate our work, so we can systematically tackle our research problem.

Chapter 3

Methodology

This chapter describes the methodology we used to produce the *Unlimited Rulebook* architecture. We start by discussing and establishing our research questions in Section 3.1. Next, in Section 3.2, we present the process we followed to design, represent, and evaluate our reference architecture. As a part of this process, we list and detail in Section 3.3 the many information sources we relied on during our research and the development of the *Unlimited Rulebook* architecture.

3.1 Research Questions

As explained in Section 1.2, our research goal is to understand and provide software architecture approaches that support the creative process of developing digital games. Our focus is on games where economy mechanics are in constant production, making software change an integral part of development. Based on the objectives listed in Section 1.2.1, we designed the following research questions:

RQ1 How does the creative process of economy mechanics in games translate into system requirements?

RQ2 How do different software designs address the requirements of implementing economy mechanics in games?

RQ3 What part of developing new economy mechanics produces the most expensive software changes?

RQ4 How can architects design game systems that minimize the cost of changes in producing new economy mechanics?

RQ1 and RQ2 determine that our research will present the state of the art and state of the practice of how software architects address the creative process of games with constant production of economy mechanics. By making clear the requirements and what designs meet which requirements, we ensure an objective guideline for our work. However, only these two questions are not enough to discuss all the challenges in this subject because game systems are developed as part of a subjective, multidisciplinary process. As different perspectives evaluate the game at different points during production, the requirements change over time. The problem with that is that changing a software architecture after it is implemented can be expensive. RQ3 and RQ4 address this issue by extending our research to understand what causes the need for architectural change so that we may avoid or mitigate the associated costs. The answer to each of these questions will be a part of the body of knowledge that constitutes the *Unlimited Rulebook* reference architecture. To do this systematically, we follow a state-of-the-art research process that has been documented, tested, and published in the software architecture literature, described next.

3.2 The ProSA-RA Process

Nakagawa *et al.* (2014) propose “a process that systematizes the design, representation and evaluation of reference architectures” called ProSA-RA. It uses RAModel, a reference model for reference architectures (Nakagawa *et al.*, 2012), as a guideline, which we briefly explain in Section 3.2.1. ProSA-RA divides the process of producing a reference architecture into four steps: information source investigation, architectural analysis, architectural synthesis, and architectural evaluation. These steps are detailed in Section 3.2.2. However, for our research, we adapted ProSA-RA into a cyclic, iterative process. We describe how we did this in Section 3.2.3. Then, in Section 3.2.4, we explain th evaluation method of each cycle.

3.2.1 The RAModel

The RAModel (Nakagawa *et al.*, 2012) determines all the elements that compose a reference architecture as well as the relationships between them. When designing a reference architecture according to the RAModel, all these elements should be accounted for (which we will do in Chapter 5). The elements are organized into four groups: domain, application, infrastructure, and crosscutting elements. Figure 3.1 summarizes the overall structure of the RAModel. For details on the elements inside each group, we recommend reading the original study from Nakagawa *et al.* (2012). Here we provide a summarized explanation of each group as a whole.

Domain Elements

This group contains all elements that regard the *space of human action* of the domain covered by the reference architecture. It covers the **practical context** on which systems derived from the reference architecture operate. In our research, that means we must identify the requirements and expectations that the field of game development imposes on the systems developed through the *Unlimited Rulebook* reference architecture.

Application Elements

The group of application elements comprises the **technical specifications** of the reference architecture, from its goals and limitations to the kinds of data processing and functional requirements it meets. For instance, games usually have to provide pixel data for rendering a full screen 60 times per second while keeping up with the simulation of their virtual world.

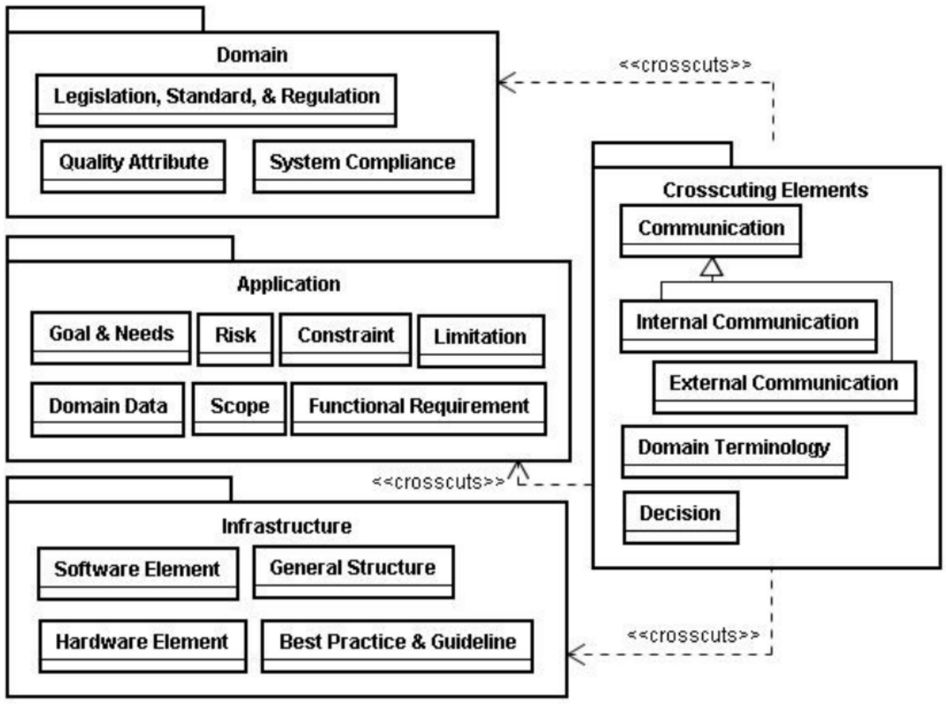


Figure 3.1: A diagram by Nakagawa *et al.* (2012) that illustrates the elements, groups, and relationships of the RAModel.

Infrastructure Elements

This group contains the **available tools** that architects use from the reference architecture to design their game systems. This includes the technologies at their disposal but also any guidelines and design knowledge that support the development of the game. For instance, game engines, design patterns, and data-driven design are elements developers use to build their games.

Crosscutting Elements

Elements of the crosscutting group are interwoven into the other groups and serve as a reference for all parts of a reference architecture. They are **interconnecting concepts** that provide basic assumptions over how the other elements interact with each other. This group includes elements such as core design decisions, domain terminology, and means of communication between different parts of the domain.

3.2.2 ProSA-RA Steps

As part of its systematic approach, ProSA-RA divides the process of producing a reference architecture into the following formal steps. Figure 3.2 summarizes the entire process.

Step RA-1: Information Source Investigation

The first step in ProSA-RA is to identify relevant sources of information regarding the domain of systems supported by the reference architecture. The set of information sources should be as

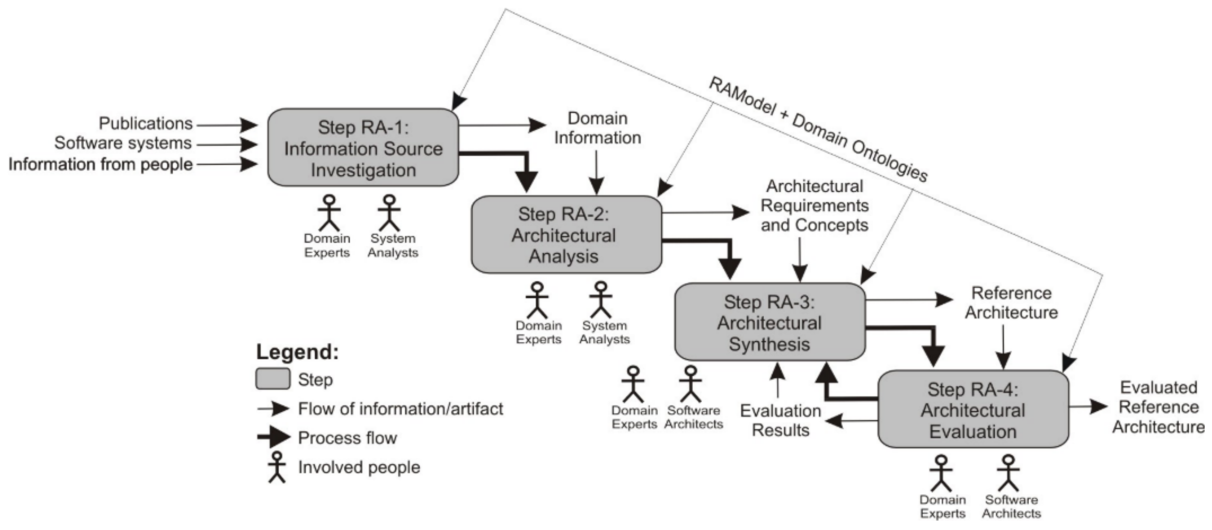

Publications
Software systems
Information from people
Step RA-1: Information Source Investigation
Domain Information
Domain Experts
System Analysts
Step RA-2: Architectural Analysis
Architectural Requirements and Concepts
RAModel + Domain Ontologies
Step RA-3: Architectural Synthesis
Reference Architecture
Software Architects
Evaluation Results
Step RA-4: Architectural Evaluation
Evaluated Reference Architecture
Legend:
Step
Flow of information/artifact
Process flow
Involved people

Figure 3.2: A diagram by Nakagawa *et al.* (2014) that illustrates the outline structure of ProSA-RA.

comprehensive as possible and there are several types of sources: expert knowledge, pre-existing software systems, publications, other reference architectures, among others. In Section 3.3, we present the sources we used for the *Unlimited Rulebook*. The following steps in the process must draw their results from the information sources indicated in this initial step, deriving conclusions from an explicit chain of causal and/or logical links that trace back to the sources.

Step RA-2: Architectural Analysis

The role of this second step is to extract actionable knowledge from the sources found in Step RA-1. This part is further divided into three sub-steps, that go from specific to general. First, **system requirements** identified in the information sources are gathered. These requirements refer to the specific systems and solutions discussed in each specific source. Second, these system requirements are mapped into more general **architectural requirements** that the reference architecture being designed should meet. That means these requirements have a higher abstraction level and broader application. Third, the architectural requirements are grouped into **domain concepts** that better represent that aspect of architectural knowledge. Chapter 4 is responsible for addressing Step RA-2.

Step RA-3: Architectural Synthesis

The third ProSA-RA step essentially describes the reference architecture itself using the RAModel as a general framework. Each of the elements in that reference model should be elaborated on, as well as their relationships. An important part of this step is using diagrams to document the many design perspectives of the reference architecture. This is done in Chapter 5. When describing the architecture elements in this step, they must be explicitly linked to the information source, requirement, or domain concept that justifies them.

Step RA-4: Architectural Evaluation

Lastly, the fourth step in ProSA-RA provides the means for evaluating the designed reference architecture. It mainly relies on a checklist method named *Framework for Evaluation of Reference Architectures* (FERA) (Santos *et al.*, 2013). In our research, we complement this method through the use of quasi-experiments and proofs-of-concept. These evaluations are presented in Chapter 6. In the original ProSA-RA, after this step, we should use the feedback from the evaluation to improve on the reference architecture — i.e., we go back to Step RA-3 as needed. However, we chose to iterate on this process differently.

3.2.3 Iterative Variation

During the development of this project, we had some practical limitations and research opportunities that led us to apply ProSA-RA as a fully cyclic and iterative process. We found that, upon evaluation of intermediate stages of the *Unlimited Rulebook* (Step RA-4), we discovered, were pointed towards, or reconsidered new information sources we had not before. Thus, after each evaluation step, we went back to Step RA-1, increasing the set of information sources, followed by updates in Step RA-2 and Step RA-3. In other words, instead of progressing straight forward and leaving iteration to the last two ProSA-RA steps, we developed a variation of the process where iteration considers all steps. This methodology was described and published in some of our previous work as a novel approach to using ProSA-RA in reference architecture research (Mizutani and Kon, 2019, 2020). We believe that being open to new knowledge input — even on later steps of this systematic process — prevents the design from becoming too rigid and naive. In some cases, as with the interviews with developers in Section 3.3.4, where access to an information source is limited, having prior experience with designing and evaluating the *Unlimited Rulebook* helped us ask more focused questions.

3.2.4 Evaluation Methods

Following the iterative approach we described in Section 3.2.3, our methodology involves evaluating the *Unlimited Rulebook* at the end of each design cycle. Our approach used two different types of empirical studies. Some studies were proofs-of-concept of varying scales where we implement prototypes or games. These proofs-of-concept aimed to intentionally produce scenarios where the *Unlimited Rulebook* would theoretically reduce the cost of implementing economy mechanics the most. This way we could assess how much it supports the creative process in extreme cases, which is one of the goals of this research. The other type of study was a quasi-experiment using computer science students. We designed the quasi-experiment using a *Latin square* approach (Campbell and Stanley, 1963) to evaluate the difference in development effort between using the *Unlimited Rulebook* or relying on the developers' *ad hoc* solutions. This would measure the benefits against the limitations of using our reference architecture. We detail the protocols and results of these evaluation studies in Chapter 6.

3.3 Information Sources

As part of Step RA-1, we list here all the information sources we investigated for the process of designing the *Unlimited Rulebook*. According to Nakagawa *et al.* (2014), the main types of sources are other reference architectures and reference models, publications in the field, software systems of the same domain, expertise from practitioners and researchers, and domain ontologies. However, we did not find any ontology that would benefit our research. On the other hand, since we saw from our early work that the academic literature still has a relatively small body of evidence in our field (Mizutani *et al.*, 2021), we extended our information sources with works from the gray literature since they complement academic research, especially where software developer communities are involved (Wen *et al.*, 2020). Next, we explain how we chose sources of each type and how they contributed to the design of the *Unlimited Rulebook*.

3.3.1 Reference Architectures and Models

From the reference architectures discussed in Section 2.2.1, we chose the architectures of Gregory (2019) and Plummer (2004) as information sources of this category. Both provide general aspects of game architecture that we use as a starting point for designing the more specific case of economy mechanics. Moreover, they complement each other since Gregory is an industry veteran and Plummer comes from an academic background.

Reference models, on the other hand, form "a standard decomposition of a known problem into parts that cooperatively solve the problem" (Bass *et al.*, 2003). Their accumulated knowledge is another source of structured information we can reuse to design the *Unlimited Rulebook*. Following the suggestion from Nakagawa *et al.* (2014), we chose the RAModel as the model used to design the *Unlimited Rulebook* itself. We also need reference models for economy mechanics, the domain problem of our research. To build that model, we found two sources that, despite not being formal reference models, are, nevertheless, complete and time-tested models for game mechanics.

The first source is the Machinations framework and DSL (Dormans, 2012a), specifically proposed as a means of modeling and simulating economy mechanics. It emphasizes the flow of in-game resources between different "resource pools". Its original purpose was to help identify and evaluate patterns of emergent behavior in the design of games. Although the framework stems from the field of game design, the author also writes about its use in software engineering practices such as model-driven development (Dormans, 2012b).

The second source to help design our reference model is the comprehensive rules from *Magic: the Gathering* (Wizards of the Coast, 2021). As the first widely successful trading card game, *Magic: the Gathering* has decades' worth of rule expansions and experience with game design. Its rules are extensive, comprehensive, and robust enough to support a wide spectrum of game mechanics — especially self-amending mechanics. This second source also complements the Machinations framework twofold. First, because it comes from industry expertise, whereas Dormans' work comes mainly from his Ph.D. research. Second, because it models how economy "qualitative actions" happen inside the game while Machinations focuses on the "quantitative actions" that distribute resources. There is a more in-depth discussion of this in Chapter 4.

3.3.2 Publications

Publications include works published by experts in the field of software and game development that provide information about the requirements and practices involved in the architecture of economy mechanics in games. We gathered publications of varied origins: peer-revied studies, theses, books, web articles, and conference talks. We also used publications from both the game development domain and software development in general, as long as they provided insight into how to improve game development according to our goals.

We chose sources using two separate approaches. The first approach was a **systematic literature review** (SLR) (Kitchenham and Charters, 2007) that we designed, performed, and published (Mizutani *et al.*, 2021). SLRs provide a systematic method of answering research questions based on the state of the art. In our case, we wanted to assess the relationship between software architecture and game mechanics from the perspective of the academic literature. The study contains the full details involving the methodology, the selected studies, and the resulting analysis. We chose to begin our research by considering game mechanics in general, instead of only economy mechanics, because there are very few studies in this field and this approach allowed us to understand how different types of mechanics interact with software architecture before specializing in one of them. We found 36 studies that matched our research questions to varying degrees. Each study was assigned a fitness score from 0 to 5 measuring how much it contributed to answering the research questions. We selected all that had a score of 3 or more to use as information sources in this thesis. Table 3.1 lists the resulting set of studies, all of which discuss practices and technologies for developing games and their mechanics.

Study	**Subject**
Llansó *et al.* (2011)	Using ontologies to validate ECS games
Tutzschke and Zukunft (2009)	Framework for pervasive games
Gestwicki (2012)	Design patterns in game development
De Freitas *et al.* (2012)	ECS engine for 2D games
Mottola *et al.* (2006)	Using data-sharing middleware in pervasive games
Wu *et al.* (2010)	Using Android for games and education
Wang and Nordmark (2015)	Software architecture and the creative process in games
George *et al.* (2013)	Modding framework applied to education
Patel *et al.* (2004)	Architecture for card games over table-top surfaces
Mössenböck (2000)	The *Twin* design pattern
Papaioannou (2005)	System design for large VR applications
Pinhanez (2000)	Architecture for story-driven interactive spaces
Valentin *et al.* (2012)	Architecture for evacuation simulations
Sanneblad and Holmquist (2003)	Platform for making interactive networked mobile games
Maggiore *et al.* (2012)	Language for general-purpose game development
Sarinho *et al.* (2018)	Feature-based approach for multiplatform quiz games

Table 3.1: Studies from our systematic literature review (Mizutani *et al.*, 2021) that had a fitness score of 3 or more, chosen as information sources for the *Unlimited Rulebook*. They are sorted according to the order they were coded in in the original review.

The second part of our information sources from publications includes all **works we found throughout our research project** that provided useful, actionable knowledge into our problem domain. This was a more subjective and far from exhaustive process, so we cannot be certain if there are even more sources in the literature we missed. This set of publications, however, has proven useful while also covering a wide spectrum of aspects in the dynamic between the creative process of games and economy mechanics. The reason they were not found in the SLR is either because they are not necessarily peer-reviewed or because of the lack of consistent terminology in this field as we point out in the SLR itself (Mizutani *et al.*, 2021). Since the volume of publications gathered in this part is larger and the domains involved were more varied, **we divided these sources into three groups**: system requirements in game development, software engineering practices in general, and software engineering practices in game development.

The first group — publications about **system requirements in game development** — has 8 publications, listed in Table 3.2. Murphy-Hill *et al.* (2014) wrote a survey that analyzes the differences between game development and software development in general. Pascarella *et al.* (2018) made a follow-up paper that analyzes the same difference but considering specifically open-source systems. The works of Kasurinen *et al.* (2017) and Politowski *et al.* (2021) collect and analyse development problems in the game industry. Schell (2020), a recurring reference in this thesis, explains at length the process behind designing games. As such, it provides an extensive source of information about what the creative process of games needs. For similar reasons, we include Hunicke *et al.* (2004)'s work on the *Mechanics-Dynamics-Aesthetics* (MDA) framework, which formalizes the relationship between the precise requirements games have in the form of mechanics and the non-functional requirements game designers aim for in the players' experience. Finally, the books by Rollings and Ernest (2006) and Adams and Dormans (2012), which discuss a variety of mechanics-related topics, give us further insight into the requirements of game development.

Publication	**Type**
Hunicke *et al.* (2004)	Conference paper
Murphy-Hill *et al.* (2014)	Conference paper
Pascarella *et al.* (2018)	Conference paper
Kasurinen *et al.* (2017)	Conference paper
Politowski *et al.* (2021)	Journal paper
Adams and Dormans (2012)	Book
Rollings and Ernest (2006)	Book
Schell (2020)	Book

Table 3.2: Publication sources about requirements in game development.

The second group — publications about **software engineering practices in general** — consists of a selection of works that discuss practices we believe support the development of games with constant production of economy mechanics. Dealing with the costs (both immediate and long-term) of choosing one design over the other in software systems is a long-standing topic of research, debate, and interest in the field of computer science. As we studied this general-purpose knowledge and compared it to game industry wisdom, we found that what developers often designed had already been studied before, and vice-versa. For instance, the recent tendency of practitioners to promote data-oriented or data-centered designs (West, 2018) — where there is a central repository with the entire game state and the subsystems all share access to it —

reminds us of the *Blackboard* architectural style and pattern, discussed in literature classics such as Shaw and Garlan (1996) and Buschman *et al.* (1996), respectively. Even the widely acclaimed *Entity-Component-System* (ECS) pattern could be interpreted as a particular case of the *Adaptive Object-Model* architectural pattern proposed by Yoder and Johnson (2002). Another similar case we will refer to in later chapters is that the rule-based programming proposed by Andrew Plotkin (2009) is, in fact, a form of *predicate dispatching*, originally proposed by Ernst *et al.* (1998). On the other hand, we included a few "gray literature" sources in this list from both renowned authors (Fowler, 2019) and relatively more informal sources (Anonymous authors, 2020; Figg, 2016) as a more subjective yet hands-on perspective from practitioners. Table 3.3 lists all the publication sources in this category.

Publication	**Type**
Ernst *et al.* (1998)	Conference paper
Yoder and Johnson (2002)	Conference paper
Buschman *et al.* (1996)	Book
Gamma *et al.* (1995)	Book
Shaw and Garlan (1996)	Book
Figg (2016)	Web article
Fowler (2019)	Web article
Anonymous authors (2020)	Web article

Table 3.3: Publication sources about software engineering practices in general.

The third group — publications about **software engineering practices in game development** — is the largest of the three. As with the first and second groups, we strived to keep a balance between academic and industry sources as well as between addressed subjects. We chose both academic works like peer-reviewed papers and Ph.D. theses and gray literature sources from practitioners, like web articles and — a common vehicle for sharing expertise in the game industry — conference talks. The most common topics in our selection are data-driven design (Bilas, 2002; Leonard, 1999; Rabin, 2000) and the ECS pattern (Bilas, 2002; Bucklew, 2015; Leonard, 1999; West, 2018, 2007; Wiebusch and Latoschik, 2015) but it also includes MVC (Olsson *et al.*, 2015), *Game Loop* (Zamith *et al.*, 2016), and other patterns (Nystrom, 2014), along with practices such as model-driven development (Zhu, 2014), software product lines (Furtado, 2012), modding (Scacchi, 2011), and the use of ontologies (Wiebusch and Latoschik, 2015). Table 3.4 lists all the publication sources in this category. Of particular interest in dealing with the constant production of economy mechanics and self-amending mechanics, we have:

1. the flexible approach of combining the ECS pattern with event-based systems Bucklew (2015) talks about;

2. the insightful talk on the use of multiplicative gameplay in *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017) by Fujibayashi *et al.* (2017); and

3. the already mentioned proposal of using rule-based programming to deal with the complexity of interactive fiction by Andrew Plotkin (2009).

Publication	Type	Practice
Callele *et al.* (2005)	Conference paper	Requirements engineering
Olsson *et al.* (2015)	Conference paper	MVC
Wiebusch and Latoschik (2015)	Conference paper	ECS with ontologies
Zamith *et al.* (2016)	Conference paper	Game Loop
Scacchi (2011)	Journal paper	Modding
Furtado (2012)	Ph.D. thesis	SPL
Zhu (2014)	Ph.D. thesis	MDD
BinSubaih *et al.* (2007)	Technical report	Reuse and portability
Rabin (2000)	Book chapter	Data-driven design
Nystrom (2014)	Book	Assorted patterns
Leonard (1999)	Web article	Data-driven design and ECS
West (2007)	Web article	ECS
Bilas (2002)	Conference talk	Data-driven design and ECS
Bucklew (2015)	Conference talk	ECS and event systems
Fujibayashi *et al.* (2017)	Conference talk	Multiplicative gameplay
Nystrom (2018)	Conference talk	Assorted patterns
Andrew Plotkin (2009)	Conference talk	Rule-based programming
West (2018)	Conference talk	Data-centered design and ECS

Table 3.4: Publication sources about software engineering practices in game development.

3.3.3 Software Systems

Our sources in the form of **software systems**, like publications, serve two purposes: determining the *requirements* of games with constant production of economy mechanics and the *practices* that help fulfill those requirements. Thus, **this group is also divided into subgroups** — two, in this case: games and development tools. While games exemplify the requirements since they are the end products, development tools provide the solutions that the industry has chosen to rely on, illustrating the preferred practices of the field.

Our **game sources** were chosen to align with the kind of game we want the *Unlimited Rulebook* to support. That is, games that have a focus on the constant production of economy mechanics (ideally including self-amending mechanics). The list we collected was not exhaustive but, instead, focused on achieving an increased variety of genres, styles, and contexts while avoiding too many redundancies. For instance, we tried to balance action games with turn-based games, old with new, offline with online, single-player with multi-player, 2D with 3D, proprietary with open-source, popular with niche. That said, we favored representatives we had more knowledge about or experience with, so we could reference them with more authority. Table 3.5 lists the 24 game systems used as information sources when applying ProSA-RA to *Unlimited Rulebook*. A few of them require some explanations. *Diablo* is originally a proprietary game from *Blizzard Entertainment* but it has been reverse-engineered completely in recent years[1]. *Magic: the Gathering* started as an analog game but has multiple digital adaptations. On the other hand, *Nomic* is an analog game with no digital counterpart but its explicitly self-amending mechanics are the reason we included it in the list.

[1]See github.com/galaxyhaxz/devilution, github.com/diasurgical/devilution, and https://github.com/diasurgical/devilutionx for details (last accesed March 24th, 2021)

The development tools we chose as sources were mainly game engines that support games with constant production of economy mechanics to varying degrees. Again, we tried to ensure variety, including almost mandatory engines such as *Unity3D* but also less known frameworks such as *Bevy*[2] and the *Halley* engine[3]. Table 3.6 lists the development tools chosen as information sources.

Game	**Genres**
BYTEPATH (a327ex, 2018)*	top-down shooter, action
Caves of Qud (Freehold Games, 2015)	rogue-like, turn-based
Cogmind (Grid Sage Games, 2017)	rogue-like, turn-based
Dungeon Crawl: Stone Soup (DCSS Devteam, 2006–2021)*	rogue-like, turn-based
Diablo (Blizzard Entertainment, 1997)*	role-playing, action
Dota 2 (Valve Corporation, 2013)	battle arena, real-time strategy
Dwarf Fortress (Bay 12 Games, 2006)	rogue-like, fortress simulation
Factorio (Wube Software, 2016–2021)	factory simulation
Final Fantasy Tactics Advance (Square Enix, 2003)	role-playing, strategy, turn-based
Guild Wars (ArenaNet, 2005)	role-playing, action
Hearthstone (Blizzard Entertainment, 2014)	card-collecting, turn-based
Loop Hero (Four Quarters, 2021)	role-playing
Magic: the Gathering (Wizards of the Coast, 1993)	card-collecting, turn-based
Minecraft (Mojang Studios, 2011)	sandbox, construction simulation
NetHack (DevTeam, 1987)*	rogue-like, turn-based
Nomic (Peter Suber, 1982)*	self-amending game
Path of Exile (Grinding Gear Games, 2013–2021)	role-playing, action
Pokémon series (Game Freak, 1996–2021)	role-playing, turn-based
Ragnarok Online (Gravity Interactive, 2002)	role-playing, action
Sid Meyer's Civilization V (Firaxis Games, 2010)	4X, turn-based
Terraria (Re-Logic, 2011)	sandbox, action
The Battle for Wesnoth (The Battle for Wesnoth Project, 2003)*	role-playing, strategy, turn-based
The Legend of Zelda: Breath of the Wild (Nintendo, 2017)	open world, action
Veloren (Veloren team and contributors, 2018)*	role-playing, open-world
Warcraft 3 (Blizzard Entertainment, 2002)	real-time stratery

Table 3.5: Game systems used as information sources and their respective references, genres, and licensing. Titles marked with a * are open source. The genres do not follow any formal standards and are here merely to help illustrate what each game is more or less about.

3.3.4 People

For **people**, we gathered sources using two different approaches. The main approach was semi-structured interviews we carried out with professional developers. The second was through informal feedback from USPGameDev, a student special interest group at the University of São Paulo. The group was founded in 2009 and conducts projects that range from game development — with dozens of published titles[4] — to events and courses. Since the author is one of the

[2]bevyengine.org (last accessed March 24th, 2021)
[3]github.com/amzeratul/halley (last accessed June 7th, 2021)
[4]See uspgamedev.itch.io (last accessed March 23rd, 2021)

Tool	Type	About
*Bevy**	game engine	general-purpose, code-only
*Godot**	game engine	general-purpose
Halley	game engine	general-purpose, code-only
Inform7	game engine	for text-based interactive fiction
RPGMaker series	game engine	for role-playing games
*Tiled**	map editor	2D only
Unity3D	game engine	general-purpose
Unreal Engine	game engine	general-purpose

Table 3.6: Game development tools used as information sources and some information about each of them. Titles marked with a * are open source.

co-founders of USPGameDev and currently the oldest active member, this source includes his expertise as well.

Semi-structured interviews are an information gathering method that tries to balance the formality of surveys with the more in-depth nature of free-form interviews. According to Adams (2015):

> Conducted conversationally with one respondent at a time, the [semi-structured interview] SSI employs a blend of closed- and open-ended questions, often accompanied by follow-up why or how questions. The dialogue can meander around the topics on the agenda—rather than adhering slavishly to verbatim questions as in a standardized survey—and may delve into totally unforeseen issues.

Based on a semi-structured interview protocol from other researchers in our research group (Leite *et al.*, 2020), we designed our protocol[5] for carrying out the interviews. The main objective was to assess the state of the practice of software architecture in the economy mechanics of games. We interviewed four active developers from different companies and working on different kinds of games. They were selected based mainly on the available contacts we had but also, as with other information sources, based on how they covered varying aspects of our research field. Table 3.7 summarizes the profile of the projects the interviewers worked on and the data collected from the interviews are in Appendix A. Note that interviewees were anonymized as per our interview protocol.

Code	Game Genres	Platform	Company Size	Development Status
I01	Idle game	Mobile	Less than 100	Released
I02	Action role-playing game	PC, console	Over 10000	Late production
I03	Card game	Mobile	500–100	Released
I04	MMORPG[6]	PC	Less than 100	Early production

Table 3.7: Information about the kinds of games and companies the developers we interviewed worked on. The codes can be used to find the data corresponding to that interview in Appendix A. Company size refers to the number of employees.

[5]https://www.ime.usp.br/~kazuo/thesis/InterviewProtocol.pdf (last accessed Apr 19th, 2021)
[6]Massive Multiplayer Online Role-Playing Game

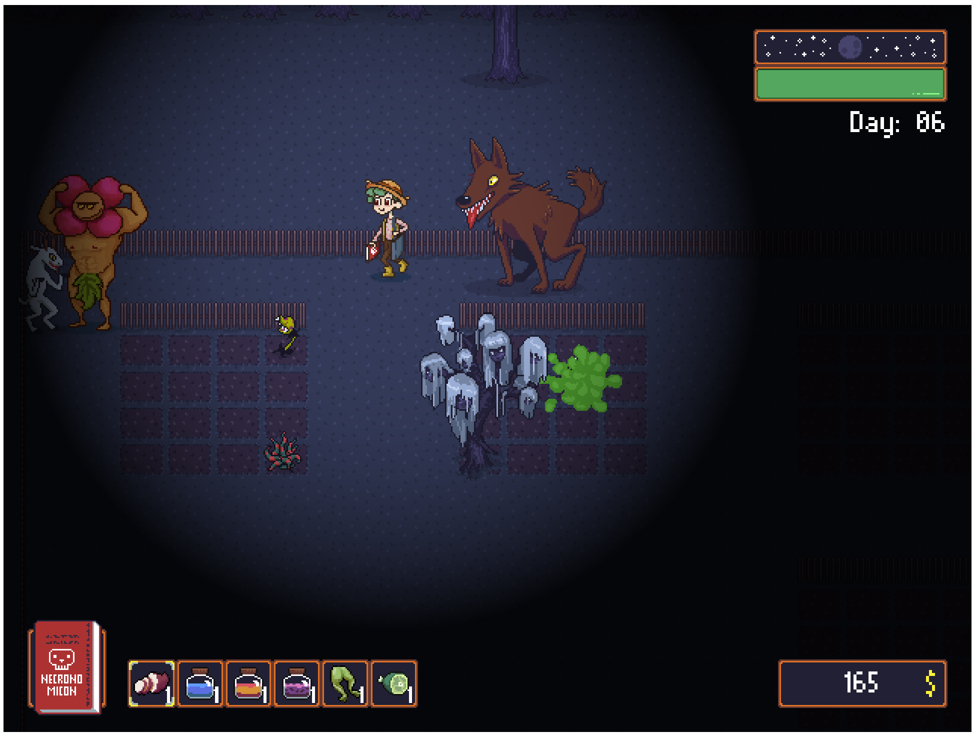


Figure 3.3: Screen capture from *It's All About Lasagna* (USPGameDev, 2018). It was developed during *Ludum Dare 41*, an international remote game jam[7] that happened in 2018. We developed the game in 72 hours under the theme "two incompatible genres".

As an information source, USPGameDev helped our research in two ways. First, by developing more than twenty games over ten years since the group's foundation, the author accumulated his knowledge and expertise in the area as a programmer and software architect. Highlights that required special attention to software design and involved economy mechanics include *Horus Eye* (USPGameDev, 2010), the first game developed by USPGameDev and written entirely from the ground up; *L.A.V.A. series L.A.M.P. edition* (USPGameDev, 2017), a survival shoot'em up where we developed a few dozen enemies and character powers during a 72-hour game jam; *It's All About Lasagna!* (USPGameDev, 2018), a horror farming simulator with a number of different monsters and crafting mechanics (see Figure 3.3). Two other important experiences for our research that we had working on USPGameDev projects are *Backdoor Route* (USPGameDev, 2020) and *Grimoire: Ars Bellica* (USPGameDev, 2021) since both are used in different evaluation steps in Chapter 6, doubling as information sources and validation projects.

The other way in which USPGameDev informs our research is through the informal exchanges the author had with other members throughout his Ph.D. program. USPGameDev has the established practice of meeting every other week to update its members about each other's projects and provide insight. This allows the constant flow of knowledge across the group and our research project benefited from the input of dozens of active members. We do acknowledge that this source lacks the means of providing a written (and more verifiable) record but thought it nevertheless important to include in this chapter.

Chapter 4

Domain Investigation

In this chapter, we analyze the domain of the creative process of developing games that focus on economy mechanics. It corresponds to the second step of ProSA-RA, **Step RA-2: Architectural Analysis**, described in Section 3.2.2. In this step of our research, we gathered system requirements pertinent to our research questions (see Section 3.1) from the information sources listed in Section 3.3. More specifically, we inspected:

1. the reference architectures and models from Section 3.3.1;
2. the publications in Table 3.2;
3. the games in Table 3.5; and
4. the interview data and USPGameDev expertise from Section 3.3.4.

Since remaining sources regard architectural practices instead of requirements, they will be referenced when we explain the *Unlimited Rulebook* architecture in Chapter 5. The process we used to gather the requirements listed in this chapter was the following. First, we consulted each of the information sources from items (1), (2), and (4) above and focused on chapters and sections of their texts that were more clearly related to our research questions. As we read them, we took note of any system requirement the authors explicitly or implicitly stated games had or should have, be them functional or non-functional, that related to economy mechanics, how to implement them, and how to support the creative process behind them. As we did this, we started organizing these requirements into *architectural requirements* and then into *domain concepts* (as explained in Section 3.2.2).

After going through all literature sources, we moved on to the games in item (3). Since it would not be practical to play all games and take note of all their features, we instead did the opposite. We went through the requirements we already had and took note of what games we knew (or believed to the best of our knowledge) met those requirements. This was a rather limited process since most games are closed-source, proprietary titles. The exception was the requirements involving economy mechanics since those are more evident by simply playing the games.

Here we describe the resulting list of 10 domain concepts we identified, their corresponding architectural requirements (33 in total), and the information sources they came from. A more complete and summarized break down of this data is available in Appendix B. We grouped the domain concepts we found into **three major groups: *Mechanics Model*, *Subsystem Integration*, and *Iterative Development***, which are described in Section 4.1, Section 4.2, and Section 4.3, respectively. Whenever we present an architectural requirement, we also include the set of information sources it came from. Figure 4.1 summarizes the relationships of concepts and requirements described in this chapter.

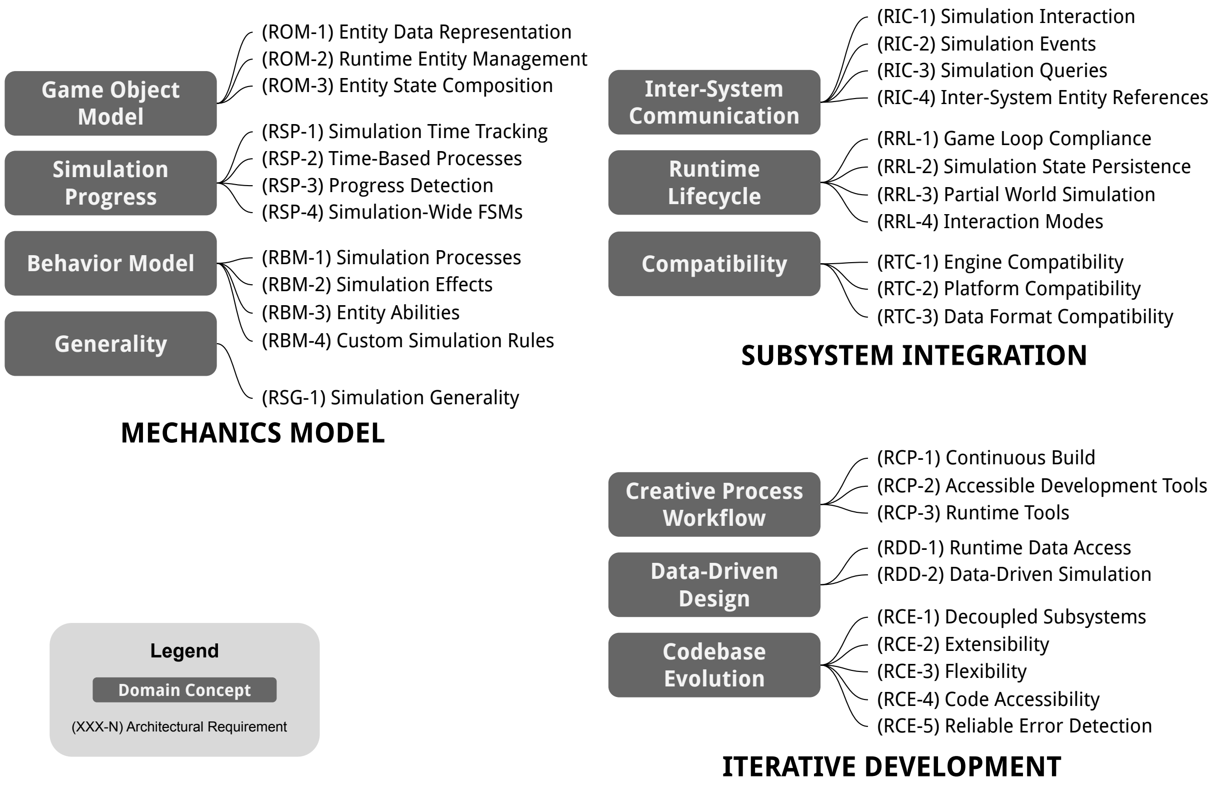

Game Object Model
(ROM-1) Entity Data Representation
(ROM-2) Runtime Entity Management
(ROM-3) Entity State Composition
Simulation Progress
(RSP-1) Simulation Time Tracking
(RSP-2) Time-Based Processes
(RSP-3) Progress Detection
(RSP-4) Simulation-Wide FSMs
Behavior Model
(RBM-1) Simulation Processes
(RBM-2) Simulation Effects
(RBM-3) Entity Abilities
(RBM-4) Custom Simulation Rules
Generality
(RSG-1) Simulation Generality
MECHANICS MODEL
Inter-System Communication
(RIC-1) Simulation Interaction
(RIC-2) Simulation Events
(RIC-3) Simulation Queries
(RIC-4) Inter-System Entity References
Runtime Lifecycle
(RRL-1) Game Loop Compliance
(RRL-2) Simulation State Persistence
(RRL-3) Partial World Simulation
(RRL-4) Interaction Modes
Compatibility
(RTC-1) Engine Compatibility
(RTC-2) Platform Compatibility
(RTC-3) Data Format Compatibility
SUBSYSTEM INTEGRATION
Creative Process Workflow
(RCP-1) Continuous Build
(RCP-2) Accessible Development Tools
(RCP-3) Runtime Tools
Data-Driven Design
(RDD-1) Runtime Data Access
(RDD-2) Data-Driven Simulation
Codebase Evolution
(RCE-1) Decoupled Subsystems
(RCE-2) Extensibility
(RCE-3) Flexibility
(RCE-4) Code Accessibility
(RCE-5) Reliable Error Detection
ITERATIVE DEVELOPMENT
Legend
Domain Concept
(XXX-N) Architectural Requirement

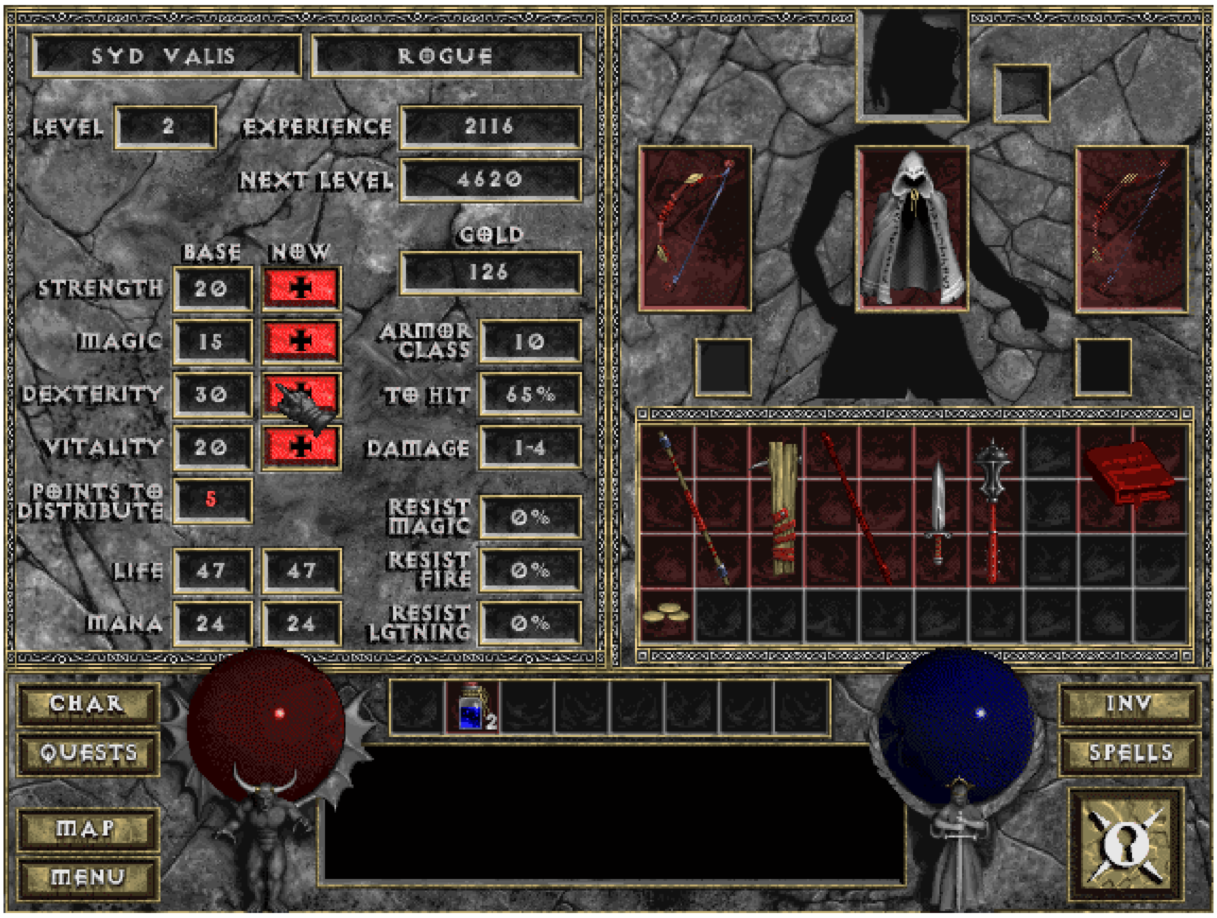
Figure 4.1: Overview of all the domain concepts and the corresponding architectural requirements we found during **Step RA-2: Architectural Analysis** of the ProSA-RA process. Each of these are described throughout Chapter 4.

