Accepted for publication in IEEE Transactions on Games. DOI: 10.1109/TG.2024.3359439

Rulebook: An Architectural Pattern for Self-Amending Mechanics in Digital Games

Wilson Kazuo Mizutani, Fabio Kon

***Abstract*—Mechanics are one of the pillars of gameplay, enabled by the underlying implementation of the game and subject to constant changes during development. In particular, self-amending mechanics adjust themselves dynamically and are a common source of coupled code. The *Rulebook* is an architectural pattern that generalizes how developers prevent coupled code in self-amending mechanics, based on a careful research process including a systematic literature review, semi-structured interviews with professional developers, and quasi-experiments. The pattern codifies changes to the game state as "effect" objects, which it matches against a dynamic pool of rules. Each rule may amend, resolve, or chain effects. By preventing the control flow of the game from becoming coupled to the specific interactions of mechanics while also promoting an extensible and flexible structure for self-amendment, our solution reduces the time developers need to iterate on the design of mechanics. This paper details the *Rulebook* pattern and presents a case study demonstrating its design process in three different implementations of open-source jam games. Together with the typification of self-amending mechanics, this article formalizes a novel, state-of-the-art toolset for architecting games.**

***Index Terms*—software architecture, object-oriented design patterns, architectural patterns, digital games, self-amending mechanics.**

I. Introduction

NOMIC is a pen-and-paper game where "changing the rules is a move" as described by Suber in his work on **self-amendment** [1]. Based on this concept, we proposed the term **self-amending mechanics** in our previous research to typify mechanics that, when used or enabled, change how other mechanics work [2]. That definition views games as *interactive simulations* [3] — a medium where users intervene and interpret the state of a virtual world — and game mechanics as the set of all intentionally valid state changes inside that simulation [4]–[6], to provide a direct association between mechanics and game subsystems. As such, implementation-wise, self-amending mechanics are simulation operations that reshape the computation of subsequent operations.

A. Motivation

By enabling diverse and thought-provoking gameplay experiences, self-amending mechanics provide surprising dynamics through unexpected interactions, offer strategic actions to choose from, incite problem-solving curiosity, and even improve immersion. They are found in most games, from the invincibility star in *Super Mario Bros.* (*Nintendo*, 1985) changing collisions with otherwise hazardous objects, to the "is" block of *Baba is You* (*Hempuli Oy*, 2019), which controls what rules apply to any type of game object.

W. K. Mizutani and F. Kon are with the Department of Computer Science of the Institute of Mathematics and Statistics, University of São Paulo, São Paulo, Brazil. E-mail: {kazuo,kon}@ime.usp.br

As part of the creative process, self-amending mechanics can exert great influence on the software architecture of games [7], [8]. On the one hand, they are prone to specification changes like all mechanics in the iterative cycle of game design [4]. On the other hand, their intervention in other mechanics encourages tightly coupled code, which is expensive to maintain because changes in it propagate to the parts coupled to it, multiplying the costs of new features and bug fixes. The dozens of lines of code that check for petrification mechanics in *NetHack* (*DevTeam*, 1987) exemplify this:[1] a new interaction (e.g., a weapon that grows stronger when petrified) has to consider changing each of these lines on a case-by-case basis — a cost that might stifle the creative process. How an architecture organizes self-amending mechanics determines how coupled the system is to that interwoven interaction. The goal of our research is to design an architectural solution that unifies the implementation of self-amending mechanics while accounting for their transient specifications and tendency toward coupling. That way, developers spend less time maintaining coupled code and more time iterating on the game design to produce better games.

B. Proposal

Based on the state of the art and state of the practice of software architecture in game mechanics [2], [9]–[11], we gathered recurrent design solutions that reduce the coupling of self-amending mechanics. In this article, we propose and document an **architectural pattern** that generalizes these solutions, supporting any game *and* its specification changes over time. We named it the ***Rulebook* pattern** due to its central role in the reference architecture of our prior research, the *Unlimited Rulebook* [2]. Architectural patterns are the "fundamental structural organization schemas" where "every development activity that follows is governed by this structure" [12].

C. Methodology

The *Rulebook* pattern derives from the same systematic process, ProSA-RA [13], that led to the *Unlimited Rulebook* reference architecture. This process included a systematic literature review [11], semi-structured interviews with professional game developers, a survey of both academic and gray literature, two proofs-of-concept, and a quasi-experiment [2], [9], [10]. This extensive analysis yielded a *reference model* with 33 architectural requirements that served as the formal foundation for both the *Unlimited Rulebook* and the *Rulebook* pattern, **which we will cite where appropriate using their**

[1] https://github.com/NetHack/NetHack

code identifiers.[2] A reference implementation and proof-of-concept validation of both results is available online under the GPL v3 license.[3] It serves as an example throughout this paper, which complements our previous research with a case study evaluating the *Rulebook* in more practical contexts.

D. Text Organization

Section II places our proposal in the context of other approaches to the same or similar problems. Section III describes the *Rulebook* pattern itself. Section IV presents the case study where we portray the proposed pattern in actual games. Last, we discuss our conclusions in Section V.

II. Related Work

Self-amending mechanics are a subset of economy mechanics [6], [14] — the subset that offers the least opportunities for general-purpose software reuse [2]. The *comprehensive rules* of *Magic: the Gathering* (*Wizards of the Coast*, 1993–2023) are an iconic reference to self-amending mechanics [15]. A game design concept related to self-amending mechanics is *multiplicative gameplay* [16].

The study of design and architectural patterns is an old theme for game developers [17]. Of particular note are the architectural patterns *Entity-Component-System* (*ECS*) [3], [18] and *Layers* [12], [3]. *ECS* organizes the game state into "entities" that combine "components" from the different domains of the system (graphics, physics, etc.), while all computation is organized into stateless functions called "systems". It is known for promoting reuse and extensibility in game systems in general, mechanics included. The *Layers* pattern divides the game system according to the dependencies between parts so that the "lower" the layer where a part is, the more parts depend on it. It helps keep the vast complexity of game engines under control. However, neither of these patterns offers any directed guidance regarding self-amending mechanics. Academic publications on architectural solutions to self-amending mechanics, or even economy mechanics in general, are scarce [11], [19], [20].

Some works take a similar approach to us in the sense that they investigate the architectural impact of applying a given architectural pattern to the development of games. Olsson *et al.* evaluate the use of the *Model-View-Controller* by measuring the cost of adding changes to the codebase [21]. Wiebusch and Latoschik, on the other hand, propose a semantic validation tool to compensate for how the lack of strict typing in the *ECS* pattern leads to subsystems being coupled to which component combinations are compatible [22].

