Unlimited Rulebook: a Reference Architecture for Economy Mechanics in Digital Games

Wilson Kazuo Mizutani
Department of Computer Science
University of São Paulo, Brazil
kazuo@ime.usp.br

Fabio Kon
Department of Computer Science
University of São Paulo, Brazil
kon@ime.usp.br

***Abstract*—In game development, mechanics are one of the basis of the entertainment experience. However, the cost of implementing, improving, and refactoring economy mechanics is high because solutions cannot be easily reused across products. We argue that a reference architecture reduces this cost by providing knowledge reuse in addition to software reuse. To achieve that, we designed Unlimited Rulebook, a reference architecture for economy subsystems in games. It builds on established techniques such as Predicate Dispatching, the Entity-Component-System pattern, and the Adaptive Object-Model architectural style to facilitate the addition and modification of entity types and mechanics to the game while reducing the cost of changing existing code. We evaluated this cost reduction empirically via quasi-experiments with university students in two game programming courses.**

***Index Terms*—reference architecture, design patterns, adaptive object-model, digital games, economy mechanics**

I. Introduction

Games are designed to entertain users. This is largely done through *game mechanics*: rules about how the game works and interacts with user input, creating the dynamics that characterize the gameplay experience [1], [2]. Game mechanics can be organized into a few categories, such as physics mechanics (position, movement, collision of objects, etc.) or narrative progression mechanics (bookkeeping of game progression, win and loss conditions, etc.) [3], [4]. *Economy mechanics* are responsible for resource accounting and transactions inside the game but in a very broad sense. They adjudicate operations from the purchase of in-game goods to character combat statistics and reserves of magic power in role-playing games, troops and buildings in strategy games, cards and information management in card games, and much more [3], [4].

However, developers can never guarantee for sure the success of their games, despite the amount of time and resources invested. To improve the odds or reduce the risks and costs involved, developers rely on strategies such as frequent playtests [2] or quality-assurance coverage [5]. One of the most common approaches is to avoid implementing games from scratch by reusing existing software. The greatest example of this are *game engines*: generic frameworks for game development that come in many formats and prices. Development teams include these tools in their pipeline to focus on programming the parts that make their game unique.

For the most part, engines support a great number of basic mechanics, especially physics mechanics since they are all based on the physics of the real world. Nonetheless, the more specific the mechanics, the less an engine can be reused. This is particularly common in games centered around economy mechanics because they do not try to simulate reality – instead, they are more metaphoric in nature, such as the "attack" statistics of a character in role-playing games. That is, economy mechanics vary more from game to game because there is no canonical model they follow. Thus, economy-centered games are more expensive to implement due to reduced opportunities for software reuse. In our research, we investigated how to help further reduce the costs of economy mechanics through knowledge reuse in the form of a reference architecture.

In the reminder of this section, we expose the exact problem we propose to reduce in Section I-A, then describe the proposal itself in more detail in Section I-B, followed by an explanation of our methodology in Section I-C.

A. Problem

In previous work [6], we identified three main challenges in the implementation of economy mechanics that reduce the opportunities for software reuse across different games. First, since economy mechanics vary widely among games, a single implementation cannot reasonably support all possible mechanics. It is not the case if we reduce the scope to a more specific genre. Engines such as the *RPG Maker* product series[1] are evidence of that. However, for the general case of economy-centered games, we can say that economy mechanics are **unpredictable**. More specifically, to simulate economy mechanics, it is a challenge to determine universal object types that model its internal state and how to access such data in a way that different games can reuse.

For instance, in an RPG, it is reasonably safe to assume (as *RPG Maker* does) that characters, items, and spells are

Under grant from São Paulo Research Agency (FAPESP, proc. 2017/18359-6.).