4.1 Mechanics Model

The first group of domain concepts involves concepts that describe what the economy mechanics of a game need to be able to do, both generally speaking and to support the creative process of games specifically. The architectural requirements of these concepts refer to simulation features the architecture of a game should support. There are four domain concepts in this group: Game Object Model, Simulation Progress, Behavior Model, and Generality.

4.1.1 Game Object Model

As explained in Section 2.1.1, a game simulation carries a runtime state divided into static and dynamic parts, and the dynamic parts are structured into **entities**. The architecture of a game should specify how its entities are implemented and how we can determine what is possible to do with them inside the simulation. Figure 4.2 shows a part of the object model visible to players in *Diablo* (Blizzard Entertainment, 1997). To support the object model, the architecture must meet these three requirements, coded as *Requirements for the Object Model* (ROM-*):


SYD VALIS
ROGUE
LEVEL 2
EXPERIENCE 2116
NEXT LEVEL 4620
GOLD 126

	BASE	NOW
STRENGTH	20	
MAGIC	15	
DEXTERITY	30	
VITALITY	20	
POINTS TO DISTRIBUTE	5	
LIFE	47	47
MANA	24	24

ARMOR CLASS 10
TO HIT 65%
DAMAGE 1-4
RESIST MAGIC 0%
RESIST FIRE 0%
RESIST LGTNING 0%

Figure 4.2: A screen capture of *Diablo* (Blizzard Entertainment, 1997). Here we can see, on the left, a panel with the player's avatar statistics (stats). Each of the numbers shown is a resource that influences the combat economy of the game. To the right, we have the player's inventory and equipment, which are entities aggregated into the player's character that further change their stats and enable different economy mechanics. These different entity types, their combat stats, and the relationships between them from only a part of *Diablo*'s Game Object Model.

(ROM-1) Entity Data Representation. First, the architecture has to consider how entities can be represented as data inside the game simulation, which itself is a larger repository of data. Specifications like data structures, the limit of instances, and basic read and write access are the foundations of this requirement. Additionally, architects need to determine what types of entities exist and how developers can define new types, if possible, as well as how they represent relationships among entities, like aggregation, hierarchy, etc. Lastly, it is common for entities to have a close integration with the physics and graphics subsystems, and this reflects in their data representation (e.g., every entity is tied to a spatial position in one of or both these systems). Information sources: Adams and Dormans (2012); Gregory (2019); Rollings and Ernest (2006); Schell (2020); Wizards of the Coast (2021); and all games from Table 3.5.

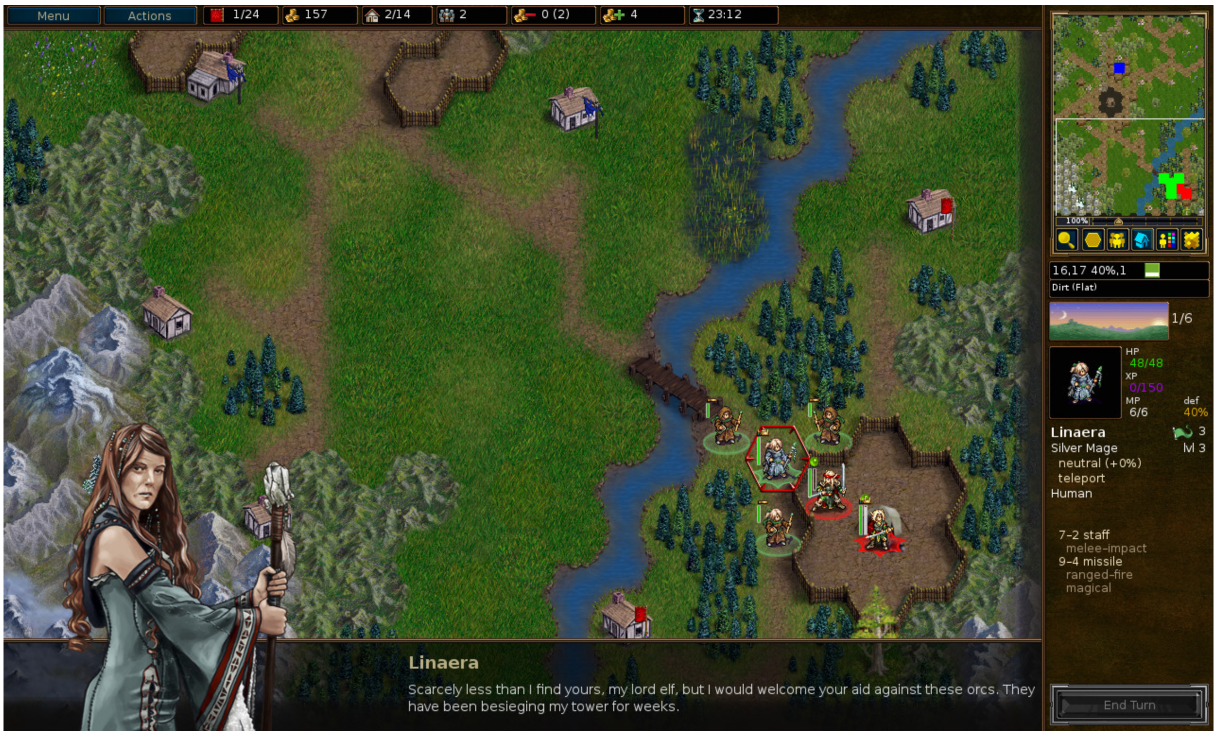
(ROM-2) Runtime Entity Management. A strong characteristic of games as software systems is that objects in memory are created and destroyed frequently and by different parts of the codebase. This process becomes a fundamental part on top of which many others are built. Thus, a dedicated mechanism for creating and destroying entities is required. Furthermore, since entity instances are so ephemeral, it is important to provide consistent ways of querying them (e.g., iterating over all monsters in the game) and referencing them (e.g., an AI storing a reference to the player to follow them over multiple game frames). Stale references are a common challenge in this regard. Information sources: Adams and Dormans (2012); Gregory (2019); Rollings and Ernest (2006); Schell (2020); and all games from Table 3.5.

(ROM-3) Entity State Composition. As we just mentioned, entities are the dynamic part of the simulation: their state change constantly. That is why, beyond their data representation, the architecture should determine what are the possible states an entity can be in, what changes are allowed, and what invariants should be kept. When it comes to economy mechanics, the state of an entity is usually composed of numeric variables, be they floating-point, integer, or symbolic values (e.g., when an entity behaves like a finite state machine). Some parts of the entity state may be procedurally derived from other parts (e.g., the damage output of a warrior is the product of their strength score and their weapon's power score). Some parts of the entity state change only temporarily (e.g., nocturnal creatures have an increased speed score at night), while others change permanently (e.g., when a character increases its experience level in a role-playing game, they become stronger). Sometimes temporary changes are very drastic (e.g., the dragon is transformed into a harmless sheep for 5 seconds). Lastly, aside from numeric states, entities also have capabilities that change over time: the abilities and custom simulation rules they carry (see Section 4.1.3). Information sources: Adams and Dormans (2012); Dormans (2012a); Gregory (2019); Rollings and Ernest (2006); Schell (2020); Wizards of the Coast (2021); and all games from Table 3.5.

4.1.2 Simulation Progress

Game simulations are not just any kind of simulation: they are temporal, real-time simulations. **Tracking the passage of time** within the game world and computing how that affects the simulation state, especially entities, is one of the domain concepts we found in our research. Figure 4.3 shows a turn-based game where simulation time plays an important tactical role. There are four architectural requirements that compose this concept, explained below and coded as *Requirements for Simulation Progress* (RSP-*):

(RSP-1) Simulation time tracking. The most basic requirement in this concept is that part of the simulation state should store and update the passage of time inside the game world. For action games, this is often a matter of storing timestamps and calculating how much virtual time has passed since a given in-game event. For instance, tracking how much time passed since the beginning of a stage in a platform adventure game such as the *Super Mario* series (Nintendo, 1985–2021). For turn-based games, however, time is measured in discrete units that bear additional data, such as whose turn it was, who went before, who is up next, etc. Information sources: Adams and Dormans (2012); Gregory (2019); Rollings and Ernest (2006); Schell (2020); Wizards of the Coast (2021); and all games from Table 3.5.



Linaera

Scarcely less than I find yours, my lord elf, but I would welcome your aid against these orcs. They have been besieging my tower for weeks.

Figure 4.3: A screen capture of *The Battle for Wesnoth* (The Battle for Wesnoth Project, 2003), a turn-based strategy role-playing game. In the middle of the sidebar to the right, we see an image of the rising sun with a "1/6" to its right. That is the day time tracker of the game, which indicates what time of the day that turn corresponds to. Different units react differently to the time of the day. Besides this, *Wesnoth* also has time-based mechanics where each settlement the player controls (the houses with flags in the image) gives them money every turn, which, in turn, they spend to train troops to increase the chance of victory.

(RSP-2) Time-based processes. The fact that games feature temporal simulations means that their simulation state changes over time. In real-time mechanics, this means some simulation routines and entity operations should execute every frame, with some direct or indirect call chain leading up to the *Game Loop*. In turn-based mechanics, the simulation subsystems need to execute these routines at key points every turn. For instance, a creature infected with poison in a role-playing game loses life points at the start of every one of *its* turns. There should be a clear definition of these entry points in the code and how developers can hook behaviors onto them. Information sources: Adams and Dormans (2012); Dormans (2012a); Gregory (2019); Rollings and Ernest (2006); Wizards of the Coast (2021); and all games from Table 3.5.

(RSP-3) Progress Detection. Time often has a hierarchical structure in games. While the player advances through a real-time battlefield, time advances in seconds. When they reach their target destination, the game takes them to the next stage, a new battlefield. This sequence of stages is an example of a higher-level form of progress: the number of stages the player has completed. This pattern might go further, adding more layers of progress to gameplay. Detecting when the current goal (or failure) states are achieved and transitioning the player through these higher-level time structures is also a requirement for the architecture of the mechanics of games. Information sources: Adams and Dormans (2012); Rollings and Ernest (2006); Schell (2020); and all games from Table 3.5.

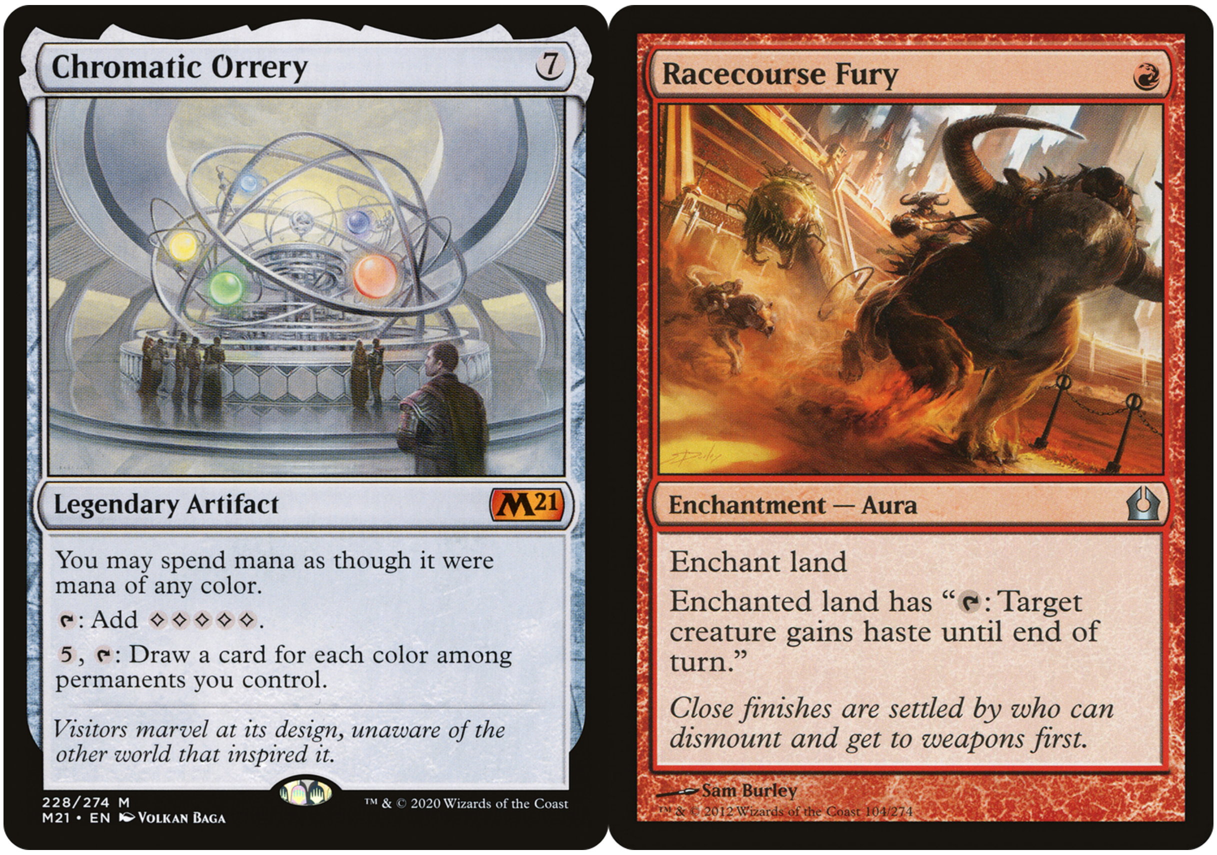
(RSP-4) Simulation-Wide Finite State Machines. Just like entities can behave like finite state machines, so can the simulation as a whole. This is very common in old-school role-playing games, where players are traveling through a world atlas in one moment, then they enter a city and are navigating its streets in another, then later find themselves in turn-based combat, only to return to world-level exploration moments after. Each of these modes of play has its own simulation mechanics and the simulation subsystems must transition between them without losing any contextual data and while keeping different representations of the same entities synchronized with each other (e.g., the player avatar during combat versus their avatar when traveling through the world map). Information sources: Rollings and Ernest (2006); Schell (2020); and all games from Table 3.5.

4.1.3 Behavior Model

The passage of time is not the only kind of state change inside the simulation. Most changes involve things like player action and the interaction between entities, sparking different forms of **simulation behaviors**. These are all the procedures and execution flows that determine how simulation changes happen in a precise and consistent manner. Here we focus especially on changes that involve economy mechanics. Figure 4.4 shows two illustrative examples from the behavior model of *Magic: the Gathering* (Wizards of the Coast, 1993). There are four architectural requirements for this domain concept, coded as *Requirements for the Behavior Model* (RBM-*) and detailed as follows:

(RBM-1) Simulation Processes. Some economy state changes are *continuous*, in the sense that every frame computes a partial step of that change. For instance, buildings in a real-time strategy game might provide a constant income of resources, like “10 wood units per second”. Through coordination with (RSP-2), this requirement states that economy architectures should support changes that are divided into partial steps over any number of game frames. The change itself is usually a simple flow of resources. It is also important to provide the means for starting and ending these economy processes as needed. Information sources: Rollings and Ernest (2006) and all *real-time* games from Table 3.5.

(RBM-2) Simulation Effects. While processes from (RBM-1) are continuous, other kinds of simulation behaviors are *discrete*: they are economy transactions that begin and end in the same game frame, computing an entire sequence of state changes in one go. We call an individual transaction an *effect* and a sequence of transactions a *chain of effects*. Effects are usually atomic: once they start, they will successfully end without any other effect taking place in the meantime because they operate under the assumption that the simulation will not reach a state they could not predict (e.g., due to an outside interference) during their execution. Examples of effects include


Chromatic Orrery 7

Legendary Artifact M21

You may spend mana as though it were mana of any color.

: Add ◇◇◇◇◇.

5, : Draw a card for each color among permanents you control.

Visitors marvel at its design, unaware of the other world that inspired it.

Racecourse Fury

Enchantment — Aura

Enchant land

Enchanted land has ": Target creature gains haste until end of turn."

Close finishes are settled by who can dismount and get to weapons first.

Figure 4.4: Two cards from *Magic: the Gathering* (Wizards of the Coast, 1993). On the left, we have *Chromatic Orrery*, a card that uses custom rules (RBM-4) to overwrite the standard rules of the game. Usually, the resource known as "mana" comes in different colors and cards require specific combinations of those colors to be played. However, *Chromatic Orrery* suspends this rule while it is in play. On the right, we have *Racecourse Fury*, a card that has the behavior of *changing the behavior* of other cards twofold. First, it gives the affected entity an ability (RBM-3). Then, the entity can use that ability to add a custom rule (in this case, the "haste" rule) to a third entity. Both these behavioral changes are temporary and can be considered a form of resource the corresponding entities carry (ROM-3).

a character taking damage, players buying items from a shop, equipping a weapon, etc. Some effects happen due to time-based entry-points as provided by (RSP-2). Others, however, depend on the activation of a higher-level structure. Information sources: Adams and Dormans (2012); Dormans (2012a); Rollings and Ernest (2006); Schell (2020); Wizards of the Coast (2021); and all games from Table 3.5.

(RBM-3) Entity Abilities. In the agent-based simulation of games, entities carry behaviors that trigger a change in the economy state. For instance, mages may be able to conjure a giant rock out of thin air or a pair of magical boots may increase the movement speed of their wearers for a few seconds whenever they speak a designated keyword. We call each of these potential behaviors an *ability*, in the sense that they describe what an entity is *able* to do and because it is the same term used in one of our reference models, *Magic: the Gathering* (Wizards of the Coast, 2021). An ability is a structure that determines under what conditions it may happen and what effects — from (RBM-2) — it causes on the world around their carriers. Abilities may be simply *activated* by a command issued to or by an entity (e.g., the player pressing a button). However, they may also happen because of a *trigger*: a particular event of interest in the simulation (i.e., a specific set of effects) or at certain times — like at the start of a character's turn, detected via (RSP-2) — as long as the simulation state satisfies a given condition (e.g., the player is still stepping on an acid pool). Abilities can be considerably complex, including features such as referring to things a previous ability did, carrying out different chains of effects depending on a given condition, or even bearing a state of its own, accumulating resources every time it is used and consuming them when appropriate. Information sources: Rollings and Ernest (2006); Wizards of the Coast (2021); and the following games:

- *Caves of Qud* (Freehold Games, 2015)
- *Dungeon Crawl: Stone Soup* (DCSS Devteam, 2006–2021)
- *Diablo* (Blizzard Entertainment, 1997)
- *Dota 2* (Valve Corporation, 2013)
- *Final Fantasy Tactics Advance* (Square Enix, 2003)
- *Guild Wars* (ArenaNet, 2005)
- *Hearthstone* (Blizzard Entertainment, 2014)
- *Magic: the Gathering* (Wizards of the Coast, 1993)
- *NetHack* (DevTeam, 1987)
- *Path of Exile* (Grinding Gear Games, 2013–2021)
- *Pokémon series* (Game Freak, 1996–2021)
- *Ragnarok Online* (Gravity Interactive, 2002)
- *Terraria* (Re-Logic, 2011)
- *The Battle for Wesnoth* (The Battle for Wesnoth Project, 2003)
- *Veloren* (Veloren team and contributors, 2018)
- *Warcraft 3* (Blizzard Entertainment, 2002)

(RBM-4) Custom Simulation Rules. Some games that focus on economy mechanics have *self-amending* mechanics: simulation rules that change, at runtime, how other mechanics are supposed to work. In other words, they are *exceptions to the rule*. We call this type of mechanics *custom rules* (as opposed to general, standard rules). In more specific terms, custom rules change how effects are supposed to happen by determining how they are executed, stipulating new effects to be chained after them, preventing effects from happening at all, or describing when a new ability trigger should happen, among a few other special cases. Some custom rules affect the simulation state as a whole (e.g., the basic movements every creature is capable of) while others pertain to specific entities (e.g., undead monsters lose life when they receive healing spells). Some custom rules say that an entity, ability, or effect should be considered as something different from what it really is (e.g., shooting magical beams with this holy sword should be treated as a ranged weapon instead of a melee weapon). Entities may "gain" custom rules just like they gain resources and abilities, causing even more exceptional cases to overrule other custom rules dynamically. When a game uses custom rules, it also needs a way of determining which rules precede which rules, i.e., a *rule adjudication mechanism*. Information sources: Adams and Dormans (2012); Dormans (2012a); Rollings and Ernest (2006); Schell (2020); Wizards of the Coast (2021); and the following games:

- *Caves of Qud* (Freehold Games, 2015)

- *Dota 2* (Valve Corporation, 2013)
- *Dwarf Fortress* (Bay 12 Games, 2006)
- *Guild Wars* (ArenaNet, 2005)
- *Hearthstone* (Blizzard Entertainment, 2014)
- *Magic: the Gathering* (Wizards of the Coast, 1993)
- *NetHack* (DevTeam, 1987)
- *Path of Exile* (Grinding Gear Games, 2013–2021)
- *Pokémon series* (Game Freak, 1996–2021)
- *Terraria* (Re-Logic, 2011)
- *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017)
- *Warcraft 3* (Blizzard Entertainment, 2002)

4.1.4 Generality

So far, in this group of domain concepts, we have seen how game simulations have a wide variety of economy mechanics. The need for supporting this variety is itself the last domain concept of the group and consists of a single architectural requirement, coded as *Requirement for Simulation Generality* (RSG-*):

(RSG-1) Simulation Generality. The *Unlimited Rulebook* aims to support any game that focuses on economy mechanics in its creative process. However, there are many possible types of economy mechanics among entities, effects, abilities, and rules. As a reference architecture, our proposal must provide guidelines for how to specialize its abstract design into the specific needs of each game's economy. Information sources: Adams and Dormans (2012); Dormans (2012a); Plummer (2004); Schell (2020); and, by definition, all games from Table 3.5 — though *Dwarf Fortress* (Bay 12 Games, 2006) and *Loop Hero* (Four Quarters, 2021) are notably hybrid games with both real-time and turn-based gameplay.

4.2 Subsystem Integration

The second group of domain concepts in the *Unlimited Rulebook* regards how the economy subsystems interoperate with other parts of a game system. After all, simulation of economy entities and behaviors is only a part of the complex architecture of games as a whole and it does not run in isolation. As we will see, many requirements from other systems end up influencing how we design and implement the economy mechanics of a game. There are three domain concepts in this group: Inter-System Communication, Runtime Lifecycle, and Compatibility.

4.2.1 Inter-System Communication

The main domain concept in this group is about defining **how subsystems communicate with the simulation** of a game. This is always a two-way interaction: things that happen outside the simulation affect the simulation and vice-versa. At the same time, we want different game subsystems to be as decoupled as possible. To do so, it is important to understand exactly what they need (or might need later in development) from each other. Figure 4.5 shows an example of how simulation and interaction subsystems exchange information using *Ragnarok Online* (Gravity Interactive, 2002) as an example. We found four architectural requirements regarding the domain concept of Inter-System Communication, coding them as *Requirements for Inter-system Communication* (RIC-*) and listing them below:

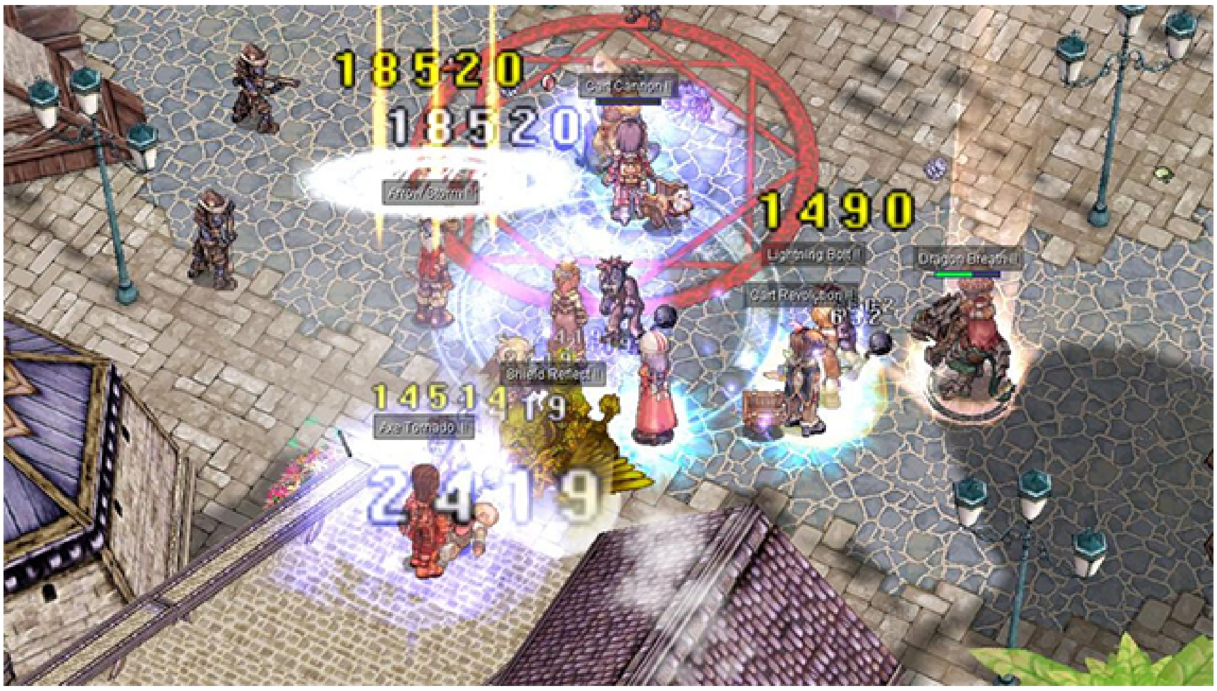


Figure 4.5: Screen capture from *Ragnarok Online* (Gravity Interactive, 2002). In this real-time MMORPG, players could activate skills by combining keyboard keys and mouse clicks to choose what skill to use and how to use it (e.g., target a spell at this or that enemy) as per (RIC-1). Conversely, whenever certain simulation events happened, the user interface would play visual effects in response to inform the player of what happened (e.g., how much damage a creature took) as explained by (RIC-2).

(RIC-1) Simulation Interaction. The first requirement is that subsystems that are not responsible for the simulation must be able to request some form of intervention on the simulation. Since simulations change their state through effects which, in turn, can be produced by abilities, there must be an API for other subsystems to activate or trigger abilities inside the simulation. For instance, allowing the user to cast a spell not only requires connecting control input to an in-game ability, but it might also require the user interface to enter specific interaction modes to, for instance, let the player pick a target for their spell — see (RRL-4). Similarly, some abilities might provide the player with choices (e.g., a pair of special boots that can be activated to either dash forward or jump high). This means the ability API also demands adherence to a certain protocol (e.g., first initiate ability, then choose targets, then confirm the ability). During this process, interface designers might want to show the players what targets are invalid and what effects that ability is likely to cause (i.e., a preview of the ability) — information that the simulation will have to provide without actually executing the ability. Sometimes abilities will not be available and the simulation must inform the other systems of that, which, in turn, have to inform the players and AIs of that as well. Speaking of AIs, subsystems must agree on whether a given entity is under the player's control or an AI's control. Information sources: Adams and Dormans (2012); Dormans (2012a); Rollings and Ernest (2006); Schell (2020); Wizards of the Coast (2021); all games from Table 3.5; and the input from USPGameDev members.

(RIC-2) Simulation Events. One way for the economy simulation to communicate with other subsystems is through events like in an event-based system or with the *Observer* pattern (Gamma *et al.*, 1995). Since economy mechanics include behaviors that might happen even without input from players, this is a way of allowing other subsystems to ask for information about *when* events of interest happen without making the simulation aware of that, thus keeping them decoupled. To do so, the architecture needs to specify that an API for registering, handling, and detecting simulation events should exist in the final game architecture. Information sources: Gregory (2019); Rollings and Ernest (2006); Schell (2020); and the following games (considering only those whose implementation we had access to):

- *BYTEPATH* (a327ex, 2018)
- *Diablo* (Blizzard Entertainment, 1997)
- *The Battle for Wesnoth* (The Battle for Wesnoth Project, 2003)
- *Veloren* (Veloren team and contributors, 2018)

(RIC-3) Simulation Queries. Sometimes, subsystems might need information from the simulation immediately, so they cannot wait for an event that brings them that information. For these cases, subsystems need a way to *query* the simulation API about its inner state, like asking where the player is right now so the graphics subsystem can know where to draw their avatar on the screen. As mentioned in (RIC-1), the subsystems outside the simulation might also need information about what abilities are available and what would likely happen if they were used right now. These should also be available as queries if possible. Information sources: Schell (2020); Wizards of the Coast (2021); the input from USPGameDev members; and the following games:

- *Caves of Qud* (Freehold Games, 2015)
- *Dungeon Crawl: Stone Soup* (DCSS Devteam, 2006–2021)
- *BYTEPATH* (a327ex, 2018)
- *Diablo* Blizzard Entertainment (1997)
- *The Battle for Wesnoth* The Battle for Wesnoth Project (2003)
- *Veloren* Veloren team and contributors (2018)

(RIC-4) Inter-System Entity References It is very common, especially in *Entity-Component-System* engines, that all subsystems of a game use a single, unified method of identifying game entities. Typically, this happens in the form of an integer or string identifier that can be used to query the state of an entity regarding different subsystems of the game. For instance, programmers could query the simulation to know the entity's current life points or they could query the graphics subsystem to gather the vertex data that make up the entity's 3D model so they can draw it. In this kind of architecture, the concept of entity is spread beyond the simulation. One way or another, the architecture of a game must define a way to reference simulation entities from code in other subsystems, and that referencing mechanism should be able to detect stale references as per (ROM-2). The entity referencing mechanism is also important for associating economy mechanics with physics mechanics (e.g., detecting *what* entities are currently inside the area of effect of a certain spell) and for synchronizing entity states over a network connection. Information sources: Gregory (2019); Rollings and Ernest (2006); the input from USPGameDev members; and all games from Table 3.5.

4.2.2 Runtime Lifecycle

The subsystems of a game are not simply modules that exchange data — they exist and run under the **runtime lifecycle of a game engine**. This lifecycle consists of the initialization and termination procedures as well as the ubiquitous *Game Loop* pattern. It also concerns restrictions on how a game system operates as a whole. Figure 4.6 illustrates how the runtime lifecycle can interact with economy mechanics using *Final Fantasy Tactics Advance* (Square Enix, 2003) as an example. Our information sources brought four architectural requirements to our attention, coded as *Requirements for Runtime Lifecycle* (RRL-*) and listed below:

(RRL-1) *Game Loop* Compliance. Given the real-time, interactive nature of games, all subsystems must abide by the fact that they most often only acquire control of the execution flow when they are directly or indirectly serviced by the *Game Loop* on a frame-per-frame basis. This means computations might need to be done in partial steps then wait for a full frame to continue what they were doing. Moreover, the key role of interactivity in games means that some parts of the subsystems will need to wait for user input, hanging on that temporary inert state for as long as it takes. Sometimes, one subsystem will have to wait for another subsystem to finish a task before continuing their own (e.g., the battle subsystem in the economy simulation has to wait for an attack animation to finish in the animation subsystem before applying damage, otherwise entities might disappear before they are visibly hit). In other words, game architectures benefit from support for asynchronous execution. Lastly, the way the *Game Loop* structures the execution flow means that the computations that happen every frame are strictly limited by the minimum target FPS of the game, which can be a severe restriction in games with complex simulations or extensive graphics and audio features. Information sources: Gregory (2019); Nakagawa *et al.* (2014); Rollings and Ernest (2006); Wizards of the Coast (2021); and all games from Table 3.5.

(RRL-2) Simulation State Persistence. A very common feature in games is that players can save their progress and continue later on, during an entirely new execution of the application. There are many ways games provide this feature to users: sometimes they can save and load their progress at any time, sometimes they can only do so in specific moments, and other times the game saves the progress automatically (i.e., it *autosaves*) at key points and/or periodically. This persistence feature implies a series of requirements. First, it demands that the game state be serializable. Second, it means the simulation should be able to pick up execution not only from their initial states but also from "advanced" states, both on the virtual world level but also on the entity level (e.g., often, games only need to restore the player's avatar state to allow continuity of the user's progress). Third, to do that, the architecture of the simulation subsystems requires a clear expectation of what states are stable enough to persist and start from (e.g., only at the start

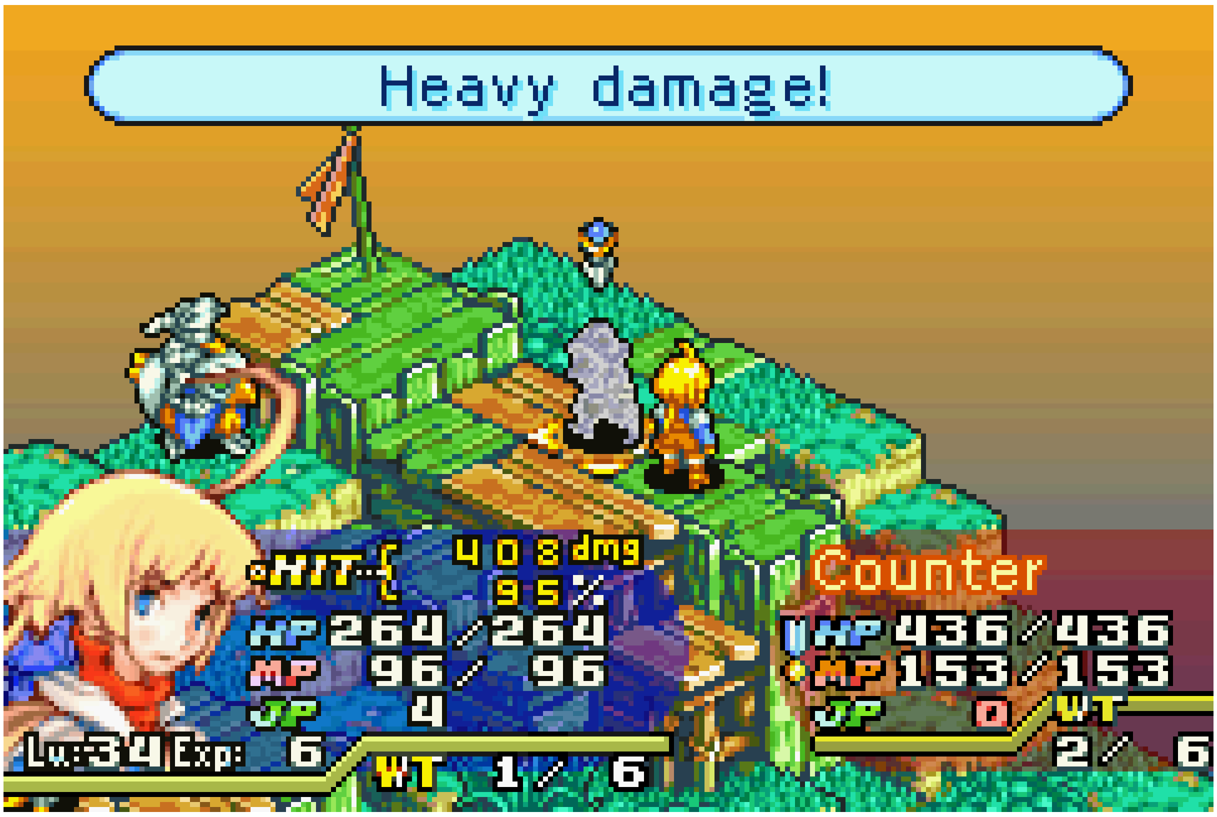


Figure 4.6: Screen capture from *Final Fantasy Tactics Advance* (Square Enix, 2003). Using abilities (called "skills") in this game involves multiple interaction steps. During a character's turn, the player has to choose (1) where and how they will move, (2) what action or skill they will use, (3) where they will target that action, (4) confirm, after reading the preview of the action (which is the moment captured in the image), if they really want to take that action, then, after the animation of the action finishes, (5) the player has to choose what direction the character will be facing from now on (because flanking is an important part of the game's strategy). Note how the preview of the action shows the estimated damage amount and to-hit chance, as well as the fact that the triggered ability "Counter" from the enemy will activate (assuming they do not die from the attack).

of a new stage). Information sources: Gregory (2019); the input from USPGameDev members; and all games from Table 3.5.

(RRL-3) Partial World Simulation. Games can have very large simulation worlds and it is not feasible to always keep their whole state in memory — especially considering other game subsystems are competing for this resource. This means that game simulations might be required to process their virtual world only one piece at a time. Some games will have, for instance, clear separations of different world areas which the players interact with only one at a time. Other games, however, place the player into a single, continuous world to navigate. In these cases, the world data is also broken into chunks and loaded one at a time, except this is done by background processes so that players never notice they do not have access to the whole world all the time (i.e., the world chunks are *streamed* to the simulation). Either way, the game architecture has to be able to simulate its virtual world in parts and any changes one part inflicts onto others has to be accounted for (e.g., pulling a lever in one room opens a door in a different, distant room, even if that room is not loaded in memory yet). Information sources: Gregory (2019); and the following games:

- *Guild Wars* (ArenaNet, 2005)
- *Minecraft* Mojang Studios (2011)
- *The Legend of Zelda: Breath of the Wild* Nintendo (2017)

(RRL-4) Interaction Modes. Playing a game usually follows a pattern: you boot up the system, load a save, play through the main interactions (e.g., find and conquest dungeons in *The Legend of Zelda* games), alternate with in-game side activities (e.g., look for and collect "heart pieces" also in *The Legend of Zelda* games), then eventually save your game, return to the title screen, and quit the application. This means not only simulation subsystems need start-up and finishing routines, but they also need to switch between different interaction modes, often tied to the simulation modes from (RSP-4), such as routing the input from the directional buttons to a menu cursor instead of making the player avatar walk around when they have the inventory window open. In other words, the simulation subsystems must be aware of the overall interaction state of the game and operate accordingly. Sometimes, initializing and finishing subsystems happen when different interaction and simulation modes begin, finish, or alternate among themselves. Information sources: Gregory (2019); and all games from Table 3.5.

4.2.3 Compatibility

The last domain concept in this group is about how game software does not exist in a vacuum: there is an entire ecosystem of tools and libraries developers, designers, and artists rely on to build a game. Architects have to consider that and plan for how **compatible** the systems they design will be with other technologies in the industry. Figure 4.7 shows the interface of the *Godot* engine and how it requires certain accommodation from the architecture of a game. We found three kinds of compatibility to look for and listed them below, coding them as *Requirements for Technical Compatibility* (RTC-*):

(RTC-1) Engine Compatibility. A considerable part of the game industry, especially smaller companies, develop games using game engines. Even more, certain engines are more clearly prominent than others. A reference architecture for games cannot ignore this and ought to consider how developers can apply it to the architecture of games developed with these engines. For instance, most WYSIWYG engines group graphics, physics, and mechanics together into a single unified object model. This means the object model from (ROM-1), (ROM-2), and (ROM-3) will not exist in a conveniently encapsulated environment — instead, most of its data and structure will be mingled with that of other subsystems. As much as we would like to have a clean separation of concerns, the reality of game development has to be accounted for. Information sources: Nakagawa *et al.* (2014) and te input from USPGameDev members.

(RTC-2) Platform Compatibility. Though this is often mitigated by the game engine used, platform support is common a concern in game development. On the one hand, the architecture has to consider the limitations (e.g., low memory and battery capacity) and affordances (e.g., touch screens, gamepads, handheld devices, etc.) of the target platform of the game. On the

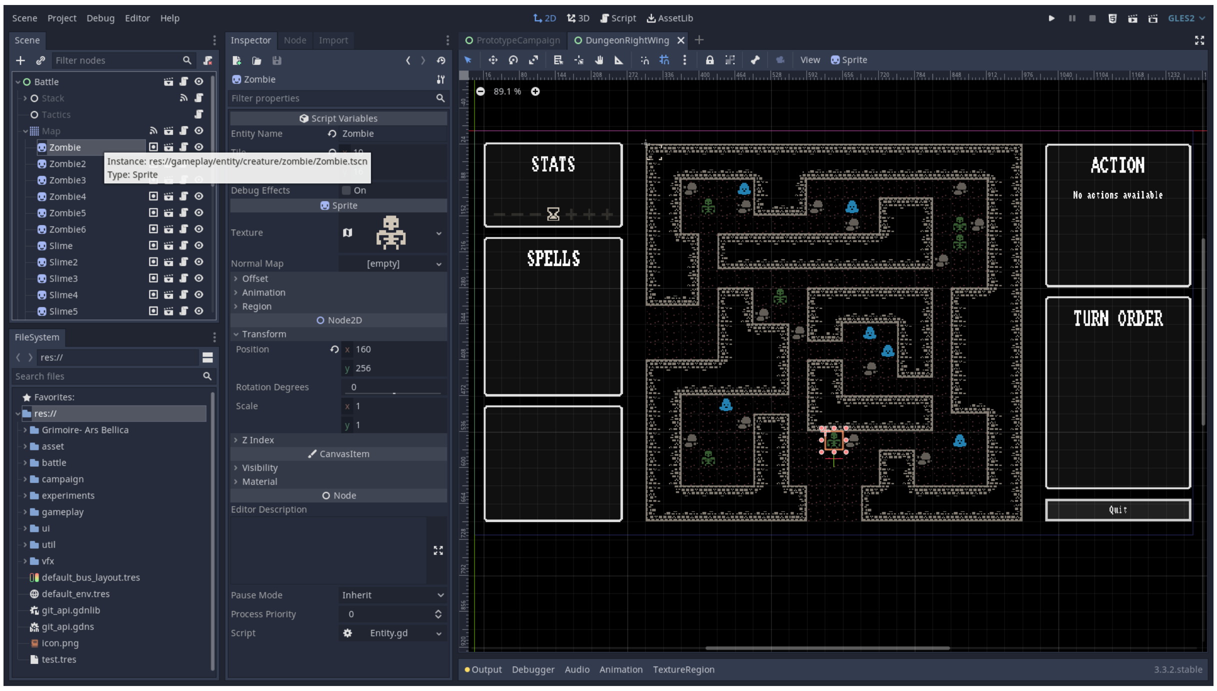


Figure 4.7: Screen capture from the *Godot* engine. Although *Godot* takes care of most platform compatibility issues for developers, it requires (or, at least, strongly incentivizes) games to be structured using its tree-based scenes paradigm (note the scene tab on the left). It also uses its own file formats for data-driven features (e.g., `*.tscn` for scene data files). Different engines, like *Unity3D*, use different object models, formats, etc. The *Unlimited Rulebook* architecture should support all of them. The game being developed in the image is *Grimoire: Ars Bellica*, one of the proofs-of-concept described in Chapter 6.

other hand, it usually also needs to allow a certain flexibility, so that multiple platforms are supported at the same time. In this regard, the more simulation subsystems can be decoupled from these implementation details, the better. Information sources: Nakagawa *et al.* (2014); Plummer (2004); the input from USPGameDev members; and all games from Table 3.5.

(RTC-3) Data Format Compatibility. When the simulation state needs to be serialized due to (RRL-2) or (RDD-2), the limitations of the available data storage formats have to be taken into consideration. For instance, a popular format such as JSON[1] can only store primitive types (e.g., integers, strings, booleans) and two compound types (arrays and associative tables). For most entity types and state variables thereof, these are sufficient but, for mechanics such as abilities (RBM-3) and custom rules (RBM-4), the implementation design is less straightforward. An additional limitation, especially when developers opt to design custom data formats, is that compatibility with version control (i.e., using plain text, line-by-line formats) is an integral part of software development and games as well. Information sources: Nakagawa *et al.* (2014); the input from USPGameDev members; and the following games (that we are certain of):

- *Dungeon Crawl: Stone Soup* (DCSS Devteam, 2006–2021)
- *Diablo* (Blizzard Entertainment, 1997)
- *Dwarf Fortress* (Bay 12 Games, 2006)
- *Factorio* (Wube Software, 2016–2021)

[1] json.org (last accessed May 26th, 2021)

- *NetHack* (DevTeam, 1987)
- *Veloren* Veloren team and contributors (2018)

4.3 Iterative Development

The third and last group of domain concepts are about how the development of games often follows an iterative format and how the architecture can respond to that. Even if some games are developed using a more cascade-like process, at some point in their production the creative process naturally lends itself to an iterative cycle. To ensure that the architecture promotes this iterative development of games and the creative process behind it, the analysis of our information sources pointed to three domain concepts: Creative Process Workflow, Data-Driven Design, and Code-Base Evolution.

4.3.1 Creative Process Workflow

When we consider the workflow of the creative process, we see that obtaining feedback on ongoing design decisions is an integral part of it. However, games are complex systems, and changing them to accommodate a new decision takes time, depending on the kind of change the designers need. This domain concept requires that as many as possible types of change are of the quickest type: changes that you can test immediately and with no (or minimal) need for technical effort. Figure 4.8 shows the world editor of *Warcraft 3* (Blizzard Entertainment, 2002), a development tool that not only reduced the effort of developing the game and its adventures, but that was also later used by the player community to make their own maps, MODs, and entirely new games inside the *Warcraft 3* engine. We divided the Creative Process Workflow problem into three architectural requirements that we describe below, coded as *Requirements for the Creative Process* (RCP-*):

(RCP-1) Continuous Build. When a team starts developing a game, it takes a while until they have a playable system. The architecture, together with the building tools and engine, can help reduce the time needed to reach that first playable. For instance, they can reduce the amount of boilerplate code needed to simply open a window and start drawing things on it (something that might easily require hundreds of lines of code if the game uses no engine at all). Once the team has a playable build, however, the next challenge is to keep an up-to-date build always available. This involves design decisions that consider compilation times, as well as tools and technologies that might automate parts or all of the build process. Information sources: Kasurinen *et al.* (2017); Murphy-Hill *et al.* (2014); Politowski *et al.* (2021); Rollings and Ernest (2006); Schell (2020).