The underlying design structure of the *Rulebook* pattern resembles two other programming techniques. The first are *rule-based systems* (or *expert systems*), where condition-action rules are used to infer subsequent states of a knowledge database [23], [24]. The main difference to our approach is that we do not require a dedicated knowledge database, and we add the notion of "effect" objects to the process, which enables self-amendment. The other technique is *predicate-dispatching* [25]. If the *Rulebook* pattern were a first-class feature in a programming language, it would support a subset of *predicate-dispatching*.

[2]See Chapter 4 of our thesis for the complete list and descriptions [2]

[3]https://gitlab.com/uspgamedev/grimoire-ars-bellica/grimoire-ars-bellica

III. The *Rulebook* Pattern

This section follows a mixture of the formats for presenting design patterns from Gamma *et al.* [26] and Buschmann *et al.* [12]. The *Rulebook* is an *architectural pattern* that (1) explicitly codifies state changes in the simulation as effect objects, (2) tracks the set of active mechanics in the simulation, and (3) matches those mechanics against effects to execute them on a case-by-case manner through dynamic dispatching. This decouples most of the game from the complexity and specification instability of self-amending mechanics, making their implementation a scalable, flexible, and incremental process. It also enables dynamically adding and removing custom mechanics to the simulation in the form of rule objects.

A. Example

Consider a game about managing a caravan as it travels across a grid-structured world, tile by tile. The players' ultimate goal is to reach a certain destination, but to do so they must ensure their crew survives, their transportation means do not fall apart, and that their supplies never run out. The game follows a turn-based execution, such that simulation time only advances when the user provides input for in-game actions. Assume there are several possible such actions, giving the player enough possibilities to strategize around. Similarly, assume there are multiple variations of caravan members and land features. The outcome of the user's actions depends on the member composition of the caravan and the features of the stretch of land they are currently in. A few examples for each of these game elements could be:

- User actions – travel onwards, gather supplies, hunt food, repair vehicles, make a camp, trade goods;
- Caravan members – navigators, hunters, engineers, bards, cooks, historians;
- Land features – forests, bridges, rivers, roads, mountains, settlements; and
- Self-amending mechanics:
  - Cooks double all food produced;
  - Engineers have a 50% chance of preventing caravan vehicles from breaking;
  - Fog randomly changes the destination of every action that involves movement.

As a last part of this example, imagine there is a land feature called "abandoned ruins" and that the developers of this hypothetical caravan game want to give it the following self-amending mechanics: the duration of movement-related actions inside that tile doubles, while supplies do not get soaked when a "rain" land feature is present and the caravan stays put during its action. We will use this scenario to discuss the architectural challenges of self-amending mechanics in general and how the *Rulebook* approaches that problem.

B. Problem

The self-amending mechanics introduced by the "abandoned ruins" in our example both involve multiple types of user actions (RBM-3, RIC-1 [2]). The increased movement duration

affects the "travel onwards" action but also affects other move-related actions like "gather supplies". The protection from the elements, in turn, affects non-movement actions, i.e., *all other types of actions*. **That means the new mechanics change *all actions in the game* – maybe dozens of them** [2].

The amount of work required depends on how user actions tie into simulation state changes. For instance, if every action is implemented in its own function, then the "abandoned ruins" mechanics likely require adding a clause in all action functions to account for either the doubled movement duration or the prevention of weather-based consequences. We could, at first, refactor the actions so they rely on two new, reusable functions – one for handling duration changes and one for checking if weather consequences apply. However, that would still require changing all action functions and would not solve the more general problem posed by self-amending mechanics (RCE-3 [2]). If the next set of mechanics introduced also involves several actions but does not fall under these two cases we added, then the architecture will require another solution, probably involving another expensive code restructuring.

This recurring phenomenon happens when the *control flow* of the simulation is coupled to the implementation of the mechanics. No matter how much we rearrange conditional branches and functions, if they rely on the behavior of the mechanics then any self-amending mechanics that amend multiple mechanics might involve multiple points in the control flow of the simulation. In other words, the key challenge of implementing self-amending mechanics is to be able to introduce new behavior in multiple execution paths while changing a minimum amount of code (RBM-4, RCE-2, RCE-3 [2]). In some cases, even non-simulation control flow might be coupled to specific mechanics, e.g., if the user interface needs to know the consequences of an action to inform the user *before* they choose that action (RIC-1 [2])

For most games, the scarcity of self-amending mechanics ensures these restructuring steps remain within a manageable scope. The genres we noted where the costs involved are non-trivial include role-playing, strategy, simulation, and management games [2], [9], [10]. Some games outside these genres fall under similar circumstances if they involve a continuous release cycle (RCP-1 [2]) with constant innovation in game mechanics, e.g., to support competitive gameplay.

C. Solution

To avoid coupling the control flow to the game mechanics, the *Rulebook* pattern represents the *intended outcome* of user actions as objects independent from the actual *state change* caused by all self-amending mechanics at play. We call the first part **simulation effects** (or just "effects") – pieces of data that describe what we *want* to happen – and the second part, **effect resolution** (or just "resolution") – the actual code that *executes the change* to the simulation state (RBM-2 [2]). What bridges these two elements to make sure that effects bring about the appropriate resolutions are **simulation rules** (or just "rules") – they help the game system adjudicate what is the correct resolution for a given effect. Rules associate a *condition* (or *predicate*) over the upcoming effect and the current simulation state with a resolution *or a modification to the effect itself*

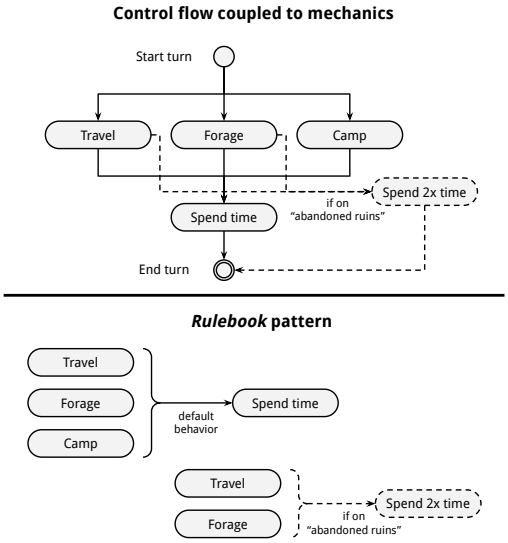

Fig. 1. A didactic representation of the difference between introducing self-amending mechanics in games where the simulation control flow is coupled to mechanics and games using the *Rulebook* pattern. Dashed lines and nodes represent code added to enable the new mechanics. In the upper part, because some but not all actions have a specific behavior variation, we need to branch into the new "Spend 2x time" mechanics from all involved parts of the codebase (the "Travel" and "Forage" steps). In the lower part, we only need to add one rule that applies to all the scenarios that matter, regardless of where and when they happen in the simulation.