[1] rpgmakerweb.com

Visit the published version of this paper at https://doi.org/10.1109/ICSA47634.2020.00014

different and separate parts of the state and that combat exchanges between two or more characters are part of the core mechanics. Thus, we can determine that RPGs can have separate object types for characters, items, and spells, and that combat effects need access only to the characters involved in a specific skirmish. In a card game, however, a card could be, at the same time, a character, an item, *and* a spell, and their mechanics could involve a *very wide* spectrum of possibilities, such as controlling the actions of an opponent. *Magic: the Gathering* (*Wizards of the Coast*, 1993) is a card game featuring all these cases. To implement it, we would need a more flexible typing system and the mechanics would require access to essentially the entire state of the economy all the time. This challenge is much less present in physics mechanics, where simulated entities can be more consistently mapped into types and the use cases for data access (collision detection, geometry inspection, etc.) fit into more typical cases. Physics engines such as Bullet[2] and Havok[3] are evidence of this.

The second problem is a characteristic that economy mechanics carry from just being mechanics: during the development of a game, their specification will change constantly. In the search for the intended game experience, projects pivot their development directions multiple times within their budget and time limits. Even after release, patches, updates, expansions, and downloadable content (DLCs) could alter any of the previously specified mechanics. This is very evident in competitive multiplayer games, where developers change the mechanics from time to time to topple currently winning strategies and to diversify the players' experience. For economy mechanics, it becomes increasingly hard to add features while minimizing changes to old code – that is, they often face the Expression Problem, a remarkably non-trivial issue to design for [7]. In this scenario, software reuse is not possible because the application specification changes significantly and retroactively all the time. Thus, we say that economy mechanics are **unstable**.

Third, economy-centered games can be very **complex**. In *Hearthstone* (*Blizzard Entertainment*, 2014), a competitive multiplayer card game, the most common result of playing a card is a change to the state of a target entity (a player or another card), be it by causing damage to it, giving it a power boost, impeding it from acting, or any one of the other dozens of effects. After some years, *Blizzard* released a card collection containing *Mayor Noggenfogger* (Figure 1), which completely changes how the targeting mechanics work, *but only when it is in play*. It is an example of how economy mechanics change themselves at runtime, requiring dynamic and adaptive behavior from the game. Software reuse is less applicable when not only the games differ in economy mechanics, but also have deep complexities in their execution, *requiring a variable number of layers of dynamic behavior.*

[2]pybullet.org/wordpress
[3]havok.com

Fig. 1. *Mayor Noggenfogger*, a card from the game *Hearthstone* (*Blizzard Entertainment*, 2014) which exemplifies the complexity of economy-centered games. Its written effect only happens when it is in play, dynamically changing how one of the core mechanics of the game works.

B. Proposal

Given the limitations of software reuse in economy-centered games, we propose the reuse of knowledge – more specifically, the reuse of architectural knowledge – to reduce costs of developing the economy subsystem of game applications. Both industry and academia have faced the challenges of economy mechanics being unpredictable, unstable, and complex, and both designed solutions for each case they faced [8]. These solutions often come in the form of design and architectural patterns, such as the Entity-Component-System (ECS) pattern [5]. Our proposal complements this knowledge with a way to consistently map economy mechanics onto these patterns to guide the design of specific architectures. This mapping is done in the form of a *reference architecture* [9]. The architecture we designed and evaluated is called the *Unlimited Rulebook* (URB).

C. Methodology

The methodology used to design, represent, and evaluate the URB architecture is an iterative version of the ProSA-RA process [10]. It is divided into four steps, which we iterate on to produce an improved version of the architecture at the end of each cycle. The first step is information gathering about the domain when we determine which sources we draw information from and investigate its characteristics and common solutions. In the second step, we analyze the collected information to understand the domain. With the support from domain experts, we find patterns in the recurring system requirements of studied sources. Next, we synthesize these specific system requirements into more general *architectural requirements* that, in turn, we group into *domain concepts*. The third step is essentially designing the reference architecture or, in the case of our iterative variant methodology, the changes

that should improve the architecture regarding the problems stated in Section I-A. Last, the fourth step is to evaluate the design and measure how well it meets our objectives, noting where it could improve in the next iteration.