(RCP-2) Accessible Development Tools. When a designer or artist wants to create new mechanics, assets, or other forms of content for the game, they ideally should not have to write any code. Dedicated editing tools for the virtual world and its entities, for instance, are a common approach, especially in WYSIWYG engines. That means the runtime architecture needs to load content produced by these external tools or, when they have an embedded editor, has to make the additional design effort of supporting advanced graphical user interfaces. Other ways of providing accessible development tools include scripting support through an embedded programming language or visual scripting tool — though we said coding is better left out of the creative process, sometimes it is the only reasonable way of adding mechanics (especially behavior mechanics) to the game. Information sources: Gregory (2019); Kasurinen *et al.* (2017); Murphy-Hill *et al.* (2014); Rollings and Ernest (2006); input from USPGameDev members; and the following games (that we are certain of):

- *Dungeon Crawl: Stone Soup* (DCSS Devteam, 2006–2021)
- *Diablo* (Blizzard Entertainment, 1997)
- *Dota 2* (Valve Corporation, 2013)
- *Dwarf Fortress* (Bay 12 Games, 2006)
- *Factorio* (Wube Software, 2016–2021)
- *Hearthstone* (Blizzard Entertainment, 2014)
- *Minecraft* (Mojang Studios, 2011)
- *Sid Meyer's Civilization V* (Firaxis Games, 2010)
- *The Battle for Wesnoth* (The Battle for Wesnoth Project, 2003)
- *Warcraft 3* (Blizzard Entertainment, 2002)

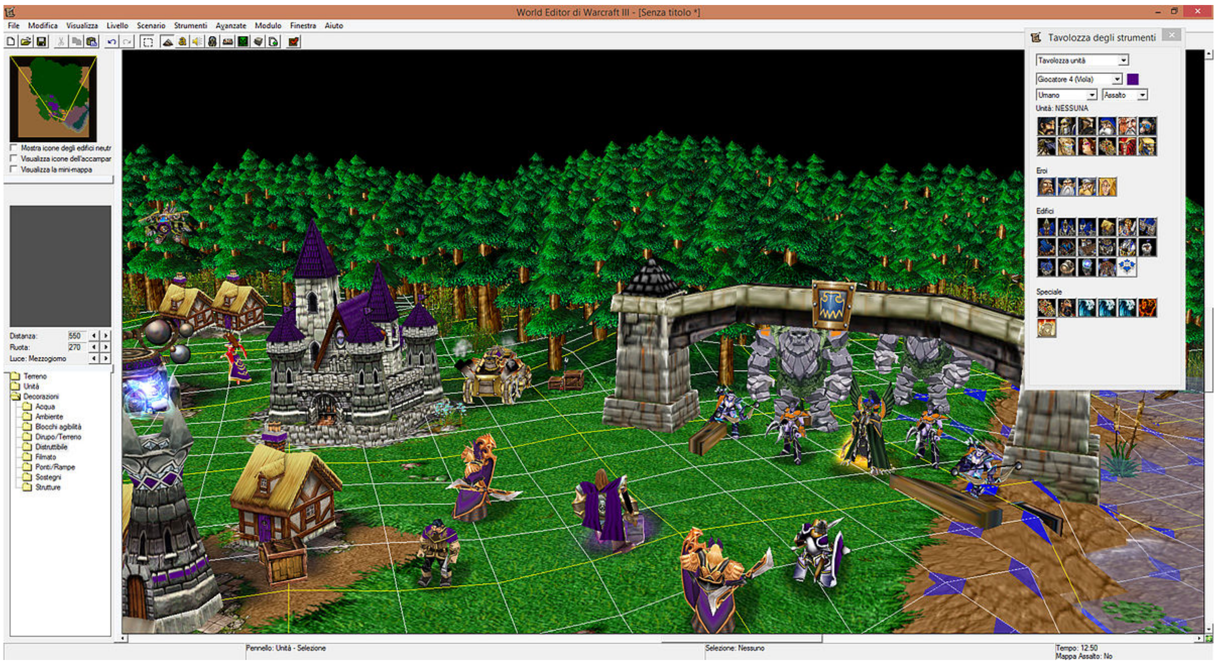


Figure 4.8: Screen capture from the *Warcraft 3* world editor (Blizzard Entertainment, 2002). This tool was shipped together with the game and provided a complete suite of features that allowed players to create custom maps and even entirely new games within the *Warcraft 3* engine. The accessibility of the interface enables even non-programmers to create their own content, though it has a scripting language more experienced programmers can take advantage of.

(RCP-3) Runtime Tools. Once the designer or artist has created new content for the game, they need to test it. When they do, they will often notice something they need to go back and change. If possible, games should be able to allow these late changes without having to close the application and start the creation process from zero. For instance, the game could have an embedded editor that could edit game data without closing the game itself. Or, at least, it could detect when a data file it loaded has changed and reload it, updating the runtime game elements accordingly. Other features that help in testing a game are cheats (special inputs that allow developers to play through the game faster and that are usually not present in release builds) and in-game consoles and inspectors that allow testers to evaluate the game state variables and see if everything is as expected. Information sources: Gregory (2019); Murphy-Hill *et al.* (2014); the input from USPGameDev members; and the following games (that we are certain of):

- *Dota 2* (Valve Corporation, 2013)
- *Factorio* (Wube Software, 2016–2021)
- *Minecraft* (Mojang Studios, 2011)
- *Terraria* (Re-Logic, 2011)
- *Warcraft 3* (Blizzard Entertainment, 2002)

4.3.2 Data-Driven Design

To support the requirements for the creative process workflow, the architecture of a game has a series of further requirements to allow it the flexibility and robustness of operating on top of the data it will only access at runtime. This domain concept essentially corresponds to the principle of **data-driven design** which, in turn, comprises the following two architectural requirements, coded as *Requirements for Data-driven Design* (RDD-*):

(RDD-1) Runtime Data Access. The basic requirement for this domain concept is that the game system must be able to load data from the disk and into its runtime memory, where it has to be structured to make it easy for subsystems to query the information they need. Most likely an entire subsystem should be dedicated to centralizing these operations and providing a consistent API for other parts of the game to rely on. This way things like hot-loading from (RCP-3) can be added without requiring developers to migrate the entire code base. It is also important to consider the memory optimizations mentioned in (RRL-3). Information sources: Gregory (2019); Politowski *et al.* (2021); Rollings and Ernest (2006); and all games from Table 3.5 except *Nomic* (Peter Suber, 1982) (which only exists as an analog game) and *BYTEPATH* (a327ex, 2018) (which implements all game mechanics in a hard-coded manner).

(RDD-2) Data-Driven Simulation. Once a game data management service is available in the architecture, there are specific requirements it must address involving the simulation subsystems. Since simulation data comprises static data about the virtual world as a whole and dynamic data from entities, those are the two main types of runtime data the game will need for its mechanics: world data and entity data. This data usually represents an *initial state* of their simulation counterparts (e.g., how geometries and entities are initially positioned in a given game stage, how many life points a creature has when they first appear, etc.). For entities, this type of data loaded from disk is sometimes called a *spawner* (Gregory, 2019, page 1065), *prefab* (a term employed by the *Unity3D* engine), or *prototype* (Nystrom, 2014, Chapter 5) — which is the term we will use here. Entity prototypes also have requirements of their own. They need to specify what *type* of entity they are used for and sometimes those types can also be defined in data files known as *data schemas* (Gregory, 2019, page 1066). Most likely, entity prototypes need to provide information about the physical, graphical, and auditory aspects of the entity, which means they need to be able to refer to *other* data entries in the game's database. Lastly, all this data has to be validated, either by the tool-side applications used to produce them, or by the game itself when it loads them, or both. Ideally, invalid data entries should be handled both gracefully (i.e., without crashing the game) and deliberately so that the system can report the failure to developers. Information sources: Gregory (2019); Murphy-Hill *et al.* (2014); Rollings and Ernest (2006); and all games from Table 3.5 except *Nomic* (Peter Suber, 1982) (which only exists as an analog game) and *BYTEPATH* (a327ex, 2018) (which implements all game mechanics in a hard-coded manner)

4.3.3 Code-Base Evolution

It is not always possible for the creative team to introduce new content and mechanics to a game without technical intervention. In which case, **the easier it is to change the code, the better**. This domain concept acknowledges that there is no perfect data-driven engine and that maintaining the game architecture as it grows is essential to minimizing the costs of the creative process in games. Most of the requirements in this category are about principles that would improve the quality of any kind of software but we focus on the challenges that are specific to games and economy mechanics. We found five architectural requirements, coded as *Requirements for Code Evolution* (RCE-*), and listed below:

(RCE-1) Decoupled Subsystems. Though not all games have a clear architectural division into subsystems, doing so and ensuring the subsystems remain as decoupled as possible has many advantages. Changes developers do to one subsystem have less chance of forcing subsystems that depend on the changed subsystem to be revised and changed themselves. A stricter approach could even allow subsystems to be replaced entirely — for instance, with a third-party implementation, which would possibly reduce development costs thanks to code reuse. At the same time, a robust API design for a subsystem might allow it to be reused in a different game with similar features (e.g., titles in the same series), becoming a form of investment in future projects. Information sources: Pascarella *et al.* (2018); Plummer (2004); Politowski *et al.* (2021).

(RCE-2) Extensibility. During the initial stages of the production of a game, its codebase is in constant growth. Subsystems are added or have new features added to them. Without a conscious effort, this is an easy trap for architectural erosion and technical debt creep. Games need to iterate fast and having runnable builds is a priority, but so is ensuring that the architecture remains sustainable and development costs manageable. Thus, this requirement deserves special attention in the early steps of the architectural design of a game. A particularly complex problem with extensibility in games is that sometimes a new feature interferes with multiple parts of the codebase (e.g., the cockatrice example from Chapter 1). The architecture for the economy mechanics must minimize the chances that this will happen by considering the particular needs and expectations of the game being developed. Information sources: Pascarella *et al.* (2018); Plummer (2004).

(RCE-3) Flexibility. Even when architects take care to organize the game architecture into subsystems, keep them decoupled, and design for the possibility that they will be extended, game features remain remarkably unpredictable. The creative process is constantly revising mechanics and gameplay aspects to achieve the desired user experience. This means that games also need to be flexible in their architectures since subsystems will have to be branched into two or more subsystems, others will have to be merged, and dependencies will be broken. Decoupling and extensibility must be achieved while also accounting for vague requirements, the possibility of large changes late in development, and, especially, the costs for deep revisions in the game economy mechanics. Information sources: Hunicke *et al.* (2004); Kasurinen *et al.* (2017); Murphy-Hill *et al.* (2014).

(RCE-4) Code Accessibility. A very common human factor in the development of any software is that the programmers that compose the team may change. When a new developer (or, for instance, an artist that for some reason had to write code for themselves) comes in contact with the code base, the overall organization into subsystems helps them know where to look for the code pertinent to them at the moment. Moreover, APIs for these subsystems should be designed to reduce the chance for error or unexpected behavior. Robust coding conventions help too. Information sources: Plummer (2004) and the input from USPGameDev members.

(RCE-5) Reliable Error Detection. Lastly, even if all the other requirements in this domain concept were already addressed, mistakes will happen during development. It is simply impossible to predict all possible liabilities. Instead, it is better to be prepared for when errors do happen. First, if possible, as many mistakes (or even just potentially dangerous lines of code) as possible should be detected as they are written — by the text editor used, a lexical and/or syntactical analyzing tool, or even the compiler. When that fails, automated tests and a robust build pipeline are the next lines of defense, though that is very challenging in game development since it's hard to emulate gameplay integration and acceptance tests. The errors that still creep into the game have to be detected at runtime, like with the invalid entity data from (RDD-2). Being able to handle runtime errors with grace and then recording as much information about its context as possible is a great contribution to the debugging process. For instance, the game could dump a snapshot of the entire simulation state (i.e., all entities and their individual states) whenever an unrecoverable crash happens. Some engines even take screenshots or record a video of the five or so seconds that led to the crash. Information sources: Gregory (2019); Pascarella *et al.* (2018).

In this chapter, we listed and explained all architectural requirements we take into consideration to design the *Unlimited Rulebook*. We organized these requirements into domain concepts which, in turn, we grouped grouped according to their relationships. Just like we cited the information sources we drew each requirement from, in Chapter 5 we cite these requirements and domain concepts to justify each aspect of the reference architecture we propose.

Chapter 5

The Unlimited Rulebook Architecture

"Whenever a card's text directly contradicts these rules, the card takes precedence."

Comprehensive Rules for *Magic: the Gathering* (Wizards of the Coast, 2021)

This chapter describes the *Unlimited Rulebook* reference architecture based on the domain concepts and architectural requirements from Chapter 4. To describe this proposal, we follow the ProSA-RA process from Section 3.2. More specifically, we divide the presentation of the *Unlimited Rulebook* into three major viewpoints: Crosscutting Viewpoint (Section 5.1), Runtime Viewpoint (Section 5.2), and Source Code Viewpoint (Section 5.3) (Nakagawa *et al.*, 2014).

In the Crosscutting Viewpoint, we discuss broad topics of the *Unlimited Rulebook*, like the terminology, the reference model, the expected use cases, and the key variabilities of the architecture. We present our basic assumptions and lay the foundation on top of which the other two viewpoints are built. In the Runtime Viewpoint, we describe how a system designed with the *Unlimited Rulebook* works in action. This includes the main execution flows, the logistics between different modules and subsystems, and the dependencies between APIs, protocols, and other parts of the game. Last, the Source Code Viewpoint describes specific implementation guidelines from the perspective of the programmers and architects. This part focus on patterns and architectural styles that can be used to fulfill the requirements of the game and its particular use of the *Unlimited Rulebook*. The ProSA-RA method originally proposed an additional viewpoint called the Deployment Viewpoint, where we would discuss how our target system fits into hardware and network layouts. However, since the *Unlimited Rulebook* covers only the runtime architecture for simulating economy mechanics, this Viewpoint is not applicable in our research — and, thus, we skip it.

Throughout this chapter, we cite the architectural requirements from Chapter 4 to clarify what our proposals are based on, forming a causal chain back to the information sources from Section 3.3. **We do so by using their codes (e.g., RDD-1)**. We suggest consulting back to Figure 4.1 for a quick reference on the requirements we cite. Most diagrams we present here all use UML 2.5 (OMG, 2015). That said, aside from Section 5.3 where we discuss specific implementation designs, our use of UML to communicate the *Unlimited Rulebook* reference architecture more commonly models *abstract elements* of the architecture and the domain of games, their creative process, and economy mechanics. As a last note, the state of the *Unlimited Rulebook* as described in this chapter is the result of the iterative process described in Section 3.2.3 — i.e., there were previous versions of the reference architecture, including some we published before (Mizutani and Kon, 2019, 2020), but what we present here is the current version of the *Unlimited Rulebook*.

5.1 Crosscutting Viewpoint

The first part of discussing the Crosscutting Viewpoint is to establish the *application context* of our reference architecture, as mentioned in Section 3.2.1. This provides the motivations for the underlying decisions we made throughout the design process of the *Unlimited Rulebook*. The **goal** of this reference architecture is to provide reusable knowledge about the architecture of economy mechanics in games to reduce the implementation costs caused by the creative process. Consequently, the **scope** of this proposal is limited to the runtime architecture of digital games and, more specifically, to the software design of the simulation of economy mechanics. At the same time, the software elements that implement the simulation of a game depend on the implementation of other parts of the game system as well as the implementation of applications outside it, and vice-versa. Thus, the **needs** of this proposal include considerations regarding the interface between the simulation of economy mechanics and other features of a game system. They also include explaining how our design meets the requirements of the creative process of making games.

The main **risk** our proposal aims to reduce are the technical costs that rise from the architecture used to implement economy mechanics. In this sense, we will often provide multiple approaches to a given design problem, presenting the benefits and shortcomings of each. The architects of each game should ponder between these options and determine what they believe minimizes the risk of technical costs for their project. The **limitations** of our proposal involve the trade-off between reusability and flexibility of implementing new economy mechanics. Providing simpler but limited reusable constructs for the creative team to produce new mechanics reduces the need for technical knowledge but increases the dependency on the technical team. Providing more flexible building tools by supporting complex combinations of operations empowers the creative team but demands more technical expertise from them. Lastly, since our proposal is technology and game agnostic, there are **constraints** on the assumptions we can make, which results in more abstract and generic designs that architects are left to translate into concrete architectures.

Based on this application context, the remainder of this Viewpoint is divided into three central aspects of architecting economy mechanics for the creative process of games: the reference model, the expected use cases, and the possible variabilities. Whenever architects have a game they believe would benefit from the *Unlimited Rulebook*, they must first understand these three aspects of the game, its intended economy mechanics, and the likely needs of both the creative and the technical teams. After that, the other viewpoints can be used to design the game architecture itself.

The reference model is a framework for analyzing a game project and determining what it will demand from the *Unlimited Rulebook* architecture. Once the requirements of a game fit into the reference model, the architects can use the *Unlimited Rulebook* to map those requirements into a specific architecture for that specific game. The reference model also provides a common terminology to discuss the many concepts involved in the design process of the architecture of the game, as well as the relationships and dependencies between the manifestations of those concepts in the actual system. We describe the reference model in Section 5.1.1.

The uses cases describe what features players and developers alike expect from the game system. As players, the use cases involve gameplay interactions: the possible actions we can issue to our avatars, the information the game is supposed to provide us in real-time, and any common behavior the simulation is expected to perform. As developers, the use cases describe the usual tasks we must perform to populate the game with content and mechanics. In this sense, we consider the software design of the game as well as the interfaces and tools it exposes to the team as part of the desired feature set of the game. Section 5.1.2 elaborates on the use cases of games developed using the *Unlimited Rulebook* architecture.

The variabilities indicate what are the game system requirements that most influence the design of the resulting architecture. They also explain what are the key design decisions the architects will have to make and their corresponding trade-offs. We present this discussion in Section 5.1.3.

5.1.1 Reference Model

The *Unlimited Rulebook* reference model formalizes the basic assumptions we have about economy mechanics and the development process of games. To begin with, the model establishes the architectural context of economy mechanics within a game system. This is summarized in Figure 5.1 and broadly derived from the work of Gregory (2019). There are a number of **architectural elements** we expect every game to have, so we can later propose which element is responsible for which part of the domain problem and its related requirements.

First, we have the **Game System**, which is the software the **Technical Team** writes code for. As with every system, we assume its source code can be divided (to varying degrees) into different types of elements. In actual games, this division does not need to be explicit or even exist. The architectural elements we discuss here represent conceptual categories of software elements. Architects applying the reference model need only acknowledge how the equivalent functionality and interactions are implemented in their game.

The first of these elements is the **Game Loop** (RRL-1)[1], which ensures the synchronization between simulation and interaction. There is usually only one instance of the Game Loop. It invokes the **Subsystems** every frame and each of these are responsible for a different domain of the Game System, such as graphics rendering, input processing, network communication, simulation updating, etc. Sometimes, the Subsystems directly manipulate the **Simulation State**. However, to avoid duplicated code of common operations, they can also rely on a number of **Services**, such as a collision detection service that groups all algorithms for checking when the shapes of simulation objects intersect each other. This way whenever a Subsystem needs to check that — like a physics Subsystem or a UI Subsystem (for detecting clicks inside specific screen areas) — they can reuse this Service (RCE-1). Besides encapsulating common simulation operations (RIC-1, RIC-3, and RIC-4), some Services also serve as adapters to lower-level **I/O**

[1]This is an example of how we are going to refer to the codes of the architectural requirements from Chapter 4.

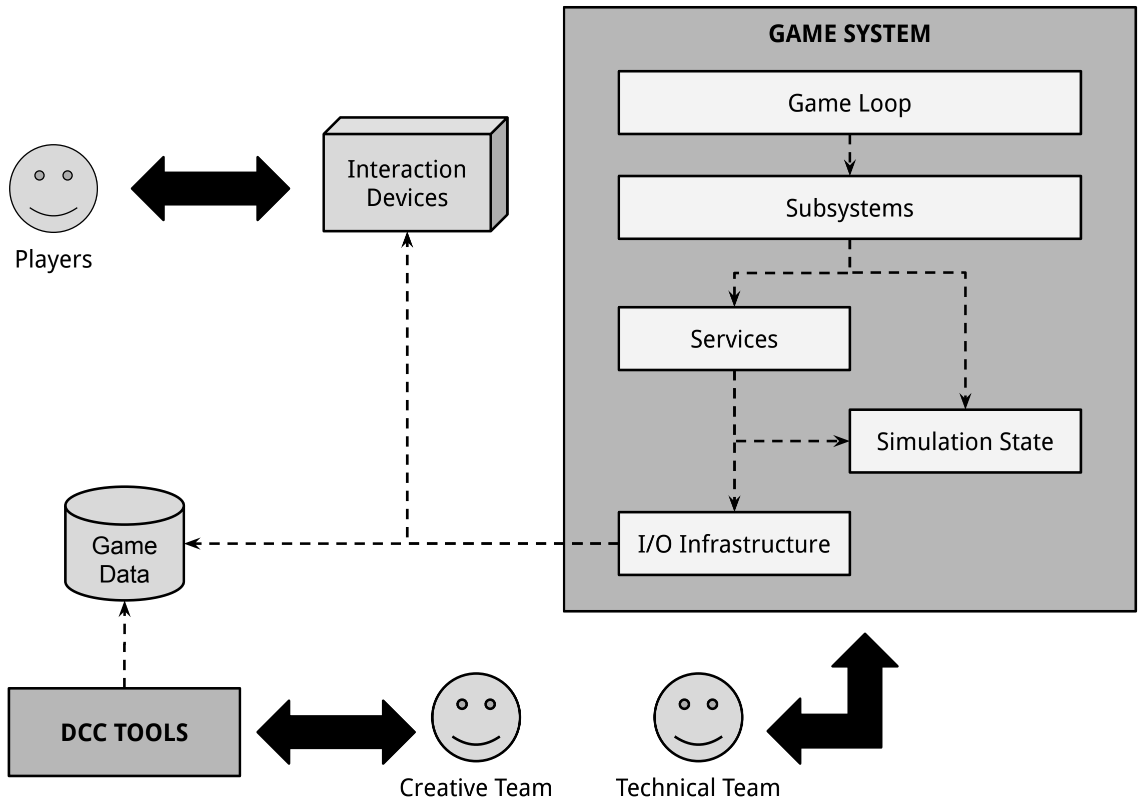


Figure 5.1: An informal UML diagram representing the architectural elements of a game — and their corresponding relationships — according to the *Unlimited Rulebook* reference model. It shows how the inner parts of a Game System depend on each other and how they relate to the different people and external systems (like DCC tools, see Section 2.1.2) involved in the development of games. We differentiate software systems from software parts by using dark gray boxes for the former and light gray boxes for the latter. The thick arrows represent interaction between people and a software or hardware component. The rest uses the standard meanings of UML diagrams.

Infrastructure (RTC-2) while others play a support role in the logistics between Subsystems and Services, like an event-handling Service (RIC-2). Essentially, Services are the specialized elements of the Game System that Subsystems use to facilitate their work (RCE-4) and that are commonly needed even across different games. Finally, the I/O Infrastructure, as opposed to the Simulation State, is responsible for the interaction features of the Game System. This involves reading from and writing to disk, where we expect the **Game Data** for assets and other forms of game content to be (RTC-3, RDD-1), as provided by the **Creative Team**. The I/O Infrastructure also includes sending and receiving data to and from all the **Interaction Devices** the **Players** interacts with to play the game (RTC-1, RTC-2): the video card, the sound card, the input controllers, the network card, etc.

Now that we have established the assumed context of our domain, we present the part of the *Unlimited Rulebook* reference model regarding the problem domain itself: the **Mechanics Model**. Like the architectural elements, the terms we discuss here refer to concepts and not actual implementation components, since the model should suit different kinds of games (RSG-1). Furthermore, there are a number of ways these elements can be designed and implemented, as we will discuss throughout this chapter. Figure 5.2 summarizes the concepts in the Mechanics

Model as well as the relations between them.

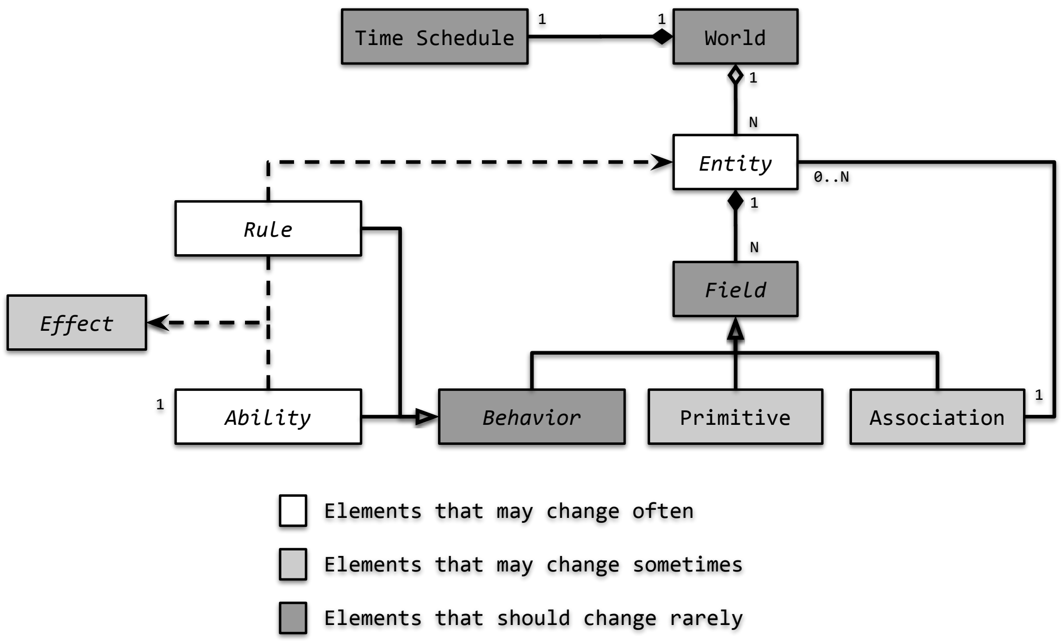

Time Schedule
World
Entity
Rule
Field
Effect
Ability
Behavior
Primitive
Association
1
1
1
N
0..N
1
N
1
1
Elements that may change often
Elements that may change sometimes
Elements that should change rarely

Figure 5.2: Mechanics Model of the *Unlimited Rulebook*. The classes to the right belong to the Game Object Model domain concept (ROM-1 through ROM-3), while the classes to the left are from the Behavior Model domain concept (RBM-2 through RBM-4). The classes in white — `Entity`, `Ability`, and `Rule` — are the main extension mechanisms of the model (RCE-2), while the classes in light gray — `Effect`, `Primitive`, and `Association` — are secondary extension mechanisms that are more expensive to change, and the classes in dark gray — `World`, `Time Schedule`, `Field`, and `Behavior` — represent mechanics that are less likely to change and can be depended on more reliably.

The Mechanics Model structures the information that makes up the Simulation State and highlights what operations it allows to both players and developers. The Simulation State is primarily composed of a virtual **World** and its **Entity** instances. There can be many types of Entities and each of them carries a part of the Simulation State within themselves, stored in their **Fields** (ROM-1, ROM-3). Fields can be of simple **Primitive** types (such as integers or even basic collections) that store information about the individual state of an Entity, including physical geometries (e.g., its position and shape) and economy resource values (e.g., its current speed and life total). Fields can also be **Associations** to other Entity instances, indicating, for instance, that an Entity is part of another Entity or that a certain group of Entity instances are under the influence of a particular Entity.

The World also stores part of the Simulation State into a **Time Schedule** regarding its internal, simulated passage of time. It keeps any information needed to work as a Simulation-wide FSM (RSP-4), such as turns and rounds and what Entities they belong to. Together, the elements of the Simulation State we have just discussed (the World, the Entities, all the Fields subtypes, and the Time Schedule) make up our solution to the Game Object Model domain concept. Furthermore, another special type of Field — the **Behavior** abstraction — works as a bridge to a different domain concept: the Behavior Model. There are two types of Behavior that

Entities can have.

The first kind of Behavior is the **Ability** abstraction. Abilities represent what Entities are capable of doing inside the simulation (RBM-3) by generating **Effect** instances (RBM-2). Effects, in turn, represent the fundamental changes that can happen to the Simulation State, as dictated by the game mechanics. As we will see, however, the Simulation State might also allow Services and Subsystems to directly create Effect instances inside the simulation, bypassing Abilities. The second kind of Behavior is **Rules**. They implement the game mechanics by determining how exactly each specific Effect type changes the Simulation State and, as a first-class construct in the reference model, allow Custom Simulation Rules to be implemented as part of the creative process (RBM-4). For that, Rules consist of a **Predicate** that specifies for what kinds of Effects and under what conditions the Rule applies, and a **Resolution** that processes Effects into changes in the Simulation State.

There are three types of changes caused by the resolution between Effects and Rules. First, they may change the state of an Entity by changing its Fields, which includes adding or removing Behaviors. Second, they may activate an Ability — we call them **Triggered Abilities** — causing a chain reaction in the simulation. Third, the Resolution may manipulate the Effect matched by its Predicate, changing it (e.g., by replacing it with a different Effect) to satisfy a special case in the Simulation State. Note that Effects, at least in the reference model, are not responsible for causing changes to the Simulation State — Rules are. Instead, Effects only provide information about the intended change. Since Rules are Entity Behavior Fields, it is possible, in this model, to put new Rules into effect as part of the simulation itself. This means the model supports self-amending mechanics since Rules produce Rules that might overrule themselves.

When it comes to the creative process of games, the elements of the Mechanics Model have varying degrees of expected extensibility and flexibility. The World structure, its Time Schedule implementation, how Entity instances store their Fields, and how Entity Behaviors present themselves to the rest of the Game System are the most rigid and expensive parts of the Simulation State. That is because too many parts of the Game System — inside and outside the Simulation State — depend on and sometimes are coupled to them. Since the World houses the Entity instances of the Simulation State, all Subsystems and Services depend on it and its interface to access the state of Entity instances. The Time Schedule affects the order in which mechanics are computed and, since games are temporal simulations by nature, it is common to design their implementation by making assumptions about what is processed before what. Just like the World controls access to Entity instances, an Entity controls access to its state by how it exposes its Fields. Lastly, using Entity Behaviors is the main way to give life to the mechanics of a Game System, which means there are many parts of the codebase that depend on the access interface of those Behaviors.

On the other hand, because these elements we just discussed are (or should be) more stable, the types of available Primitives, the kinds of possible Associations between Entity instances, and the set of applicable Effects are more tolerant to change, albeit not the cheapest. Both Primitives and Associations have the Field abstraction to shield other parts of the Game System to the addition and change of existing implementations. That said, changed Primitives require changed operations to read and write to them, and changed Associations change how Subsystems and Services lookup related Entity instances in the Simulation State. Effects are the basis upon which Abilities and Rules work, so changes to them propagate to these other elements but not much farther than that. All the elements in this mid-tier category in terms of extensibility and flexibility share a common characteristic: they are the building blocks for the elements we really want to be cheaper to change. That is, the real role of these elements is *reusability*, which is why they come with an embedded cost to change.

The elements developers change most, as part of the creative process of implementing economy mechanics, are Entity types, Abilities, and Rules. As game projects move from preproduction to production, ideally the implementation of the elements we discussed before should be more or less stable, so the Creative Team can focus on coming up with the mechanics the game needs to be successful as a product. This often involves mainly creating stages, creatures, items, characters, quests, etc., but also how all of these *act* inside the simulation. That is why the elements that we most expect to be extensible (RCE-2) and flexible (RCE-3) are Entity types, Abilities, and Rules, and why, whenever possible, we employ Data-Driven Design on their implementations, as we will discuss throughout this chapter.

5.1.2 Use Cases

The final users of games are the Players and, within the context of economy mechanics, Players rely on three main types of use cases to consider in the *Unlimited Rulebook*. Figure 5.3 summarizes these use cases. First, as Players play through different executions of the Game System, they start new "simulation sessions", save their progress, then continue it later. We call these use cases New Game (starting a new session), Save Game (storing the state of the current session to the disk), and Load Game (restoring the state of a session from the disk). These cases are specializations of the more abstract *Progress Persistence* use case.

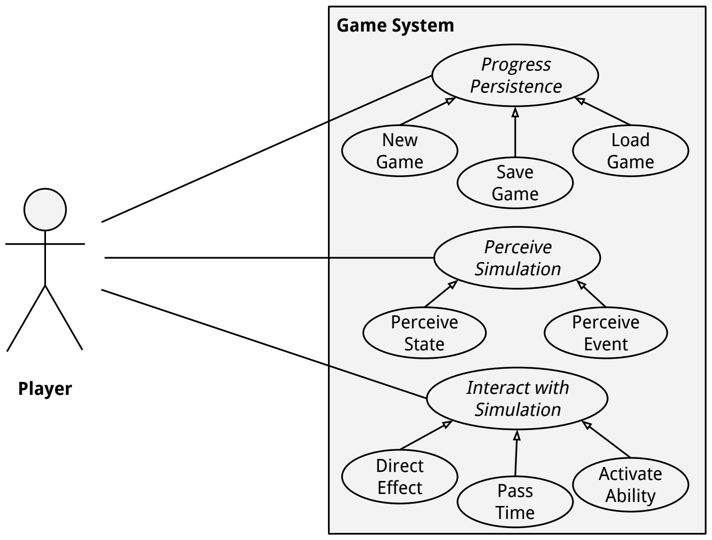


Figure 5.3: Player interactions with a Game System. These use cases guide us when designing the major execution control flows between Subsystems, Services, the I/O Infrastructure, and the Simulation State of the game.

The second common use case for Players is *Perceive Simulation*, which, in terms of the economy mechanics and their simulations, is specialized into two more specific use cases: Perceive State and Perceive Event. Perceive State happens when the Game System presents the current, real-time Simulation State by querying the data it needs to produce audiovisual representations every frame. Perceive Event happens when a relevant change in the Simulation State is informed to other Subsystems and Services (usually through whatever Event Dispatching Service is available, see Section 5.2.2), and the Game System notifies the Players of the change somehow (e.g., by starting an animation or sound effect).

The third and last generic use case for Players is Interact with Simulation, which specializes in Direct Effect, Activate Ability, or Pass Time. Direct Effect is used for straightforward interactions, like moving an avatar with an analog controller. For economy mechanics, though, Activate Ability is a more common interaction because it offers a more robust and data-driven means of queueing Effects into the Simulation. This is what happens when the Players use a spell from their characters, choosing where or who they want to affect it with. Even in an action game like a first-person shooter, shooting with a gun might Activate an Ability if developers want to insert custom Effects into that action as part of the creative process. Lastly, Pass Time is an interaction that represents the fact that Game Systems have real-time simulations and, even when the Players are not inputting any commands into the software, changes still happen to the Simulation State. Section 5.2 details how the Player's use cases translate into runtime execution and communication between architectural elements.

As a reference architecture, the *Unlimited Rulebook* also reserves concepts for the "use cases" of the developers, from both the technical and creatives teams. That is, the architecture of the Game System has both teams as its "clients". Figure 5.4 illustrates the main types of use cases architects must keep in mind when designing for their teams.

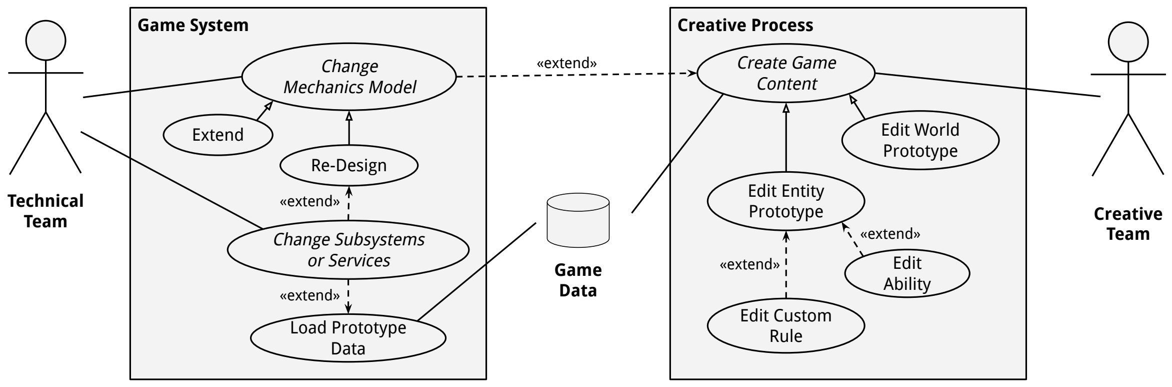


Figure 5.4: Interactions between the Creative Team, the Technical Team, the Game Data, the Creative Process, and the Game System. These use cases guide us when designing the different levels of extensibility the Game System should have to accommodate the demands of the development process.

Here the *Unlimited Rulebook* uses the creative process as the starting point for understanding use cases. In that sense, the Creative Team's main use case is to Create Game Content. Broadly speaking, there are two types of content creation we care about: Editing Entity Prototypes and Editing World Prototypes (we discuss the Prototype concept in more detail in Section 5.2.2). Both provide the Game System with the data that determines the initial state of Entities and

Worlds. Editing Entity Prototypes might also involve Editing Abilities and Editing Custom Rules, which are trickier content types to expose to the Creative Team. Whatever content is produced, it is stored as Data for the Game System to load at runtime (as per the Data-Driven Design domain concept). However, sometimes the development tools available are not enough to achieve the kind of content the Creative Team wants, which is when the use case must be extended with an intervention from the Technical Team.

The Technical Team has two general types of use cases: Changing the Mechanics Model and Changing Subsystems or Services. When demand from the Creative Team requires Changing the Mechanics Model, ideally we want the change to simply Extend the current implementation of the Mechanics Model. In other words, it should involve mostly adding code to a constant number of entry points in the codebase and it should not break any dependencies that would ripple changes throughout the architecture. When a change breaks dependencies and requires reworking the relationships between architectural elements, we call the use case Re-Designing, and it is the worst case we most want to avoid through the *Unlimited Rulebook*. Changing Subsystems or Services is what the Technical Team usually does to provide interactive features to the Game System (e.g., improving the rendering pipeline). However, sometimes the Re-Design of the Mechanics Model also requires this use case since Subsystems and Services might depend (i.e., be coupled to) certain aspects of the current implementation of the Simulation State. Changing Subsystems and Services might also be needed when new types of Game Data come from the Creative Process and the Game System does not know how to load them into Prototypes (e.g., implementing new data structures to store a new type of data that is more complex than the types used until then).

5.1.3 Variability

Since reference architectures support a family of systems in a certain domain, there is always some degree of variability they allow. Here, we present the main variabilities of the *Unlimited Rulebook*, which will influence how architects translate their games from our reference model into a particular architecture. Section 5.2, and Section 5.3 reference these variabilities whenever key design decisions are involved. The variabilities listed here are in no particular order.

Time Mechanics

How does time pass inside the game simulation? For many games, it simply follows the user's time, i.e., it runs in Real-Time, with some pausing, fast-forwarding, and/or slow-motion features. Other games, however, have a variety of asynchronous simulation timelines which we will broadly refer to here as Turn-Based — in the sense that the simulations take turns to progress time, waiting for external inputs when necessary. Turn-Based Game Systems have more leeway with performance but usually require Simulation-Wide FSMs (RSP-4). Some Game Systems mix multiple types of Time Mechanics.

Deterministic Simulation

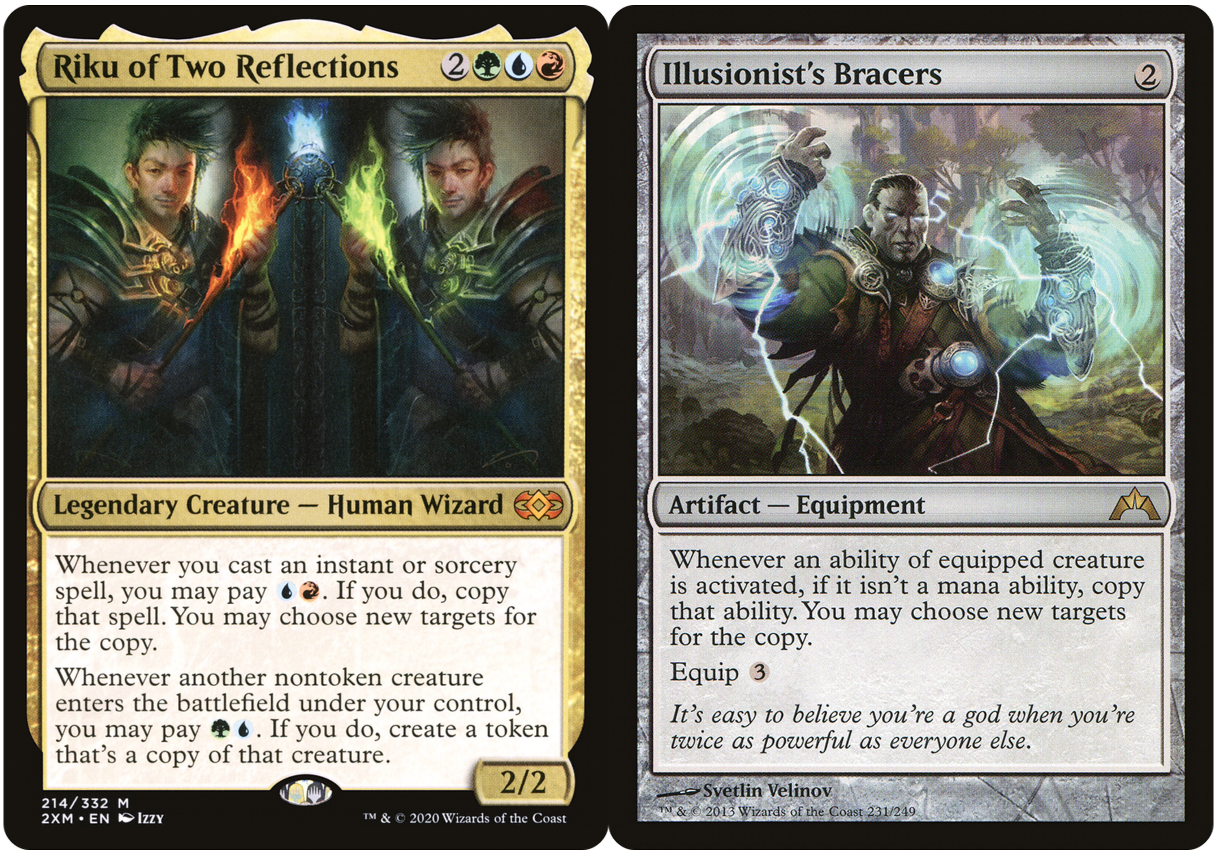
Does the simulation have to be deterministic? To what degree? There are a few reasons why architects might want their Game Systems to have deterministic simulations. One reason is that it enables a simpler network synchronization mechanism for online multiplayer games (like in the game from interview I03, refer to Appendix A). Another reason is that it makes the simulation reproducible: developers can replay a scenario that leads to a crash and debug the Simulation State step-by-step, or they can check if a player's save file has not been tampered with by evaluating whether the Simulation State it describes could be achieved from its initial state, or you can simply provide Players the feature of watching replays of their gameplay sessions. A third reason is that the "more deterministic" the simulation is, the easier it is to avoid inconsistent states caused by unreliable execution order of routines, and the safer it is to implement Behaviors through Data-Driven Design since the simulation becomes more robust. Overall, a Deterministic Simulation improves software quality at the cost of using more complex design constructs and patterns.

Action Protocol

What kinds of Abilities are there? What parts of the Simulation State do they interact with and how is the Player or Game System supposed to refer to them? This question is one of the main intersections between simulation and interaction. To represent the simulation subjects affected by an Ability, we must use abstractions that refer to elements (e.g., Entities) inside the Simulation. These "Ability parameters" can be as simple as virtual coordinates indicating where the Player wants to aim their next spell and as complex as targeting the Effects of a different Ability. In the case of Triggered Abilities, their parameters must be provided by the trigger mechanism itself (e.g., the Ability triggered by a trap Entity must know what Entity triggered it). Figure 5.5 shows two examples of Abilities that target other Abilities, require multiple types of Player interaction, and could be considered complex to design for.