(RBM-4 [2]). Together, these pieces allows us to introduce individual rules that operate on any number of effect types, grouping multiple changes into a single extension to the game. Figure 1 offers some insight into how that works.

D. Participants and collaborations

The *Rulebook* assumes the game stores the simulation state behind an abstraction layer (RIC-1, RIC-3, RIC-4 [2]). We will call it the `World` of the game (see Figure 2). The *Rulebook* pattern has two other key abstractions: `Effect` and `Rule`.

`Effect` objects describe a desired change to the simulation state, with usually no special behavior by itself. `Rule` objects are responsible for testing whether an `Effect` instance given the current state of the `World` matches its predicate, and for modifying that `Effect` or applying its resolution to the `World`. The most straightforward way the `Rule` class does this is by providing a single abstract method, `handleEffect(world, effect)`. Implementations of this method start by testing their predicate against the provided arguments, executing the assigned behavior if successful.[4]

A central `Rulebook` object is responsible for, given an `Effect` instance, finding all the `Rule` objects that participate in its resolution process, then applying them.[5] This process modifies the `Effect` instance partway, forming a pipeline of rules that *collectively shape simulation effects to achieve any particular, special-case resolution desired*. A single rule can negate, extend, or completely replace any effect (RBM-4 [2]).

To make the most of the *Rulebook* pattern, state changes should always be done by creating an `Effect` instance

[4]Example: https://tinyurl.com/grimoire-example-rule
[5]Example: https://tinyurl.com/grimoire-rulebook

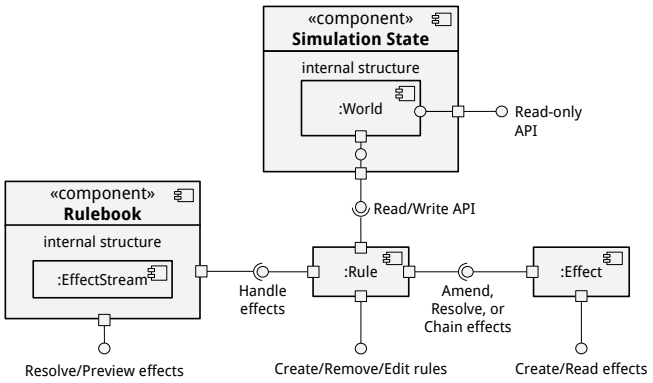

Fig. 2. Key participants in the *Rulebook* pattern. The unconnected interfaces define the available operations over the simulation: read the state, create new effects, resolve effects, preview effects to read them, and add or remove rules.

and passing it to the `Rulebook` instead of accessing the `World` directly. Since this subjects all simulation behavior to the effect resolution process, it is *always* possible to change *any* mechanics using `Rule` objects. This added level of indirection decouples the simulation control flow from the mechanics, minimizing the cost of changing them (RCE-1 [2]). The `Rulebook` encapsulates any complexity involved in finding, navigating, and invoking methods in the `Rule` objects currently active in the simulation.

Sometimes the resolution of an `Effect` may create other `Effect` instances. In these scenarios, it is common for the `Rulebook` to have an associated `EffectStream` to keep track of all pending `Effect` instances. Once the `Rulebook` starts processing an `Effect`, it stores and retrieves any further `Effect` instances in and from the `EffectStream` until no more `Effect` instances arise or after an established limit is reached. Figure 2 shows the complete process sustained by the *Rulebook* pattern and its participants.

E. Implementation considerations

As an architectural pattern, using the *Rulebook* to make a game is a key decision in the early phases of development [12]. Just like a team would not switch over to an *ECS* pattern without a cautious consideration, adopting the *Rulebook* pattern is not without its costs either. This is particularly evidenced by the need to centralize all state changes to the `World` in the `Rulebook` object and its `Rule` instances. Having only part of the state changes follow the pattern could make the overall maintenance cost higher than simply carrying on without it.

For similar reasons, even if a team adopts the *Rulebook* from start, the eventual need to change specific implementation decisions in the pattern itself also impose a risk to the project scope, though to a relatively lesser extent. The architects of a game with self-amending mechanics ought to consider the possible variations of the pattern and the consequences of each. This section discusses such design decisions and the considerations our research arrived at.

1) Storage of `Rule` instances: One of the central benefits of the *Rulebook* pattern is that rules can be dynamically added to and removed from the simulation since they are instantiated as objects. However, there are two types of rules regarding *when* the game adds them to the simulation. The first type is added during the start-up process and is never removed. The second is added and removed *always associated with a simulation element*, such as characters and items or even whole maps and environmental features (ROM-2 [2]).

On the one hand, the recurrent associations between `Rule` objects and simulation elements suggest that `Rule` should *belong* to those elements as instance variables or something similar (e.g., a component in an *ECS* architecture).[6] On the other hand, keeping a centralized storage of all `Rule` instances supports the use of specialized data structures to optimize queries for rules that match a given `Effect`, as discussed in Section III-E2. Either way, the *Rulebook* pattern mitigates eventual changes to that decision since it encapsulates inside the `Rulebook` object how `Rule` objects are accessed.

2) Rule-matching optimizations: Based on examples like *Magic: the Gathering* and the very complex *Path of Exile* (*Grinding Gear Games*, 2013), games with self-amending mechanics could require thousands of rules. The straightforward approach described in Section III-D for implementing the `Rule` class, however, scales poorly with the number of rules present in the simulation. That is because the `Rulebook` always needs to call `handleEffect` on all rule instances. Next are a few alternatives to keep in mind.

One option is to filter rules based on the spatial relationship of the simulation objects involved. Every rule might specify an "area of effect" so that the `Rulebook` can rely on collision detection systems — with support for optimizations such as spatial partition algorithms [17] — to only invoke the `handleEffect` method in the `Rule` objects that have their spatial preconditions satisfied and the rule then further tests the situation (RIC-3 [2]). In the grid-structured game of the example, one could limit resolution to only `Rule` objects tied to the current tile (which is enough to solve the example).

Another approach is to filter rules based on the effects they operate on instead of the simulation elements involved. For instance, rules could be indexed by the specific subtypes of `Effect` they target. Using this method, supporting rules that affect multiple types of effects might be less trivial.