Self-Amending Mechanics

What Behaviors in the Simulation State are subject to amendment by other Behaviors? What kinds of amendments should be possible? To describe software design and its possible implementations, we say that Behavior A amends Behavior B if the occurence of Behavior A requires code to handle an unforeseen special case of Behavior B. For example, suppose a role-playing game has a typical healing spell whose basic Behavior is to remove damage from creatures. Imagine then that the Creative Team wants the game to have "undead creatures", that take damage from healing spells instead of losing damage. In other words, being "undead" amends the Behavior of healing. The more Self-Amending Mechanics the Creative Team expects to rely on, the more complex and consistent the architecture of the Behavior Model has to be to reduce their costs. Essentially, more of the Behaviors will have to be implemented through Rule specializations. This makes the cost of adding simpler mechanics more expensive since every new kind of Simulation State change added requires at least one new Rule to be written. However, when a new Behavior is intended to amend a previously implemented one, the effort needed is greatly reduced. The cockatrice case from Chapter 1 is an example of this, since its petrifying Behavior amends multiple touch-based Behaviors and, in this case, would only require adding new Rules instead of finding and changing already implemented Rules.


Riku of Two Reflections 2

Legendary Creature — Human Wizard

Whenever you cast an instant or sorcery spell, you may pay . If you do, copy that spell. You may choose new targets for the copy.

Whenever another nontoken creature enters the battlefield under your control, you may pay . If you do, create a token that's a copy of that creature.

2/2

214/332 M
2XM • EN Izzy

™ & © 2020 Wizards of the Coast

Illusionist's Bracers 2

Artifact — Equipment

Whenever an ability of equipped creature is activated, if it isn't a mana ability, copy that ability. You may choose new targets for the copy.

Equip 3

It's easy to believe you're a god when you're twice as powerful as everyone else.

Svetlin Velinov

™ & © 2013 Wizards of the Coast 231/249

Figure 5.5: Two cards from *Magic: the Gathering* (Wizards of the Coast, 1993). The card on the left, "Riku of Two Reflections", has a Triggered Ability that requires *at least* three different types of parameters. First, it requires an "instant or sorcery spell" the Player just cast. Second, it requires a confirmation from the Player in the form of a conditional payment in "you may pay [. . .] If you do". Third, it needs all the parameters the copied spell requires (such as targeting a creature on the battlefield). The card on the right, "Illusionist's Bracers", has a similar Triggered Ability. However, its first parameter must be another activated Ability (instead of a spell, which is usually a whole card), and it always happens as long as the triggering Ability fits its criteria. Note that both these cards could potentially copy Abilities that also copy other Abilities, requiring a *variable* number of parameters that can only be determined as the Player fills them, one by one.

State Modifier Mechanics

Do Entities have Fields that are derived from the state of other Fields? How? As mentioned in (ROM-3), sometimes parts of the state of an Entity are procedurally derived from other parts of its state. A very common example is character equipment in role-playing games: the state of the pieces of equipment an Entity carries influences the state of its combat statistics. The more complex and flexible the Creative Team wants these State Modifiers to be, the more forethought has to be put into designing the interfaces through which Subsystems and Services consult an Entity's state since that will be an unavoidable dependency spread across the entire Game System.

Game Object Model

What kinds of Entities are there? Will there be more? The greater the variety and need for mixing Entity types there are, the more flexible and expensive it is to implement the Game Object Model. At the same time, if the plan is to keep expanding the Game System even after the initially envisioned set of Entities is implemented, the more the team will benefit from designing the Game Object Model upfront.

Development Technology

What engine will the Game System be written in? What other libraries and frameworks are available or needed? This can both lessen or increase the burden of the software architects. Sometimes game engines already come with reliable Game Object Model implementations, Event Dispatching Services, among other features that greatly shorten the time needed to have the Game System up and running. Sometimes, however, the technological restrictions (e.g., limitations in the target platform hardware or programming language used) require additional effort to be invested to fully exploit the potential of using the *Unlimited Rulebook*.

Development Methodology

What is the release plan? What is the business model? What are the priorities? Understanding the context of the Game System under development helps determine a reasonable scope of its architectural design. The *Unlimited Rulebook* is particularly concerned with the *release cycle* since that dictates for how long the Creative Team will keep adding new mechanics to the game. Wang and Nordmark (2015) show evidence that not only the Creative Team but also the *management team* of a game should have a say on expensive architectural decisions.

Team Profile

How much workforce do the Technical and Creative Teams have? What tools do they have experience with? If the Creative Team is comfortable with scripting, for instance, that reduces the need for developing in-house graphical editors for Entity Prototypes. Understanding the capabilities of the team behind the Game System allows the architect to better prioritize their efforts, focusing on the tools and parts of the design that will benefit development the most.

5.2 Runtime Viewpoint

In this section, we describe the *Unlimited Rulebook* architecture from the viewpoint of how Game Systems designed with it should operate during runtime. Since the focus of the *Unlimited Rulebook* is the economy mechanics implemented in the Game System, the Runtime Viewpoint of this reference architecture is mostly about how Subsystems and Services interface with the Simulation State and what are their expected execution paths. To recapitulate, while Subsystems implement core features of a Game System that tie directly into the Game Loop and often depend on the particular requirements of a game (e.g., specific graphics pipelines, mechanics, networking interfaces, etc.), Services implement lower-level, general features that are common among different

Game Systems (at least in the context of games with an emphasis on economy mechanics). Some Services encapsulate sequences of operations over the Simulation State and some encapsulate the means for communication between different Subsystems. Using these archetypical Services as building blocks, the *Unlimited Rulebook* guides the design of the internal relationships of a Game System. In practice, the exact Services a Game System needs will vary. That said, Section 5.2.2 presents a core set of Services most games will likely need. However, to understand the role of each Service, we must first establish what parts of the Simulation State are publicly exposed to other parts of the Game System Section 5.2.1. We finish discussing the Runtime Viewpoint by describing how the overall simulation of game mechanics works in Section 5.2.3.

5.2.1 Simulation State Access

Figure 5.2 in Section 5.1.1 shows us how we conceptually organize the data that makes up the Simulation State. We chose the term Simulation State because it suggests that this architectural element is responsible for storing information but not for implementing operations more complex than simply reading or writing values to itself (e.g., it might contain setters and getters, but not the routines that fully implement a simulation feature). Instead, it *supports* a number of operations through the methods of access it exposes while the actual implementation of these operations lies in the Services of the Game System, so that any Subsystem or Service may reuse these operations as they see fit. That said, this is only a conceptual and didactic distinction to help determine how simulation data is shared between different parts of the Game System. We group the methods of accessing the Simulation State into two categories: Direct Access and Rule-Mediated Access. Figure 5.6 shows the different methods of access discussed in this section.

Direct Access involves reading and writing directly to World instances of the Simulation State. Since they carry all Entity instances and the Time Schedule data for simulating time, they form the central repository of data of the Simulation State. Accessing World instances is a low-level operation, where Subsystems and Services can read and write data to the Simulation State by interacting directly with Entity states. For that reason, it is also the method most coupled to what types of Entities exist and how they work. For instance, a specific Simulation State design might have a vehicle Entity type that has a "speed" Field. If later in development we want the speed of vehicles to be reduced by the amount of weight they transport, all Services and routines that accessed that attribute must now decide whether they should also check the luggage of that vehicle Entity. On the other hand, accessing the speed value of a vehicle Entity is always straightforward: programmers just look up the Entity instance in the World and inspect its "speed" Field.

Rule-Mediated Access is a higher-level access method that offers flexibility and extensibility for the Creative Process but requires more steps to be used. In particular, they support Custom Simulation Rules (RBM-4), i.e., the way they are handled varies according to what Rule-type Behaviors from Entities are currently in effect. Direct Access, on the other hand, simply bypasses all that and provides immediate access to the data of the Simulation State. There are two layers to Rule-Mediated Access: the Effect layer and the Ability layer. The Effect layer offers the simplest access method, it simply requires the programmer to create Effect instances that describe the change they want to make to the Simulation State then find and apply the relevant

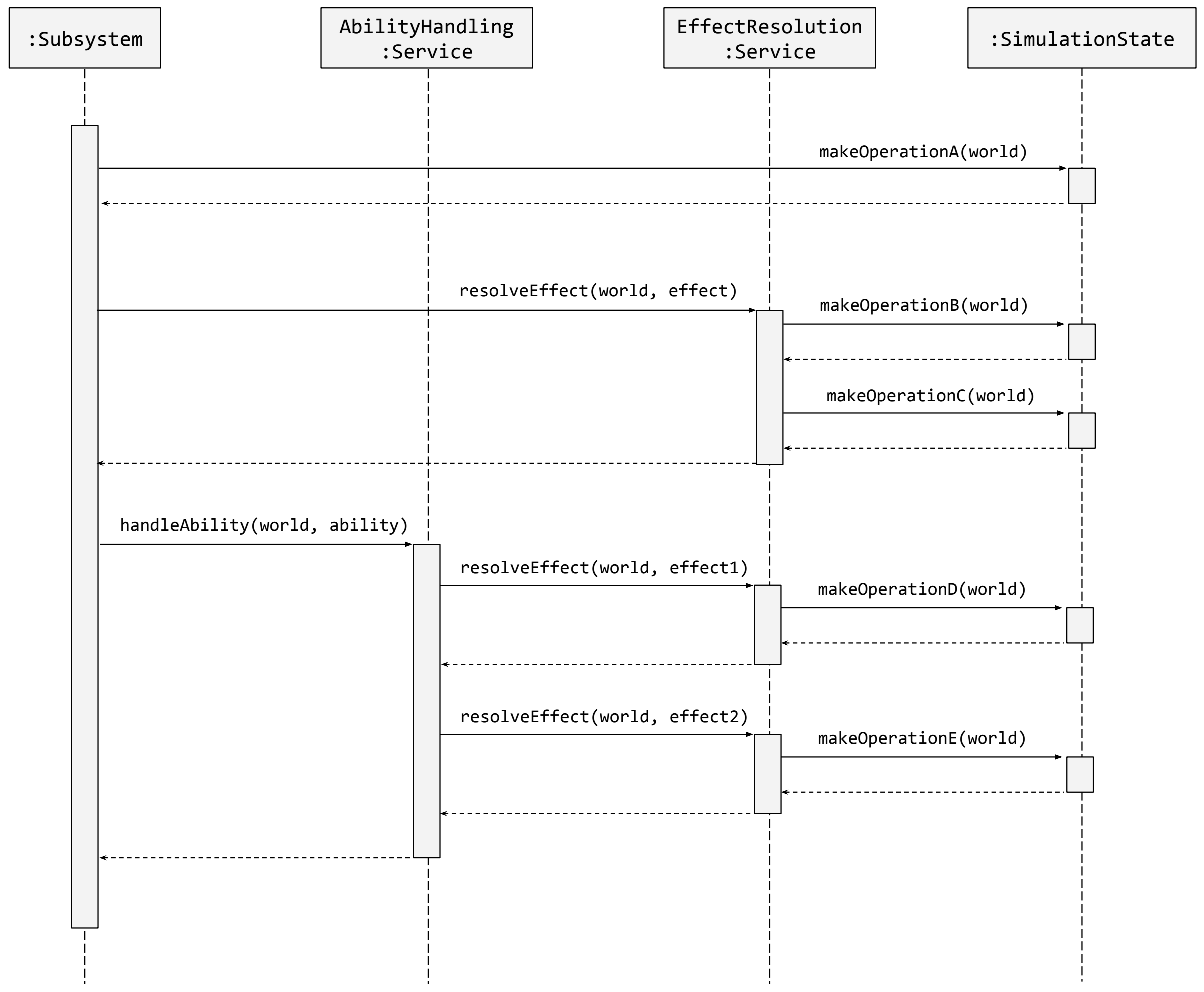


Figure 5.6: Different methods of access to the Simulation State. In the diagram, a hypothetical and generic Subsystem first makes a Direct Access, followed by two Rule-Mediated Accesses. The first Rule-Mediated Access uses only the Effect layer via the Effect Resolution Service. The second Rule-Mediated Access uses both Ability and Effect layers, additionally relying on the Ability Handling Service.

Rules — this is called Effect Resolution, and we will present the Service that implements it in Section 5.2.2. The Ability layer adds a second level of complexity to the access of the Simulation State. This method requires the use of Abilities, which encapsulate the process of producing the desired set of Effect instances so that Entities can store these Abilities as Behavior Fields. For this reason, using the Ability layer requires multiple steps: creating an Ability, which creates Effects, which, in turn, passes through an Effect Resolution process before finally providing the desired access to the Simulation State. In other words, going through the Ability layer requires going through the Effect layer too. Like with the Effect Layer, Game Systems can employ an Ability Handling Service to provide reuse of this process.

On one hand, Rule-Mediated Access via the Effect layer is useful for specific needs of Subsystems and Services that decide Direct Access would make them too coupled to the Simulation State because it is a one-time effort that can still be changed as needed. On the other hand, access via the Ability layer is useful for Simulation State operations whose details the Subsystem or Service does not know since they can simply delegate those details to the Ability. An example would be drinking a potion in a role-playing game: the code that handles this cannot determine the exact Effect without checking all possible cases, so it can instead associate an Ability to the potion Entities and rely on them for specifying the desired Effects. The *Unlimited Rulebook* proposes these different methods of access to leverage different needs and circumstances during the development of the Game System, balancing flexibility, extensibility, and reuse.

5.2.2 Core Services

As explained in Section 5.1.1, each Service is responsible for a specific set of common operations, both within a single Game System and across different Game Systems. The *Unlimited Rulebook* assumes this encapsulation and separation of concerns to help illustrate what features a Game System might need according to its design variabilities. To do this, our reference architecture establishes the **Core Services** we believe are likely needed for the creative process of games that focus on economy mechanics and what the role of each of these Services is during the execution of a Game System. This section describes the Core Services and how the system as a whole relies on them. We divide the Services into two broad categories: Simulation Support Services and Inter-System Services.

Simulation Support Services

These Services essentially automate the Simulation State access methods described in Section 5.2.1. They read the Simulation State and operate on it following the constraints and protocols dictated by the mechanics of the Game System. The actual use of some of these Services is more clearly illustrated in Section 5.2.3 when we discuss the overall execution of the Game System simulation.

Query Processing. Though operating over the Simulation State using Direct Access is straightforward, Game Systems often use a number of more specific but recurring queries. For instance, listing all Entity instances that have a certain characteristic (e.g., "all creatures currently affected by poison") so that an operation that only applies to them can be made (e.g., "check whether any creature managed to cure itself of poison"). The Query Processing Service is mostly responsible, thus, for finding and listing Entity instances and/or the corresponding Fields that are of interest to other Services and Subsystems. In other words, it encapsulates query-like Direct Access operations to provide reuse and, sometimes, decouple Subsystems and Services from Direct Access methods by adding a level of indirection.

Effect Resolution. This Service is responsible for interpreting Effect instances to apply the operations they represent over the Simulation State. To do this, the Effect Resolution Service determines what Rules currently apply to a given Effect (using their Predicates), adjudicates what Rules have priority over others, then applies them (using their Resolutions). It might rely on the Query Processing Service to find Rule instances but otherwise mostly operates over the Simulation State using Direct Access.

Ability Handling. Since Effects describe the entirety of the operation they represent (e.g., "an explosion should happen in these exact coordinates"), we have to create new instances to apply them more interactively (e.g., causing explosions in the coordinates the player is currently pointing at). We also do not always need to know the exact Effects caused by an Ability, since that might depend on the magic spell, item, skills, cards, or whatever Entity the player uses to carry out the interaction. Thus, the Ability Handling Service takes both Ability and interaction data (e.g., player input) as arguments and uses them to produce the exact Effects that represent the outcome of that interaction. However, as we explained in Section 5.1.1, not all Abilities come from explicit interactions from outside the simulation: Triggered Abilities activate due to some Effect in the simulation, and detecting their triggers and handling their execution is also part of the Ability Handling Service's role.

Progress Tracking. The Time Schedule of the Simulation State is managed by this Service (RSP-1), which can be accessed by Subsystems to ensure synchronization with the Game Loop (e.g., advance a Turn-Based simulation when the player passes their turn) and by Services to, for instance, detect when the Simulation State has progressed significantly (RSP-3) (e.g., check if a timer has run out of time). The Progress Tracking Service might use the Effect Resolution Service with some form of periodicity to produce continuous processes in the Simulation State (RBM-1).

Inter-System Services

These are Services that provide an interface between the Simulation State and other parts of the Game System, though some might be used without interacting with the Simulation State as well. Inter-System Services allow, in particular, the interoperation between the Simulation State and the I/O Infrastructure.

Prototype Loading. Programmers can create new World and Entity instances into the Simulation State using any of its access methods but, by doing so, they must manually enter the exact values for the Fields of each created Entity or World. This also makes programmers a bottleneck for the Creative Team to add and change the mechanics of the game. Instead, what most games and engines do, is draw the values that specify World and Entity instances from the Game Data (RDD-1), which the Creative Team has more access to. More specifically, these systems establish a type of Game Data known as **Prototype** (RDD-2) that represents the initial state of a possible World or Entity instance. The Prototype Loading Service processes these Prototypes and creates the corresponding instances in the Simulation State.

A single Prototype (e.g., representing a tree Entity) may produce any number of instances (e.g., is used to fill up an entire forest in the World with tree Entities that are all created with the same Field values). After their creation, however, the World and the Entity instances that came from a Prototype are likely to diverge and each follows its own sequence of states over the simulation. The Prototype Loading Service also functions as an adapter to whatever data storing format the Game System relies on (RTC-3).

The access method used by this Service to create World and Entity instances may be either Direct Access or Rule-Mediated Access. The latter is preferable when even the insertion of new Entity instances into the Simulation State might be subject to Rules. For instance, in *Warcraft 3* (Blizzard Entertainment, 2002), creating new units for the Player's army is closely regulated by how much food you can provide them, and simply adding a new unit instance could break those rules. Figure 5.7 shows how the Prototype Loading Service interfaces between the Simulation State and the I/O Infrastructure — in this case, the Service uses Direct Access for simplicity.

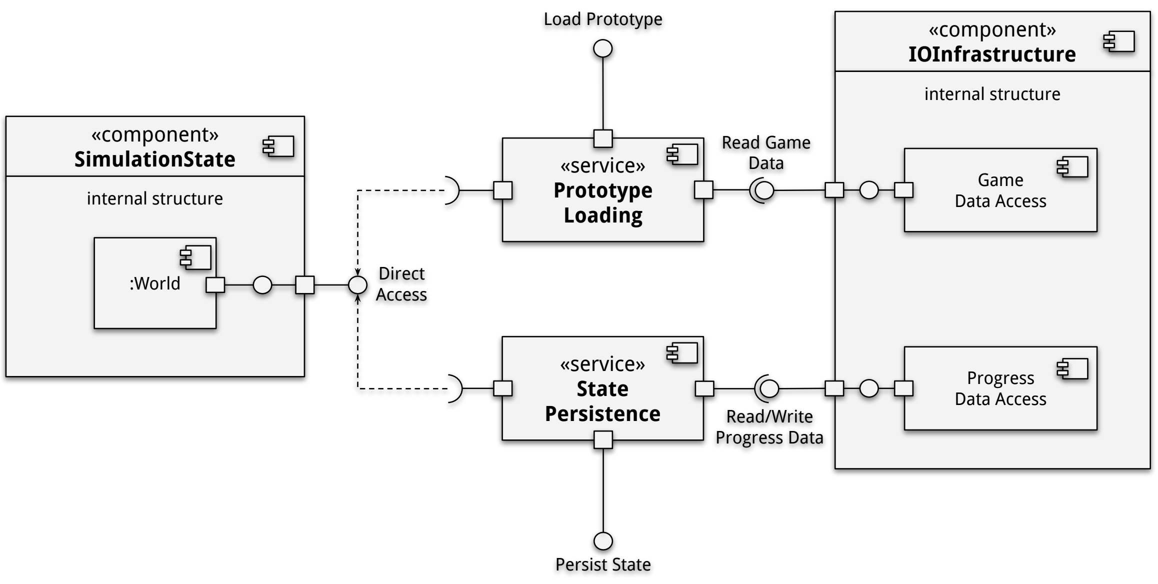


Figure 5.7: Prototype Loading and State Persistence Services interface between the Simulation State and the I/O Infrastructure. For clarity, the diagram simplifies some assumptions (like the access method used by the Prototype Loading Service).

State Persistence. When the progress a player has made while using the Game System must be kept for future executions, parts of the Simulation State need to be stored and written to disk then later read and loaded back into the states they had previously. This Service abstracts both these procedures (RRL-2). Different from the Prototype Loading Service, State Persistence is more likely to use Direct Access into the Simulation State because being able to restore a Player's progress is a usability feature, not a mechanics feature (and, thus, probably dispenses adjudication of simulation rules). The persisted data this Service accesses via the I/O Infrastructure is also not Game Data produced by the Creative Team. It is *user data* that the Game System stores in the end user's machine, instead of data produced at development time. Depending on how much of the Simulation State is persisted, using the State Persistence Service might require the specification of *Stable Simulation Checkpoints* when it is safe to either capture or restore the Simulation State without incurring in an inconsistent state. Figure 5.7 also shows how the State Persistence Service interfaces between the Simulation State and the I/O Infrastructure.

Event Dispatching. Any part of the Simulation State can notify Subsystems and Services that a relevant operation occurred by sending an **Event** via the Event Dispatching Service (RIC-2). In turn, Subsystems and Services can register their interest in being notified of specific Events using this too. This way, a UI Service can, for instance, be notified when an Entity takes damage and prepare to add an animated text on the screen telling the Player how strong that damage was. Figure 5.11 in Section 5.2.3 gives a more explicit example of the Event Dispatching Service complementing other forms of communication with the Simulation State.

5.2.3 Subsystem Activity

A Game System may have any number of Subsystems but, to implementing economy mechanics, there are three main typical Subsystems the *Unlimited Rulebook* proposes: Input Processing, Simulation Update, and Output Rendering. These are the most common steps in the Game Loop (RRL-1). At the same time, games often display different Interaction Modes that change how Input, Simulation, and Output behave over the execution of the Game System (RRL-4).

Figure 5.8 depicts an **Input Processing** Subsystem at work in very broad terms. When the Game Loop invokes this subsystem, it first gathers the current state of all relevant Input Interaction Devices (or just Input Devices), such as what buttons are currently pressed, what is the current axis position of analog controllers, what is the current angle the gyroscope detects right now, etc. Then, based on the current Interaction Mode, the Input Processing Subsystem determines what kind of interaction the user is trying to do with the Simulation State. However, instead of directly accessing the Simulation State to perform that interaction, the Input Processing Subsystem merely *communicates* the **Simulation Update** Subsystem of the intended interactions.

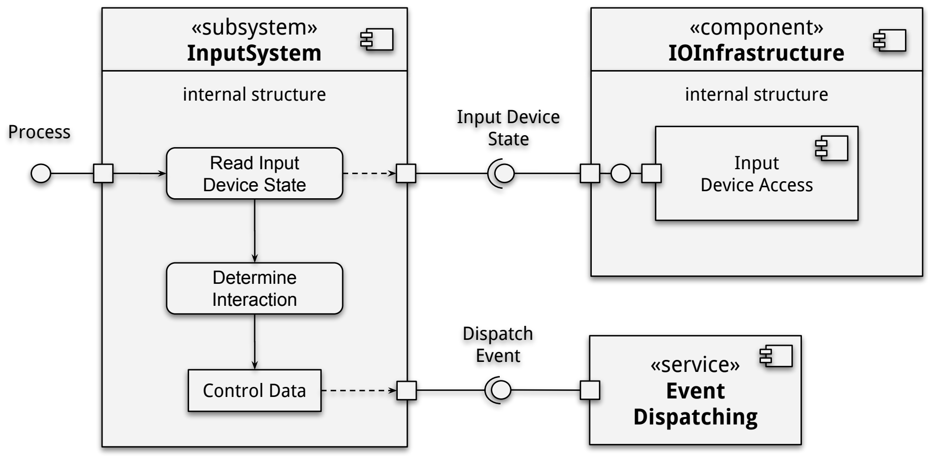


Figure 5.8: The basic proposed behavior of the Input Process Subsystem. The elements shown in the internal structure of the `InputSystem` component form a "nested" Activity Diagram, while the rest works just like a usual Component Diagram. The Input Process Subsystem reads data from Input Devices using the I/O Infrastructure, computes how that input data translates into control data for the simulation, then sends the control data via the Event Dispatching Service.

In Figure 5.8 and Figure 5.9, this is represented by the use of the Event Dispatching Service

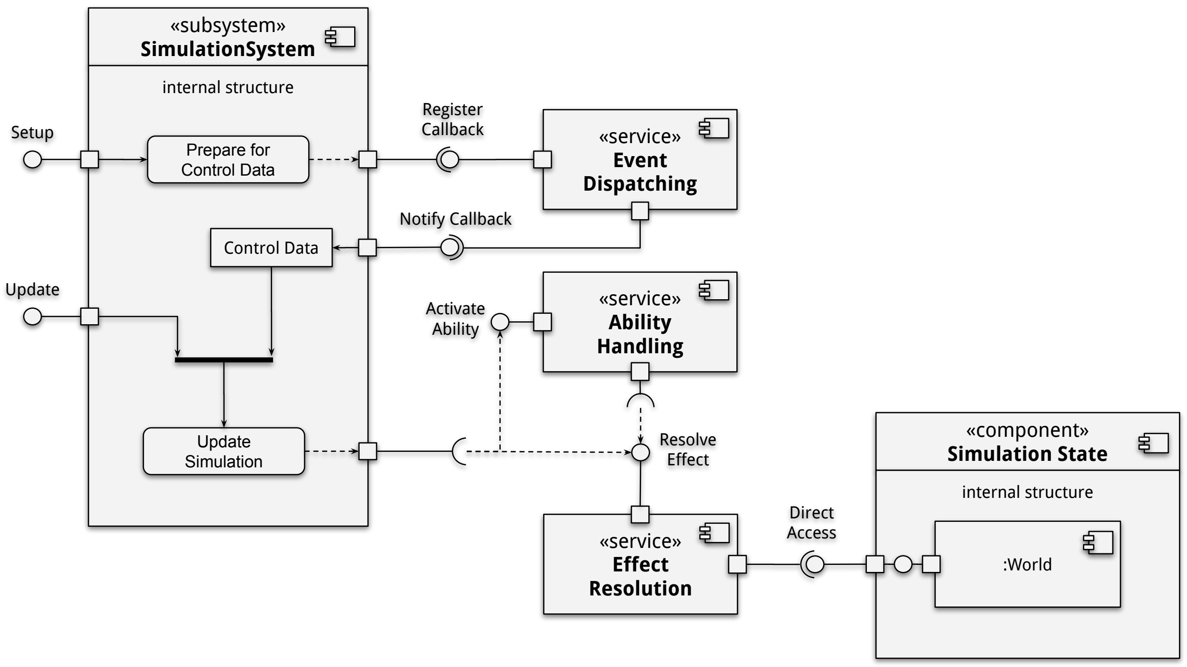


Figure 5.9: The proposed behavior of the Simulation Update Subsystem. The elements shown in the internal structure of the `SimulationSystem` component form a nested Activity Diagram, while the rest works like a usual Component Diagram. The Simulation Update Subsystem registers for incoming events containing control data when the Game System first starts up. After that, whenever such data is received, it waits until the next time the Game Loop invokes it to use that control data (even if it is empty because the Player made no inputs during that frame) to update the Simulation State using the Ability Handling and Effect Resolution Services.

but, in practice, any way of sending the control data between these Subsystems is valid. By not directly interacting with the Simulation State, this approach to the Input Processing Subsystem decouples the Simulation Update Subsystem from it. That is, the Simulation Update Subsystem can more safely assume that it is the only Subsystem to cause changes to the Simulation State, since all user interactions pass through it. This can be used to increase the determinism of the simulation, because the Simulation Update Subsystem may now decide the exact order in which to compute the mechanics of the simulation. This Subsystem, in turn, relies on the Ability Handling and Effect Resolution Services discussed in Section 5.2.2 as shown in Figure 5.9.

Figure 5.10 gives a more detailed picture of how the Simulation Update Subsystem works. First, as soon as the Player requests to either start ("new game") or restore ("load game") a simulation, the Subsystem prepares the Simulation State accordingly and proceeds to wait for update calls. This "standby" state composes the Stable Simulation Checkpoint which, as we saw in Section 5.2.2 regarding the State Persistence Service, might be a necessary mechanism. The Simulation Update Subsystem always comes back to the Stable Simulation Checkpoint after its work is done for the current game frame. In the case of Turn-Based simulations, it might spend multiple frames without moving from that state when it is the Player's turn and there is no control data to proceed with the simulation yet. When it is time to advance the simulation, as we saw in Figure 5.3 from Section 5.1.2, there are three possible scenarios.

The first scenario is when time simply passes in the simulation. It could be a Real-Time

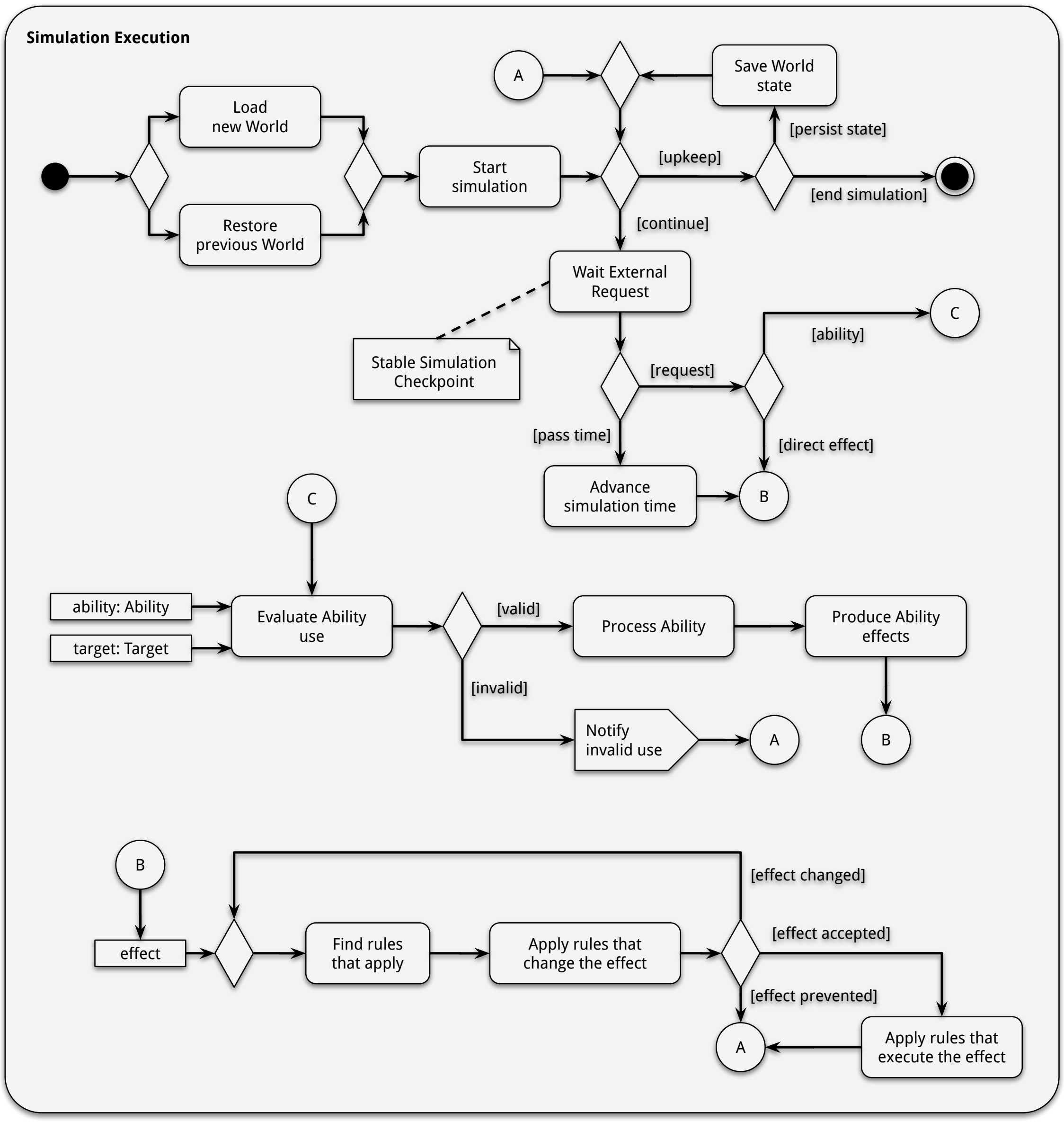


Figure 5.10: Execution flow of the Simulation Update Subsystem across multiple Game Loop frames. It is didactically divided into three sections. The first, in the upper part, represents the basic behaviors of the Subsystem over its lifecycle. The second, in the middle part, shows how the Subsystem and the Ability Handling Service activate Abilities in the simulation. The third, in the lower part, demonstrates how the Subsystem and the Effect Resolution Service resolve the Effects that are applied to the Simulation State.

simulation where the simulation advances every frame, or it could be a Turn-Based simulation and the simulation moved from one actor's turn to the next. Whatever the case, the Subsystems produces one or more Effects representing the passage of time and uses the Effect Resolution Service to apply them to the Simulation State (following from the connector B in Figure 5.10). The second and third scenarios happens when the control data for that game frame contains directives to act over the Simulation State. Depending on the mechanics involved, the Subsystem could either directly produce the corresponding Effects (and go through connector B as well) or delegate that to a corresponding Ability using the Ability Handling Service (following from

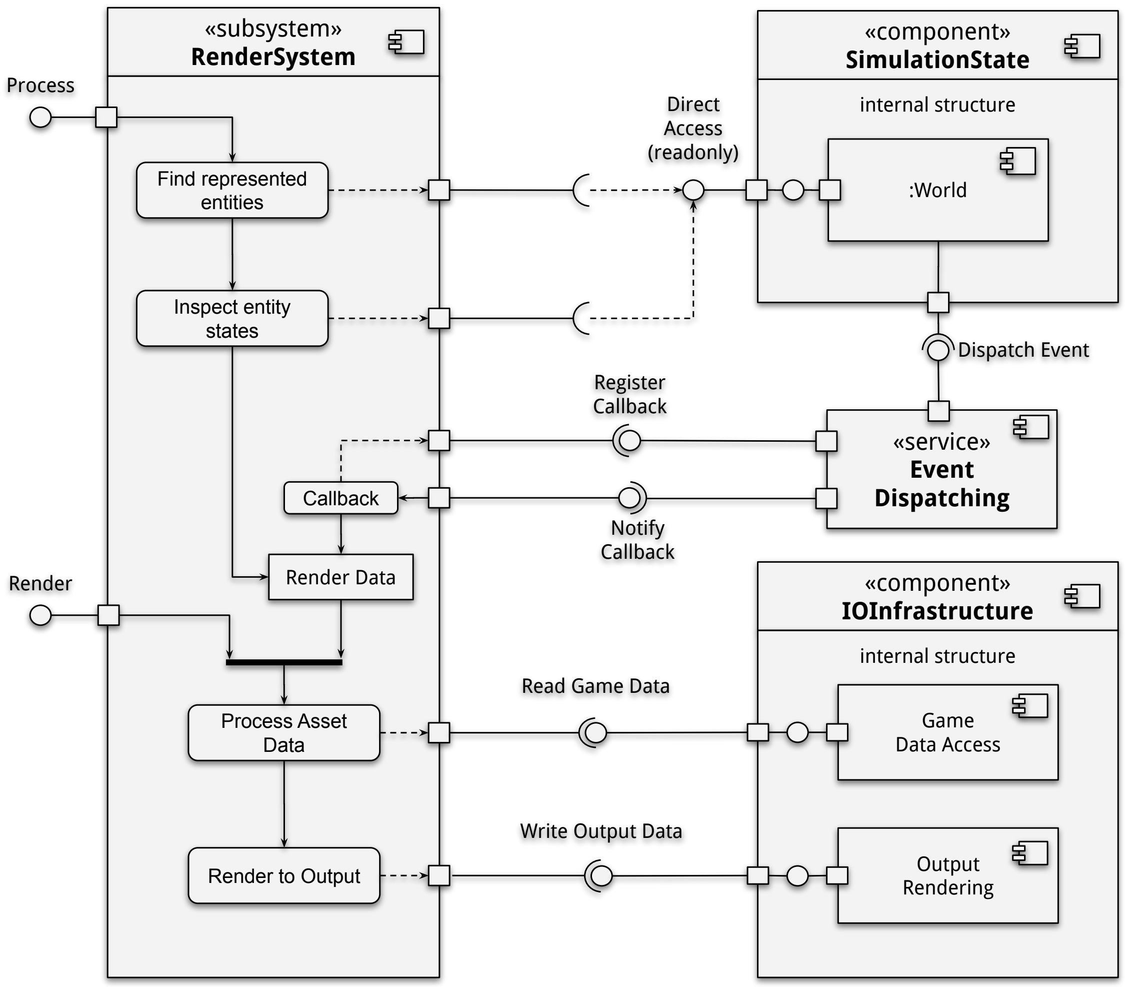


Figure 5.11: The proposed behavior of the Output Rendering Subsystem. The elements shown in the internal structure of the `RenderSystem` component form a nested Activity Diagram, while the rest works like a usual Component Diagram. The Output Rendering Subsystems lists all currently perceptible Entities by the Player then inspects their state to produce render data. Events from the Simulation State add to that data using callbacks from the Event Dispatching Service. Lastly, to render the result, the Subsystems gathers all render data and loads any necessary asset from Game Data before sending the result to an Output Device.

connector C). There might be multiple Effects and Abilities processed in a single frame and the Simulation Update Subsystems should determine how many of them it completes before finishing its work for that frame. In particular, by resolving some Effects, Triggered Abilities might come up, which would create a new Ability request (following from connect C again). Some time between each full resolution, the Subsystem might enter an upkeep phase where it checks if the simulation has come to a terminal state and/or whether it should persist its state so the Player can restore it later (using the State Persistence Service). Note that State Persistence always happens right before the Stable Simulation Checkpoint, so that the simulation is always restored to a state where it can promptly restart running.

Lastly, Figure 5.11 shows how an Output Rendering Subsystem interacts with the Simulation State and the Event Dispatching Service to produce the data for displaying the current state of the simulation and send that data to the appropriate Output Interaction Devices (or just Output Devices). Though the Subsystem uses mostly Direct Access, it only *reads* data from the Simulation State. If necessary, it could also use Rule-Mediated Access to preview information about Effects that might be applied. Based on the data collected from the Simulation State through Direct Access, Rule-Mediated Access or events received from the Event Dispatching Service, The Output Rendering Subsystem gathers all the render data it needs for that frame. Then, when the Game Loop asks the Subsystem to render its output, it processes that render data, loading any assets from Game Data as needed, then sends the result to the Output Device it is responsible for. Note that this workflow was kept generic because it can be used for graphics or audio rendering, or (theoretically) any other form of rendering the hardware in question supports.

5.3 Source Code Viewpoint

In this section, we discuss the more in-depth details of how to design the implementation of the individual parts of a Game System using the *Unlimited Rulebook*. We group these system parts in two groups, the first, related to the Object Model, in Section 5.3.1, and the second, related to the Behavior Model, in Section 5.3.2. In each case, we will approach the matter of Iterative Development where appropriate. The designs discussed throughout this section provide architects with alternatives they must choose from when using the *Unlimited Rulebook*. There is no single way to implement the specific parts that make up the economy mechanics of a Game System. The Variabilities considered in Section 5.1.3, in particular, weigh in on these decisions. That is why each of the following topics presents a series of approaches and discusses the benefits and costs of each one.

5.3.1 Object Model Design

In this section we address the Game Object Model and Generality domain concepts of the Mechanics Model presented in Section 4.1. The Object Model concept is implemented by the World, Entity, and Field elements of the reference model discussed in Section 5.1.1 and illustrated in Figure 5.2. However, we refrain from detailing the Behavior specialization of Fields, since that is left for the design of the Behavior Model in Section 5.3.2.

World Design and Entity Management

The Game Object Model is responsible for detailing how the World of the Simulation State is structured, what possible types of objects it can store, what their possible states are, and how they relate to each other. There are three different things the World stores: static simulation data (e.g., spatial geometries), dynamic simulation data (i.e., Entity instances), and simulation time data (i.e., the Time Schedule). The Simulation State does not necessarily keep all the World data in a continuous structure — it can be broken into pieces spread over the codebase. Ideally, however, keeping this data in a contiguous segment of memory supports cache optimization (Gregory, 2019; Nystrom, 2014) and might improve code legibility, because programmers can analyze the entire Simulation State in a single place (West, 2018).

Since much of simulating the virtual world of a game involves processing the data that makes up the Simulation State, how it is stored and accessed becomes a fundamental dependency for programming the mechanics of the game. Thus, the storage mechanism for each part of the World should fit their corresponding characteristics. Static simulation data may begin at varying sizes (e.g., simulating World instances of different sizes) but usually do not change once loaded. Dynamic simulation data not only begins at different sizes but changes during the simulation, sometimes very fast. The size of simulation time data varies the least and is considerably smaller than the other two. When implementing the World, appropriate collection structures (e.g., arrays, look-up tables, etc.) should be taken into consideration according to how each kind of data will change and be accessed over time. It might be reasonable to enforce a limit to how much data a World instance can carry. The exact implementation, however, should be encapsulated by the Query Processing Service, to reduce how coupled other Services and Subsystems are to the data layout of the World (RCE-1).

The implementation of the Time Schedule varies significantly between Real-Time and Turn-Based simulations. With Real-Time mechanics, its role is to track the passage of in-game seconds and cue time-sensitive features such as timers, animations, sound events, etc. In economy mechanics, the main time-sensitive behaviors are temporary Entities and Fields: they are assigned a duration after which they should be automatically disposed of (e.g., eating food in *The Legend of Zelda: Breath of the Wild* gives the Player a power-up for several minutes). In these cases, storing timestamps and associating them with the corresponding Entity or Field should be enough. For Turn-Based mechanics, the Time Schedule is responsible for asserting the order in which Entities make actions inside the Simulation, as well as time-sensitive features that are based on counting turns instead of seconds. This means the Time Schedule should store more information in this case — usually a list referring to Entity instances that represent the turn order. In both the Real-Time and the Turn-Based case, the Progress Tracking Service reads and writes to this part of the Simulation State with a frequency that makes sense for the game, producing time-based Effects accordingly (e.g., a "begin new turn" Effect).

For economy mechanics, we are especially interested in how Entity instances are stored in the World. Most economy-centered gameplay revolves around changing the state of Entity instances to represent the flow of resources in the simulation (Adams and Dormans, 2012; Dormans, 2012a; Rollings and Ernest, 2006). For that, however, Services and Subsystems need to be able to find the right Entity instances in the World structure. Industry authors state that relying on low-level programming features such as pointers and raw variable references is too unsafe for this since Game Systems are very prone to creating stale references (Gregory, 2019; West, 2018). This happens when a pointer, for instance, points to an Entity instance that has been already freed, and maybe some other Entity data has been written where the old instance laid in memory. Keeping copies of pointers or raw references to Entity instances for later use is particularly dangerous since it is easy to dereference them without checking their validity first.

To prevent this, game programmers prefer to use something more indirect than pointers but also lighter than reference-counting mechanisms. This usually takes the form of an identifier (e.g., the index of the Entity instance in an array inside the World structure) but can also use a more robust approach such as a generational index (West, 2018) or smart handle (Gregory, 2019, pages

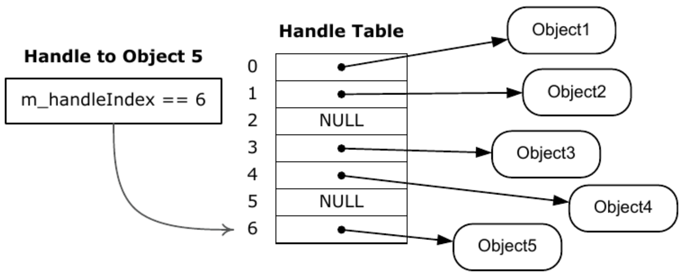


Figure 5.12: A handle-table implementation for References as proposed by Gregory (2019). By adding a level of indirection, programmers are forced to check whether their References are still valid before accessing the Entity instances they refer to.

1082–1085) (see Figure 5.12). Whatever its exact implementation is, the *Unlimited Rulebook* calls it a **Reference** to the Entity instance. Thus, a Reference is a piece of information the Simulation State provides to other parts of the Game System so they can refer to Entity instances *currently present* in the simulation World. It works with whatever Entity storage mechanism the Game System uses to ensure that Entities referred to are valid (ROM-2). Subsystems and Services can safely store References for later use since they *must* always check them against the current World state to see if they are still valid. Since looking up Entity instances using References in the World is a very common task inside a Game System, the *Unlimited Rulebook* once again assigns the Querying Processing Service to reduce both duplicated code and coupling to the implementation details of the World (RIC-3).

Entity and Field Design

Entities make up the dynamic simulation data of the Simulation State. Their individual states may change with every game frame. That state is, in turn, composed of the state of the Fields an Entity has. Fields carry the smallest, indivisible amount of data that has a meaningful purpose in the simulation of an Entity. It could be the position, speed, weight, fuel capacity, or any other simulated value that is part of what represents an Entity digitally. Fields can be simple Primitives (e.g., integers, character strings, etc.), which include simple collections of Primitives (e.g., a list of numbers). Fields can also represent associations between Entity instances. This is done using References. For instance, a character inventory in a role-playing game could be a Field that is a collection of References to the Entity instances of the items they carry (e.g., weapons, potions, armor, etc.). Fields can contain Behaviors as well but we will leave that for Section 5.3.2.

How architects design the implementation of Fields as parts of Entities depends on how it represents different Entity types. Each type has a specific layout of Fields its instances store. We saw in Section 2.2.2 four different design and architectural patterns for implementing "type systems" for Entities of the Object Model:

1. **Inheritance-based Types**, where types are defined using object-oriented classes and inheritance, promoting reuse of common features by moving them into parent classes. It relies on the typing system of the underlying programming language but new types or major changes can only be done by the Technical Team (and can be very expensive, see

I04 in Appendix A). For programmers with less experience in game development, it might be a more didactic approach (Mizutani *et al.*, 2021).

2. ***Entity-Component-System***, where the type of an Entity is determined by the set of *components* or *properties* it has. This pattern decomposes Entities into simpler objects that are combined to produce specific Entity "types". The approach requires some upfront effort but pays off by allowing the Creative Team to create new Entity types by choosing components to compose an Entity via Data-Driven Design. Making new components altogether still requires some form of programming but their reusability reduces the chance of needing more component types each time. Figure 5.13 shows one possible implementation of this pattern.

3. ***Composite***, where an Entity comprises a tree of node objects, each inherited from a base node class. It allows "deeper" structures for Entity types by making more flexible use of inheritance. Works well with graphics pipelines and has the same extensibility characteristics of the *Entity-Component-System*: easy to combine new Entity "types", hard to create new building blocks, but the long-term effort pays off. Since Entities possess a more complex structure, however, referencing each of its nodes is less straightforward than the ECS.

4. ***Adaptive Object-Model***, which completely generalizes object models by combining a series of design patterns, including the *Property* pattern which closely resembles the *Entity-Component-System*. This approach allows virtually all new types of Entities to be implemented by the Creative Team, being fully compatible with Data-Driven Design. It is also the most expensive to implement but Yoder and Johnson (2002), authors of the pattern, explain that developers need not use all the sub-patterns of the *Adaptive Object-Model* — only the ones that make sense for their system. One possible approach is shown in Figure 5.14: the composed design pattern known as *Type Square* (Yoder and Johnson, 2002).

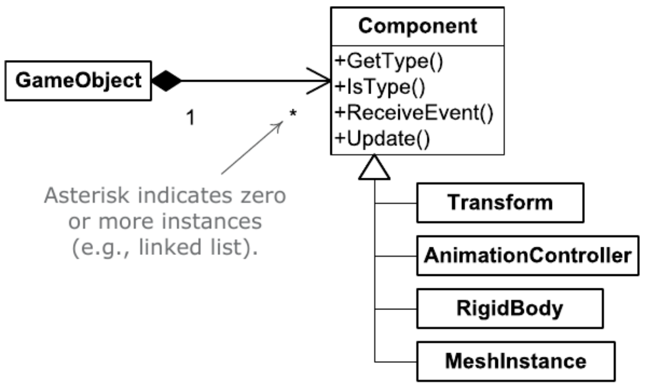


Figure 5.13: A possible implementation of the ECS design pattern as proposed by Gregory (2019). In this case, `GameObject` plus its currently associated `Component` instances make up the "type" of an Entity, and all components classes share a common abstraction so that the `GameObject` can store a list with virtual references to them.

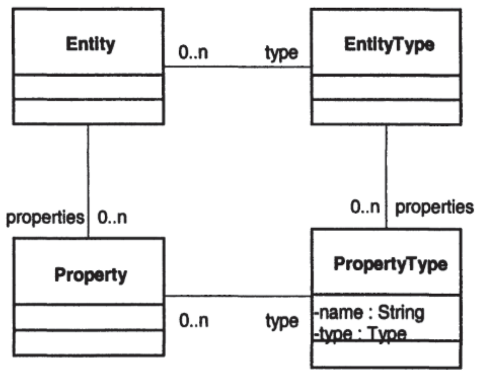


Figure 5.14: The *Type Square* design pattern as proposed by Yoder and Johnson (2002). It uses both the *Property* and *Type Object* design patterns together to allow the run-time definition of new types of objects.

We identify a tendency where game architects favor composition-based patterns over purely inheritance-based ones when they know their games have a considerable volume of content and mechanics. It is especially useful because the Creative Team can make new Entity types without writing code, since the object composition of an Entity instance can be loaded from Game Data (RDD-2), as long as the basic building blocks are already implemented. Architects, thus, increase the extensibility (RCE-2) of the economy mechanics of a Game System by reusing basic objects to make up specific Entity types. This extensibility translates into reduced implementation costs, supporting a faster creative process.

The exact design and implementation of the Object Model where it concerns Entities, Fields, and types of Entities, is left to the architects that use the *Unlimited Rulebook*. There is no silver bullet approach and every method has its downsides. For instance, typical *Entity-Component-System* implementations lack reliable type consistency (Llansó *et al.*, 2011). Here, the Technical Team must discuss with the Creative Team what are the expected types of Entities, their corresponding Fields, and the roadmap of economy mechanics for the Game System they are developing together. This is further influenced by the choice of technologies since game engines might enforce one implementation over others (RTC-1), and probably the business model as well. The decision of the Object Model design is usually hard to change afterward, since any code that directly reads from and writes to the Simulation State will assume that design even if only indirectly. As we will see in the Behavior Model, Abilities, Effects, and Rules try to reduce this, but that only makes them the ones coupled to the Object Model instead.

Data-Driven Design applied to the Object Model

Section 5.2.2 introduced the Prototype Loading Service as a Data-Driven Design method to define Entity and World instances using Prototypes. The general principle of the *Unlimited Rulebook* is to support the creative process of games by reducing the costs of adding and changing economy mechanics. Thus, compatibility with Data-Driven Design should be a key factor in the choice of Object Model design. In this sense, architects using the *Unlimited Rulebook* should consider how their Object Model interfaces with Prototypes.

Adding a new Entity instance to the Simulation State boils down to determining the initial values of its Fields (Gregory, 2019, pages 1062–1063). That is what Prototypes do. How expensive it is for the Creative Team to specify new Entity instances thus depends mainly on how expensive it is to produce new Prototypes. That, in turn, depends on what data format Prototypes specify Field values with. For the creative process, the most expensive way to make Prototypes is to implement them as hard-coded procedures that create and populate the corresponding Entity instance, such as in the *Factory* design pattern (Gamma *et al.*, 1995). It is a straightforward approach that, however, prevents non-programmers from the Creative Team from inserting new Entity types into the Simulation State. Instead, they rely on — and possibly overload — the Technical Team.

The first step into making Prototypes more accessible to the Creative Team, as we mentioned before, is to implement them based on Game Data. It is a similar approach to the *Builder* pattern from Gamma *et al.* (1995). The Prototype works as the builder, storing information that represents an Entity instance. Again, in games, that data is usually the initial value of Fields the Entity has. The Prototype Loading Service works as the director, filling the data of the Prototype. However, it does so by reading that data from the Game Data stored on disk, which the Creative Team has control of. The Prototype is then able to create an Entity instance in the Simulation State.

There are two design issues the *Unlimited Rulebook* leaves for architects to solve here. The first one is how to reduce coupling between Prototypes and specific Entity types. Since each Entity type defines the layout of Fields its instances have, one would expect that the Prototype must know how to populate each specific type of Entity. This can lead to a complex and hard-to-maintain code, as seen in interview I03. One way to solve this, though not all technologies support it, is through reflection: by allowing the Prototype to dynamically determine, at runtime, the Field composition of an Entity type, it can then automate the process of populating instances. This is when the compatibility of the Object Model comes into play. The design used could reduce the cost of coupling Prototypes to Entity types or, hopefully, decouple them entirely. For instance, an *Entity-Component-System* design can distribute the coupled Prototype code between components, reducing its maintenance cost. Alternatively, an *Adaptive Object-Model* using the *Type Square* design may reproduce the *Type Object* pattern for Fields, making the Entity type layout iterable at runtime. More specifically, by specifying Field types via a separate, instantiated object, Entity types can be formally defined as a sequence of “Field type” objects, effectively emulating reflection. Enabling reflection also simplifies the implementation of the State Persistence Service, for similar reasons.

The *Adaptive Object-Model* has a higher upfront development cost, but it pays off during the production phase because reuse of Field types increases. During early production there will be Field types the Game System does not know how to translate from Prototypes. For instance, the programmers might not have foreseen that Entity Fields could be 4×4 matrices of floating-point numbers. Thus, they need to add support for this new Field type so that new Prototypes can be made listing values of that type. This incurs a more expensive extension of the Mechanics Model. How expensive it is depends on the Prototype Loading implementation and the kind of Field involved. That said, the cost of extending Field types becomes rarer the longer a game is in development since it is less likely that more required but unforeseen Field types will keep coming up. In the end, we repeat, the design choice of both Object Model and Prototype design should take into consideration the particular needs of the Game System along with the Creative Team expectations for the economy mechanics.

The second design issue of Prototypes is how exactly the Game Data they are built from is stored. A minimalist approach, for instance, is to use static hard-coded definitions[2]. In essence, Game Data for Prototypes can be:

1. Stored...
    (a) in-code or
    (b) in a database packed with the Game System; and
2. Represented and edited...
    (a) in plain text or
    (b) via a specialized editing tool.

Where the mechanisms for options 1a and 2a are cheaper to implement but harder for the Creative Team to use than options 1b and 2b, respectively. Other things to consider when choosing how to store Prototypes as Game Data is version control support (RTC-3), how to handle invalid data (RCE-5), and the computational cost of loading data from disk at runtime (RDD-1). Ideally, if the Prototype Loading Service is implemented as a self-contained software component, it can be more easily switched to different implementations. This promotes flexibility (RCE-3) in the Game Data representation of Prototypes since the exact procedures of the Prototype Loading Service are provided through a common abstraction.

5.3.2 Behavior Model Design

Here we address the Behavior Model and Generality domain concepts, though we include discussion regarding the Simulation Progress concept too. The Behavior Model comprises the Behavior Fields of Entities — Abilities and Rules — and their interaction with Effects and the Time Schedule. With this, we cover design approaches to all elements of Figure 5.2. We divide the design approaches for the Behavior Model into three groups: the imperative approach, the *Command* approach Gamma *et al.* (1995), and the dynamic dispatch approach. We present them from simpler but less extensible to more complex but more extensible designs.

Imperative Approach

As explained in Section 5.2.1, there are two ways of accessing the Simulation State: Direct Access and Rule-Mediated Access. Direct Access is straigtforward but coupled, while Rule-Mediated Access is more bureaucratic but more extensible and, as we will see, more compatible with Data-Driven Design. The Behavior Model is essentially responsible for implementing Rule-Mediated

[2]See, for instance, the source file that defines all monsters of *NetHack* (DevTeam, 1987) at https://github.com/NetHack/NetHack/blob/NetHack-3.7/src/monst.c (last access July 16th, 2021).

Access. It provides an API combining the Effect Resolution and Ability Handling Services that other parts of the Game System must rely on. The main role of these Services is to guarantee the extensibility (RCE-3) of the economy mechanics. They achieve this by uncoupling code (RCE-1) that consumes that API from the exact implementation of the mechanics (refer back to Figure 5.9) and by providing means for the Creative Team to add and change Entity Behaviors despite their procedural nature.

Rule-Mediated Access does not need to be complicated or overly formal. The Technical Team can implement Effect Resolution as a set of system routines that compose the basic possible operations over the Simulation State that are subject to the Rules of the economy mechanics. For instance, there could be one routine for each Effect type, and its implementation starts by asserting what Rules currently active in the Simulation State match their Predicate to the Effect implemented by the routine. This could be a manual, hard-coded series of if-statements. For every matched Predicate, the corresponding Rule Resolution is written out.

Changing — or even just reading — the Simulation State without using these basic operations (i.e., using Direct Access instead) has no guarantees that economy mechanics are properly employed. For instance, though we could directly read the "power" Field of a "creature card" in *Magic: the Gathering* (Wizards of the Coast, 1993), that does not mean its *effective* power has that exact value at a given time during gameplay. That is because there might be a plethora of economy mechanics Rules in place that manipulate the value of that Field, subject to change at a moment's notice. Figure 5.15 shows an example of this. The Effect Resolution routines would make sure to follow all the simulation Rules so that using them reliably reproduces the desired economy mechanics. More specifically, Effect Resolution ensures that games with State Modifier Mechanics are properly implemented.

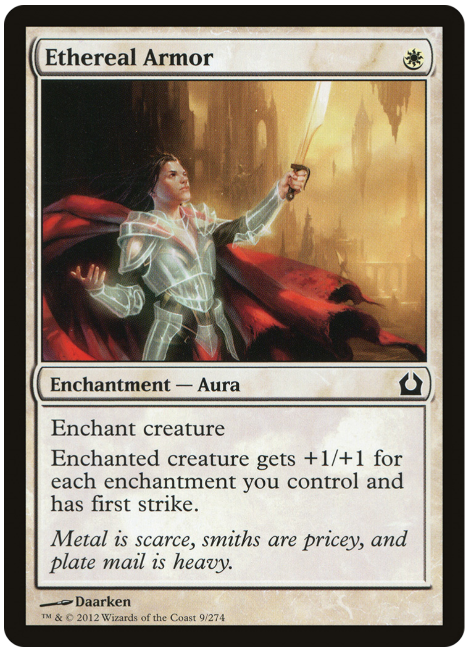

Ethereal Armor
Enchantment — Aura
Enchant creature
Enchanted creature gets +1/+1 for each enchantment you control and has first strike.
Metal is scarce, smiths are pricey, and plate mail is heavy.
Daarken
™ & © 2012 Wizards of the Coast 9/274

Figure 5.15: The *Ethereal Armor* card from *Magic: the Gathering* (Wizards of the Coast, 1993). The economy mechanics of the game dictate that, while in play, this card changes the effective "power" and "toughness" Fields of the "creature card" Entity it enchants by an amount that depends on the state of other "card" Entities in play.

By using hard-coded routines, however, there would be no formal representation for Rules in the codebase — programmers just implement them as part of resolving Effects. This is a straightforward approach that is reasonable for games where economy mechanics lack an escalating number of Custom Simulation Rules. If that is not the case, there are a series of limitations in the design. The maintenance cost of adding or changing Rules, especially when they interfere with multiple Effects, is potentially expensive (remember the cockatrice example from Chapter 1). It also increases the cost of adding new Effects, since the programmer has to consider all Rules that might apply to it. Additionally, only the Technical Team can add and change both Effects and Rules in this design. Finally, since Rules are Behavior Fields that belong to Entity instances, programmers still need to represent that information as a Field. The possibilities reinforce the hard-coded and *ad hoc* nature of the design, such as using boolean values to indicate whether an Entity has a certain Rule active or not (e.g., a “this creature is immune to poison” flag), which is then checked in the routines of the Effect Resolution Service, coupling it to their representation as a Field.

Since Abilities are stand-alone constructs — as opposed to Effects and Rules, which only produce a result when matched together — it is slightly simpler to store them as Fields. It either requires support for first-class functions or some other way of referring to the routine that implements the Ability. For instance, it could be a constant integer reserved to represent each Ability together with a switch-case statement that executes the appropriate Ability given its identifying constant. Similar to Effect Resolution, Ability Handling may be written as a higher-level set of routines that reuse the Effect Resolution routines to implement the in-game actions of simulation Entities. One could even eliminate the separation between Effect Resolution and Ability Handling, using a single layer of routines. We find it useful, however, to separate them, because Ability Handling:

- involves validating the interaction between Subsystems (RIC-1);
- allows storing per-Entity mechanics as Ability Fields (RBM-3); and
- promotes a higher-level creative process of producing Abilities from reusable Effects (RCP-2), though still dependent on programmers.

A consideration that stems from this when implementing the Behavior Model is that some Rules control what Entities *are allowed to do* in terms of Effects and Abilities — hence the validation step of Ability Handling. For instance, the *Silence* spell in *Dungeon Crawl: Stone Soup* (DCSS Devteam, 2006–2021) prevents all creatures from taking actions that require vocalizations (e.g., casting other spells) for a certain duration and within a certain radius of whoever cast *Silence*. When a preventive Rule like this is in effect and the Player attempts to activate one of the forbidden Abilities (or one that produces forbidden Effects), there are two possible outcomes. One outcome is that nothing happens and the Player wasted valuable time (and maybe resources too). Another outcome is that the Ability Handling Service properly notifies the code that invoked it that that course of action is fruitless, and the Player gets a new chance to decide what to do at no cost. The latter outcome is usually preferable in terms of user experience (though there are exceptions) but involves more steps and defensive coding to implement. Games are likely to require both outcomes given specific circumstances, so the design of the Behavior Model should accommodate both.

The validation process of Effects and Abilities has an additional matter to consider. Validating a change to the Simulation State means verifying whether that change is valid given the *current* state. As soon as an Effect, for instance, begins execution, that state is no longer the same and the assurances of the validation crumble one after the other. If only one Effect takes place at a time, this is acceptable because we can assume that the code that implements the resolution of an Effect and its Rules knows the changes that are going to be made. In other words, if only one Effect is resolved at a time (i.e., they are “atomic” regarding each other) validation is stronger.

Both preventing and detecting bugs become easier. As for Abilities, which potentially produce multiple Effects, it is left for the architects to decide how atomic they are among themselves and under what circumstances. Making Abilities entirely atomic, in the sense that they guarantee a complete execution from beginning to end, is hard because programmers have to predict the sequence of consequences of the entire Effect chain produced. This is even harder in Real-Time games where Abilities might take multiple frames to execute and are subject to intervention from physics mechanics (e.g., the Player avatar falls into a hole midway through their multi-slash sword attack).

To summarize, we believe Ability validation requires or, at least, is best supported by separating Effect Resolution from Ability Handling. In the following approaches we propose, we will show how this separation further improves validation mechanisms.

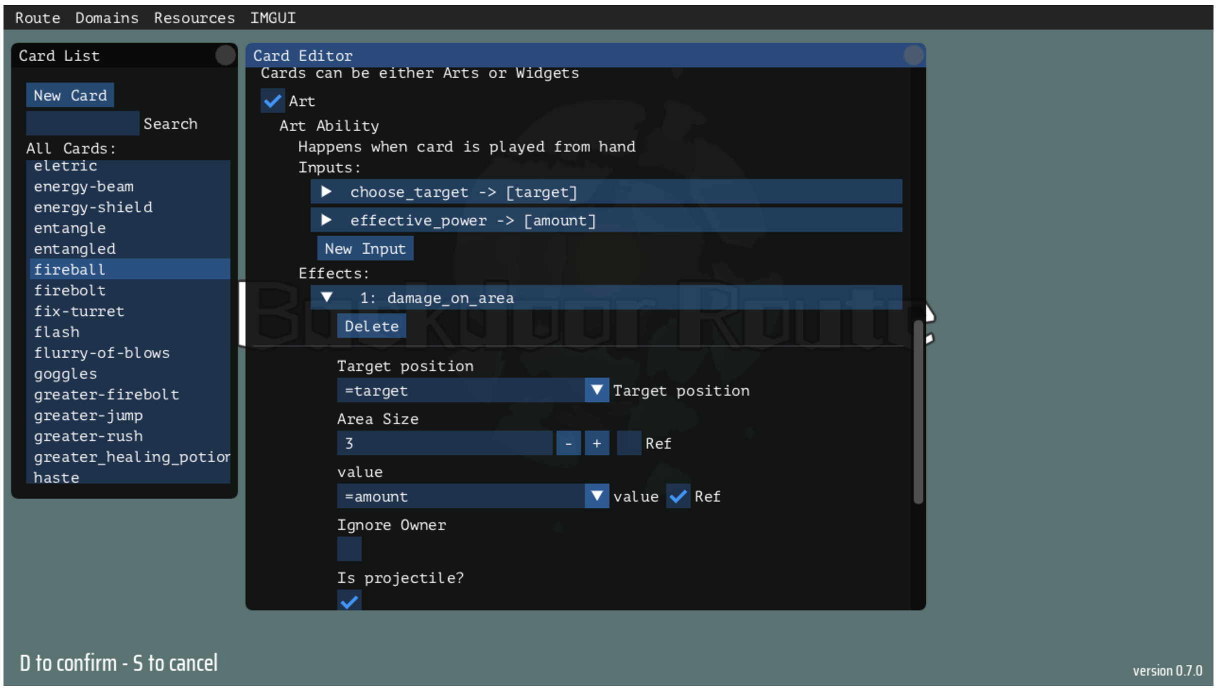
Command Approach

The greatest restriction of implementing Abilities, Effects, and Rules imperatively (i.e., as hard-coded routines) is that it is expensive for the Creative Process. Notably, it lacks support for Data-Driven Design. Practitioners offer alternatives such as using a *scripting language* (Gregory, 2019, pages 1135–1157), the *Bytecode* design pattern proposed by (Nystrom, 2014, Chapter 11), or the *Interpreter* pattern from the *Adaptive Object-Model* (Gamma *et al.*, 1995; Yoder and Johnson, 2002). What all these approaches have in common is that they turn the basic operations of the Simulation State into executable units that can be ordered flexibly by a higher-level structure — a script[3], a bytecode sequence, or a *Composite* tree, respectively. We can see these executable units as manifestations of the *Command* design pattern (Gamma *et al.*, 1995) and, by interpreting them as the Effect and Rule resolutions of the economy mechanics, we find a more extensible approach to the design of the Behavior Model.

In other words, instead of hard-coded routines, programmers can implement the resolution of an Effect as *Command* objects. The implementation of the command methods is similar to the imperative approach, but parameters can be stored as attributes and loaded from Game Data. That way, Abilities specify what Effects they produce in what order. Abilities themselves can be implemented as *Command* instances as well, as suggested by Nystrom (2018), or they can use one of the methods above for increased extensibility and support for Data-Driven Design. Just like with Prototypes, dedicated graphical tools further promote the Creative Process of creating and editing Abilities (see Figure 5.16). In this approach, the Effect Resolution Service becomes an input stream for Effects over the Simulation State, and the Ability Handling Service turns into a *compiler* of Effect commands that feeds that stream. Alternatively, a simpler approach is to implement Effects and Abilities as a single layer as we discussed before, in which case they become *Command* instances that handle broader operations over the Simulation State directly, but lose reusability and support for Data-Driven Development.

An important advantage of seeing Abilities as "compilers of Effects" is that validation becomes more evident. Architects can join the processes of compiling and validating Effects so that an

[3]In the case of scripting, it is often necessary to register bindings for the lower-level routines into the virtual machine that runs the scripts. These routines would be the basic operations over the Simulation State. See some of our previous work (Aluani and Mizutani, 2013) for more information on the subject.



```
Card List
  New Card
  Search
  All Cards:
  eletric
  energy-beam
  energy-shield
  entangle
  entangled
  fireball
  firebolt
  fix-turret
  flash
  flurry-of-blows
  goggles
  greater-firebolt
  greater-jump
  greater-rush
  greater_healing_potior
  haste

Card Editor
 Cards can be either Arts or Widgets
 ✓ Art
   Art Ability
     Happens when card is played from hand
     Inputs:
       ▶ choose_target -> [target]
       ▶ effective_power -> [amount]
       New Input
     Effects:
       ▼ 1: damage_on_area
       Delete

       Target position
       =target              ▼ Target position
       Area Size
       3                  - +   Ref
       value
       =amount              ▼ value ✓ Ref
       Ignore Owner

       Is projectile?
       ✓
```

Figure 5.16: *Backdoor Route* (USPGameDev, 2020), a game that mixes rogue-like and card game mechanics. The image shows the in-game ability editor for cards. Each ability is composed of a series of "input" and "effect" fields. Though the exact implementation details differ, the general pattern of Abilities producing a sequence of Effects can be noticed. Since Effects in *Backdoor Route* are implemented to have a similar role to *Command* objects, it is possible to store instances of Effects as Game Data and edit them visually like this.

Ability either outputs an entirely valid Effect stream or it fails and outputs nothing. This does not prevent Effects from canceling each other but increases the robustness and opportunities for debugging the Game System. Programmers can, for instance, keep a record of all Effects that successfully happened in the Simulation State, producing an auditable log of the in-game events. Such a log could, theoretically, be fed into a new game simulation to reproduce a sequence of steps like in a *tool-assisted speedrun*. This way one could reproduce gameplay sequences that are known to lead to crashes, for instance, and debug them more consistently. Furthermore, by turning Abilities into first-class values (e.g., *Composite* trees), it is possible to not only properly store them as Entity Fields, but also in a queue or other data structure when the simulation needs to schedule their execution in a specific order. This allows implementing complex mechanics such as the *stack* from *Magic: the Gathering* (Wizards of the Coast, 1993). Its rules determine that Players must first place the Ability they desire to activate (e.g., casting a spell) on top of the stack, then all Players are allowed to respond to that Ability by activating more Abilities and further stacking them. Once no one wishes to activate any more Abilities (and no further Triggered Abilities occur), the Ability on the top of the stack happens atomically. After that, the process starts over. Some cards can even affect Abilities that are in the stack (e.g., spells that negate other spells), making it an integral part of the Mechanics Model of *Magic: the Gathering*.

The limitation of the *Command* approach is that it still couples Effects and Rules together in the Effect Resolution process. Though it makes Abilities more compatible with Data-Driven Design, this feature does not extend to Rules. They are still hard-coded into the individual implementations of each Effect Resolution, which, in turn, are still coupled to how Rules are stored as Entity Fields. In particular, the *Command* approach is not enough to make the task of implementing cockatrices in *NetHack* (DevTeam, 1987) significantly easier, for instance.

Dynamic Dispatch Approach

By objectifying Abilities and Effects, the *Command* approach makes them more extensible and compatible with Data-Driven Design. This last approach essentially tries to do something similar to Rules. By turning Rules into *first-class* types, Entities may properly encapsulate them as Behavior Fields, and the opportunities for loading Rules from Game Data increase. For that, we need to first determine how a Rule might be implemented so that we can design a way to move it into an object. Let us look at an example from a real game.

During the *Game Developers Conference* of 2017, Fujibayashi *et al.* (2017) presented what they called the "chemistry engine" of *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017) and its role in providing *multiplicative gameplay*. This engine divides the Entities of the game into "elements" and "materials". Elements include the typical fire, cold, wind, and electricity, but also some game-specific phenomena such as "weapon strikes". Materials relate to a key characteristic of non-element Entities that determines how they react to each element. For instance, whether an Entity is made of flesh, wood, metal, cloth, etc. Each type of element produces a corresponding Effect when it comes in contact with another Entity instance, be it element or material. The way the Effect resolves depends on a series of Rules. For our discussion, we will refer to the fire element as a simplified example. In *The Legend of Zelda: Breath of the Wild*, when an Entity comes in contact with a fire Effect (see Figure 5.17):

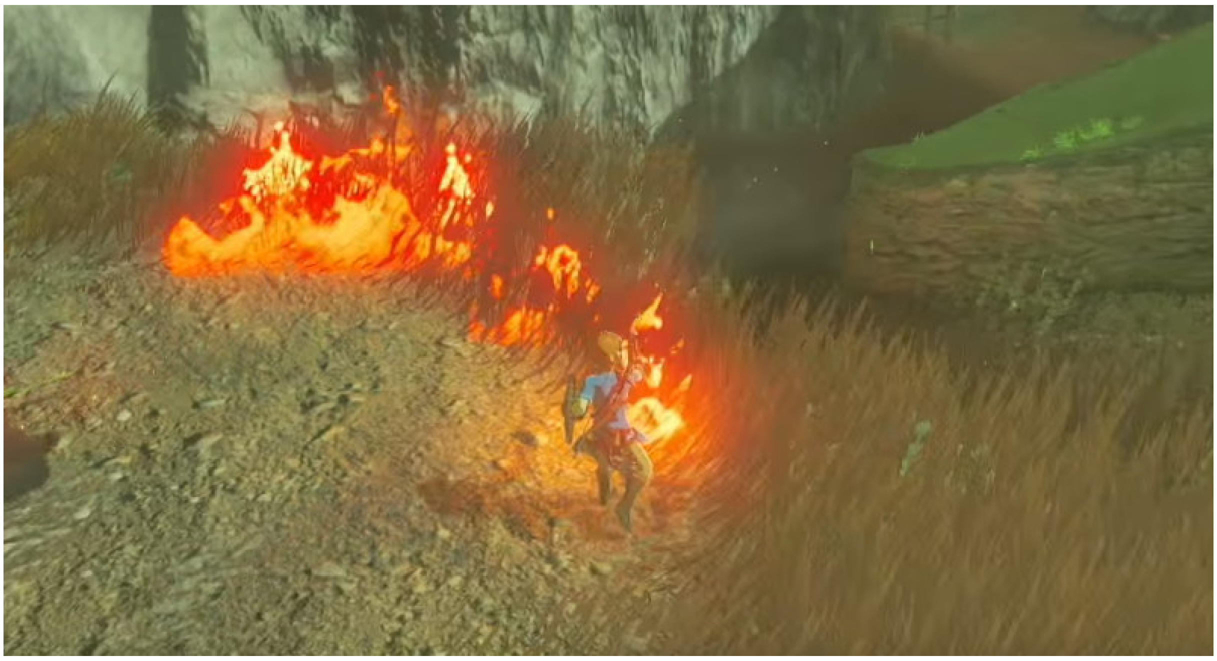


Figure 5.17: *The Legend of Zelda: Breath of the Wild* (Nintendo, 2017). The screen capture shows a player using a torch to make grass catch fire, which is only one of many possible Rules that apply to the "affected by fire" Effect in the game.

1. If it has a plant, wood, or flesh material, it catches fire starts taking damage until the fire is put out or the Entity is destroyed.
2. If it has a food material, it becomes cooked.
3. If it has an ice material, it melts a little for every second it is exposed.
4. If it has an explosive material, it explodes.
5. If the Entity is specifically a "lumber" object, it turns into a campfire Entity.

Each of these cases is a Rule that contributes to the resolution of the fire Effect. Often, only one of them comes into effect but, sometimes, more than one might happen. Using the imperative and *Command* approaches, the programmers could simply write these cases as a series of if-then-else statements. However, we want to implement them as some kind of object instead so we can apply Data-Driven Design to their creative process. After all, as the chemistry engine of *The Legend of Zelda: Breath of the Wild* suggests, each of these Rules is an attribute ("material") of the Entity instances affected by the Effect. If we write the Rules as part of the Effect implementation, we are coupling the Effect to the Field layouts of Entity types and increasing the cost of changes.

We do not know how Fujibayashi *et al.* (2017) implemented the Rules of *The Legend of Zelda: Breath of the Wild*'s chemistry engine but we have solutions proposed by practitioners among our information sources that address similar design issues. Bucklew (2015) uses the *Entity-Component-System* pattern as a basis for his design. The components in his solution implement a common interface with a `FireEvent` abstract method (as in "fire an event" and not "an event about fire"). Events in Bucklew's proposal correspond to Effects in the *Unlimited Rulebook*. In Bucklew's game, *Caves of Qud* (Freehold Games, 2015), when an Effect happens, it is sent to any affected Entity instances which, in turn, forward it to each of its components via the `FireEvent` method. Every component, thus, has a chance to react to that Effect. If we used that to implement *The Legend of Zelda: Breath of the Wild*, we could implement materials as components. For instance, the "food" material would be a component that, upon receiving the Effect for coming into contact with fire, changes its Entity's state to represent the corresponding cooked variation of the food it was. An alternative way of seeing this design is as a mixture of the *Decorator* and *Chain-of-Responsibility* patterns (Gamma *et al.*, 1995).

This solution succeeds in making Rules an explicit part of Entities as one of their Fields (in this case, one of their ECS components). As Bucklew explains, by joining this with Prototypes, the Creative Team can now assign Rules to Entities more flexibly. Since it uses the ECS pattern, though, new Rules still need to go through the Technical Team, but that overall maintenance cost is reduced. The main limitation of Bucklew's design is that the `FireEvent` method implements *all* Rules related to that component because it checks the type of incoming Effects and acts accordingly on a case-by-case basis. If we could separate Rules into individual objects, we could take the ECS one step further and make components out of combinations of Rule Fields.

That is where a different solution from our information sources comes in. Andrew Plotkin (2009) proposes a more radical approach to Rules. He defends that Rules should not be part of objects but independent constructs, referring to this method as *rule-based programming* — though a more precise term would be *predicate-dispatching* (Ernst *et al.*, 1998). In this paradigm, a Rule is the composition of an identifier, a predicate, and a resolution. Identifiers can be called like typical functions or routines and even receive arguments, but there is no predetermined implementation. Instead, there must be a *dispatching mechanism* that finds a Rule with the same identifier and a predicate that evaluates a true value and uses the resolution of that Rule as the implementation for the call. By making Predicates determine the dispatched implementation of functions, this design generalizes dynamic dispatches to an extreme. Though predicate-dispatching was originally proposed as a programming language feature (Ernst *et al.*, 1998), it is possible to implement it as a design pattern (and we did so in our early proofs-of-concept, see Section 6.1).

In the *Unlimited Rulebook*, identifiers translate into Effect types and the arguments into attributes of those Effects. Thus, the Predicate and Resolution form Rules, as explained back in Section 5.1.1, but an implicit part of the Predicate is the type of Effect the Rule applies to. With this, predicate-dispatching manages to successfully isolate Rules into self-contained constructs: a tuple more or less in the form of (Effect Type, Predicate, Resolution). In particular, it finally reduces the cost of adding cockatrices to *NetHack*. When we add them as a new Entity type to the game, we include with it new Rules that match all touch-related Effects. The programmers will not need to find all the specific cases spread throughout the codebase where a touch-related Effect might happen, because predicate-dispatching allows them to "inject" the behavior as an extension.

Nonetheless, predicate-dispatch mechanisms assume programmers write Rules like they would common routines. We need to adjust that to how a Game System works and in a way that allows non-programmers to assign Rules to Entities as well. There are two key assumptions we learned through Bucklew's and Plotkin's works that allow us to adapt the design. The first is that *all* Rules resolve a specific type of Effect — i.e., they all have an "identifier" to invoke them. The second is that to apply an Effect to the Simulation State, it must be dispatched to a particular Entity instance. Even if that is not always the case (i.e., some Effects are "global") we can fall back to a placeholder singleton Entity to catch these stray Effects. Given these two assumptions, we can merge Plotkin's and Bucklew's solutions to have both the flexibility close to pure predicate-dispatching and the extensibility of adding Rules to Entities via composition.

Starting from Bucklew's design, instead of using a catch-all handler for Effects in each component, we propose a Rule Field object that implements a single Rule at a time. Entities can have as many Rule Fields as necessary but they must be able to list them as a basic Direct Access operation. Every Rule Field object has a Predicate and a Resolution. However, these do not match the concepts from predicate-dispatching one-to-one. Using the first assumption, our Predicate is (at least partially) stored as a "type mask" used to detect Effect types resolved by the Rule in question. Its exact implementation depends on the programming language used but can probably be represented by instance variables (as opposed to methods). This is similar to implementing a multi-dispatch manually but has the added benefit that these "type mask" variables can be derived from Game Data. The Resolution, however, has to be an abstract method or, at most, a scripted routine. This method might implement any remaining condition checks to fulfill the Predicate, if necessary. It becomes possible to make reusable Rules because not only are their Predicates exposed mainly as Data-Driven attributes, but also any other parameter it could read from a stored value can be exposed too, just like with the *Command* approach. By using Buckle's design we also rely on the second assumption, since Effects must be dispatched to Entity instances, which then iterate over all their Rule Fields and find all whose Predicate matches the Effect. Then, the game-specific implementation of the Effect Resolution Service adjudicates what Rule Fields should be applied and executes their Resolutions. This design:

- removes the specific dependency on the *Entity-Component-System* pattern as the Object Model;
- allows Rules as Entity Fields, keeping the creative process centered on the composition of Entities;
- decouples Rule Resolution from Rule storage as a Field;
- supports as much Data-Driven Design as possible;
- keeps each Rule object with very limited responsibility, promoting their reuse;
- with this limited responsibility, the implementation of each Rule translates into shorter, more maintainable code pieces (like with the original predicate-dispatch approach); and
- provides a formal framework for self-amending mechanics by turning Rules into first-class objects.

In the end, this approach ends up leaving the concept of predicate-dispatching behind by using the assumptions and simplifying it into a more common dynamic dispatch mechanism. One could say it now works more like a double-dispatch, where the implementation of an Effect Resolution is dispatched by the Effect and the Entity types together. Some practitioners argue in favor of using these forms of advanced dispatching mechanisms in games (Moll, 2021). Either way, since this is not proper predicate-dispatching anymore, we prefer to refer to this third approach to the design of the Behavior Model as simply the *dynamic dispatch* approach. The notion of separating Effect Resolution into a Predicate and a Resolution, however, benefits validation mechanisms, as we have discussed. It also enables *previewing* what an Ability or Effect will do with more precision. What prevents us from doing so, usually, is that resolving one Effect might invalidate the next (and there is nothing much we can do about this short of simulating the future) and that some Rules change Effects as part of their Resolution (refer to Section 5.1.1). We can handle the latter by handling Rules that apply Effects and Rules that change Effects differently. The Effect Resolution Service should process only Rules that change the incoming Effects first and only when there are no other Rules of this type to process should it move on to the Rules that apply Effects (this was already illustrated in Figure 5.10). For instance, if a magic shield Entity is capable of redirecting projectile Effects (i.e., changing their target destination), that Rule has to be processed before the Rules that apply the projectile Effect. This simple prioritozation mechanism should suffice for even relatively complex games, though *Magic: the Gathering*, for instance, has a priority sequence among Rules that change Effects to guarantee the determinism of the mechanics (Wizards of the Coast, 2021, rule 613).

Lastly, there are some performance issues to consider depending on the size of the Simulation State and how many Rules can adjudicate a given Effect. In a game like *Magic: the Gathering*, any card in the game can potentially overrule any Effect, so a naive implementation would have the Effect Resolution Mechanism pass the Effect to all card Entities, which in turn pass it to all their Rule Fields, even if they have nothing to do with that Effect. In games that simulate Worlds with clearer spatial characteristics usually need only verify Entities that are near the Effect. Even then, there would be some wasted computational resources spent checking against Rules that are unlikely to handle the Effect. A possible optimization is to use a look-up table of what Rules currently handle a given Effect type instead of blindly iterating over Entity instances. The cost of this is a more complex design where this look-up table has to be kept up-to-date with Entities coming and going inside the simulation.

This chapter presented the reference architecture we developed as the main byproduct of our research project — the *Unlimited Rulebook*. Using the ProSA-RA method, we explained this reference architecture by discussing three Viewpoints: the Crosscutting Viewpoint, the Runtime Viewpoint, and the Source Code Viewpoint. Next, in Chapter 6, we evaluate our proposal using a number of different methods across each of its design iterations.

Chapter 6

Evaluation

Following the ProSA-RA method as discussed in Section 3.2, this chapter presents our evaluation of the *Unlimited Rulebook* reference architecture. ProSA-RA suggests a checklist method for validating reference architectures (Nakagawa *et al.*, 2014) but the proposal is specifically tailored to evaluate embedded systems. Instead, we chose to rely on the expertise of the computer systems research group of the University of São Paulo and carry out a series of empirical studies. This allowed us to work together with the University of São Paulo Institute of Mathematics and Statistics to provide two new game programming courses: one for Summer students and one for undergraduate and graduate students. These courses gave us opportunities to test the *Unlimited Rulebook* in practice.

In the end, there were two types of evaluation methods we employed. We performed proofs-of-concept and a quasi-experiment. The proofs-of-concept were divided into two parts. First, the early proof-of-concept was a small-scale prototype developed to better understand and validate the use of predicate-dispatching as an Effect Resolution mechanism (from Section 5.3.2). This prototype and its results are described in Section 6.1. The second part of the proofs-of-concept was a mid-scale game designed to evaluate the *Unlimited Rulebook* as a whole. We present it in Section 6.3. All proofs-of-concept were designed to stress the *Unlimited Rulebook* by using games where new, self-amending mechanics are supposed to be constantly produced in a (theoretically) endless development cycle. The quasi-experiment is explained in Section 6.2. It used students from the game programming courses mentioned above to validate whether using the *Unlimited Rulebook* would reduce the effort in implementing certain games.

Our methodology was an iterative variation of ProSA-RA as explained in Section 3.2.3. There were three full cycles in our research project, each with its evaluation steps. However, this chapter does not present them in chronological order. Instead, each section points out the iteration in which its corresponding studies happened.

6.1 Early Proof-of-Concept

This proof-of-concept was part of the architectural evaluation step in the second iteration of the *Unlimited Rulebook*. During this iteration, our main advancements were on the Behavior Model since it was when we studied and applied the concepts of rule-based programming and predicate-dispatching (Andrew Plotkin, 2009; Ernst *et al.*, 1998). Since the literature on the subject approached the technique as a programming language feature, we wanted to validate whether it could be emulated as a design pattern and retain its practicality. Thus, we designed and studied a possible implementation in the form of a proof-of-concept prototype. I developed it myself during November 2019. The study protocol in this part was as follows.

Choice of games. We chose to develop a prototype to implement parts of an existing game. This way, besides eliminating the requirement of designing a game, we reduced the artificiality of the study. In particular, the game we chose — *Magic: the Gathering* (Wizards of the Coast, 1993) — had complex economy mechanics with self-amending rules and a business model where new content is always being produced, constantly evolving the design of the game.

Design Directives. We would implement *only mechanics*, i.e., the Simulation State and the minimum of Services needed to run it. This was done to simplify the process and to eliminate the noise of having to implement the numerous parts that make up a game system. The Object Model was designed using the "pure" and data-centered variation of the *Entity-Component-System* pattern, as suggested by West (2018) and Plummer (2004). We used the Lua programming language with the *LÖVE* engine in mind since it would be later used in our quasi-experiment (see Section 6.2). The main objective of the architectures we developed was to provide a streamlined process of adding mechanics to the prototype. At this moment, we dismissed Data-Driven Design to further reduce the scope and noise of the study, using only a code-based approach.

Evaluation Criteria. The final evaluation of this study regarding the *Unlimited Rulebook* consisted of qualitative analysis, including feedback from USPGameDev members. We discussed and estimated the development cost of extrapolating the prototypes to full-fledged games. The conclusions are documented at the end of the study. As a proof-of-concept, the fact that its implementation was possible without additional features further validates the design since it suggests a certain "completeness" to the solution.

Prototype: *Magic: the Gathering* mechanics using the *Unlimited Rulebook*

In this proof-of-concept, we chose a very small subset of *Magic: the Gathering* rules regarding how creature cards can be destroyed. We will present a brief explanation of the pertinent rules, the gameplay cases we aimed to implement, the architecture of the proof-of-concept, and some discussion on the results. The implementation can be found at gitlab.com/unlimited-rulebook/prototype-example (last accessed August 12th, 2021) under the MIT license.

Since the goal was to validate the use of predicate-dispatching to implement self-amending mechanics, we chose a few very specific rules to focus on. They were chosen to be illustrative and as easy to understand as possible, given the potential complexity of *Magic: the Gathering*. In this game, players take the role of wizards wielding magic to defeat each other but, in practice, most of the action is performed by *creature cards* fighting each other in the virtual battlefield of the match. Every creature has two basic combat statistics: *power* and *toughness*, represented by integer numbers and written as 2/2, for instance. As they combat, creatures accumulate *damage* also in the form of integer values. If at any moment the amount of damage on a creature equals or surpasses its toughness value, the creature is considered destroyed, and its card is moved from the battlefield zone to the graveyard pile. The basic fighting rules say that, when two creatures fight, they each cause damage to the other equal to their own power value.

The power and toughness of a creature, however, can change according to the state of the game. The $+1/+1$ and $-1/-1$ counter mechanics are an example of this. These counters are usually represented by tokens or coins and they change both the power and toughness of the creature according to their values. For example, a 2/2 creature with one $+1/+1$ counter effectively counts as a 3/3 creature instead. In case there are opposing counters (e.g., one $+1/+1$ and one $-1/-1$), they cancel each other until there are no more opposing pairs and any counters left after that remain on the creature. When creatures fight, the game must use their current, up-to-date, effective power and toughness values to determine damage and whether any is destroyed. The counter mechanics are our first amendment to the basic fighting mechanics.

Other mechanics can further complicate the outcome of a fight between two creatures. If a creature has the “indestructible” keyword written on it, then it cannot be “destroyed”. As we have seen, receiving a lethal amount of damage “destroys” a creature, so indestructible creatures simply ignore those mechanics, accumulating any amount of damage without ever leaving the battlefield. Thus, the indestructibility rules further amend the fighting mechanics. Nonetheless, if, for any reason, the creature ends up with zero or less toughness (not damage, but the total toughness value it has), it still “dies”. This happens due to the technicality that having zero toughness is not worded as being “destroyed” — it simply becomes an “invalid” creature and cannot exist anymore, by design. In particular, creatures with the “wither” keyword also have their fighting mechanics amended so that any damage they cause to creatures is transformed into $-1/-1$ counters instead. In other words, creatures with “wither” can still defeat creatures considered “indestructible”. The cards in Figure 6.1 have examples of cards using these mechanics and the particular interaction between indestructibility and wither laid out explicitly.