A third alternative is to query rules based on their predicate broken into logic clauses. Ernst *et al.*'s predicate-dispatch approach, for instance, suggests a complex but powerful matching algorithm for that [25]. Since there are similarities between the *Rulebook* pattern and rule-based systems, some classic solutions in that field – such as the Rete algorithm [24] – might be useful as well.

3) Solving `Rule` conflicts: Two `Rule` objects conflict whenever both handle the same incoming `Effect` with non-commutative operations. In such cases, whichever gets to run first might prevent the other from doing so because either the `Effect` or the simulation state changed. A conflict resolution mechanism (usually part of the `Rulebook` object) determines a priority between rules and can improve the determinism of the simulation.

A straightforward but still considerably flexible approach is to always match against all rules but in a consistent order. This way, those that go first act as "higher priority", shaping the

[6]Example: https://tinyurl.com/grimoire-component-with-rules

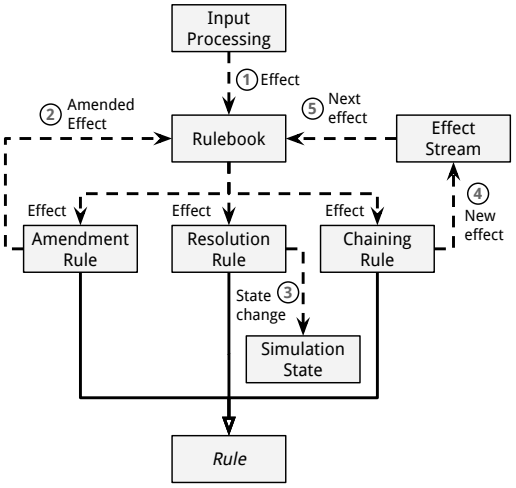

Fig. 3. Flow of effects using the Amend-Resolve-Chain (ARC) variation of the *Rulebook* pattern, which prioritizes rules according to what they do to effects. Following the numbered markers: (1) the game feeds an effect to the `Rulebook` (user input is just an example); (2) that effect passes through all applicable amendment rules, which change its data; (3) after there are no amendments left, the effect never changes again and goes to the resolution rules, which apply it to the simulation state; (4) once all simulation state changes end, chaining rules then take the effect and the simulation state into account to schedule new effects; and (5) at some point in the future, the `Rulebook` polls those queued effects to repeat the cycle with them.

`Effect` and the simulation state before other rules have a chance to. When changes to an `Effect` during its resolution should give previously considered `Rule` objects a new chance at matching, the `Rulebook` can detect this change and reset the iteration from the start — always keeping the matching order consistent — to give earlier rules a new chance at matching. In this case, one can prevent soft locks by never letting a `Rule` handle the same `Effect` more than once.[7]

The drawback of this approach is its scalability. As Plotkin claims, keeping track of the order of thousands of rules might be unsustainable [27]. He argues that any specific heuristic would always miss any number of corner cases, motivating him to propose using the rule system itself to determine which rules win a conflict. We explored this approach in an early proof-of-concept of our previous research [2].[8] Though it worked, it made the code considerably more complex.

Our investigation showed we can prioritize some rules over others consistently. By ordering rules by whether they amend, resolve, or chain effects [2], developers maximize the information each type of rule has about the effect being processed. Though there is no guarantee that all games are compatible with this, it is a practical rule of thumb to follow.

For example, the "travel onwards" action in the caravan game, when executed inside a tile with the "abandoned ruins" and "rain" land features, may prioritize rules as follows. First, amendment rules from the ruins double the duration property of the travel effect. Second, resolution rules move the caravan and advance time twice as usual in the simulation — because the slow-down rule ran beforehand. Third, chaining rules from the rain chain a new effect for soaking some of the caravan's supplies, which only happens if the caravan did move. If any rule amended or resolved the travel effect differently, then the chaining would be appropriately avoided.

We call this three-step division of rules the Amend-Resolve-Chain (ARC) variation of the *Rulebook* pattern (Figure 3). It has the added benefit of allowing us to "preview" the resolution of an effect *by only applying the amendment rules* and looking into the resulting `Effect` object. Since it will not change further, it is significantly easier to deduce its resolution, especially if the simulation is deterministic. Even when a game does not use this variation, it is still useful to talk about rules in terms of what their role would be in the "ARC pipeline".

F. Example Resolved

Assuming the hypothetical caravan game adopted the *Rulebook* pattern, we will illustrate how that could be implemented and how one could add the "abandoned ruins" mechanics. For this game, the straightforward approach of using a single method for the `Rule` class (`handleEffect`) is enough. Keeping all `Rule` objects in a central list is also enough, as is implementing the `Rulebook` by iterating over that list and letting every `Rule` try to handle every `Effect`.

Every user action is initially translated into an `Effect` instance. For didactic purposes and simplicity, `Effect` objects can be dynamically-typed, JSON-like associative tables. That way, the "travel onwards" action might translate, for instance, into an `Effect` as simple as {`"travel": "east", "duration": 10`}. Then, a series of amendment rules might look at the rain in the current tile and chain a {`"soak_supples": true`} effect afterwards. Algorithm 1 illustrates a *very simplified* implementation of this.

As for the "abandoned ruins" land feature, at first glance, we can see that there are likely two amendment rules needed. The first would test for effects it considers to involve movement and the second would test for all other effects. However, if other rules in the future need to test for the same conditions, then we would have to duplicate that piece of code. Instead, we will use a third, higher-priority amendment rule to tag the effect as being movement-related. Following this approach, Algorithm 2 shows the simplified pseudo-code implementation required to enable the "abandoned ruins" mechanics. Because the *Rulebook* pattern allows us to amend simulation effects, we only needed to add three rules to cover mechanics that potentially involved *all* effects in the game.

G. Known uses and related patterns

The *Rulebook* pattern is a generalization of several specific solutions found in the game developer community. Among these, two were central to the design we arrived at: Plotkin's rule-based programming for interactive fiction [27] — based on the actual implementation of the *Inform7* engine[9] — and Bucklew's "components and events" design [28] for the games *Caves of Qud* (*Freehold Games*, 2015) and *Sproggiwood* (*Freehold Games*, 2014). At the same time, there are patterns that either resemble the *Rulebook* in some aspect or fulfill part of its features but do not solve the particular problem of self-amending mechanics. Using the example caravan game where applicable, we elaborate on some of these relationships.