Given these mechanics, the prototype was implemented to correctly determine the outcome of the following *Magic: the Gathering* scenarios:

1. Fight: 1/1 creature A versus 2/2 creature B. Result: creature A dies and creature B survives with 1 damage point.

2. Fight: 1/1 creature A with “indestructible” versus 2/2 creature B. Result: creature A survives with 2 damage points and creature B survives with 1 damage point.

3. Fight: 1/1 creature A with two $+1/+1$ counters versus 2/2 creature B. Result: creature A survives with 2 damage points and creature B dies.

4. Fight: 1/1 creature A with “indestructible” versus 2/2 creature B with “wither”. Result: creature A dies and creature B survives with 1 damage point.

5. Fight: 1/1 creature A with one $+1/+1$ counter and “indestructible” versus 2/2 creature B with “wither”. Result: both creatures die.

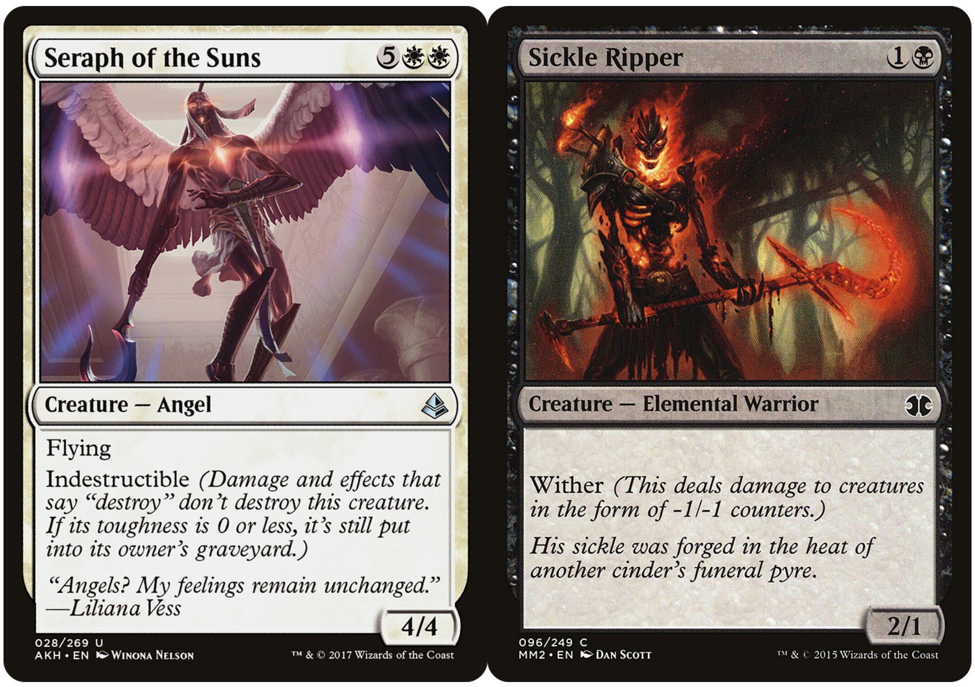

Seraph of the Suns
5
Creature — Angel
Flying
Indestructible (Damage and effects that say "destroy" don't destroy this creature. If its toughness is 0 or less, it's still put into its owner's graveyard.)
"Angels? My feelings remain unchanged." —Liliana Vess
4/4
028/269 U
AKH • EN Winona Nelson
™ & © 2017 Wizards of the Coast
Sickle Ripper
1
Creature — Elemental Warrior
Wither (This deals damage to creatures in the form of -1/-1 counters.)
His sickle was forged in the heat of another cinder's funeral pyre.
2/1
096/249 C
MM2 • EN Dan Scott
™ & © 2015 Wizards of the Coast

Figure 6.1: Two cards from *Magic: the Gathering* (Wizards of the Coast, 1993). The card on the left, "Seraph of the Suns", has the "indestructible" keyword and cannot be destroyed by receiving damage. It could, theoretically, be defeated by the card on the right, "Sickle Ripper", because the latter has the "wither" keyword.

Figure 6.2 shows the architecture of the prototype we developed. The `Record` class — implemented as a Singleton (Gamma *et al.*, 1995) — worked as the World of the Simulation State. It used the *Entity-Component-System* pattern, with the `Entity` class holding only an identifier used to find its components (called `Property` in this implementation). The `RuleEngine`, `RuleSolver`, and `RuleSet` classes are implemented together with the Effect Resolution Service. The `Rule` class corresponds almost one-to-one to the Rule abstraction of our final reference model. In this proof-of-concept, programmers add Rules to the simulation by defining new `RuleSet` instances. This was supposed to represent new "mechanics packages" being added to the game (e.g., a new card with its custom rules). Effects in this implementation exist implicitly as the combination of the name associated with rules and the parameters used to "invoke" that name. One thing that differs from the final design of the *Unlimited Rulebook* is that, in this proof-of-concept, there are "query Effects" that the code uses to read (as opposed to writing to) the state of Entities while also abiding by the Rules in effect. Listing 6.1 shows a commented example of the `RuleSet` that introduces the fighting mechanics of *Magic: the Gathering*.

We implemented the indestructibility and wither mechanics as follows. To determine whether a creature should die, we first defined an `is_dead` query Effect with the general Rule that a creature is dead if its toughness value is zero or less. We then added a `RuleSet` with the damage mechanics, defining that creatures are also dead if the query Effect `has_lethal_damage` returns true. The general rule was that it did so only if there is as much damage as there is toughness in a creature. Indestructibility worked by assigning a `RuleSet` that overruled the `has_lethal_damage` to always return false if the creature was marked as indestructible. The

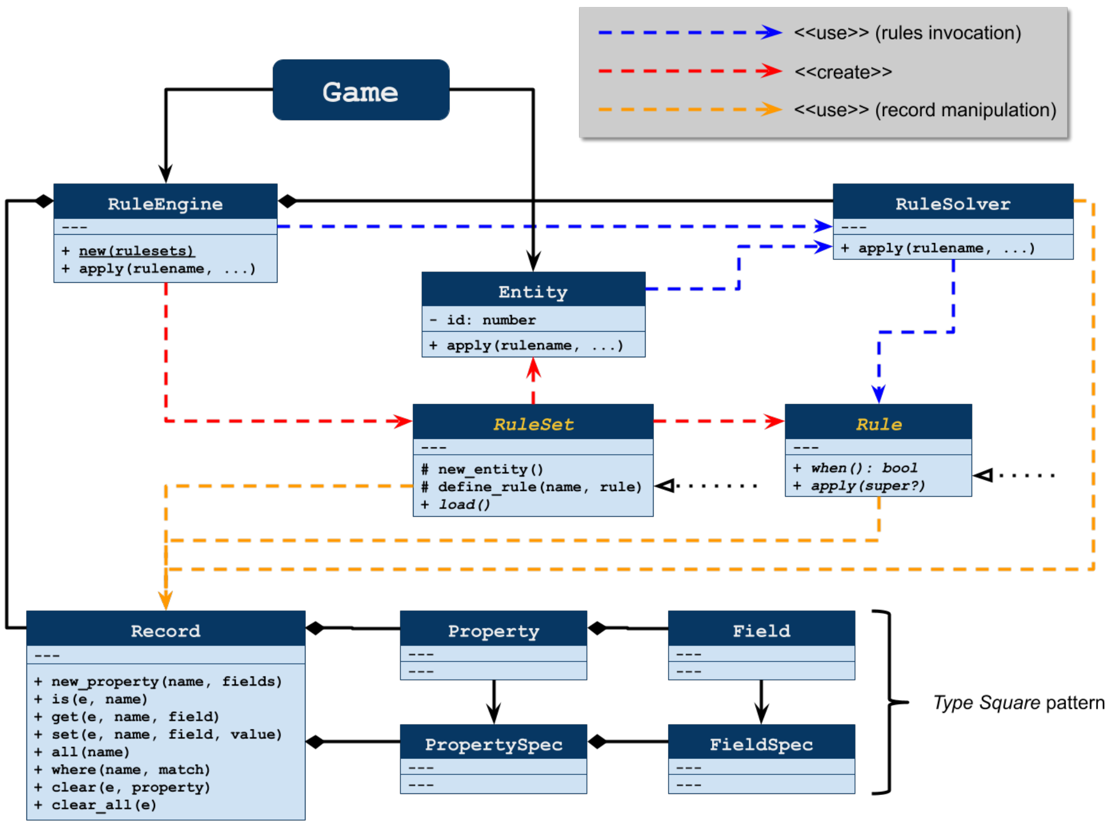


Figure 6.2: Class diagram for the architecture of the second iteration of the *Unlimited Rulebook*, also documented in previous work (Mizutani and Kon, 2020). Developers add new mechanics by extending the base `RuleSet` class which, in turn, specializes the base `Rule` class by defining its name, parameters, predicate, and resolution.

wither mechanics, on the other hand, required the implementation of counter mechanics — which naturally caused creatures to die because of the general rule for `is_dead`. After that, it was simply a matter of overruling the `cause_damage` Effect to place $-1/-1$ counters instead of adding damage. With this, the proof-of-concept passed all the test scenarios. We wrote a more extensive example using this version of the *Unlimited Rulebook* in previous work (Mizutani and Kon, 2020).

Our qualitative analysis of this proof-of-concept was that it provided a versatile and extensible tool for adding economy mechanics and was particularly proficient in supporting self-amending mechanics. Writing `RuleSet` entries and building the mechanics by combining properties and rules allowed fine control over the depth of the gameplay while keeping a streamlined process for adding content. Since this design only allowed access to the Simulation State via the Effect Resolution Service, there was no Direct Access alternative, making the simulation almost completely encapsulated. While this reduced the risk of broken dependencies, it incurred some maintenance costs that we only noticed in the quasi-experiment that followed. What we did notice at this point was that it overly encapsulated design made it hard to integrate the simulation with other subsystems, especially for asynchronous interactions with the Player. For instance, there was no easy way to stop the resolution of an Effect midway to display the in-progress result of that

```lua
-- Populates the brand-new ruleset with rules
function (ruleset)

  -- Reference to the World to look up Entities
  local r = ruleset.record

  -- Defines a rule to resolve the "fight" Effect, which takes e1 and e2
  -- as the creature card Entities fighting
  function ruleset.define:fight(e1, e2)
    -- Rule Predicate
    function self.when()
      return r:is(e1, "creature") and r:is(e2, "creature")
    end
    -- Rule Resolution
    function self.apply()
      local power1, power2 = e1.power, e2.power
      -- The lines below invoke the "cause_damage" effect twice
      e1:cause_damage(e2, power1)
      e2:cause_damage(e1, power2)
    end
  end

end
```

Listing 6.1: The RuleSet for fighting mechanics, written in Lua.

interaction to the user before resuming the resolution. Another issue we realized (though it never became an actual problem) was that all Rules had to exist in the Simulation State all the time. This might not only be unfeasible in a larger game due to memory limitations, but it might also make it too slow to iterate over all possible Rules looking for the ones that match their Predicate to the Effect invoked. The main takeaway from this empirical study on predicate dispatching was that the notion of separating a function signature from its implementations and defining the latter in terms of particular, overruling cases, provided both flexibility and extensibility to the Mechanics Model of a game.

6.2 Quasi-Experiment

This was part of the architectural evaluation step in the second iteration of the research and development of the *Unlimited Rulebook*. It was a quasi-experiment: an empirical study where we compare the outcome of a process with and without a certain treatment (i.e., using our reference architecture) but the tested samples are only partially randomized (Campbell and Stanley, 1963). To do so, we used students from game programming courses as subjects. The goal was to measure the costs of using the *Unlimited Rulebook* compared to letting students design *ad hoc* approaches.

6.2.1 Protocol Design

The quasi-experiment was performed as part of two different courses. One was a paid but publicly available Summer course offered at the Institute of Mathematics and Statistics of the University of São Paulo (IME-USP) and the other was a course offered to undergraduates as part of their Bachelor in Computer Science degree, also at IME-USP. We taught at both courses with the help of teaching assistants that were not involved in the research. The institute authorized the study and it was also submitted to and approved by the Ethics Committee of the School of Arts, Science, and Humanities from the University of São Paulo (EACH-USP).

The Summer course lasted for two months and had two four-hour classes per week. Though it was publicly open to paid enrollments, it required students to know the basics of programming but not necessarily game development or software architecture. Classes were half theorical and half practical and happened in a laboratory where students had access to computers. The participants carried out all tasks, including the ones that were part of the quasi-experiment, during class at the laboratory. Given the limited amount of machines available, there were only 20 open spots.

The undergraduate course lasted a full semester, with two weekly classes of two hours each. It was offered to students in their senior years, so we expected them to know advanced programming and to have prior experience with software architecture. We did not expect them to know game development, though, as there are very few courses in this area in IME-USP. All classes in the undergraduate course were theorical and used conventional classrooms without computers. Thus, students performed their tasks at home or at the university's open laboratories. There were 40 open spots.

Given the circumstances of each course, the quasi-experiment protocol was as follows. After teaching the most basic concepts for game programming, we gave some introductory classes to software architecture applied to games to reduce the difference between student experiences. In these classes, we included the same principles behind the *Unlimited Rulebook*: reusability, flexibility, and extensibility. This was when the quasi-experiment began in each course.

Students who agreed to participate had to fill in a form to record their technical profile and had to sign a consent document. Students were free to abstain from the study and whether they participated would not change their learning activities, grades, or evaluation in any way. To those who accepted to participate, we assigned them to teams of three, randomly chosen in a way that balanced the experience level in each team according to the technical profile they filled in.

During the quasi-experiment, students had to implement the economy mechanics of two games with different genres: one turn-based role-playing game and one real-time strategy tower defense game. We gave them the code with the basic structure of each game except for the Simulation State and related Services. This was to place students on as much equal footing as possible, given that developing even the basic features of a game system is a considerably complex task. It also helped to reduce the amount of time each assignment took. The idea of using two game genres was to test the *Unlimited Rulebook* in different situations where we assumed economy mechanics would differ significantly. Both games were to be developed using *LÖVE* (see Section 2.1.2).

The quasi-experiment had two stages, one for each game. In the first stage, the *control stage*, half of the student teams were randomly assigned one of the games and the other half was assigned the other game. Students were to implement the economy mechanics as they saw fit. At this point, we had not yet explained any detail about the *Unlimited Rulebook* to any of them. In the second stage, we repeat this process with two changes. The first was that we now taught students about the *Unlimited Rulebook* reference architecture, providing them with the reference implementation used in the prototype from Section 6.1. The second change was that each team now had to implement the economy mechanics of the other game — the one they had not worked with yet. This study based on controlling two variables (the used architecture and the game genre developed) follows the quasi-experiment design known as *Latin Square* (Campbell and Stanley, 1963). Table 6.1 shows the resulting layout.

	First project	**Second project**
Group 1:	real-time tower defense game, free design	turn-based role-playing game, URB
Group 2:	turn-based role-playing game, free design	real-time tower defense game, URB

Table 6.1: Latin square design of our quasi-experiments.

For each game, there was a list of economy mechanics (characters, items, skills, etc.) that had to be implemented. Each stage had the same amount of time for students to develop their projects. At the end of each stage, we evaluated how the software architecture the students designed affected the development costs of the economy mechanics we asked them to do. We compared the results to determine whether the *Unlimited Rulebook* reduced the costs of developing new mechanics. This analysis involved both measuring implementation effort and reflecting on the subjective input from students.

6.2.2 Data Collection

The Summer course edition of the quasi-experiment worked as a pilot for the Undergraduate course edition that would happen later. 6 students participated in the pilot and we considered its sample too small to draw any conclusions from. We noted, however, that students agreed that the learning curve to apply the *Unlimited Rulebook* effectively was steep. In the undergraduate course version of the quasi-experiment, 28 students participated. The demographic information we collected from participants regarded mostly their previous experience in the key knowledge fields related to evaluating the *Unlimited Rulebook*: general programming, object-oriented programming, design patterns, software architecture, game development in general, and game programming specifically. As for the analysis of the *Unlimited Rulebook*, aside from metadata fields, the data we collected from students at the end of each stage consisted of three Likert-scale questions and three open-ended questions:

1. How hard was it to implement the economy mechanics? (Likert scale)
2. How much did the architecture help in implementing the economy mechanics? (Likert scale)
3. How often did you have to change the architecture throughout development? (Likert scale)
4. What aspects of the architecture helped development the most? (Open-ended)
5. What aspects of the architecture hindered development the most? (Open-ended)
6. What are your comments and suggestions on the architecture? (Open-ended)

We crossed the data from the Likert-scale questions with the type of architecture used and the genre of the game developed to analyze whether there was any significant difference in the answers. For the open-ended questions, we coded the responses by extracting relevant and recurrent themes or subjects into keywords. We then analyzed the frequency of the codes to determine the overall evaluation of the *Unlimited Rulebook*.

As per the consent term the participants signed, we cannot share the raw data publicly but any interested party may request them from us directly. As long as the same terms are agreed to, we can provide the data set produced by these empirical studies.

6.2.3 Results

As discussed in our previous work (Mizutani and Kon, 2020), data from the quantitative questions (i.e., using the Likert scale) was mostly inconclusive. There was no noticeable difference in the costs of using the *Unlimited Rulebook* or an *ad hoc* architecture, as shown in Figure 6.3, Figure 6.4, and Figure 6.5. The differences between the distribution of answers according to the genre of the games developed presented no discernible pattern either.

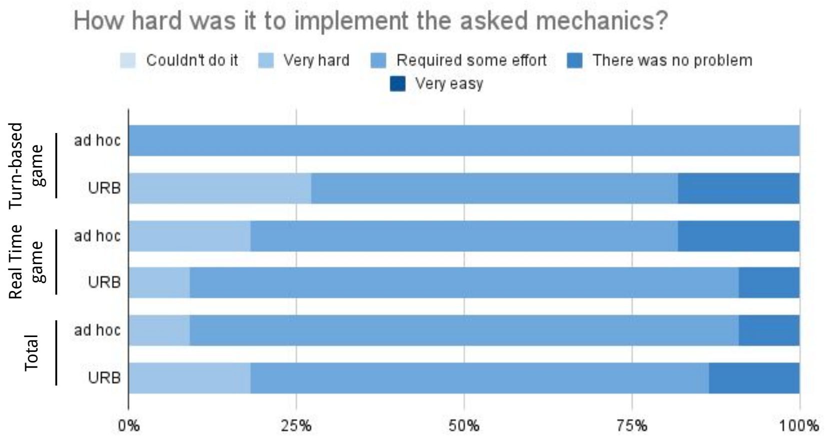


Figure 6.3: Distribution of answers to the first Likert-scale question of the quasi-experiment.

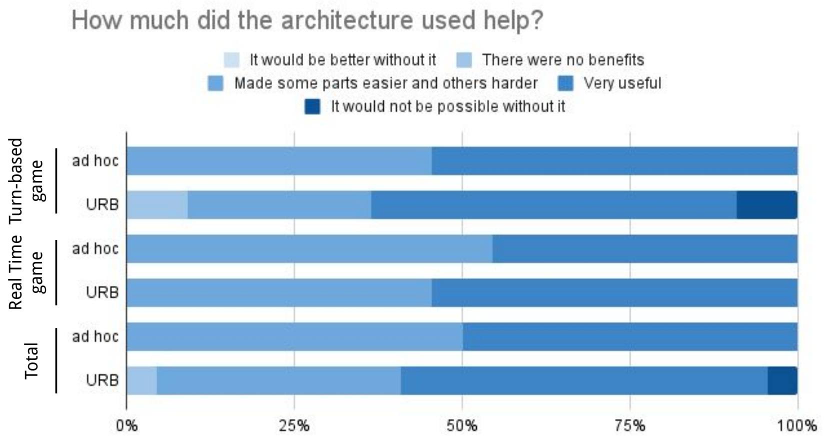


Figure 6.4: Distribution of answers to the second Likert-scale question of the quasi-experiment.

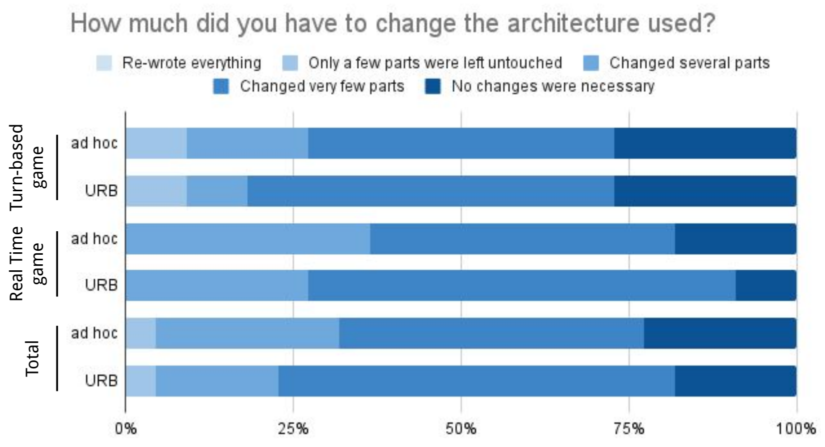


Figure 6.5: Distribution of answers to the third Likert-scale question of the quasi-experiment.

The open-ended questions, on the other hand, provided clearer results. Figure 6.6, Figure 6.7, and Figure 6.8 show the most common topics among open-ended answers. The aspect of the *Unlimited Rulebook* that most (34.5%) helped students develop their projects was the predicate-dispatching approach to the Behavior Model, while the aspects that most hindered them were the learning curve, in first (27.6%), and the strict division between the World and the Rules, in second (20.7%). There were no prominent comments or suggestions for the *Unlimited Rulebook* but the general sentiment was that it was over-engineered for very particular types of games. All these results were taken into consideration in the third design cycle of the *Unlimited Rulebook*.

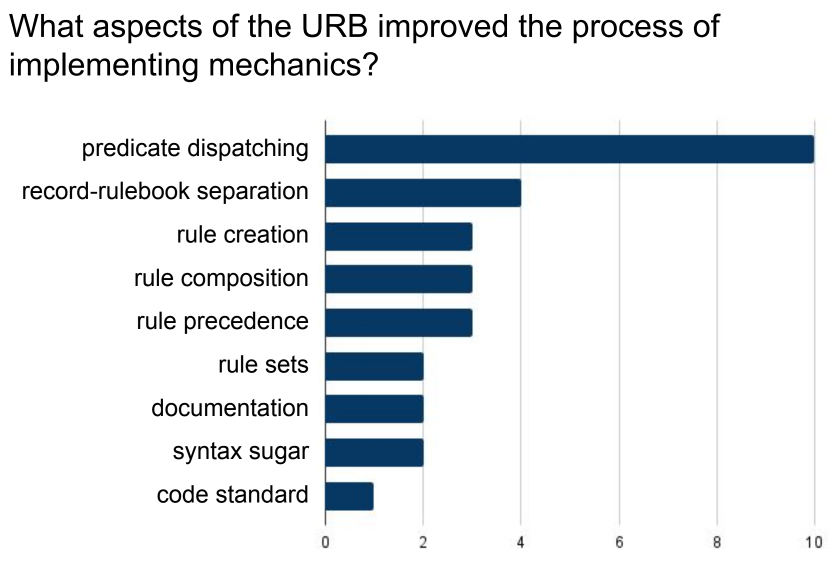


Figure 6.6: Frequency of common topics among answers to the first open-ended question of the quasi-experiment.

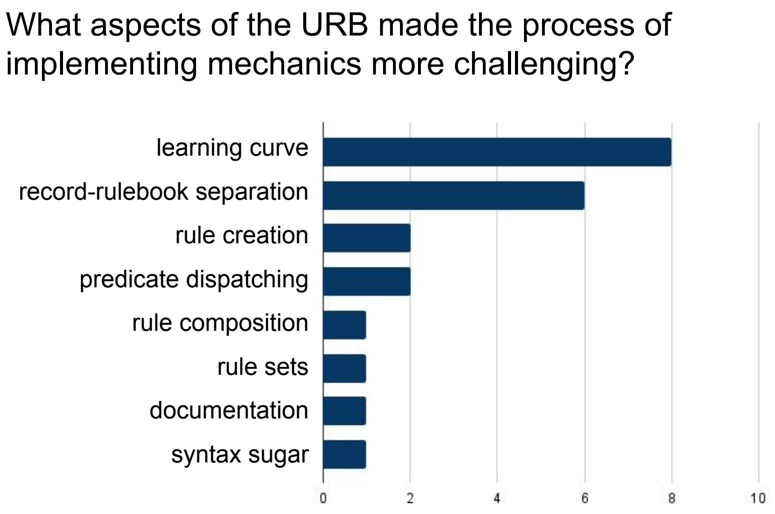


Figure 6.7: Frequency of common topics among answers to the second open-ended question of the quasi-experiment.

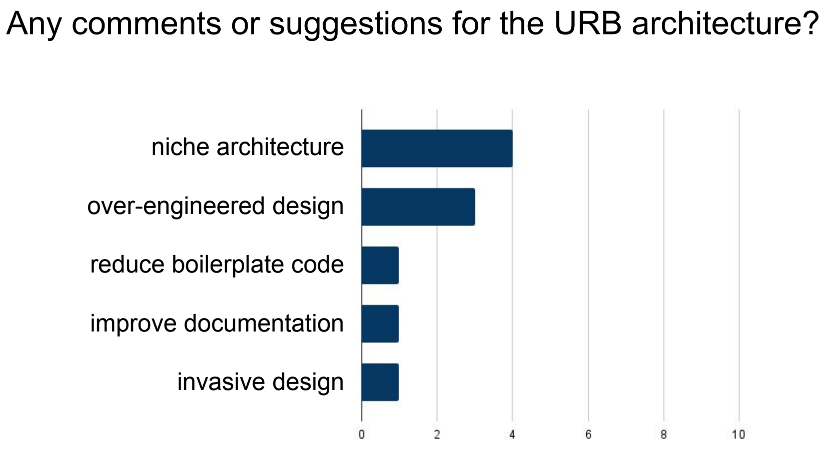


Figure 6.8: Frequency of common topics among answers to the third open-ended question of the quasi-experiment.

6.2.4 Discussion

We speculate a number of reasons for the inconclusiveness of the quantitative analysis in our previous work (Mizutani and Kon, 2020). The two main ones are the students' struggle in the second stage of the quasi-experiment and the scope of the games they developed. Their struggle was because, despite having the same amount of time for both projects, the second one (where they used the *Unlimited Rulebook*) had effectively less time due to the convergence of deadlines and final exams at the end of the semester. The notable decline in the students' grades was evident even though the second stage was supposed to be easier. After all, they had more experience after the first project and after the extra classes between them. We believe this corroborates our speculation that the second project ended up being more taxing on students for external factors, which might have reduced the effects of using the *Unlimited Rulebook*. As for the scope of the games, we believe them to be an issue because a 1-month class project would not fit a game large enough to surface the costs of implementing economy mechanics. At the same time, a single, larger project would be harder to compare against a control sample. We would have to adapt the quasi-experiment to accommodate half the students using the *Unlimited Rulebook* while the others would artificially avoid it. Another factor to consider is that, by providing students with an initial implementation of the Simulation State and Mechanics Model, we might isolate the measurement of the costs of making economy mechanics, but we missed the opportunity to account for the upfront costs of making that initial implementation.

From the results in the qualitative analysis, we steered the design of the *Unlimited Rulebook* in the third iteration as follows. It seems that, while predicate-dispatching conceptually improved flexibility and extensibility, it both had a steep learning curve and made development too rigid by using the Rules as the single access method to the Simulation State. In the current design of the *Unlimited Rulebook* we showed in Chapter 5, we re-designed the Ability, Rule, and Effect abstractions in a more object-friendly way that we believe is both simpler to reason about and to implement. At the same time, we acknowledged that game systems need the short-term flexibility of Direct Access to the Simulation State, even if we know that will create long-term costs. This happens because, as we discussed many times regarding the creative process in games, being able to play the game as soon as possible is a fundamental part of the game design process.

6.3 Final Proof-of-Concept

The final proof-of-concept was the evaluation method of the third and final design cycle of the *Unlimited Rulebook*. It incorporated the most up-to-date elements of the reference architecture. We carried out the proof-of-concept out in the form of a mid-sized game we called *Grimoire: Ars Bellica*. I developed it from January to August, 2021, with assorted contributions from USPGameDev members, especially in the last weeks when we finished the final version for this thesis.

6.3.1 Study Design

Our goal with *Grimoire: Ars Bellica* was to validate whether the *Unlimited Rulebook* improved the production of economy mechanics compared to its previous iterations. We also wanted to evaluate aspects of the reference architecture we had not yet considered. Thus, we wanted to assess whether our proposal had the right balance between reusability, extensibility, flexibility, learning curve, and the pragmatism for developing a game as close as possible to a "real product".

As a game, we designed *Grimoire: Ars Bellica* to resemble a number of economy-centered games we studied throughout our research. Our main inspirations were classic rogue-like games such as *NetHack* (DevTeam, 1987) and *Dungeon Crawl: Stone Soup* (DCSS Devteam, 2006–2021), games from *The Legend of Zelda* series (Nintendo, 1986-2021) and the main *Pokémon* series (Game Freak, 1996–2021), the *Magic: the Gathering* (Wizards of the Coast, 1993) card game, and action role-playing game *Path of Exile* (Grinding Gear Games, 2013–2021). Thus, we aimed for a game with turn- and grid-based control, multiplicative and emergent gameplay, collectible elements that provided unique powers, exploration, and tactical combat. To specifically validate the Behavior Model designs of the *Unlimited Rulebook* using Abilities, Effects, and Rules, we needed those unique powers to offer self-amending mechanics. We chose them as the central pillar in the design of the gameplay and used the other elements to provide a "playground" for those powers. We chose to use "magic spells" like in *Magic: the Gathering* to represent the unique powers and came up with a standard fantasy setting to provide context and motivation for those mechanics. The resulting game was as follows.

In *Grimoire: Ars Bellica*, the user plays the role of a scholar in the arcane arts tasked with invading a monster-filled land to both study lost magic and restore peace to the region. The player navigates the virtual world by traveling between local maps like in the first titles of *The Legend of Zelda* franchise. Each map is a 21×21 grid where anything moves in discrete steps and acts on a turn-based order. The main action available to the player is to cast spells from their *grimoire* — a tome where they scribe their arcane knowledge. They learn spells by studying ancient runes spread throughout the land but the grimoire can store only five spells at a time. The spells can be used to vanquish monsters, surpass environmental obstacles, and survive in hostile territory. To win the game, the player must find and activate a magical crystal protected by a very dangerous monster, deep into the cursed land.

We developed *Grimoire: Ars Bellica* using *Godot* to both reduce the amount of work needed and to assess the compatibility between the *Unlimited Rulebook* and a production-ready game engine. We kept the graphics very simple using low-resolution pixel-art where most sprites are 1-bit textures with frames of 16×16 pixels. Whenever possible, we used free-licensed assets from the community. We completely dismissed the soundtrack of the game, at least for this thesis. The user interface of the game uses both the keyboard and the mouse to play, being mainly aimed at computer users.

The development process followed an agile methodology using short milestones as iterations. After each milestone, we would publish a build of the game to kazuo256.itch.io/grimoire-ars-bellica. Our builds were all playable on the browser to ascertain the accessibility of the game. We intended from the beginning to publish this proof-of-concept as a playable game and to continue its development after the thesis until we have a complete product. To simulate the development process for a "real" game, we used two methods. First, we kept under consideration its potential success as an entertainment system and the costs of maintaining its development even after it fulfilled its most immediate purpose in this research. Second, we gathered informal feedback from USPGameDev members that played the game after every intermediate release. In these feedbacks, we focused on how easily they understood the gameplay and were capable of engaging in the challenges proposed by the mechanics. With this, we could, to a certain extent, treat the game as if it would be distributed to "real" players outside the development team. On the other hand, development ended up consisting of mostly a single person[1], which limits how much *Grimoire: Ars Bellica* represents game development using the *Unlimited Rulebook* in general. The source code of the game is available under the GPL 3.0 license at gitlab.com/uspgamedev/grimoire-ars-

[1] This was in great part due to the COVID-19 pandemic, which severely disturbed the schedule of the developers originally involved, including me.

bellica/grimoire-ars-bellica.

The study aimed to measure how many significantly different economy mechanics we managed to implement and to analyze what were the cheapest and most expensive types of mechanics to produce. This information was informally derived from a qualitative analysis of the resulting architecture of the game as well as from the effective variety of mechanics we managed to implement by the end of the study. We also maintained a development journal (devlog)[2] where we documented our efforts in developing *Grimoire: Ars Bellica* and we also measured the number of hours we put into the project.

6.3.2 Implementation

Here we briefly describe how each of the elements of the *Unlimited Rulebook* manifests in *Grimoire: Ars Bellica*. Since we used *Godot* (see Section 2.1.2), though, many of these elements were already implemented from the start:

- **Game Loop**. *Godot* has its own Game Loop implementation that developers do not usually need to modify or replace. The key characteristic of their implementation is that the execution flow of the game comes from two different steps in the Game Loop: physics frames, used for simulation logic, and idle frames, used for rendering logic. This had no major impact on our implementation — we just chose which to use on a case-by-case basis following *Godot* best practices.

- **Subsystems**. *Godot* implements Subsystems very explicitly, though it calls them *servers* (e.g., `VisualServer`, `PhysicsServer`). The Subsystem for simulation features other than physics works differently. Every frame, it traverses the Simulation State and runs specific script routines depending on the Game Loop steps we explained above. These scripts are attached to individual Entities as we will see now.

- **Simulation State**. The engine provides a Composite-based (Gamma *et al.*, 1995) framework for structuring and populating its Simulation State. The basic composition units are *nodes* and trees of nodes are called *scenes*. The composed state of the entire tree of the game at a given time constitutes its Simulation State. Each node can have one script attached to it and *Godot* looks for routines with specific names to invoke in these scripts. There are dozens of built-in types of nodes and each script attached to them is an inherited class that extends those types. This way each particular game extends the Simulation State format to fit their needs.

- **I/O Infrastructure**. As expected from a production-ready engine, *Godot* provides all essential infrastructure to implement a game. In particular, it provides interactive features such as input, graphics, and sound processing that developers can take for granted most of the time. More importantly, however, *Godot* is completely implemented for Data-Driven Design: it stores scenes as data files and allows node scripts to specify custom variables

[2]https://gitlab.com/uspgamedev/grimoire-ars-bellica/grimoire-ars-bellica/-/wikis/DevLog/Index (last accessed August 25th, 2021)

that developers (and especially the Creative Team) can edit directly in the graphical user interface of the engine.

- **Services**. From the Core Services the *Unlimited Rulebook* expects, *Godot* provides Prototype Loading — by allowing programmers to easily instantiate scenes stored in files — and Event Dispatching — using its built-in signal mechanism.

- **DCC Tools**. Being a WYSIWYG Data-Driven engine, *Godot*'s editor has multiple DCC Tools built into it: map editors, animation players, node inspectors, etc. By using its own UI features to implement its editor, extending *Godot* with new DCC Tools is quite accessible and we made extensive use of this feature.

The elements that are not included as *Godot* features are the Subsystem for simulating economy mechanics and the Services for Ability Handling, Effect Resolution, Query Processing, Progress Tracking, and State Persistence. We describe ahead how we implemented them in *Grimoire: Ars Bellica*. As for the Creative and Technical Teams, we were the only developers involved and, thus, played both roles at the same time — which is one of the threats to the validity of this study.

Since the default Simulation State provided by *Godot* is generic we specialized it to represent the state for the particular case of *Grimoire: Ars Bellica*. By defining standard structures and patterns to follow, we formalized how the elements from the *Unlimited Rulebook* reference model fit into the game architecture. Starting from the Object Model, the **World** element is divided into two parts. Gameplay in *Grimoire: Ars Bellica* happens in grid-based maps and the player can travel between these maps. These local maps are the "lower layer" of the World. The "upper layer" is the layout of the overworld matrix that describes how each local map connects to neighboring maps. Figure 6.9 illustrates this "world map" in an older version of *Grimoire: Ars Bellica*.

We implemented each local map as a *Godot* node tree in which the root node contained the grid information and nodes directly below that represented **Entities**. The simulation only ran one map at a time, so Entities from one location did not affect those from other maps. To implement Entity types, we used the *Entity-Component-System* pattern: nodes directly below an Entity node were *Properties* that, together, composed the type of the Entity. Figure 6.10 exemplifies this in the *Godot* scene that makes up the player Entity Prototype. We designed the overall scene tree of the game so that Entities had unique paths from the root node to their respective nodes. This allowed us to use those paths as **References** to them, so if a path no longer pointed to an existing Entity node, it must have been deleted. All Entities had some basic physics-related Fields (e.g., position) and all other **Fields** came from Property nodes. For instance, in the turn-by-turn dynamics of the game, the order in which Entities acted depended on their `initiative` Fields, which only exists when they have the `Turn` Property — otherwise, that Entity did not act during turns, likely because it was an inanimate object such as a tree or rock. Our **State Persistence Service** simply iterates over all maps, Entities, and Properties to serialize the Fields in a structured way that allows the game system to restore its state in future executions. Most of the serialization and de-serialization steps are already provided by *Godot*.

The turn mechanics using the `initiative` Field relate to how we implemented the **Time Schedule** element. The **Progress Tracking Service** invoked by our economy simulation Sub-

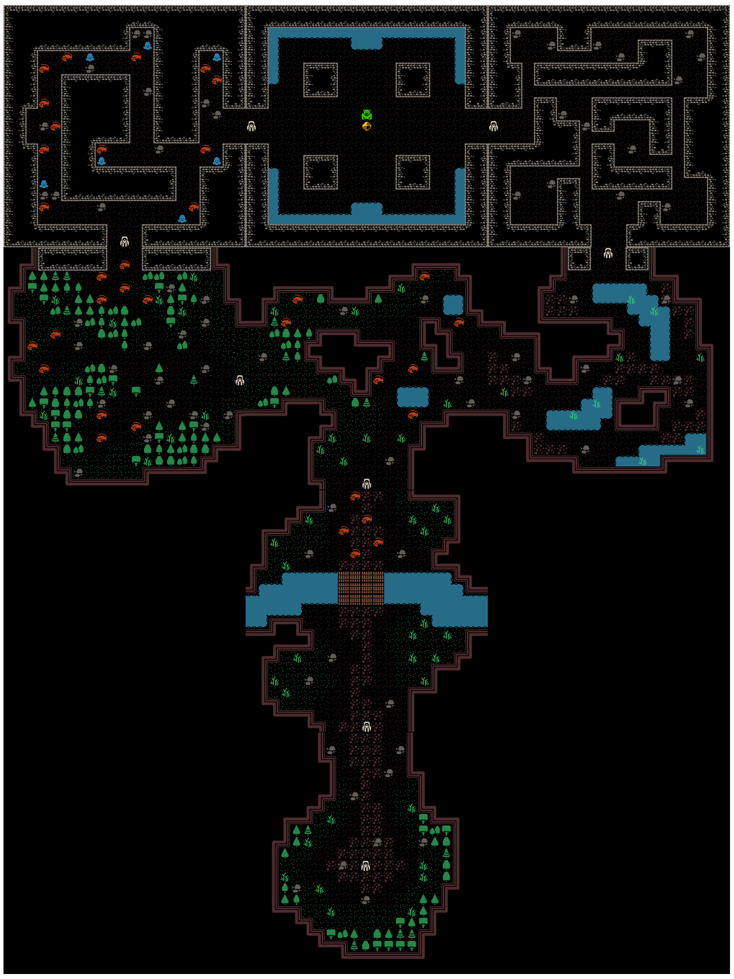
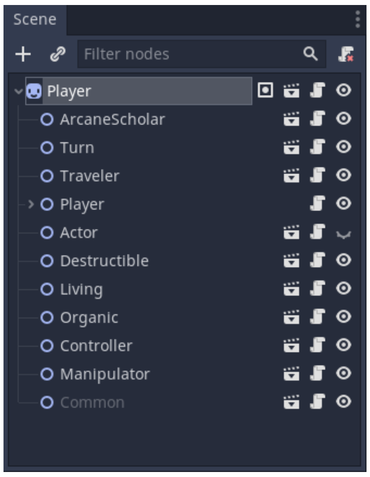


Figure 6.9: An old version of the "world map" of *Grimoire: Ars Bellica*. It had 8 maps laid out in a 3×4 matrix. Players started out at the bottom map and moved North towards the top-most central room with the magic crystal and a boss monster. They could choose between the Western and Eastern routes.



Scene

Filter nodes

Player
ArcaneScholar
Turn
Traveler
Player
Actor
Destructible
Living
Organic
Controller
Manipulator
Common

Figure 6.10: Node composition of the player Entity prototype in *Grimoire: Ars Bellica*. As usual in most games, the player Entity was one of the most complex. The "Common" Property is greyed out because it is inherited from the base Entity Prototype (Prototypes in *Godot* have inheritance).

system kept track of Entity `initiative` values and iterated over them, using the *State* pattern (Gamma *et al.*, 1995) to handle any user interaction with each game turn. For instance, it waited on a `SelectTile` state whenever a spell required the player to aim at a particular position in the map. That state changes the user interface to guide the user into selecting the tile, then moved on to other states after they were done. After all Entities had a turn, a new round started and new `initiative` values were computed. Faster Entities were more likely to act earlier during a round.

Following the *Unlimited Rulebook* reference model, some Fields were Behavior Fields: either Abilities or Rules. The player's **Abilities** came mainly from their spells but monsters had innate Abilities (e.g., biting, spilling webs, breathing fire, etc.) and any Entity could have Triggered Abilities. Abilities implemented the *Interpreter* pattern (Gamma *et al.*, 1995) to allow Data-Driven combinations into unique mechanics. The nodes in the pattern were *Godot* nodes. The result of our **Ability Handling Service** "executing" an Ability was a series of **Effect** objects. We wrote a base Effect class and inherited it into specific Effect types to represent the many possible operations over the Simulation State. We used the "complete" design version of the *Unlimited Rulebook* Behavior Model where we separate **Rules** from Effects. That way, to "execute" an Effect, our **Effect Resolution Service** dispatched it to the Entity directly affected by it. Then, it iterated over its Properties and the Rule Fields they carried, matching the Effect type to a list each Rule had describing the Effects they could resolve. When a match occurred, the Effect Resolution Service called a method in the Rule passing the Effect as an argument, and each Rule specialization implemented its own resolution.

As specified in the Runtime Viewpoint of the *Unlimited Rulebook*, we classified how Rules interacted with the Simulation State into "changes incoming Effects" and/or "applies them to the Simulation State". Each Rule had two abstract methods it could implement that would be called by the Effect Resolution Service: `_process_effect` and `_apply_effect`. The first both received the Effect as an argument and had to return an Effect as a result. This allowed it to change the Effect object entirely, and the Effect Resolution Service would adjust accordingly. The second method, `_apply_effect`, was not supposed to change the Effect object (though this cannot be enforced in *Godot*'s scripting language). It could only change the Simulation State by accessing a reference to the local map node it also received as an argument. Let us illustrate this mechanism by using the turn order mechanics we described before.

Every round, the Progress Tracking Service sends `RollInitiativeEffect` instances into the Simulation State using the Effect Resolution Service. Instances are sent to all Entities that have the `Turn` Property. The `Turn` Property adds a Rule Field to its owner Entity that randomly determines its `initiative` Field value upon receiving a `RollInitiativeEffect` instance (see Listing 6.2). However, since Entities may act at different speeds, we added an attribute to the `RollInitiativeEffect` that indicates the speed at which the Entity is currently acting. By default, its value is zero, meaning the Entity acts at "average" speed. However, when an Entity has the Actor Property, it not only has a custom speed but can also perform special actions (e.g., the player casts spells) that may be faster or slower depending on the action itself. Thus, the Actor Property gives Entities a Rule that adds the total speed modifier of the action they intend to make that round to their current initiative roll (see Listing 6.3). Note that the initiative Rule uses the `_apply_effect` method since it changes the Simulation State by assigning a value to the `initiative` Field of Entities. The speed Rule, on the other hand, uses the `_process_effect` method because it changes the values of the `RollInitiativeEffect` attributes *before* the Effect is applied. In *Grimoire: Ars Bellica*, all "process" Rules concerning a given Effect object execute before the "apply" Rules. This way, the game supports self-amending mechanics because its Rule objects can intercept Effects and manipulate exactly how they will affect the simulation.

```
extends Rule

func _apply_effect(_map: Map, effect: Effect):
  var roll_effect := effect as RollInitiativeEffect
  var turn := get_self_property() as Turn
  turn.initiative = Dice.roll(2, 6) + roll_effect.get_speed()
```

Listing 6.2: The Rule for determining initiative mechanics in *Grimoire: Ars Bellica* when Entities have the `Turn` Property. Written in GDScript.

```
extends Rule

func _process_effect(_map: Map, effect: Effect) -> Effect:
  var roll_effect := effect as RollInitiativeEffect
  var actor := get_self_property() as Actor
  roll_effect = roll_effect.change_speed_by(actor.SPEED)
  if actor.next_action != null:
    var mod := actor.next_action.ability.action_speed_modifier
    return roll_effect.change_speed_by(mod)
  else:
    return effect
```

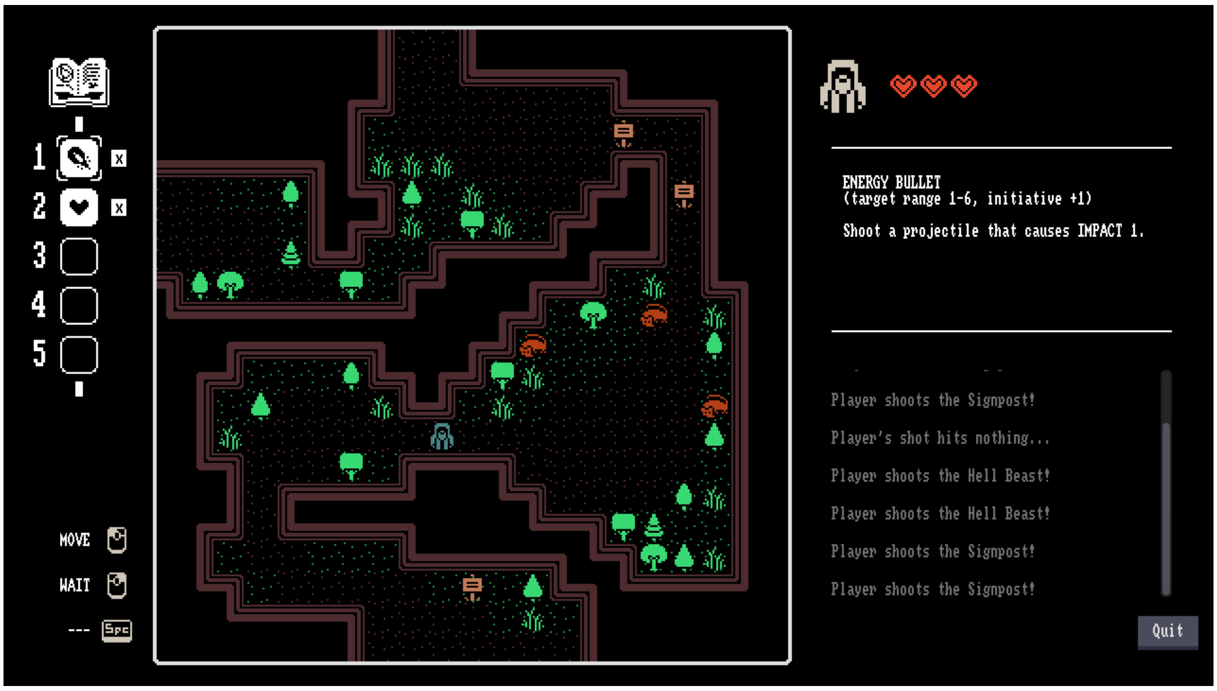
Listing 6.3: The Rule for adding the Entity's speed to their initiative roll in *Grimoire: Ars Bellica* when Entities have the `Actor` Property. Written in GDScript.

6.3.3 Results

We had three iterations of *Grimoire: Ars Bellica* since early 2019. We dropped the first two because the *Unlimited Rulebook* had evolved its design and because the team was reduced to a single developer. The third and current iteration is the one we presented here and corresponds to a much simpler game proposal with a reduced scope. We started developing this version in January 2021, right after we finished the third iteration of the *Unlimited Rulebook*. Development reached its first alpha release in August 2021 (see Figure 6.11) after over 180 hours of work, though we are still adding fixes and smaller improvements. Besides, as we said in Section 6.3.1, we intend to continue the development of *Grimoire: Ars Bellica* at least until we publish it as a complete game. There were eight intermediate releases during this first development stage[3]. The total of economy mechanics we implemented was:

- 23 unique Entity types, among which
  - 1 was the player avatar,

[3]https://gitlab.com/uspgamedev/grimoire-ars-bellica/grimoire-ars-bellica/-/tags (last accessed August 25th, 2021)



```
ENERGY BULLET
(target range 1-6, initiative +1)

Shoot a projectile that causes IMPACT 1.
```

```
Player shoots the Signpost!
Player's shot hits nothing...
Player shoots the Hell Beast!
Player shoots the Hell Beast!
Player shoots the Signpost!
Player shoots the Signpost!
```

Figure 6.11: Current *Grimoire: Ars Bellica* version as of this writing. The blue-robed figure near the center is the player's avatar and the orange rat-like creatures are monsters the player faces. To the left, the UI shows the player's grimoire and the spells they know. To the right, we see information about the current state of the player's avatar as well as tooltips for what the cursor is currently hovering and a message log to help describe the virtual world events to the player.

  – 1 was a special "environment" Entity for universal Rules,
  – 1 was the magic crystal at the end of the game,
  – 10 were different monsters with special abilities,
  – 6 were magical constructs produced by abilities (e.g., magical flames), and
  – 4 were common obstacles (e.g., rocks);

- 43 different Properties that could be combined to create new Entity types, among which
  – 17 were generic common Properties (e.g., `Turn`, `Actor`, etc.),
  – 11 were temporary conditions caused by Abilities,
  – 5 implemented IA strategies for monsters, and
  – 10 were custom, unique Properties for specific Entities;
- 20 unique Effect types;
- 42 unique Abilities, among which
  – 7 were common Abilities (e.g., walking, waiting, etc.),
  – 1 was a quest-related Ability (winning the game),
  – 27 came from spells, and
  – 7 were innate monster Abilities; and

- 46 unique Rules, among which
  - 26 were common Rules (e.g., initiative, speed, etc.),
  - 5 were custom Rules for specific Entities,
  - 10 were custom Rules for specific spells, and
  - 5 were reusable but more advanced Rules.

	Lines of code			**Methods**			**Classes**		
	Total	Core	Ext.	Total	Core	Ext.	Total	Core	Ext.
Simulation State/Rule	759	63	696	70	10	60	46	1	45
Simulation State/Ability	732	59	673	47	3	44	21	2	19
Ability Handling Service	664	137	527	83	25	58	13	4	9
Simulation State/Effect	577	122	455	136	16	120	22	2	20
UI Service	522	-	-	68	-	-	22	-	-
Simulation State/Field	479	74	405	60	9	51	30	3	27
Simulation State/World	256	256	0	32	32	0	4	4	0
State Persistence Service	199	199	0	16	16	0	5	5	0
Query Processing Service	139	139	0	15	15	0	4	4	0
Progress Tracking Service	138	138	0	15	15	0	1	1	0
Simulation Subsystem	104	104	0	20	20	0	1	1	0
Animation Service	104	-	-	12	-	-	6	-	-
Simulation State/Entity	91	91	0	8	8	0	2	2	0
Effect Resolution Service	69	69	0	5	5	0	2	2	0
AI Service	69	-	-	5	-	-	1	-	-
Simulation State/Primitive	64	0	64	4	0	4	2	0	2
Prototype Loading Service	61	61	0	5	5	0	3	3	0
Event Dispatching Service	18	18	0	3	3	0	1	1	0
Total Simulation Code	4350	1530	2820	519	182	337	157	35	122
Total		5045			604			186	

Table 6.2: Code analysis of *Grimoire: Ars Bellica* as of version 0.1. Each lines of the table breaks down the amount of lines of code, methods, and classes that belong to each element of the reference model, sorted by lines of code. The table also indicates how much of a given metric refers to code that is considered "core" and code that constitutes an "extension" (Ext.). Extension code is highly coupled to core code but less coupled to other extension code. Thus, in general, extension code is also cheaper to read and write. It is also more specific to the game and its mechanics, whereas core code is more generic and abstract.

As for the implementation details, Table 6.2 provides a simple overview. It shows how the different elements of the reference model of the *Unlimited Rulebook* ended up distributed across the codebase. The table also illustrates the distribution between core code and code that implements specific mechanics. For instance, there is only 1 abstract class for all Rules, which is inherited and implemented throughout 46 specific child classes (hence the "64 unique Rules" above). Note that Entity types in *Grimoire: Ars Bellica* are not defined by classes but via Prototypes, which, in this case, were *Godot* scenes. This means they are not part of the code but part of the game data — i.e., Entity types are Data Driven — and do not show up in Table 6.2. The high ratio of content and extension code compared to core code in key elements of the economy mechanics (Rules, Abilities, Effects, and Fields), suggest that:

1. The architecture of *Grimoire: Ars Bellica* is highly *extensible*;
2. Core code for economy mechanics was *reused* profusely;
3. Core code for economy mechanics offered *flexibility* to implement a wide variety of objects and behaviors;
4. Changing core code was easy because the amount of code to work through was comparatively small;
5. At the same time, extension code is inevitably coupled to core code and changes to the latter might require widespread changes in the former; and
6. Most of the final code consists of extension code, indicating that, to some extent, we spent most of our effort in the creative process of the game rather than in the technical process.

Let us elaborate on a few of these statements. Creating new Entities in *Grimoire: Ars Bellica* resulted in a very streamlined process. Besides using *Godot*'s WYSIWYG editor to easily support parametrical variations (Entities with different hit points, movement types, speed, etc.) the *Entity-Component-System* design supports the combination of reusable Properties into dozens of new Entity types. When the existing Properties are not enough, the effort needed to create new ones depends on the type of Property desired. Condition status Properties and Properties that carry only Rule Fields are the easiest to produce because adding Rules to Properties is also a mix-and-match process. Adding custom Fields to Properties requires specializing the Property class and so does adding custom Rules. For the most part, these end up being scripts that are seldom larger than the editor screen. The hardest part of making custom Properties is making sure to implement the `_save` method used by the State Persistence Service. As for custom Rules, the hardest part is properly handling the Effect objects and understanding how the Effect Resolution Service will process them.

Effects are the most expensive to produce and maintain. The current implementation offers a very limited array of attributes an Effect can carry. It was a limitation we added to the design because it made it easier to create specialized DCC tools for writing Abilities. If the number of possible attributes types an Effect can have is small, then the Ability editor can handle all possible cases manually. Adding new Effect attribute types thus requires changing the Ability editor too, hence its increased cost. Adding new Effects using pre-existing attributes is considerably easier.

Effects and Rules have another design limitation. Since our Effect Resolution Service only dispatches a particular Effect object to the specific Entities directly affected by it, it is hard for Rules in other Entities to access that Effect. As an example, the *Stasis Field* spell in *Grimoire: Ars Bellica* reduces the speed of Entities that are inside the area of its field. That means the Rules of the Stasis Field should be able to change Effect instances that pass through Entities inside it, but they cannot. We circumvented this by creating a special Effect type called `SenseEffect`, which represents an Entity *perceiving* an Effect in another Entity. Only Entities with the special

AreaSensor Property do this, though, because, otherwise, every Effect would pass through all Entities in a map[4]. This approach only works because the Effect that causes Entities to change positions is handled by the special *Environment* Entity, and it has a specific Rule that gives AreaSensor Entities an opportunity to affect position-related Effects. In other words, the current design of *Grimoire: Ars Bellica* couples Rules and Effects to the Entity that operates them.

Though not directly related to economy mechanics, a major bottleneck in the architecture of *Grimoire: Ars Bellica* is the UI and the interaction states controlled by the Progress Tracking Service. Adding new interaction states, especially for giving users ways to control their spells, is expensive. It easily requires a few hundred lines of code per state plus thorough debugging. Another bottleneck, this time mostly unrelated to programming, is that Abilities should have visual effects to communicate the players of their outcomes. Finding the right assets to use and scripting their animations took a considerable amount of our time in this regard. Similarly, creating new maps for *Grimoire: Ars Bellica* involves skill and time spent in level design to populate the game with the appropriate terrains, entities, and challenges.

One of the most experimental features we added to *Grimoire: Ars Bellica* was the possibility of *previewing* Effects. Since all Effect-changing Rules run before all Effect-application Rules we adapted the Effect Resolution Service to allow performing only the Effect-changing Rules for a given Effect and returning the result. This way we can, for instance, preview a RollInitiativeEffect dispatched to a specific Entity then inspect its speed bonus attribute to obtain the *effective* speed of the Entity — i.e., the speed that in fact applies to its initiative after processing all its Rules. We used this to support our Query Processing Service in a few cases. It was particularly useful to preview whether an Entity could move into a given position or not, since multiple physics and economy Rules affected mechanics related to this operation and we would rather not couple the queries to their specifics. Instead, we simply preview the result as if it would happen and act based on that.

6.3.4 Discussion

There are some limitations to what we can assess in this proof-of-concept. Though *Grimoire: Ars Bellica* was a larger game than the ones developed in the quasi-experiment of the second design iteration, it still is not on the same scale as most commercial games we used as information sources in this research. Having a single developer that played the role of both Technical and Creative Teams prevents us from asserting how much the *Unlimited Rulebook* really improves the creative process in general. As a game intended for distribution to end-users, the amount of playtesting we did was not enough to simulate the burden of a real Technical Team. All in all this study was still limited in scope, and the conclusions we draw from it require further validation.

That said, the study provided valuable information regarding the capabilities of the *Unlimited Rulebook*. By using a production-ready engine and pre-made assets whenever possible, we know that most of the effort we put into the game went into implementing economy mechanics from both technical and creative perspectives. The only major field we had to spend time on other than that was the UI — though an analysis of the commit history and devlog could provide

[4]We did this for a while until it caused severe performance issues.

stronger evidence regarding this statement. Given this, the study allowed us to experience the benefits and risks of using the *Unlimited Rulebook* for economy mechanics with some degree of isolation from external noise. Thus, it provided us insight into the capabilities of the reference architecture in a more or less “ideal environment”.

The results show us that the amount of content we produced is comparable to that of the smaller commercial games from the interviews in Section 3.3.4, considering we had a smaller Technical Team. We attribute part of this to how much *Godot* simplified many steps in the development. However, we can attest to how certain design patterns we used supported the creative process. The *Entity-Component-System*, the implementation we used for the Object Model as suggested by the *Unlimited Rulebook*, increased the reuse, flexibility, and extensibility of the architecture. This can be seen in how Properties became easy to mix into Entities, how implementing new ones required less effort than implementing an entirely new Entity type, and how the investment of making new Properties often paid off because other Entities used them. We saw this in the AI Properties we implemented in *Grimoire: Ars Bellica* because all 10 monsters in the game usually needed only 3 out of the 5 different AI Properties, so we had many opportunities to reuse them. The same happened with Properties that defined what types of damage Entities are resistant and vulnerable to. For instance, the `Undead` Property makes an Entity vulnerable to Life energy, so we can simply add it to any Entity to make it behave like an undead creature. If we need to extend what defines an undead, we simply update the `Undead` Property and the change propagates to all Entities that have that Property. We have considered using this to procedurally create new Entity types but left the idea for future improvements in the game. The *Unlimited Rulebook* guided us into how to make the best use out of the design paradigms *Godot* had to offer, promoting the design of an Object Model that becomes cheaper and cheaper to maintain as development advances.

Godot was also very helpful with its built-in Event Dispatching Service due to its support for co-routines. Sometimes, a spell had to play an animation and wait for it to finish before applying the Effects. Using *Godot* signals, we simply yield the computation of the spell so it waited for a signal indicating the animation had finished. Then, it resumed right where it stopped and we did not need any boilerplate code to restore its computation state. In the future, it might be worth including asynchronous programming support as one of the key variabilities in the *Unlimited Rulebook*. At the same time, because we implemented Abilities using the *Interpreter* pattern (Gamma *et al.*, 1995) following the Behavior Model guidelines, pausing the execution of an Ability between *Command* (Gamma *et al.*, 1995) instances was an effective and straightforward approach.

We used the Ability, Effect, and Rule pattern to implement the Ability Handling and Effect Resolution Service for the Behavior model — the most complete solution proposed in the *Unlimited Rulebook*. The implementation required a few design revisions until we reached the current approach, so we could not avoid wide changes to the codebase. Given the results, however, we believe we reduced the costs of these large changes considerably. For instance, when we introduced Effects and Rules to complement the *Interpreter* implementation of Abilities, the parts of the game system that consumed the Ability API barely changed. This was because we simply kept using the *Interpreter* pattern but, instead of changing the Simulation State directly, its computations relied on the new Effect Resolution Service. This was evidence that mapping our design elements into reusable Services, a basic concept in the reference model of the *Unlimited Rulebook*, reduced the effort required to extend and even re-design the implementation of economy mechanics. The part that required more effort, in this case, was re-creating all *Interpreter* nodes to follow the new paradigm, then updating all Abilities with the new node types where appropriate. This means the *Interpreter* pattern decouples Ability from other Subsystems and Services, but couples Abilities to the specifics of the pattern implementation. The earlier developers identify issues in this regard and fix them, the better, since the game will only have more and more Abilities as production moves forward. Now that we integrated that knowledge into the current version of the *Unlimited Rulebook*, we believe architects will be able to make more informed decisions that prevent this kind of cost.

There were a few other insights we had on the Effects and Rules mechanism that only became evident as we tried to produce monsters and spells with more unique Abilities. The fact that Effects only passed through the Rules of their primarily affected Entity forced us to use roundabout methods to make Rules that change how multiple Entities behave. The `SenseEffect` reduced this issue but we believe allowing Effects to pass through more than one Entity to be a more universal solution. However, that will require a more sophisticated dispatching mechanism that avoids passing an Effect through all Entities and Rules needlessly. Since the *Unlimited Rulebook* suggests architects use the Query Processing Service to locate Entities of interest, it can encapsulate how to find the Entities an Effect should be dispatched to. This way, the Effect Resolution Service becomes decoupled from how the dispatching process works.

Another limitation, as we explained in Section 6.3.2, was that Effect attributes were very rigid. To reduce the costs of adding new attribute types without increasing the cost of maintaining Ability editors, we consider using a simpler version of the *Entity-Component-System* or other pattern related to the *Adaptive Object-Model* to flexibilize Effect attributes while still supporting Data-Driven Design. That way, new Effect types could be made by combining pre-existing "Effect components". Making "multiple levels" of composition-based patterns seems like a particularly useful pattern in economy mechanics, based on our experience with *Grimoire: Ars Bellica*. While *Godot*'s composite-based paradigm might be a strong influence towards that impression, it also seems like a natural step after the insights the *Unlimited Rulebook* provides regarding composition over inheritance designs in the Object Model. For instance, we noticed that Rules, being nodes attached to Property nodes, allowed us to create new Properties by combining Rules the same way we create new Entities by combining Properties. We cannot tell whether this pattern could go further without causing any issue in the architecture. Last, we introduced the Effect preview feature very late in development and did not have many opportunities to ascertain its benefits and risks. It requires some careful design to prevent baseless recursion — it happened when we tried to preview long Entity movements because doing so required previewing how Entities made short movements. We also speculate the preview system could have performance issues without a more sophisticated design for more complex usage.

As we developed *Grimoire: Ars Bellica*, the *Unlimited Rulebook* gave us a design framework to work with. We could assess the trade-off of our implementation decisions because we knew the role they played in the simulation and what architectural elements depended on those decisions.

The guidelines from Section 5.3 provided a roadmap for the architecture, reducing the need to research possible solutions and weighing their benefits and risks through prototypes and speculation. The runtime perspective from Section 5.2 allowed us to think about the design more clearly and locate issues through a more informed process. More importantly, the reference model in Section 5.1 gave us a *language* to express, ponder, and criticize the architecture behind the economy mechanics of *Grimoire: Ars Bellica*.

In this chapter, we evaluated the *Unlimited Rulebook* reference architecture using two different methods spread through three design iterations of the project. With this, we finished detailing our research. Next, we move on to the conclusions of this thesis and elaborate on possible future works.

Chapter 7

Conclusions

"Surely, there is some reason you are going through all the trouble of trying to design great games."

Schell (2020)

I chose to graduate as a computer scientist because I wanted to make games. I wanted the knowledge, experience, and tools that would allow me to turn into reality the many game ideas that haunt me to this day. What I found, however, was that making a simple sprite move over the computer screen required hundreds of lines of code — if not code I wrote, then code inside an engine. There was always some *latency* between having an idea and seeing it running. The frustration this caused naturally led me to dive deeper and deeper into the field of software architecture, looking for means to reduce that latency.

In this research project, we determined one of the many causes behind this problem and this, in turn, enabled us to design a solution for it. We learned about economy mechanics and self-amending rules as concepts that embodied the types of game features we had struggled with in the past. We understood that, because these mechanics usually break away from standards and conventions (compared to physics mechanics), it can be very challenging to design universal solutions. The conclusion we arrived at is that supporting any possible rule involves, in the worst-case scenario, first-class support to exceptions to the rule. That is how the central design of the *Unlimited Rulebook* came to be.

However, economy mechanics and the creative process behind them do not exist in isolation. Digital games are complex systems developed in complex environments. Thankfully, many of the other parts of a game system and its development process have decades' worth of research and industry experience. Our job was to shape our solution in a way that acknowledged and properly interfaced with the other elements that make up a game system. With this, we reinforced its purpose of reducing the friction between having ideas and reifying them. The *Unlimited Rulebook* we proposed is, thus, what we call a *partial reference architecture* — it guides the design of a specific part of a game system but includes the means for seamlessly integrating that part with the other parts.

That said, our work is far from complete. As we saw in one of our previous publications, the field of software architecture in games lacks a formal body of knowledge regarding the implementation design of mechanics (Mizutani *et al.*, 2021). That makes the *Unlimited Rulebook* one of the first steps into this "new" field. As such, the research opportunities are still many and we sincerely hope that more computer scientists and software engineers show interest in the subject. For now, we end this thesis by summarizing our main achievements, discussing the conclusions we drew from our research, and what we consider essential lines of research for the future of the *Unlimited Rulebook*.

7.1 Achievements

The following are the achievements of our research and how each of them addresses the research questions we listed on Section 3.1. As part of our research project, we published three academic works:

- A **systematic literature review article** on software architecture applications in the field of game mechanics (Mizutani *et al.*, 2021), published in the *Entertainment Computing journal*. Although the last published, the systematic literature review was one of the first steps in our research. With it, we formed an initial answer to both RQ1 and RQ2 based on the state of the art of software architecture applied to game mechanics. It also contributed to Step RA-1 of the ProSA-RA method.
- A **short paper** on the architectural requirements of economy mechanics (Mizutani and Kon, 2019), published in the *Proceedings of the 2019 Brazillian Symposium on Computer Games and Digital Entertainment* (SBGames 2019). It was our first time publishing the results of Step RA-2. This data was the basis for answering RQ3 and RQ4 later on.
- A **full paper** on the second iteration of the *Unlimited Rulebook* (Mizutani and Kon, 2020), published in the *Proceedings of the 2020 International Conference on Software Architecture* (ICSA 2020). The discussion it includes about self-amending mechanics composes one of our main answers to RQ3 and the design of the *Unlimited Rulebook* at the time was our latest take on RQ4. Included progresses on both Step RA-3 and Step RA-4.

Aside from these, other important achievements in our research include:

- A new introductory course to game programming at the University of São Paulo under its Summer school program, open to the general public. Part of the process for conducting a quasi-experiment to validate our answer to RQ4 through the *Unlimited Rulebook* in Step RA-4.
- A new undergraduate course on game programming aimed at senior computer science students. Served the same role as the previous one but applied to the second iteration of the *Unlimited Rulebook*.
- *Grimoire: Ars Bellica*, a fully playable proof-of-concept demonstrating the results of our research, which includes a reference implementation of the *Unlimited Rulebook*. Provides evidence to our approach to RQ4 as part of Step RA-4.

- The still-in-progress formalization of the Ability-Effect-Rule design pattern. This is another part of our answer to RQ4.
- A new software architecture perspective on the design and implementation of mechanics in game systems, in particular of economy mechanics with self-amending rules. It is the general rationale our research used for addressing the reserach questions as a whole, establishing the relationships betweeb the creative process of economy mechanics, the development pipeline of games, and the software architecture of games as interactive simulations.

7.2 Discussion

Much of the journey behind the *Unlimited Rulebook* was about giving shape and structure to concepts we were familiar with but did not know how to address. The very first obstacle of this sort was trying to explain why to focus on economy mechanics as a niche for game software architecture. As with any classification method, our division of mechanics into physics, economy, and progression (Section 2.1.4) serves a purpose rather than establishes any fundamental truth. What was particularly special about economy mechanics then? It was only after we understood that games, because they are simulations, represent complex but non-existent systems (the abstract virtual world) with a simpler system (the implementation of the virtual world), that we were able to phrase the issue properly. The simpler and more symbolic the representation, the more arbitrary it is and the harder it becomes to design universal solutions. Similarly, we only managed to state the challenge of extending economy mechanics by looking at the self-amending rules of *Nomic* (Peter Suber, 1982) and the issues of programming interactive fiction presented by Andrew Plotkin (2009). We believe that finding the right names and models to describe our research was one of the key steps this thesis took to establish the foundation for this field of research.

On the other hand, our progress towards giving form to our knowledge also led to very well-known territory. In particular, a question we often wrestled with was: what does it mean for an architecture to make a game easier to develop? The concepts of technical debt (Fowler, 2019), reusability, flexibility, extensibility, and learning curve — among many others — are all things we take for granted as we graduate as computer scientists and software engineers, but are challenging to define formally and even harder to measure objectively. Each of them has several lines of research dedicated to unraveling their intricacies. We learned this over and over again in the struggle to design, apply, and analyze the quasi-experiment to validate the *Unlimited Rulebook*. Our lasting impression from that empirical study is that objectively measuring certain qualities of software architectures is impractical. Small, controlled samples fail to attest to real-world game systems because our architecture is particularly aimed at larger, long-lasting projects. However, big games like these take time to produce and, to provide a comparable reference to assess the qualities of the architecture, would require at least two implementations. At the same time, the formalities for proposing studies involving programmers demand solid planning and institutional infrastructure. We believe there are many other ways to validate an architecture but we ran out of scope, time, budget, and opportunity.

That said, the results of the *Unlimited Rulebook* itself are evident. While it does not include any entirely unprecedented idea, it fulfills its prime role for existence: it provides a *frame of reference*. We believe this becomes evident in the Ability-Effect-Rule pattern we proposed in Section 5.3.2. Even though we relied mainly on works from widely different contexts — the chemistry engine from Fujibayashi *et al.* (2017), the event mechanism from Bucklew (2015), the rule-based programming approach from Andrew Plotkin (2009), the complex rules of *Magic: the Gathering* (Wizards of the Coast, 2021), and the open-ended self-amending rules from *Nomic* (Peter Suber, 1982) — we still found what we believe to be the pattern in these solutions. Predicate-dispatching is a programming language feature but, if we consider Abilities as compilers of Effects and Rules as dispatching mechanisms, we can argue how our proposal meets the criteria for self-amending rules while maintaining a structure game developers are more likely to assimilate. We hope to see more study on this pattern and, more importantly, its use in commercial, "real" games.

At the same time, we found that isolated solutions cannot solve everything. Thanks to the works of Gregory (2019) and Plummer (2004), which are *other reference architectures* for games, we developed a reference model to express the role of economy mechanics simulation among the many gears that make a game system tick. We identified specific challenges in the synchronization with input and output Subsystems, requiring coordination with the Progress Tracking and Event Dispatching Services. In particular, we realized that, sometimes, what makes a change in the economy mechanics expensive is how coupled they are to completely different Subsystems. For instance, making Abilities that use new types of targeting mechanics require new interaction states in the Game System. Not to mention that most new forms of mechanics involve making new assets and implementing them into the Output Rendering Subsystems. In this sense, our study adds to the mass of support for Data-Driven Design in games and provides reusable knowledge of ways to integrate it with economy mechanics.

7.2.1 Applications of the *Unlimited Rulebook*

As a reference architecture for game development, we designed the *Unlimited Rulebook* for systems with certain characteristcs. We assumed the gameplay of the target products emphasize economy mechanics and that the project either (a) bolsters a large volume of gameplay content or (b) involves a long development lifetime filled with content updates, or both. There are no hard assumptions about specific team layouts but we did favor the possibility that non-programmers would create content and mechanics for the game. These constraints suggest certain profiles the *Unlimited Rulebook* ought to work benefit more. For instance, competitive multiplayer games with a growing rooster of unique entities — like collecting card games or multiplayer on-line battle arenas — fit most of the design assumptions. Other genres that meet only one or two of the assumptions include *rogue-likes* (complex economy mechanics, sometimes with long development lifetimes) and sandbox games (large volume of gameplay content), but many role-playing and strategy games would benefit too, such as *Path of Exile* (Grinding Gear Games, 2013–2021), *Warcraft 3* (Blizzard Entertainment, 2002), and *Sid Meyer's Civilization V* (Firaxis Games, 2010). A particular case where the *Unlimited Rulebook* would greatly reduce development effort and costs is in the development of series of a games, like in the *Pokémon* franchise (Game Freak, 1996–2021), because the core mechanics are similar enough that the developers can write a single reusable framework for all titles.

However, most of this assessment is theoretical, given that information sources we based the *Unlimited Rulebook* on. The games we managed to evaluate the *Unlimited Rulebook* with were predominantly role-playing games with turn-based gameplay, and all were single-player, 2D games meant to run only on computers and not mobile devices or consoles. In other words, the evidence we used to design the *Unlimited Rulebook* strongly suggests that it should support a wide variety of games but we could not validate that empirically yet. It should be perfectly possible, for example, to use the *Unlimited Rulebook* in serious games and even other forms of digitally gamified experiences. At the same time, we infer that, for some types of games, using the *Unlimited Rulebook* would incur more costs than it would save. For instance, simple action-platformer games like the classics from the *Super Mario* franchise (Nintendo, 1985–2021) do not have enough economy mechanics to benefit from the reusability and exxtensibility of our reference architecture.

There are two more or less unexpected applications of the *Unlimited Rulebook* we stumbled upon during the late stages of the research but did not have the time to formalize as part of the thesis. One of them is *procedural content generation*, a practice where part of the content of a game is produced by algorithms that rely on pseudo-randomness to provide variety. The classic example is for generating the topography of the virtual world of games but, with the *Unlimited Rulebook*, developers can procedurally generate content for economy mechanics too. Since the *Ability-Effect-Rule* pattern reifies Rules into reusable object types that designers can mix-and-match to create new unique entities, the same could be theoretically done via procedural content generation. This idea came up during the design process of *Grimoire: Ars Bellica*, when we imagined it would be possible to generate unique monsters by randomly combining compatible Rules — effectively producing unique ecologies each time the game was played. If we managed to attach Rules to spells too, we could even generate random spells. In the end, we left the idea behind due to lack of time, so we cannot say for certain whether it would be possible to produce interesting and balanced content for a game like this.

We found out the second unexpected application when we developed *Legend of Slime*[1] during a game jam between USPGameDev and game research groups from other colleges in our state. We managed to use a simplified version of the *Ability-Effect-Rule* pattern much like the original proposal of Bucklew (2015). In this implementation, Abilities were hard-coded and Effects were common associative tables (i.e., dictionaries), with Rules written using only the bare minimum infrastructure required. Despite being a 48-hour jam, the limited scope of the project still benefitted from this partial use of the *Unlimited Rulebook*, because it allowed us to implement over a dozen interactions between effects and rules in a short amount of time while still keeping a coherent architecture that could be extended further. This suggests that under the right circumstances, the benefits of the *Unlimited Rulebook* are still present even if some of its design assumptions are not present.

Despite all the promise we see in the possible applications of the *Unlimited Rulebook*, the main challenge we must face going forward is its accessibility. Right now, we are essentially the only ones that understand it and know how to use it. To fully achieve our goal of reducing the

[1]https://kazuo256.itch.io/legend-of-slime (last access November 21st, 2021)

costs of developing economy mechanics in games, we need game architects to learn the *Unlimited Rulebook*. While our research shows the theoretical and practical value of this architectural solution, communicating it in an accessible and didactic way to actual programmers is an entirely separate issue. We imagine it might even be the case of re-designing some parts of the *Unlimited Rulebook* to simplify or streamline its presentation and widespread use. Now that we have working results, it is time to plan ahead the next steps.

7.3 Future Work

As the last part of this thesis, we list here a few among many possible works that would further contribute to the field of architecting economy mechanics in digital games.

More design cycles for the *Unlimited Rulebook*. There will always be room for improvement. If possible, we would like to continue researching the *Unlimited Rulebook* and iterating over its design. Many of the future works proposed here, in particular, involve methods for evaluating our reference architecture to determine more of its limitations and provide insight into how to improve its design.

Formalization of the Ability-Effect-Rule pattern. While similar to the *Decorator* and *Chain-of-Responsibility* patterns (Gamma *et al.*, 1995), we believe the Ability-Effect-Rule pattern for implementing Behavior Models in games consitutes a new design pattern altogether. Publishing a paper or article on this subject would further contribute to the field of software architecture in games.

Architectural requirements for economy mechanics over different stages of the development cycle of games. Because developing a game incurs different demands over its lifetime, we believe it is important to consider how the architectural requirements we gathered in this research relate to each development stage of a game. Which are more important early on? What should architects prioritize as the game system matures?

User interaction and asynchronous programming. In the development of *Grimoire: Ars Bellica*, one of our conclusions was that the cost of adding new user interaction features (e.g., new targeting methods) was one of the key limitations to the creative process of making new spells. We believe further investigation into how to seamlessly integrate the Ability-Effect-Rule pattern with Subsystems and Services is needed, especially regarding control modes and the opportunities that asynchronous programming (e.g., co-routines) provide.

Effect previews as Simulation State queries. In a related topic, we only scratched the surface of what is possible and what challenges come with previewing Effects as a way to assess the Simulation State. Previewing the consequences of a player's action is a problem we have struggled with in the past because it often leads to duplicated code (the code that executes and the code that previews). We believe that using the Effect Resolution Service to apply only Effect-processing Rules offers a potential alternative to hard-coded previews that is reusable (because it uses part of the execution code), flexible (because it promptly adapts to new execution forms), and extensible (because making new types of preview scales together with making new Effects).

Using the *Unlimited Rulebook* as an architectural analysis tool. Our original plans for evaluating the *Unlimited Rulebook* included a case study of *Backdoor Route*, a deck-building rogue-like game developed by USPGameDev (USPGameDev, 2020). We would analyze its architecture using the *Unlimited Rulebook* as a reference to provide insight into how the game could improve its support for the creative process of economy mechanics. This would, in turn, demonstrate that the *Unlimited Rulebook* can be used to assess existing implementation besides its usual role in designing new architectures.

Commit history analysis on *Grimoire: Ars Bellica*. Thanks to version control tools, we have definitive records of the evolution of codebases. By analyzing the distribution of code contributions to economy mechanics compared to other kinds of features, we could, for instance, have a clearer picture of how *Grimoire: Ars Bellica* benefited from using the *Unlimited Rulebook* and what costs it paid for that choice.

Feedback from industry experts. A survey study aimed at game industry experts would provide a more formal validation to the *Unlimited Rulebook* while still contemplating the subjectiveness of individual experiences. The design of the study could draw from FERA, a checklist for reference architectures used in embedded systems (Santos *et al.*, 2013), and adapted for game systems.

Education about and diffusion of the *Unlimited Rulebook*. Though not a research project, we believe that teaching game developers about the *Unlimited Rulebook* — and the many learned lessons that came from it — is a crucial follow-up measure to promote further study in the field of software architecture and economy mechanics in games.

Making more games using the *Unlimited Rulebook*. Similarly, the best way to complement formal research on the *Unlimited Rulebook* is to put it into practice. By making games of varying genres, platforms, and target audiences, we can produce more data and experience regarding our reference architecture. It is especially important to see other people using it to collect their impressions. In particular, experimenting with real-time games is something we did not habe time to do in this thesis.

More important, however, is that maybe now it will be easier to exorcise the endless game ideas that haunt me at night.

Appendix A

Semi-Structured Interview Data

Table A.1, Table A.2, Table A.3, and Table A.4 contain the data we collected from the semi-structured interviews I01, I02, I03, and I04, respectively. The purpose and protocol of the interviews is available in Section 3.3.4. Some fields are missing because either they did not apply to the game in question or because the interviewee could not answer for any reason. The game from I04 was in the process of switching from a prototype implementation, which was completely discarded, and a new, definitive implementation written from scratch. Because of this, some fields have information pertaining to one or both the implementations — in which case we explicitly tell where the answer comes from.

Interviewee's Profile	
Background	Masters in Computer Science
Company Position	Senior Developer
Professional Experience	4 years
Company Experience	4 years
Team Size	5–10 employees
Company Size	Around 30 employees
Game Traits	
Genre	Idle Game
Target Platform(s)	Mobile
Development Time	2 months for initial development, 5+ years of ongoing updates
Core Mechanics	Unit purchasing, unit upgrades, currency production per unit
Architecture Design	
Component Reuse	In-house fork of open-source engine, in-house utility libraries, and third-party physics engine
Object Model	*Ad hoc*, object-oriented design
Entity Creation Workflow	Hard-coded tables in code
Prototype Loading	Copy data field by field
Ability Handling	No custom abilities
Triggered Abilities	Does not apply
Ability Targeting	Does not apply
Event Dispatching	Used together with the *Property* pattern (see below)
Effect Resolution	Mostly hard-coded but often relied on an in-house, reusable implementation of the *Property* pattern (Yoder and Johnson, 2002) to track a Field with a scalar value that changed with some periodicity; it provided common Rules for in-game currencies that accumulated over time and could be spent or bought
Production Scale	
Total number of Entities	30–100
Total number of Abilities	Less than 30
Total number of Rules	Less than 30
Major architectural changes	2
Architectural Analysis	
Main issue(s)	The code is old, obsolete, and poorly designed; scales terribly
Likely cause	At the time, the company produced multiple games in very short periods of time, leading to unsustainable practices
Reason for not solving	Would require re-writing the game from scratch, which the company cannot afford
Learning curve	New programmes are often caught off-guard by unexpected collateral behavior that is not clear from the API of in-house libraries

Table A.1: Interview I01 data.

Interviewee's Profile	
Background	Bachelor in Mathematics
Company Position	Gameplay Programmer
Professional Experience	8 years
Company Experience	1 year
Team Size	10 members but the entire project has over 200 employees
Company Size	Over 1800 employees
Game Traits	
Genre	Action Role-Playing Game
Target Platform(s)	PC, console
Development Time	2 years and going
Core Mechanics	Real-time fantasy combat, hit points, stamina, other combat statistics, potions, passive abilities, skills with cooldown, upgradeable equipment, air jumps, reward system
Architecture Design	
Component Reuse	In-house code from previous games
Object Model	Characters have hierarchy, parts of state kept in singletons
Entity Creation Workflow	DCC tools integrated into engine for characters and equipment
Prototype Loading	Prototypes are directly copied to memory as the initial entity state
Ability Handling	Partially implemented by a state machine
Triggered Abilities	Supported via a state machine combined with the Event Dispatching mechanisms described below
Ability Targeting	Information unabailable
Event Dispatching	Two Services, one synchronous and one asynchronous
Effect Resolution	Has a stack for Rules that mostly manipulate character statistics; Rules come from equipment, potions, areas affected by other entities; Rules may add Triggered Abilities and have Predicates the check the character state via code
Production Scale	
Total number of Entities	30–100
Total number of Abilities	30–100
Total number of Rules	100–500
Major architectural changes	Constantly
Architectural Analysis	
Main issue(s)	Monolithic singletons with accummulated technical debt; multiple instances of duplicated code
Likely cause	Legacy code reused across many games over decades, which the company spares no resources to work on between projects; change only happens when it becomes inevitable
Reason for not solving	Would require refactoring the entire codebase because any part has access to read from and write to the singletons
Learning curve	Steep but the company culture of teaching new employees through direct interaction with seniors works well; documentation is incomplete and obsolete

Table A.2: Interview I02 data.

Interviewee's Profile	
Background	Masters in Game Design and Development
Company Position	Game Engineering Manager
Professional Experience	4 years
Company Experience	4 years
Team Size	8 members
Company Size	500–1000 employees
Game Traits	
Genre	Real-Time Strategy Card Game
Target Platform(s)	Mobile
Development Time	1 year plus 6 months of updates
Core Mechanics	Real-time combat between troops, cards, spendable energy that recharges with time, cooldowns, areas of effect, combat statistics, multiple possible card effects
Architecture Design	
Component Reuse	Used the *Cocos-2dx*[1] engine
Object Model	*Entity-Component-System*
Entity Creation Workflow	Manually written JSON files
Prototype Loading	Hard-coded in a monolithic factory class
Ability Handling	Cards only instantiate entities, components enable multiple automatic Abilities
Triggered Abilities	Mostly attached to collisions and timers, only resulted in new entity instances (with a few exceptions)
Ability Targeting	Only one: drag and drop cards
Event Dispatching	Custom Service using engine built-in support
Effect Resolution	Command objects scheduled a few ticks ahead for determinism; no Rules ever applied besides default behavior of Effects
Production Scale	
Total number of Entities	30–100
Total number of Abilities	1–30
Total number of Rules	0
Major architectural changes	3 (unrelated to simulation)
Architectural Analysis	
Main issue(s)	Entity factory expensive to maintain; some Entity components too complex; duplicated code for starting matches
Likely cause	Maintenance costs were still cheaper than refactoring
Reason for not solving	Lack of company culture for investing in software architecture (which changed with this project after the qualities of its architecture were recognized)
Learning curve	Less experienced developers did not see the benefits of Data-Driven Design at first

Table A.3: Interview I03 data.

Interviewee's Profile	
Background	Masters Computer Science
Company Position	Lead Software Engineer
Professional Experience	8 years
Company Experience	3 years
Team Size	10 engineers
Company Size	10–50 employees
Game Traits	
Genre	Massive Multiplayer Online Role Playing Game
Target Platform(s)	PC
Development Time	1 year prototype, 1 year pre-production, 6 months production
Core Mechanics	Real-time fantasy combat, equipment, spells, skills, hit points, mana, cooldown, potions, proficiency levels with experience points, combat statistics, statuses, crafting
Architecture Design	
Component Reuse	Developed with *Unity3D*
Object Model	Prototype was inheritance-based; final game uses ECS
Entity Creation Workflow	*Unity3D* prefabs; final game includes making new components via code and a DCC tool Abilities using visual programming
Prototype Loading	*Unity3D* prefabs
Ability Handling	Prototype used hardcoded Abilities; final game uses the *Interpreter* pattern
Triggered Abilities	Only physical triggers
Ability Targeting	Final game has specific *Interpreter* nodes for quering input
Event Dispatching	Information unavailable
Effect Resolution	Prototype had hardcoded resolution with a reusable Service for applying simple Rules that affected combat statistics; final game will rely on ECS but exact implementation is unknown
Production Scale	
Total number of Entities	100–500 (prototype)
Total number of Abilities	1–30 (prototype)
Total number of Rules	1–30 (prototype)
Major architectural changes	3 (one of which was abandoning the prototype)
Architectural Analysis	
Main issue(s)	Inheritance-based object model was unsustainable
Likely cause	Prototype scope was very small and clear; development cycles were too long
Reason for not solving	Though they did leave the old design behind, they insisted for a while because the game worked and the results kept the team excited
Learning curve	New programmers need about one month to get used to the new ECS architecture

Table A.4: Interview I04 data.

Appendix B

Architectural Requirements

Table B.1 presents all the system requirements we collected from information sources, either publications or games. We refrained from citing games using LaTeX hyperlinks to save space but they can all be found in the Ludography at the end of the thesis and in Table 3.5. We took the liberty of using some shorthands to further reduce the space used by the lists of games, like "All games" and "Same as above" (referring to the table line immediately above it). Sometimes there are no game sources listed. This either means that using a game as an information for that requirement is not applicable or it means that we could not find evidence in any game regarding that requirement. That said, since we did not play all games, there are likely many cases where more game sources apply. We only listed the ones we were sure about.