[7]Example: https://tinyurl.com/grimoire-prevent-soft-lock

[8]Example: https://tinyurl.com/prototype-reentrant-rulebook

[9]https://ganelson.github.io/inform-website/

Algorithm 1 Methods for rules initialization, user input translation into travel effects, rule adjudication algorithm, resolution rule for default travel behavior, and chain rule for rain soaking caravan supplies. The field "travel" inside the effect contains the traveling direction or `null` for any other effect, while the "duration" field contains the number of in-game time units the effect takes to execute. When chaining effects, let us assume that it keeps a reference to the previous effect so other rules can further evaluate the circumstances.

```
function GAME::INIT( )                          ▷ Partial implementation
    rlbk ← self.GETRULEBOOK()
    rlbk.ADD(TRAVELRULE.NEW())
    rlbk.ADD(RAINRULE.NEW())
function GAME::UPDATE( )                        ▷ Partial implementation
    i ← self.GETUSERINPUT()
    if i.ISMOVEMENT() then
        w ← self.GETWORLD()
        d ← i.GETDIRECTION()
        e ← {"travel" : d, "duration" : 10}
        self.GETRULEBOOK().RESOLVE(w, e)
function RULEBOOK::RESOLVE(w, e)
    for rule ∈ self.GETRULES() do               ▷ Insertion order
        rule.HANDLEEFFECT(w, e)
        if e.ISCANCELLED() then return
function TRAVELRULE::HANDLEEFFECT(w, e)
    if e["travel"] ≠ null then
        w.MOVECARAVAN(e["travel"])
        w.PASSTIME(e["duration"])
function RAINRULE::HANDLEEFFECT(w, e)
    L ← w.GETCARAVAN().GETLANDFEATURES()
    if "rain" ∈ L then
        self.CHAINEFFECT(e, {"soak_supplies" : true})
```

Algorithm 2 Amendment rules that together implement the mechanics for the "abandoned ruins" land feature, assumed to be registered in an order that makes sense. The first tags all effect related to movement, the second doubles the duration of any effect tagged as movement, and the third negates rain complications on effects chained from non-movement effects. The parameters **w** and **e**, the fields "travel" and "duration", and effect chaining work as in Algorithm 1. The "gather" field contains the type of resource the caravan is looking for or `null` for any other effect. If more effects should fit the first rule, that is the only place we ever need to change for the purposes of detecting movement-related effects.

```
function MOVEMENTRULE::HANDLEEFFECT(w, e)
    if e["travel"] ≠ null or e["gather"] ≠ null then
        e["movement"] ← true
function LOSTINRUINSRULE::HANDLEEFFECT(w, e)
    L ← w.GETCARAVAN().GETLANDFEATURES()
    if e["movement"] = true and "ruins" ∈ L then
        e["duration"] ← e["duration"] * 2
function SHELTERINRUINSRULE::HANDLEEFFECT(w, e)
    L ← w.GETCARAVAN().GETLANDFEATURES()
    if e["soak_supplies"] = true and "ruins" ∈ L then
        e_0 ← e.PREVIOUSEFFECT()
        if e_0["movement"] ≠ true then
            e.CANCEL()
```

1) Component or ECS: Both these patterns use composition over inheritance to often reuse mechanics across different game simulation elements, reducing the cost of defining new types [3], [17], [18]. In the caravan game, these components might be land features such as the "abandoned ruins". However, that does not change the work required when different, separate user actions (e.g., "travel" and "gather") need to detect the presence of that component because those patterns do not specify where those actions are implemented in the codebase.

2) Command: Representing actions as objects, as the *Rulebook* does with effects, is similar to how the *Command* allows programmers to turn functions into objects [17], [26]. The "travel" action could be one such object. That said, the pattern says nothing about how the action is actually implemented – in fact, its goal is to abstract that away. There are no guidelines for changing the behavior of multiple *Command* objects.

3) Chain of Responsibility or Decorator: Chain of Responsibility involves passing a request object along a series of *handlers* that might do something with it or prevent it from going forward, while *Decorator* stores objects inside nested *decorators* composed to dynamically shape the behavior of those objects [26]. Both patterns could provide a pipeline where mechanics (e.g., doubling turn durations inside "abandoned ruins") are processed sequentially as handlers and decorators. As such, the *Rulebook* could use them as part of its implementation, but the patterns *by themselves* give no insight into how those rules fit into the larger context of game simulation to promote self-amending mechanics.

4) Observer: A mechanism for raising "events" without knowing which functions will "catch" them [17], [26], similar to the interaction between effects and rules in the *Rulebook* (e.g., the double turn duration rule is an "observer" of the "travel" event). In fact, Bucklew calls effects "events" [28] but we use the term "effect" to set it apart from typical event systems and because it more or less matches the concept of "effect" in the rules of *Magic: the Gathering* (*Wizards of the Coast*, 1993) [15]. Regardless, *Observer* can be used as part of the *Rulebook* pattern but also lacks the wider guiding structure for supporting self-amending mechanics by itself.

5) Blackboard: The way the *Rulebook* treats effects as pure data containers that rules collectively read from and write to resembles the *Blackboard* architectural pattern [12]. In this pattern, all subsystems of an application operate on a shared data repository. However, that repository usually contains the entirety of the working state of the system, not just the representation of individual operations like with effects.

IV. Case Study

To evaluate the *Rulebook* pattern, we carried out a case study based on real-world games with self-amending mechanics. We followed Runeson and Höst's guidelines for conducting case studies in software engineering [29]. Following their criteria, this is a *descriptive, interpretive, qualitative study*.

TABLE I
JAM GAMES INVESTIGATED IN CASE STUDY

	Legend of Slime [10]	***Dungeon Architect*** [11]	***Honey BZZZness*** [12]
Jam	Crossover Jam	Ludum Dare	Ludum Dare
Edition	2021 [13]	50 [14]	53 [15]
Team size	4	4	7
Duration	48h	72h	72h
Engine	Godot 3.3	Godot 3.4	Godot 4.0
Genre	Puzzle	Base-building	Clicker

The goal was to evaluate the architectural consequences of using the *Rulebook* pattern, analyzing its design process. We needed games that (1) use the *Rulebook* pattern, (2) are finished in some capacity, (3) provide access to their source code, and preferably (4) disclose the implementation design process behind using the pattern. To that end, we chose games among jam submissions the authors participated in that not only adopted the *Rulebook* pattern but were complete enough to provide a clearer picture of the pattern in action. The jams took place over the last two years and our participation, while unrelated to the current research, was still influenced by the experiences we had with the *Rulebook* pattern. We chose three games to offer multiple perspectives on the pattern. They are all available online under the GPL v3 license and the design process was accessible to this study due to our direct participation. Table I has a brief overview of each game. The analysis follows the reference model of the *Unlimited Rulebook* reference architecture [2], which sees games as interactive simulations processed by applying effects that change its state. We sought to answer the following research questions:

RQ1 What motivated the adoption of the *Rulebook*?
RQ2 How were the design decisions made in each case?
RQ3 What were the architectural consequences in each case?

For RQ1, we relied on the technical and gameplay requirements of each game — including design specifications, team composition, and available development time — to understand implementation decisions. To answer RQ2, we inspected the source code of each game and briefly described the key classes that enable the *Rulebook* pattern. We took note of any peculiarity and made a short list of notable rules where self-amendment was achieved. Last, we answered RQ3 by highlighting the development opportunities and challenges faced during each jam due to the *Rulebook* pattern.

A. Results

This section divides the results by game, providing brief descriptions to contextualize the collected information. We present the games in chronological order of development.

1) Legend of Slime: A 2D game where a slime merges with other slimes to absorb and combine powers, using them to solve puzzles (Figure 4). The slimes' powers are based on typical elemental forces (fire, water, wind, etc.) and provide unique abilities (e.g., a lightning zap).

The large number of combinations between elements and their interactions with the environment motivated using the *Rulebook* pattern. However, there were concerns that learning the pattern would take time (being in a 48-hour jam). Thus, the developers favored an approach that relied on the more familiar *Component* pattern [17], widely known for its use in the popular *Unity* engine.[16] Thus, we based the design on Bucklew's variant [28], which dismisses the central `Rulebook` object, cutting on the boilerplate.

The team implemented Effects as JSON-like objects. Each effect had a single type, emulating a tagged union.[17] Rules were the main type of components used to compose simulation objects, which worked as "individual rulebooks" the game could apply effects to. Rule components could implement multiple rules with its single effect-handling method. Conflict-solving followed the order of the components, with earlier rules being able to shut down an effect before it reached other rules. Only two effects required special treatment outside of rules because they involved map tiles inaccessible to individual simulation objects.[18]

There were two main self-amendment cases in *Legend of Slime*. The first is that, by default, simulation objects that collided with an obstacle stopped moving, but a few rules allowed players to enter a blocked space, such as when they absorbed another slime. The other is when players gained the power from another slime, because if they already had a previous power they might combine instead. To do this, the rule of each power amended the effect to gain other powers.

The game achieved over a dozen different puzzle mechanics and still managed to feature a complete sequence of stages using all of them. We tribute that in great part to the "rules as components" design. In particular, grouping multiple rules in a single component made it easier to add and remove them at runtime. However, since most effects only interacted with a single entity, conventional abstract methods for each effect type might have achieved similar results. Though the *Rulebook* enabled the self-amending mechanics of combining slime powers, it still incurred code duplication because each power had a rule to combine with each other compatible power. For instance, both the fire[19] and water[20] powers had a rule to mix into the wind power. Future development in *Legend of Slime* would have to consider refactoring rules into more reusable parts to reduce long-term maintenance costs.

2) Dungeon Architect: A 2D game where the player builds a series of grid-based dungeon layouts to delay a delving party of heroes as much as possible (Figure 5). The party traverses the grid from block to block more or less randomly, and each block has self-amending mechanics that determine

[10] https://github.com/uspgamedev/crossover-slime
[11] https://gitlab.com/uspgamedev/ld50
[12] https://gitlab.com/uspgamedev/jams/ld53-clicker
[13] https://itch.io/jam/crossover-jam-2021
[14] https://ldjam.com/events/ludum-dare/50/dungeon-architect
[15] https://ldjam.com/events/ludum-dare/53/honey-bzzzness
[16] https://unity.com/
[17] Example: https://tinyurl.com/slime-tagged-effect
[18] https://tinyurl.com/slime-workarounds
[19] https://tinyurl.com/slime-fire-with-water
[20] https://tinyurl.com/slime-water-with-fire

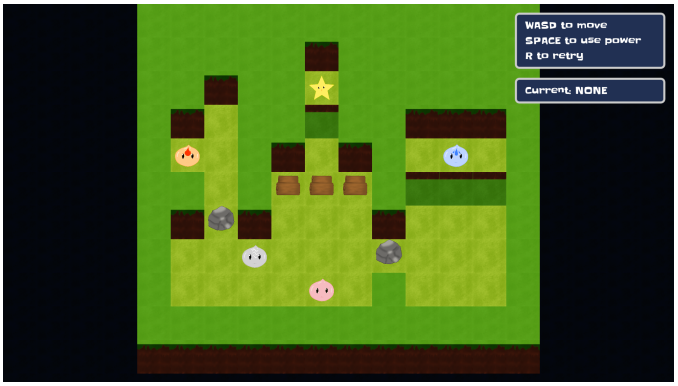

Fig. 4. Screen capture of *Legend of Slime*, one of the games investigated in the case study. In this game, the player controls a slime that combines with other slimes to wield different elemental powers to solve puzzles. For instance, water slimes fill gaps with water to cross them safely.

Fig. 5. Screen capture of *Dungeon Architect*, one of the games investigated in the case study. In this game, players place blocks to build a dungeon with the goal of making the incoming party of heroes take as much time as possible — the more they stay inside, the more currency the players receive.

how much time the party takes in each room based on the party composition and other nearby blocks.

Dungeon Architect has simple mechanics and most of the effort went into the graphics. That said, developers needed a flexible way to implement dungeon block mechanics as the designers came up with ideas over the course of the jam. There would only be two effects so there was no need for the ARC variant or effect streaming, and the previous approach of type-tagged JSON-like effects and rules as components sufficed. This time there was an overseeing `Rulebook` object because rules had to reach any effect in the simulation.

Since all active block rules were elegible to process any effects, the `Rulebook` object used a simple mechanism to filter rules that expected a specific type of effect. As mentioned, there were only two types of effect in the game: one for computing how much time the party spent in a block and one for determining the connection between adjacent blocks. As such, almost all self-amending mechanics involved rules that affected the party's traversal of each block depending on the dungeon layout. An illustrative example would be the "treasure room" block, which increased the time spent in it if any adjacent blocks had a dangerous challenge, to represent the party taking their time to celebrate their achievement.[21]

Most dungeon blocks ended up with unique self-amending mechanics, enabling less reuse of rule components than in *Legend of Slime*. Despite that, the flexibility of the *Rulebook* pattern let us spend less time handling the interactions between self-amending mechanics and more time polishing the game as a whole. The centralized `Rulebook` allowed rules to be applied to effects beyond the simulation object they were attached to, supporting interesting interactions between different block types. *Dungeon Architect* also achieved an acceptable gameplay length given the time constraints and team size.