(ROM-1) Entity Data Representation

System Requirement	Publication Sources	Game Sources
Agent-based simulation	Gregory (2019) Rollings and Ernest (2006) Adams and Dormans (2012) Schell (2020)	All games
Entity types	Gregory (2019) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012) Schell (2020)	All games
Entity type relations (e.g., hierarchy, composition)	Gregory (2019) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012)	All games
Entities w/ spatial properties (e.g., capacity occupation)	Rollings and Ernest (2006) Adams and Dormans (2012)	All games

Schell (2020)

(ROM-2) Runtime Entity Management		
System Requirement	Publication Sources	Game Sources
Dynamic creation and destruction of entities	Gregory (2019) Rollings and Ernest (2006) Adams and Dormans (2012) Schell (2020)	All games
Entity queries	Gregory (2019)	—
Entity references	Gregory (2019)	—

(ROM-3) Entity State		
System Requirement	Publication Sources	Game Sources
FSM support for entities	Gregory (2019) Schell (2020)	—
Entity fields with numeric state	Dormans (2012a) Rollings and Ernest (2006) Adams and Dormans (2012) Schell (2020)	All games
Abstract resources	Adams and Dormans (2012)	All games
Entity fields with symbolic state	Rollings and Ernest (2006) Schell (2020)	All games
Resource modifiers	Dormans (2012a) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012)	All games
Procedural resource modifiers	Wizards of the Coast (2021) Rollings and Ernest (2006) Dormans (2012a) Adams and Dormans (2012)	*Dungeon Crawl: Stone Soup* *Hearthstone* *Magic: the Gathering*
Permanent entity upgrades	Rollings and Ernest (2006) Schell (2020)	—

Entity transformation modifiers	Wizards of the Coast (2021)	*Caves of Qud* *Dungeon Crawl: Stone Soup* *Dota 2* *Final Fantasy Tactics Advance* *Magic: the Gathering* *NetHack* *Path of Exile* *Pokémon* series *Warcraft 3*
Ability as Fields	Rollings and Ernest (2006) Wizards of the Coast (2021)	All games that have abilities
Rules as Fields	Rollings and Ernest (2006) Wizards of the Coast (2021)	All games that have rules EXCEPT: *Terraria* *The Legend of Zelda: Breath of the Wild*

(RSP-1) Simulation time tracking

System Requirement	Publication Sources	Game Sources
Simulation time	Gregory (2019) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012)	All games
Turn tracking	Rollings and Ernest (2006) Schell (2020)	All turn-based games

(RSP-2) Time-based processes

System Requirement	Publication Sources	Game Sources
Turn-based routines	Dormans (2012a) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012)	All turn-based games
Real-time, per-frame routines	Gregory (2019)	All real-time games

(RSP-3) Progress Detection

System Requirement	Publication Sources	Game Sources
In-game goal and failure detection	Rollings and Ernest (2006) Adams and Dormans (2012) Schell (2020)	All games

(RSP-4) Simulation-Wide Finite State Machines

System Requirement	Publication Sources	Game Sources
Simulation mode tracking	Rollings and Ernest (2006) Schell (2020)	All games

(RBM-1) Simulation Processes

System Requirement	Publication Sources	Game Sources
Processes (per-frame state changes)	Rollings and Ernest (2006)	All real-time games
Process begin/end triggers	Rollings and Ernest (2006)	All real-time games

(RBM-2) Simulation Effects

System Requirement	Publication Sources	Game Sources
Effects (discrete state changes)	Dormans (2012a) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012) Schell (2020)	All games
Chained Effects (composed state changes)	Dormans (2012a) Adams and Dormans (2012) Wizards of the Coast (2021)	All games
Time-based Effects (periodic state changes)	Dormans (2012a) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012)	All games

(RBM-3) Entity Abilities

System Requirement	Publication Sources	Game Sources
Abilities	Wizards of the Coast (2021) Rollings and Ernest (2006)	*Caves of Qud* *Dungeon Crawl: Stone Soup*

		Diablo *Dota 2* *Final Fantasy Tactics Advance* *Guild Wars* *Hearthstone* *Magic: the Gathering* *NetHack* *Path of Exile* *Pokémon* series *Ragnarok Online* *Terraria* *The Battle for Wesnoth* *Veloren* *Warcraft 3*
Time-based Ability triggers	Wizards of the Coast (2021) Rollings and Ernest (2006)	Same as above
Event-based Abilities triggers	Wizards of the Coast (2021) Rollings and Ernest (2006)	*Caves of Qud* *Dungeon Crawl: Stone Soup* *Dota 2* *Final Fantasy Tactics Advance* *Guild Wars* *Hearthstone* *Magic: the Gathering* *NetHack* *Path of Exile* *Pokémon* series *Warcraft 3*
Linked Abilities	Wizards of the Coast (2021)	*Magic: the Gathering*
Abilities w/ Conditions	Wizards of the Coast (2021)	*Dungeon Crawl: Stone Soup* *Guild Wars* *Hearthstone* *Magic: the Gathering* *NetHack* *Pokémon* series
Abilities with Memory	Wizards of the Coast (2021)	*Guild Wars* *Hearthstone* *Magic: the Gathering*

(RBM-4) Custom Simulation Rules

System Requirement	Publication Sources	Game Sources
Rules	Wizards of the Coast (2021) Schell (2020)	Any included in the following groups
Effect resolution rules	Wizards of the Coast (2021) Schell (2020)	*Caves of Qud* *Dwarf Fortress* *Hearthstone* *Magic: the Gathering* *NetHack* *Pokémon* series *Terraria* *The Legend of Zelda: Breath of the Wild*
Effect composition rules	Wizards of the Coast (2021)	Same as above
Treatment replacement rules	Wizards of the Coast (2021)	*Caves of Qud* *Magic: the Gathering* *NetHack* *Pokémon* series
Procedural effect resolution rules	Dormans (2012a) Wizards of the Coast (2021) Adams and Dormans (2012)	*Caves of Qud* *Hearthstone* *Magic: the Gathering* *NetHack*
Effect replacement or prevention rules	Rollings and Ernest (2006) Wizards of the Coast (2021)	*Caves of Qud* *Hearthstone* *Magic: the Gathering* *NetHack* *Pokémon* series
Customizable trigger rules (determining WHEN they happen)	Rollings and Ernest (2006) Wizards of the Coast (2021) Adams and Dormans (2012)	*Caves of Qud* *Dota 2* *Guild Wars* *Hearthstone* *Magic: the Gathering* *Path of Exile*

		Pokémon series *Warcraft 3*
Rule modifiers	Wizards of the Coast (2021)	*Caves of Qud* *Hearthstone* *Magic: the Gathering*
Rule overriding precedence	Wizards of the Coast (2021)	Same as above

(RSG-1) Simulation Generality

System Requirement	Publication Sources	Game Sources
Multi-genre support	Plummer (2004)	—
Real-time and turn-based support	Dormans (2012a) Adams and Dormans (2012) Schell (2020)	*Dwarf Fortress* *Loop Hero*

(RIC-1) Simulation Interaction

System Requirement	Publication Sources	Game Sources
Activated Abilities	Wizards of the Coast (2021) Rollings and Ernest (2006)	*Caves of Qud* *Dungeon Crawl: Stone Soup* *Diablo* *Dota 2* *Final Fantasy Tactics Advance* *Guild Wars* *Hearthstone* *Magic: the Gathering* *NetHack* *Path of Exile* *Pokémon* series *Ragnarok Online* *Terraria* *The Battle for Wesnoth* *Veloren* *Warcraft 3*
Ability targeting	Wizards of the Coast (2021)	Same as above
Modal Abilities	Wizards of the Coast (2021)	*Hearthstone*

		Magic: the Gathering
Transaction enablers	Dormans (2012a) Wizards of the Coast (2021) Rollings and Ernest (2006) Adams and Dormans (2012) Schell (2020)	All games
Action handling by the simulation	Rollings and Ernest (2006) USPGameDev Schell (2020)	All games EXCEPT: *BYTEPATH*
Action validation between UI and simulation	Rollings and Ernest (2006) USPGameDev Schell (2020)	Same as above
Entity ownership management	Wizards of the Coast (2021) USPGameDev	All games EXCEPT: *BYTEPATH* *Diablo* *Factorio* *Loop Hero* *Minecraft* *Path of Exile* *Ragnarok Online* *Terraria* *The Legend of Zelda: Breath of the Wild* *Veloren*
Integration between simulation and interaction modes	Rollings and Ernest (2006) USPGameDev Schell (2020)	All turn-based games PLUS *Loop Hero*

(RIC-2) Simulation Events

System Requirement	Publication Sources	Game Sources
Event registration and handling	Gregory (2019)	*BYTEPATH* *Diablo* *The Battle for Wesnoth* *Veloren*

Simulation events to other subsystems	Rollings and Ernest (2006) Schell (2020)	Same as above

(RIC-3) Simulation Queries

System Requirement	Publication Sources	Game Sources
Simulated visibility	Wizards of the Coast (2021) Schell (2020)	*Caves of Qud* *Dungeon Crawl: Stone Soup* *Diablo* *Dota 2* *Dwarf Fortress* *Factorio* *Hearthstone* *Magic: the Gathering* *NetHack* *Sid Meyer's Civilization V* *Terraria* *Warcraft 3*
Simulation previews	USPGameDev	*Dungeon Crawl: Stone Soup* *Factorio* *Final Fantasy Tactics Advance* *Sid Meyer's Civilization V* *The Battle for Wesnoth* *The Legend of Zelda: Breath of the Wild* *Warcraft 3*

(RIC-4) Inter-System Entity References

System Requirement	Publication Sources	Game Sources
Interoperation with AI subsystems	Rollings and Ernest (2006) USPGameDev	All games with AI
Entity integration with other subsystems	Gregory (2019)	All games
Effects that affect physics state	Rollings and Ernest (2006) USPGameDev	All games EXCEPT: *Hearthstone* *Magic: the Gathering*

Network entity synchronization	Gregory (2019)	All multiplayer on-line games

(RRL-1) *Game Loop* Compliance

System Requirement	Publication Sources	Game Sources
Real-time, per-frame servicing	Gregory (2019)	All games
Asynchronous prompts	Wizards of the Coast (2021) Rollings and Ernest (2006) USPGameDev	All turn-based games PLUS *Loop Hero*
Performance	Gregory (2019) Nakagawa *et al.* (2012)	All games

(RRL-2) Simulation State Persistence

System Requirement	Publication Sources	Game Sources
Entity persistence	Gregory (2019) USPGameDev	All games EXCEPT *BYTEPATH*
World persistence	Same as above	Same as above

(RRL-3) Partial World Simulation

System Requirement	Publication Sources	Game Sources
Runtime world chunk streaming	Gregory (2019)	*Guild Wars* *Minecraft* *The Legend of Zelda: Breath of the Wild*

(RRL-4) Interaction Modes

System Requirement	Publication Sources	Game Sources
System start-up and shut-down	Gregory (2019)	All games
Interaction modes	USPGameDev	All games
High-level progression tracking	Gregory (2019)	All games

(RTC-1) Engine Compatibility

System Requirement	Publication Sources	Game Sources
Engine compliance	Nakagawa *et al.* (2012) USPGameDev	—

(RTC-2) Platform Compatibility

System Requirement	Publication Sources	Game Sources
Cross-platform support	Plummer (2004) Politowski *et al.* (2021)	All games
Platform compliance	Nakagawa *et al.* (2012)	—

(RTC-3) Data Format Compatibility

System Requirement	Publication Sources	Game Sources
Compliance w/ External data formats	Nakagawa *et al.* (2012)	—
Data formats w/ version control support	Nakagawa *et al.* (2012)	*Dungeon Crawl: Stone Soup* *Diablo* *Dwarf Fortress* *Factorio* *NetHack* *Veloren*

(RCP-1) Continuous Build

System Requirement	Publication Sources	Game Sources
Support for early builds	Politowski *et al.* (2021) Schell (2020) Rollings and Ernest (2006)	—
Reduced build costs	Politowski *et al.* (2021)	—
Software designs that support agile methodologies	Murphy-Hill *et al.* (2014) Kasurinen *et al.* (2017) Schell (2020) Rollings and Ernest (2006)	—

(RCP-2) Accessible Development Tools

System Requirement	Publication Sources	Game Sources
World editor support	Gregory (2019) Murphy-Hill *et al.* (2014)	*Factorio* *Minecraft* *The Battle for Wesnoth* *Warcraft 3*
Entity editor support	Gregory (2019) Murphy-Hill *et al.* (2014) USPGameDev	*Diablo* *Dwarf Fortress* *Hearthstone* *Warcraft 3*
Scripting support	Gregory (2019) Murphy-Hill *et al.* (2014) Rollings and Ernest (2006) USPGameDev	*Factorio* *Hearthstone* *Minecraft* *Sid Meyer's Civilization V* *The Battle for Wesnoth* *Warcraft 3*
Promotion of the creative process	Kasurinen *et al.* (2017) Murphy-Hill *et al.* (2014)	*Dungeon Crawl: Stone Soup* *Dota 2* *Factorio* *Hearthstone* *Minecraft* *Sid Meyer's Civilization V* *Warcraft 3*

(RCP-3) Runtime Tools

System Requirement	Publication Sources	Game Sources
Cheats	Gregory (2019) Murphy-Hill *et al.* (2014)	*Minecraft*
In-game inspector	Gregory (2019) Murphy-Hill *et al.* (2014)	*Factorio* *Terraria*
In-game console	Gregory (2019) Murphy-Hill *et al.* (2014)	*Dota 2* *Factorio* *Minecraft* *Terraria* *Warcraft 3*
Live data editing	Gregory (2019)	—

support / Hot-loading	Murphy-Hill *et al.* (2014)	

(RDD-1) Runtime Data Access

System Requirement	Publication Sources	Game Sources
Runtime data loading	Gregory (2019) Rollings and Ernest (2006) Politowski *et al.* (2021)	All games EXCEPT: *BYTEPATH* *Nomic*
Runtime data storage	Gregory (2019) Politowski *et al.* (2021)	Same as above
Runtime data retrieval	Gregory (2019) Politowski *et al.* (2021)	Same as above

(RDD-2) Data-Driven Simulation

System Requirement	Publication Sources	Game Sources
Entity prefabs	Gregory (2019) Murphy-Hill *et al.* (2014) Rollings and Ernest (2006)	All games EXCEPT: *BYTEPATH* *Nomic*
Content-asset integration	Gregory (2019) Murphy-Hill *et al.* (2014)	Same as above
Entity type schemas	Gregory (2019)	—
Error-handling for data loading	Pascarella *et al.* (2018)	All games EXCEPT: *BYTEPATH* *Nomic*

(RCE-1) Decoupled Subsystems

System Requirement	Publication Sources	Game Sources
Low inter-subsystem dependency	Plummer (2004) Pascarella *et al.* (2018) Politowski *et al.* (2021)	—
Subsystem encapsulation	Plummer (2004)	—

Code reusability across games	Pascarella *et al.* (2018)	*Final Fantasy Tactics Advance* *Pokémon* series

(RCE-2) Extensibility		
System Requirement	Publication Sources	Game Sources
Low cost for adding or replacing subsystems	Plummer (2004) Pascarella *et al.* (2018)	—
Low cost for extending subsystems	Plummer (2004)	—

(RCE-3) Flexibility		
System Requirement	Publication Sources	Game Sources
Support for late changes	Kasurinen *et al.* (2017)	—
Flexibility to handle vague requirements	Murphy-Hill *et al.* (2014)	—
Low cost for changing mechanics	Hunicke *et al.* (2004)	—

(RCE-4) Code Accessibility		
System Requirement	Publication Sources	Game Sources
Subsystem APIs with shallow learning curve	Plummer (2004) USPGameDev	—

(RCE-5) Reliable Error Detection		
System Requirement	Publication Sources	Game Sources
Preemptive error detection	Pascarella *et al.* (2018)	—
Automatic testing	Pascarella *et al.* (2018)	—
State snapshot	Gregory (2019)	—
Improved error-handling	Pascarella *et al.* (2018)	—

Table B.1: All architectural requirements, the specific system requirements that make them up, and the information sources (publications and games) we collected them from.

References

Adams and Dormans(2012) Ernest Adams and Joris Dormans. *Game Mechanics: Advanced Game Design*. New Riders. Cited on pages 2, 27, 28, 48, 57, 58, 59, 60, 61, 62, 63, 65, 97, 153, 154, 155, 156, 158, 159, 160

Adams(2015) William C. Adams. *Handbook of Practical Program Evaluation*, chapter Conducting Semi-Structured Interviews, pages 492–595. John Wiley & Sons, 2015. Cited on pages 52

Aleem *et al.*(2017) Saiqa Aleem, Luiz Fernando Capretz and Faheem Ahmed. Game development software engineering process life cycle: A systematic review. *Journal of Software Engineering Research and Development*. doi: 10.1186/s40411-016-0032-7. URL http://dx.doi.org/10.1186/s40411-016-0032-7. Cited on pages 17

Aluani and Mizutani(2013) Fernando O. Aluani and Wilson K. Mizutani. Projeto ouroboros: Sistema de integração automatizada entre c++ e linguagens de script. Completion of Course Work at Institute of Mathematics and Statistics from University of São Paulo, December 2013. Cited on pages 105

Andrew Plotkin(2009) Andrew Plotkin. Rule-Based Programming in Interactive Fiction. Online article (last accessed Feb 21, 2019), May 2009. URL https://eblong.com/zarf/essays/rule-based-if/. Cited on pages 36, 38, 49, 50, 108, 114, 141, 142

Anonymous authors(2020) Anonymous authors. What makes software flexible. Online wiki article (last accessed April 1st, 2021), 2020. URL https://wiki.c2.com/?WhatMakesSoftwareFlexible. Cited on pages 49

Bass *et al.*(2003) L. Bass, P. Clements and R Kazman. *Software Architecture in Practice*. Addison-Wesley. Cited on pages 5, 19, 31, 36, 46

Bilas(2002) Scott Bilas. A data-driven game object system. Online presentation (last accessed Jun 18, 2019), 2002. URL https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf. Cited on pages 49, 50

BinSubaih *et al.*(2007) Ahmed BinSubaih, Steve Maddock and Daniela Romano. A Survey of 'Game' Portability. Technical report, Department of Computer Science, University of Sheffield. Cited on pages 50

Bogost(2006) Ian Bogost. *Unit Operations. An Approach to Videogame Criticism*. MIT Press. Cited on pages 10

Bucklew(2015) Brian Bucklew. Data-driven engines of qud and sproggiwood. Video from conference talk (last accessed April 1st, 2021), 2015. URL https://www.youtube.com/watch?v=U03XXzcThGU. Cited on pages 49, 50, 108, 142, 143

Buschman *et al.***(1996)** Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. *Pattern-Oriented Software Architecture*, volume 1. John Wiley & Sons, Chichester, UK. Cited on pages 5, 19, 33, 34, 49

Callele *et al.***(2005)** D. Callele, E. Neufeld and K. Schneider. Requirements engineering and the creative process in the video game industry. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 240–250. doi: 10.1109/RE.2005.58. Cited on pages 50

Campbell and Stanley(1963) D. T. Campbell and J. C. Stanley. *Experimental and Quasi-Experimental Designs for Research*, pages 1–71. 1963. doi: 10.1093/obo/9780195389678-0053. Cited on pages 45, 118, 120

De Freitas *et al.***(2012)** Leonardo G. De Freitas, Luiggi Monteiro Reffatti, Igor Rafael De Sousa, Anderson C. Cardoso, Carla Denise Castanho, Rodrigo Bonifácio and Guilherme N. Ramos. Gear2D: An extensible component-based game engine. In *Foundations of Digital Games 2012, FDG 2012 - Conference Program*, pages 81–88. doi: 10.1145/2282338.2282357. Cited on pages 47

Dormans(2012a) Joris Dormans. *Engineering Emergence - Applied Theory for Game Design*. Ph.D thesis, University of Amsterdam. Cited on pages 46, 58, 59, 61, 62, 63, 65, 97, 154, 155, 156, 158, 159, 160

Dormans(2012b) Joris Dormans. The Effectiveness and Efficiency of Model Driven Game Design. In *Entertainment Computing - ICEC 2012*, pages 542–548. Springer Berlin Heidelberg. Cited on pages 46

Dubbelman(2016) Teun Dubbelman. Narrative Game Mechanics. In *International Conference on Interactive Digital Storytelling*, pages 39–50. doi: 10.1007/978-3-319-48279-8_4. Cited on pages 27

Ernst *et al.***(1998)** Michael Ernst, Craig Kaplan and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conference on Object- Oriented Programming*, pages 186–211. doi: 10.1007/BFb0054092. Cited on pages 36, 49, 109, 114

Figg(2016) Thomas Figg. Write code that is easy to delete, not easy to extend. Online article (last accessed March 31st, 2021), 2016. URL https://programmingisterrible.com/post/139222674273/how-to-write-disposable-code-in-large-systems. Cited on pages 49

Folmer(2007) Eelke Folmer. Component based game development–a solution to escalating costs and expanding deadlines? In *International Symposium on Component-Based Software Engineering*, pages 66–73. Springer. Cited on pages 32

Fowler(2019) Martin Fowler. Techincaldebt. Online article (last accessed Jan 5th, 2021), 2019. URL https://martinfowler.com/bliki/TechnicalDebt.html. Cited on pages 13, 49, 141

Fujibayashi *et al.***(2017)** Hidemaro Fujibayashi, Satoru Takizawa and Takuhiro Dohta. Breaking Conventions with The Legend of Zelda: Breath of the Wild. Conference talk (last accessed Match 31st, 2021), 2017. URL https://www.youtube.com/watch?v=QyMsF31NdNc. Cited on pages 49, 50, 107, 108, 142

Furtado(2012) André Wilson Brotto Furtado. *Domain-Specific Game Development*. Ph.D thesis, Universidade Federal de Pernambuco. Cited on pages 37, 49, 50

Gamma *et al.***(1995)** Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education India. Cited on pages 5, 19, 21, 22, 33, 35, 36, 49, 65, 101, 102, 105, 108, 116, 126, 129, 135, 144

George *et al.*(2013) Sébastien George, Élise Lavoué and Baptiste Monterrat. An environment to support collaborative learning by modding. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8095 LNCS, pages 111–124. doi: 10.1007/978-3-642-40814-4_10. Cited on pages 47

Gestwicki(2012) P Gestwicki. The entity system architecture and its application in an undergraduate game development studio. In *Proceeding of International Conference on the Foundations of Digital Games*, pages 73–80. ACM. Cited on pages 47

Gregory(2019) Jason Gregory. *Game engine architecture, third edition.* CRC Press. Cited on pages 10, 11, 12, 14, 15, 16, 17, 19, 21, 22, 23, 32, 33, 34, 38, 46, 57, 58, 59, 65, 66, 67, 68, 71, 72, 73, 74, 77, 96, 97, 98, 99, 101, 105, 142, 153, 154, 155, 160, 161, 162, 164, 165, 166

Grey(2017) Darren Grey. *Procedural Generation in Game Design*, chapter When and Why to Use Procedural Generation, pages 3–12. CRC Press, 2017. Cited on pages 14

Hunicke *et al.*(2004) Robin Hunicke, Marc Leblanc and Robert Zubek. MDA: A formal approach to game design and game research. In *In Proceedings of the Challenges in Games AI Workshop, Nineteenth National Conference of Artificial Intelligence*, pages 1–5. Press. Cited on pages 9, 27, 28, 48, 74, 166

ISO(2017) ISO. International Standard ISO/IEC/IEEE 24765. *Systems and software engineering — Vocabulary, 2nd Ed.*, 2017. URL https://www.iso.org/standard/71952.html. Cited on pages 13

Järvinen(2008) Aki Järvinen. *Games without Frontiers: Theories and Methods for Game Studies and Design.* Ph.D thesis, University of Tampere, Finland. Cited on pages 27

Johnson(1997) Ralph E. Johnson. Components, Frameworks, Patterns. In *Proceedings of the 1997 Symposium on Software Reusability*, SSR '97, pages 10–17, New York, NY, USA. ACM. doi: 10.1145/258366.258378. Cited on pages 17

Kahney(2006) Leander Kahney. Vaporware: Better late than never. Archived online article (last accessed Jan 5th, 2021), 2006. URL https://web.archive.org/web/20090110154932/http://www.wired.com/science/discoveries/news/2006/02/70143?currentPage=2. Cited on pages 13

Karen Collins(2008) Karen Karen Collins. *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design.* The MIT Press. Cited on pages 12

Kasurinen *et al.*(2017) Jussi Kasurinen, Maria Palacin-Silva and Erno Vanhala. What Concerns Game Developers? A Study on Game Development Processes, Sustainability and Metrics. pages 15–21. doi: 10.1109/WETSoM.2017.3. Cited on pages 48, 70, 71, 74, 163, 164, 166

Kitchenham and Charters(2007) Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature reviews in Software Engineering Version 2.3. Technical report, Software Engineering Group from the School of Computer Science and Mathematics of Keele University and Department of Computer Science of the University of Durham. Cited on pages 47

Krasner *et al.*(1988) Glenn E Krasner, Stephen T Pope *et al.* A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49. Cited on pages 19, 33

Landsteiner(2015) Norbert Landsteiner. Inside spacewar! a software archeological approach to the first video game. Online series (last accessed Jan 5th, 2021), 2015. URL https://www.masswerk.at/spacewar/inside/. Cited on pages 13

Larsen and Schoenau-Fog(2016) Bjarke A. Larsen and Henrik Schoenau-Fog. The Narrative Quality of Game Mechanics. In *International Conference on Interactive Digital Storytelling*, pages 61–72. doi: 10.1007/978-3-319-48279-8. Cited on pages 27

Lee *et al.***(2020)** Daniel Lee, Dayi Lin, Cor-Paul Bezemer and Ahmed E. Hassan. Building the perfect game – an empirical study of game modifications. In *International Conference on Interactive Digital Storytelling*, pages 2485–2518. Cited on pages 18

Leite *et al.***(2020)** Leonardo Leite, Fabio Kon and Paulo Meirelles. Interview protocol for discovering organizational structures. Online article (last accessed Apr 19, 2021), June 2020. URL http://ccsl.ime.usp.br/devops/2020-06-14/interview-protocol.html. Cited on pages 52

Leonard(1999) Tom Leonard. Postmortem: Thief: The dark project. Online article (last accessed Jun 18, 2019), 1999. URL http://www.gamasutra.com/view/feature/3355/postmortem_thief_the_dark_project.php. Cited on pages 49, 50

Llansó *et al.***(2011)** David Llansó, Marco A. Gómez-Martín, Pedro P. Gómez-Martín and Pedro A. González-Calero. Explicit domain modelling in video games. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, FDG '11, page 99–106, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/2159365.2159379. Cited on pages 37, 47, 100

Maggiore *et al.***(2012)** Giuseppe Maggiore, Pieter Spronck, Renzo Orsini, Michele Bugliesi, Enrico Steffinlongo and Mohamed Abbadi. Writing real-time. Net games in Casanova. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7522 LNCS, pages 341–348. doi: 10.1007/978-3-642-33542-6_30. Cited on pages 47

McLaughlin(2013) Martyn McLaughlin. New gta v release tipped to rake in £1bn in sales. Online article (last accessed Jan 5th, 2021), 2013. URL https://www.scotsman.com/whats-on/arts-and-entertainment/new-gta-v-release-tipped-rake-ps1bn-sales-2463312. Cited on pages 13

Mizutani(2017) Wilson K. Mizutani. *VORPAL: A Middleware for Real-Time Soundtrack in Digital Games.* Master's dissertation, Institute of Mathematics and Statistics from University of São Paulo. Cited on pages 12, 16

Mizutani and Kon(2019) Wilson K. Mizutani and Fabio Kon. Toward a reference architecture for economy mechanics in digital games. In *Proceedings of the Brazilian Symposium on Games and Digital Entertainment (SBGames)*, pages 623–626. Cited on pages 45, 76, 140

Mizutani and Kon(2020) Wilson K. Mizutani and Fabio Kon. Unlimited rulebook: a reference architecture for economy mechanics in digital games. In *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, pages 58–68. Cited on pages 45, 76, 117, 121, 123, 140

Mizutani *et al.***(2021)** Wilson K. Mizutani, Vinícius K. Daros and Fabio Kon. Software architecture for digital game mechanics: A systematic literature review. *Entertainment Computing.* doi: https://doi.org/10.1016/j.entcom.2021.100421. Cited on pages 5, 46, 47, 48, 99, 139, 140

Moll(2021) Thomas Moll. Julia used Multiple Dispatch! It's Super Effective! Online article (last accessed Jul 30, 2021), July 2021. URL https://www.moll.dev/projects/effective-multi-dispatch. Cited on pages 110

Mössenböck(2000) Hanspeter Mössenböck. Twin — A design pattern for modeling multiple inheritance. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1755, pages 358–369. doi: 10.1007/3-540-46562-6_31. Cited on pages 47

Mottola *et al.***(2006)** Luca Mottola, Amy L. Murphy and Gian Pietro Picco. Pervasive games in a mote-enabled virtual world using tuple space middleware. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames '06*, pages 1–8. doi: 10.1145/1230040.1230098. Cited on pages 47

Murphy-Hill *et al.***(2014)** Emerson Murphy-Hill, Thomas Zimmermann and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings - International Conference on Software Engineering*, pages 1–11. doi: 10.1145/2568225.2568226. Cited on pages 10, 48, 70, 71, 72, 73, 74, 163, 164, 165, 166

Nakagawa *et al.***(2014)** Elisa Y. Nakagawa, Milena Guessi, Jose C. Maldonado, Daniel Feitosa and Flavio Oquendo. Consolidating a process for the design, representation, and evaluation of reference architectures. In *Proceedings - Working IEEE/IFIP Conference on Software Architecture 2014, WICSA 2014*, pages 143–152. doi: 10.1109/WICSA.2014.25. Cited on pages 6, 42, 44, 46, 66, 68, 69, 75, 113

Nakagawa *et al.***(2011)** Elisa Yumi Nakagawa, Pablo Oliveira and Antonino Martin Becker. Reference Architecture and Product Line Architecture: A Comparison. *European Conference on Software Architecture (ECSA)*, pages 2–5. ISSN 0103-2569. Cited on pages 5, 32, 37

Nakagawa *et al.***(2012)** Elisa Yumi Nakagawa, Flavio Oquendo and Martin Becker. RAModel: A reference model for reference architectures. In *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012*, pages 297–301. doi: 10.1109/WICSA-ECSA.212.49. Cited on pages 42, 43, 162, 163

NetHack Wiki(2019) NetHack Wiki. Cockatrice. Online wiki article (last accessed Feb 23rd, 2021), 2019. URL https://nethackwiki.com/wiki/Cockatrice. Cited on pages 1

Nordmark(2012) Njål Nordmark. *Software Architecture and the Creative Process in Game Development*. Master's dissertation, Norwegian University of Science and Technology. Cited on pages 13

Nystrom(2014) Robert Nystrom. *Game Programming Patterns*. Genever Benning. Cited on pages 1, 19, 20, 34, 36, 49, 50, 72, 96, 105

Nystrom(2018) Robert Nystrom. Is There More to Game Architecture than ECS? Video from conference talk (last accessed April 1st, 2021), 2018. URL https://www.youtube.com/watch?v=JxI3Eu5DPwE. Cited on pages 50, 105

Olsson *et al.***(2015)** Tobias Olsson, Daniel Toll, Anna Wingkvist and Morgan Ericsson. Evolution and Evaluation of the Model-View-Controller Architecture in Games. In *International Workshop on Games and Software Engineering*. doi: 10.1109/GAS.2015.10. Cited on pages 33, 34, 49, 50

OMG(2015) OMG. Unified modeling language 2.5. Online reference (last accessed Jan 7th, 2021), 2015. URL https://www.omg.org/spec/UML/2.5. Cited on pages 75

Osborn *et al.***(2017)** Joseph C. Osborn, Noah Wardrip-Fruin and Michael Mateas. Refining operational logics. In *ACM International Conference Proceeding Series*. Cited on pages 27

Papaioannou(2005) Georgios Papaioannou. Interactive dynamics for large virtual reality applications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3746 LNCS, pages 307–316. doi: 10.1007/11573036_29. Cited on pages 47

Pascarella *et al.***(2018)** Luca Pascarella, Fabio Palomba, Massimiliano Di Penta and Alberto Bacchelli. How is video game development different from software development in open source? In *Proceedings - International Conference on Software Engineering*, pages 392–402. doi: 10.1145/3196398.3196418. Cited on pages 48, 73, 74, 165, 166

Patel *et al.***(2004)** Shwetak N. Patel, John A. Bunch, Kyle D. Forkner, Logan W. Johnson, Tiffany M. Johnson, Michael N. Rosack and Gregory D. Abowd. The design and implementation of multi-player card games on multi-user interactive tabletop surfaces. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3166, pages 339–344. doi: 10.1007/978-3-540-28643-1_42. Cited on pages 47

Pinhanez(2000) Claudio S Pinhanez. The scd architecture and its use in the design of story-driven interactive spaces. In *Managing Interactions in Smart Environments*, pages 239–250. Springer. Cited on pages 32, 47

Plummer(2004) Jeff Plummer. *A Flexible And Expandable Architecture for Computer Games.* Master's dissertation, Arizona State University. Cited on pages 23, 24, 32, 34, 46, 63, 69, 73, 74, 114, 142, 159, 163, 165, 166

Politowski *et al.***(2020)** Cristiano Politowski, Fabio Petrillo, Gabriel Cavalheiro Ullmann, Josias de Andrade Werly and Yann-Gaël Guéhéneuc. Dataset of video game development problems. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 553–557, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3379597.3387486. Cited on pages 13

Politowski *et al.***(2021)** Cristiano Politowski, Fabio Petrillo, Gabriel C. Ullmann and Yann Gaël Guéhéneuc. Game industry problems: An extensive analysis of the gray literature. *Information and Software Technology*, 134(February). doi: 10.1016/j.infsof.2021.106538. Cited on pages 48, 70, 72, 73, 163, 165

Rabin(2000) Steve Rabin. *Game Programming Gems*, chapter 1.0 "The magic of data-driven design", pages 3–7. Charles River Media, 2000. Cited on pages 3, 14, 17, 49, 50

Rollings and Ernest(2006) A. Rollings and A. Ernest. *Fundementals of game design.* Cited on pages 48, 57, 58, 59, 60, 61, 62, 65, 66, 70, 71, 72, 73, 97, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165

Sanneblad and Holmquist(2003) Johan Sanneblad and Lars Erik Holmquist. OpenTrek: A platform for developing interactive networked games on mobile devices. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2795, pages 224–240. doi: 10.1007/978-3-540-45233-1_17. Cited on pages 47

Santos *et al.***(2013)** José Filipe Marreiros Santos, Milena Guessi, Matthias Galster, Daniel Feitosa and Elisa Yumi Nakagawa. A checklist for evaluation of reference architectures of embedded systems (s). In *Proceeding of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, volume 13, pages 1–4. Cited on pages 45, 145

Sarinho *et al.*(2018) Victor T. Sarinho, Gabriel S. De Azevedo and Filipe M.B. Boaventura. AsKME: A Feature-Based Approach to Develop Multiplatform Quiz Games. In *2019 Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, pages 38–47. IEEE. doi: 10.1109/SBGAMES.2018.00014. Cited on pages 32, 38, 47

Scacchi(2011) Walt Scacchi. Modding as an Open Source Approach to Extending Computer Game Systems. *International Journal of Open Source Software & Processes*, 3(3):36–47. doi: 10.4018/jossp.2011070103. Cited on pages 49, 50

Scacchi(2017) Walt Scacchi. Practices and Technologies in Computer Game Software Engineering. *IEEE Software*, 34(1):110–116. Cited on pages 18

Scacchi and Cooper(2015) Walt Scacchi and Kendra M. L. Cooper. *Computer games and software engineering*. CRC Press. Cited on pages 19

Schell(2020) Jesse Schell. *The Art of Game Design, Third Edition*. CRC Press. Cited on pages 2, 9, 13, 27, 28, 48, 57, 58, 59, 60, 61, 62, 63, 65, 70, 139, 153, 154, 155, 156, 158, 159, 160, 161, 163

Schreier(2020) Jason Schreier. In world of video game development, chronic overtime is endemic. Online article (last accessed Jan 5th, 2021), 2020. URL https://www.japantimes.co.jp/news/2020/10/02/business/video-game-development-crunch-overtime/. Cited on pages 13

Shaw and Garlan(1996) Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. Cited on pages 19, 36, 49

Sicart(2008) Miguel Sicart. Defining Game Mechanics. *International Journal of Computer Game Research*, 8(2). URL http://gamestudies.org/0802/articles/sicart. Cited on pages 27

Team(2020) Factorio Team. Friday facts 360 - 1.0 is here! Online article (last accessed Feb 8th, 2021), aug 2020. URL https://factorio.com/blog/post/fff-360. Cited on pages 18

Tutzschke and Zukunft(2009) Jan Peter Tutzschke and Olaf Zukunft. FRAP: A framework for pervasive games. In *EICS'09 - Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 133–141. doi: 10.1145/1570433.1570459. Cited on pages 47

Valentin *et al.*(2012) Julien Valentin, Florent Coudret, Eric Gouardères and Wilfrid Lefer. Human behaviour modelling for simulating evacuation of buildings on fire. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7145 LNCS, pages 12–23. doi: 10.1007/978-3-642-29050-3_2. Cited on pages 47

Viana and Nakamura(2014) Bruno Santos Viana and Ricardo Nakamura. Immersive interactive narratives in augmented reality games. In *Proceedings of the Third International Conference on Design, User Experience, and Usability. User Experience Design for Diverse Interaction Platforms and Environments - Volume 8518*, pages 773–781, New York, NY, USA. Springer-Verlag New York, Inc. doi: 10.1007/978-3-319-07626-3_73. URL http://dx.doi.org/10.1007/978-3-319-07626-3_73. Cited on pages 11

Wang and Nordmark(2015) Alf Inge Wang and Njål Nordmark. Software Architectures and the Creative Processes in Game Development. In *International Conference on Entertainment Computing*. doi: 10.1007/978-3-319-24589-8. Cited on pages 14, 16, 47, 86

Wen *et al.***(2020)** Melissa Wen, Leonardo Leite, Fabio Kon and Paulo Meirelles. Understanding floss through community publications: Strategies for grey literature review. In *Proceedings of the 2020 IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE:NIER)*, pages 89–92. doi: 10.1145/3377816.3381729. Cited on pages 46

West(2018) Catherine West. Using Rust For Game Development, 2018. URL https://kyren.github.io/2018/09/14/rustconf-talk.html. Cited on pages 23, 32, 34, 48, 49, 50, 96, 97, 114

West(2007) Mick West. Evolve your hierarchy. Online article (last accessed Jun 18, 2019), January 2007. URL http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy. Cited on pages 49, 50

Wiebusch and Latoschik(2015) D Wiebusch and M Latoschik. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems, SEARIS 2015 (2017)*, pages 25–32. doi: 10.1109/SEARIS.2015.7854098. Cited on pages 49, 50

Williams *et al.***(2011)** James R Williams, Simon Poulding, Louis M Rose, Richard F Paige and Fiona A C Polack. Identifying Desirable Game Character Behaviours through the Application of Evolutionary Algorithms to Model-Driven Engineering Metamodels. In Myra B Cohen and Mel Ó Cinnéide, editors, *Search Based Software Engineering*, pages 112–126, Berlin, Heidelberg. Springer Berlin Heidelberg. Cited on pages 38

Wizards of the Coast(2021) Wizards of the Coast. Magic: the Gathering's Comprehensive Rules. Online reference (last accessed March 24th, 2021), 2021. URL https://magic.wizards.com/en/game-info/gameplay/rules-and-formats/rules. Cited on pages 46, 57, 58, 59, 61, 62, 65, 66, 75, 110, 142, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162

Wu *et al.***(2010)** Bian Wu, Alf Inge Wang, Anders Hartvoll Ruud and Wan Zhen Zhang. Extending Google Android's application as an educational tool. In *DIGITEL 2010 - The 3rd IEEE International Conference on Digital Game and Intelligent Toy Enhanced Learning*, pages 23–30. doi: 10.1109/DIGITEL.2010.38. Cited on pages 47

Yoder and Johnson(2002) Joseph W. Yoder and Ralph Johnson. The Adaptive Object-Model Architectural Style. In *Working Conference on Software Architecture*, pages 3–27. doi: 10.1007/978-0-387-35607-5_1. Cited on pages 35, 49, 99, 100, 105, 148

Yoshikawa(2018) Livia Maki Yoshikawa. A retroatividade nostálgica do chiptune. Completion of Course Work at Arts and Communications School from University of São Paulo, December 2018. Cited on pages 12

Zamith *et al.***(2016)** Marcelo Zamith, Luis Valente, Bruno Feijo and Esteban Clua. Game loop model properties and characteristics on multi-core CPU and GPU games. In *Proceedings of SBGames 2016*, pages 100–109. Cited on pages 20, 49, 50

Zhu(2014) Meng Zhu. *Model-Driven Game Development Addressing Architectural Diversity and Game Engine-Integration*. Ph.D thesis, Department of Computer and Information Science Faculty, Norwegian University of Science and Technology. Cited on pages 37, 49, 50

Ludography

a327ex(2018) a327ex. BYTEPATH, 2018. Cited on pages 51, 65, 72, 73

Abrakam Entertainment S.A.(2016–2020) Abrakam Entertainment S.A. Faeria, 2016–2020. Cited on pages 18, 30

ArenaNet(2005) ArenaNet. Guild Wars, 2005. Cited on pages 51, 62, 63, 68

Bay 12 Games(2006) Bay 12 Games. Dwarf Fortress, 2006. Cited on pages 51, 63, 69, 71

Blizzard Entertainment(1997) Blizzard Entertainment. Diablo, 1997. Cited on pages 51, 57, 62, 65, 69, 71

Blizzard Entertainment(2014) Blizzard Entertainment. Hearthstone, 2014. Cited on pages 30, 31, 51, 62, 63, 71

Blizzard Entertainment(2002) Blizzard Entertainment. Warcraft 3, 2002. Cited on pages 51, 62, 63, 70, 71, 72, 91, 142

Cygames(2016) Cygames. Shadowverse, 2016. Cited on pages 30

DCSS Devteam(2006–2021) DCSS Devteam. Dungeon Crawl Stone Soup, 2006–2021. Cited on pages 51, 62, 65, 69, 71, 104, 124

DevTeam(1987) DevTeam. Nethack, 1987. Cited on pages 1, 51, 62, 63, 70, 102, 107, 124

EA Redwood Shores(2008) EA Redwood Shores. Dead Space, 2008. Cited on pages 24

Firaxis Games(2010) Firaxis Games. Sid Meyer's Civilization V, 2010. Cited on pages 30, 51, 71, 142

Four Quarters(2021) Four Quarters. Loop Hero, 2021. Cited on pages 51, 63

Freehold Games(2015) Freehold Games. Caves of Qud, 2015. Cited on pages 51, 62, 65, 108

Game Freak(1996–2021) Game Freak. Pokémon, 1996–2021. Cited on pages 10, 22, 51, 62, 63, 124, 142

Game Freak(2019) Game Freak. Pokémon Sword & Shield, 2019. Cited on pages 11

Gravity Interactive(2002) Gravity Interactive. Ragnarok Online, 2002. Cited on pages 51, 62, 64

Grid Sage Games(2017) Grid Sage Games. Cogmind, 2017. URL https://www.gridsagegames.com/cogmind/. Cited on pages 51

Grinding Gear Games(2013–2021) Grinding Gear Games. Path of Exile, 2013–2021. Cited on pages 18, 31, 51, 62, 63, 124, 142

Mojang Studios(2011) Mojang Studios. Minecraft, 2011. Cited on pages 26, 51, 71, 72

Nintendo(2017) Nintendo. The Legend of Zelda: Breath of the Wild, 2017. Cited on pages 27, 49, 51, 63, 107

Nintendo(1985–2021) Nintendo. Super Mario series, 1985–2021. Cited on pages 58, 143

Nintendo(1986-2021) Nintendo. The Legend of Zelda series, 1986-2021. Cited on pages 124

Peter Suber(1982) Peter Suber. Nomic: A Game of Self-Amendment, 1982. URL http://legacy.earlham.edu/~peters/nomic.htm. Cited on pages 51, 72, 73, 141, 142

PlatinumGames(2017) PlatinumGames. Nier: Automata, 2017. Cited on pages 24

Re-Logic(2011) Re-Logic. Terraria, 2011. Cited on pages 51, 62, 63, 72

Square Enix(1986–2020) Square Enix. Dragon Quest series, 1986–2020. Cited on pages 38

Square Enix(2003) Square Enix. Final Fantasy Tactics Advance, 2003. Cited on pages 51, 62, 66, 67

Square Enix(1987–2020) Square Enix. Final Fantasy series, 1987–2020. Cited on pages 22, 38

The Battle for Wesnoth Project(2003) The Battle for Wesnoth Project. The Battle for Wesnoth, 2003. Cited on pages 51, 59, 62, 65, 71

Toy *et al.*(1980) Michael Toy, Glenn Wichman and Ken Arnold. Rogue, 1980. Cited on pages 1

USPGameDev(2020) USPGameDev. Backdoor Route, 2020. URL https://uspgamedev.itch.io/backdoor-route. Cited on pages 53, 106, 145

USPGameDev(2021) USPGameDev. Grimoire: Ars Bellica, 2021. URL https://kazuo256.itch.io/grimoire-ars-bellica. Cited on pages 53

USPGameDev(2010) USPGameDev. Horus Eye, 2010. URL https://uspgamedev.itch.io/horus-eye. Cited on pages 53

USPGameDev(2018) USPGameDev. It's All About Lasagna!, 2018. URL https://uspgamedev.itch.io/its-all-about-lasagna. Cited on pages 53

USPGameDev(2017) USPGameDev. L.A.V.A. series L.A.M.P. edition, 2017. URL https://uspgamedev.itch.io/lava-series-lamp-edition. Cited on pages 53

Valve Corporation(2013) Valve Corporation. Dota 2, 2013. Cited on pages 51, 62, 63, 71, 72

Veloren team and contributors(2018) Veloren team and contributors. Veloren, 2018. Cited on pages 51, 62, 65

Wizards of the Coast(1993) Wizards of the Coast. Magic: the Gathering, 1993. Cited on pages 30, 51, 60, 61, 62, 63, 85, 103, 106, 114, 116, 124

Wube Software(2016–2021) Wube Software. Factorio, 2016–2021. Cited on pages 18, 29, 51, 69, 71, 72