3) Honey BZZZness: A 2D mobile game where the player manages the honey production of a bee hive (Figure 6). They tap the screen to collect pollen, which they can convert into honey. Honey, in turn, is used to hire bees that perform a variety of tasks, produce wax to extend the hive, and unlock skills in a skill tree. All these elements affect the production of the hive differently through self-amending mechanics.

The ARC variant of the *Rulebook* was partly adopted because the *cost* of many in-game actions would change based on self-amending mechanics. By separating amendment rules from resolution rules, the game could preview effects to determine its real cost taking all rules into consideration.[22] Otherwise, it would have to compute the cost for the user interface then again when actually resolving the effect. Some rules had no particular simulation object to attach to (e.g., the rule that by default locks production of all bees until unlocked via skill tree) and there were different types of rule-bearing objects this time (bees, hive expansions, and skills). Because of that, rule storage was more flexible: rules could be attached to *any* node in the scene tree.

That resulted in the most different approach among the studied games. Effects had a proper class encapsulating a JSON value, but did not rely on a type tagging mechanism. Instead, the stored value was a shallow dictionary where each field was called a *trait*, emulating the *Component* pattern. Rules were divided into `MofidyRule` (amendment) and `ApplyRule` (resolution). The central `Rulebook` object this time offered the option of just *previewing* the result of an effect. Processed effects were dispatched to all rule nodes in the Godot scene tree, no matter where they were. A set of "core" rules was kept in a single place while the rest came attached to whatever simulation element introduced it (bees, hive expansions, or skills).

The two main forms of self-amendment in *Honey BZZZness* were mechanics that changed the values of resource transactions (e.g., how much honey was produced when processing pollen) and mechanics that enabled other mechanics (e.g., unlocking new bees in the skill tree). Transaction effects shared traits in their calculations that allowed us to reuse code across multiple rules. Mechanics that enabled other mechanics had some effects be cancelled by default, then introduced rules via unlocked skills that overwrote that behavior.[23]

Being the most complex implementation of the *Rulebook* studied, there was a steeper learning curve for the team,

[21] https://tinyurl.com/dungeon-treasure-room

[22] https://tinyurl.com/bzzzness-preview-cost

[23] https://tinyurl.com/bzzzness-unlock-bee-skill

which led to some misunderstandings on how to use the pattern and cost some extra time to implement. Together with the feature-packed user interface, the complexity of the mechanics contributed to the team barely finishing the game in time for release, with many mechanics left out. Because the rules were spread out in the scene tree, it was harder to provide them access to the simulation state, requiring some time-consuming workarounds. Despite those issues, the ARC approach prevented complications from rule conflicts entirely while the effect preview feature proved to be a valuable tool throughout development, suggesting the resulting game might have been even less complete without the pattern.

B. Discussion and implications

Understanding what type of self-amending mechanics the game had beforehand played a key role in the design process. In the particular case of jam games, because of the tight schedule, the design specification is written alongside the programming progress, so even a slightly increased variety of self-amending mechanics (slime powers, dungeon blocks, bee types) meant a certain expectation of unpredictable changes. That motivated the developers to invest in a decoupled structure for mechanics using the *Rulebook* (RQ1).

Design decisions depended on the constraints of the effects and rules in each game, as well as the composition of the team (RQ2). The variety of effects dictated how they would be stored, with all three games relying on JSON-like values since stronger typed alternatives involved extra boilerplate code that was not compatible with the scope of a jam. The lifetime cycle of rules was the key factor for determining how to store them — whether they existed alongside simulation elements and whether they could be added and removed at any time. The need for previewing effects decided if rules would follow the ARC variation. Finally, the experience and size of the team influenced whether the developers chose to fully adopt the *Rulebook* pattern or only partially apply its principles. It is worth noting the developers also never relied on streaming or chaining mechanisms, though they often had to resolve effects during the resolution of other effects synchronously.

The main architectural consequence (RQ3) across all games was that developers could implement the unplanned, mid-jam design specifications of self-amending mechanics without changing multiple parts of the codebase. The implementation of the core elements of the *Rulebook* pattern required 15 lines of code[24] at worst, and 4[25] at best, making the adopted variations of the pattern fast and practical to include and start working with. Furthermore, using rules as *Godot* nodes promoted reusability in the workflow. The only situation where self-amending mechanics required multiple changes was when rules needed unforeseen access to specific parts of the simulation state - because that state was not stored in a cohesive module, a divergence from the assumptions of Section III-D.

C. Threats to Validity

We note that part of these benefits found in this study could stem from the expertise of the teams instead of the pattern

[24]https://tinyurl.com/bzzzness-rulebook
[25]https://tinyurl.com/dungeon-rulebook

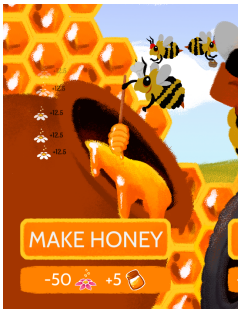

Fig. 6. Screen capture of *Honey BZZZness*, one of the games investigated in the case study. In this game, you manage a bee hive to produce honey for the queen. You collect pollen by tapping the screen and can use skills, bees, and hive expansions to produce honey, wax, and other things. By balancing your resources you can expand the hive and defend against predators.

used. That said, the same observable benefits (unforeseen mechanics requiring few code changes) were present despite the differences in each team composition and game genre. That suggests tangible advantages of the *Rulebook* pattern — the key architectural aspect in common among the studied games.

The conclusions taken might not hold for games outside the scope of game jams. In places where we were content with workarounds, a longer project might have required a revision of previous design decisions mid-development, a scenario the study did not cover. We also used the same engine and covered only three genres, so there might be other aspects of the *Rulebook* pattern that did not play a role in our investigation. As an initial study, however, it fulfills its goal of illustrating the essentials of the proposed pattern. Besides, games known to use the pattern to great effect, such as *Caves of Qud*, do feature different engines and genres, and a larger scope [28].

Last, since we were part of the teams that developed the studied games, the analysis is likely to carry biases. The greatest bias is that, as researchers of software architecture and self-amending mechanics, some design decisions might sound logical to us but not to someone not familiar with these subjects. That means that others might have come to different conclusions, especially regarding RQ3. At the same time, these different conclusions would, in turn, carry the bias of *not* being familiar with the pattern. We need both these perspectives and more to fully picture the *Rulebook*, and this case study provides the first pieces of the puzzle.

V. Conclusions

Games are a creative medium and self-amending mechanics empower developers with a wide design space to express themselves and build engaging dynamic worlds to explore. However, they are a critical part of the architecture prone to becoming a bottleneck for new changes. To allow these mechanics to interact with each other in intricate manners, games should decouple themselves from them while also providing a flexible structure for their self-amendment.

The *Rulebook* is a **general solution** that fulfills these requirements based on extensive research following a systematic process. The benefits it provides are subject to the idiosyncrasies of the self-amending mechanics and development process, with some genres and environments bearing clearer advantages. As an architectural pattern, it provides developers

with clear guidelines for devising their implementations, with each variant offering different advantages and disadvantages. The case study performed further supports the pattern through real-world implementations of three open-source jam games and the design process behind them. The *Rulebook* pattern formalizes what was only scattered knowledge into a tool now available to developers and researchers alike.

A. Future Work

Though self-amending mechanics have always existed, their typification for the purposes of architectural design is still a novel concept. As such, there are several opportunities for innovative research. As more games consciously adopt the *Rulebook* pattern — instead of incidentally intersecting its ideas — more characteristics, limitations, and variations will become evident. In particular, performing more studies — especially empirical studies or investigations into larger, commercially successful games — compose the kind of research we hope to work with in the future.

We have plans for a new case study where we critically analyze larger open-source games we have not participated in to determine how the *Rulebook* pattern could improve their architecture. Given how games often strive for performance, there are also many optimization opportunities in the rule-processing aspect of the *Rulebook* pattern. We touch upon only a few in Section III-E2. One particular approach that we believe to have great potential is the parallelization of rules — a challenging problem because rules share memory access to both effects and the simulation state.

ACKNOWLEDGMENTS

Grant 2017/18359-6, São Paulo Research Foundation (FAPESP).

REFERENCES

[1] Peter Suber. (1982) Nomic: A Game of Self-Amendment. Available online at http://legacy.earlham.edu/ peters/nomic.htm (last accessed July 23th, 2022).
[2] W. K. Mizutani, "The unlimited rulebook: Architecting the economy mechanics of games," Ph.D. dissertation, Department of Computer Science, University of São Paulo, 2021, available: https://www.teses.usp.br/teses/disponiveis/45/45134/tde-22122021-205515/publico/texto.pdf.
[3] J. Gregory, *Game engine architecture, third edition*. CRC Press, 2019.
[4] J. Schell, *The Art of Game Design, Third Edition*. CRC Press, 2020.
[5] R. Hunicke, M. Leblanc, and R. 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*. Press, 2004, pp. 1–5.
[6] E. Adams and J. Dormans, *Game Mechanics: Advanced Game Design*. New Riders, 2012.
[7] N. Nordmark, "Software Architecture and the Creative Process in Game Development," Master's thesis, Norwegian University of Science and Technology, 2012.
[8] A. I. Wang and N. Nordmark, "Software Architectures and the Creative Processes in Game Development," in *International Conference on Entertainment Computing*, 2015.
[9] W. K. Mizutani and F. Kon, "Toward a reference architecture for economy mechanics in digital games," in *Proceedings of the Brazilian Symposium on Games and Digital Entertainment (SBGames)*, Oct. 2019, pp. 623–626.
[10] ——, "Unlimited rulebook: a reference architecture for economy mechanics in digital games," in *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, Nov. 2020, pp. 58–68.
[11] W. K. Mizutani, V. K. Daros, and F. Kon, "Software architecture for digital game mechanics: A systematic literature review," *Entertainment Computing*, vol. 38, p. 100421, Mar. 2021.
[12] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. Chichester, UK: John Wiley & Sons, 1996, vol. 1.
[13] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. 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*, 2014, pp. 143–152.
[14] J. Dormans, "Engineering Emergence - Applied Theory for Game Design," Ph.D. dissertation, University of Amsterdam, 2012.
[15] Wizards of the Coast. (2023) Magic: the Gathering's Comprehensive Rules. Online reference (last accessed March 15th, 2023). [Online]. Available: https://magic.wizards.com/en/game-info/gameplay/rules-and-formats/rules
[16] H. Fujibayashi, S. Takizawa, and T. Dohta. (2017) Breaking Conventions with The Legend of Zelda: Breath of the Wild. Conference talk available online at https://www.youtube.com/watch?v=QyMsF31NdNc (last accessed Match 13th, 2022).
[17] R. Nystrom, *Game Programming Patterns*. Genever Benning, 2014.
[18] C. West. (2018) Using Rust For Game Development. [Online]. Available: https://kyren.github.io/2018/09/14/rustconf-talk.html
[19] A. Ampatzoglou and I. Stamelos, "Software engineering research for computer games: A systematic review," *Information and Software Technology*, vol. 52, no. 9, pp. 888–901, 2010.
[20] L. B. Morelli and E. Y. Nakagawa, "A Panorama of Software Architectures in Game Development," in *International Conference on Software Engineering and Knowledge Engineering*, 2011, pp. 752–757.
[21] T. Olsson, D. Toll, A. Wingkvist, and M. Ericsson, "Evolution and Evaluation of the Model-View-Controller Architecture in Games," in *International Workshop on Games and Software Engineering*, 2015.
[22] 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)*, 2015, pp. 25–32.
[23] C. Grosan and A. Abraham, *Intelligent systems*. Springer, 2011, vol. 17.
[24] I. Millington and J. Funge, *Artificial intelligence for games*. CRC Press, 2009.
[25] M. Ernst, C. Kaplan, and C. Chambers, "Predicate dispatching: A unified theory of dispatch," in *European Conference on Object- Oriented Programming*, 1998, pp. 186–211.
[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
[27] A. Plotkin. (2009, May) Rule-Based Programming in Interactive Fiction. Online article availble at https://eblong.com/zarf/essays/rule-based-if/ (last accessed March 15th, 2023).
[28] B. Bucklew. (2015) Data-Driven Engines of Qud and Sproggiwood. Video from conference talk available online at https://www.youtube.com/watch?v=U03XXzcThGU (last accessed March 13th, 2022).
[29] P. Runeson and H. Martin, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, pp. 131–164, 2009.

Wilson Kazuo Mizutani received his PhD degree from the University of São Paulo in 2021 with a doctoral thesis on software architecture applied to game development, awarded best thesis in the 21st Brazilian Symposium on Computer Games and Digital Entertainment. His research fields include game development, software architecture, computer graphics, and agile development.

Fabio Kon is a Full Professor of Computer Science at the University of São Paulo. He has 30 years of experience in research on software development, with contributions to Software Architecture, Software Engineering, Agile Methods, Object-Oriented Patterns, Distributed Systems, and Empirical methods. He is an ACM Distinguished Scientist.