For our information sources in the first step, we performed a systematic literature review [11] (still unpublished) analyzing the architectures used for game mechanics in general. We then gathered knowledge from industry authors and the gray literature, which we discuss in Section II. Additionally, we studied the implementation of open source games and engines and consulted with USPGameDev[4], a student special interest group from the University of São Paulo. There are plans for more information source investigations in future iterations of the URB architecture (see Section VI). The architectural requirements and corresponding domain concepts as well, as the process through which we arrived at them, are published in previous work [6]. In this paper, we focus on the latest version of the URB architecture, which has been evaluated through proofs-of-concept and two quasi-experiments.

D. Text Structure

This text is organized as follows. Section II discusses other works related to our research and how they contribute to each other. Section III describes the URB architecture, its reference model, the used architectural and design patterns, and presents how those relate to each other. We follow with a small usage example of the URB architecture in Section IV. In Section V, we explain how we evaluated the URB architecture and the results of these evaluations. Section VI lists the next steps in our research to converge the URB's design. Last, Section VII discusses the results and the impact our work has on the field.

II. Related Work

The study of software architecture in games is very diverse but relatively sparse. Morelli and Nakagawa's systematic mapping [12] found 33 studies on the subject in 2011, and Ampatzoglou and Stamelos' systematic literature review [13] on the broader subject of software engineering in games found 84 studies, of which less than 10% regarded software architecture or software design. Notable studies usually focus on networked multiplayer games [14], such as Zhu *et al.*'s systematic review of game network architectures [15] and Zhu's dissertation on model-driven game development [16], especially their proposed *Game World Graph* (GWG) framework.

That said, game mechanics are rarely well-defined in software architecture studies. Our systematic literature review [11] on this narrower field found only 36 pertinent studies, almost ten years after Morelli's initial mapping. Very few studies in our review explicitly referred to the architecture of game mechanics. BinSubaih *et al.* used the term "G-factor" (as in "*game* factor") to determine the part of a game responsible for game state, object model, and game logic [17]. In studies regarding the use of the Model-View-Controller (MVC) architectural pattern [18] in games [19], [20], mechanics coincide with the *Model* part of the application. There is no study (besides our previous work) about the architecture of *economy* mechanics specifically.

The architecture of game mechanics has a very different treatment in the industry. First, the term "mechanics" is much more common [2], [3], which facilitates discussing and studying the field. Wang and Nordmark's survey with industry professionals [8] found that "the game concept heavily influences the software architecture" and that "the creative team can affect the software architecture through the creation of a game concept, by adding in-game functionality", indicating how the industry recognizes the strong relation between game mechanics and software architecture. That is why, in this field, we consider the grey literature just as important as academic literature.

In this sense, Gregory [5] explains how high-budget games require a dedicated subsystem to support game mechanics in its engine. He advocates the use of architectures with *data-driven design*, a very widespread practice in the game industry that makes as much as possible of a game be specified as data instead of being coded into its executable application. Rabin [21] explains that "without this flexibility [of data-driven design], change is costly, and every change involves a programmer". That is, data-driven design is one of the methods used by game developers to reduce cost through software reuse.

Gregory [5] also presents the *Entity-Component-System* (ECS) architectural pattern and its variant, the *Property-Centric Object Model* – or a *"pure" ECS*. We will discuss the ECS in more detail in Section III-B, but it is a pattern known both to industry authors [5], [22], [23] and academic literature [24]. It is also the architecture of the most popular commercial game engine, *Unity3D*[5]. The ECS pattern is known for its flexibility in representing complex object models [5], [22], [24].

A last and very important group of related work is the material published by the community of roguelike[6] developers. In particular, the closest we found to architectures that reduce the cost of unpredictable, unstable and complex mechanics was Plotkin's talk on rule-based programming [25]. Despite the similarities, Plotkin's solution is not quite a rule-based system, as we will explain in Section III-B.

Though the gray literature offers more tools for reducing the cost of economy mechanics, we still found no equivalent to a reference architecture. That is, no work formally explains how to map the economy mechanics of a game into a specific architecture, even using one or more of the aforementioned practices and patterns. Our proposal is an entirely new solution to the problem that builds on previous findings from both industry and academia.

[4] github.com/uspgamedev

[5] unity.com

[6] A subset of role-playing games where resource management, permanent death, and highly complex interaction of mechanics are core features. It is a relatively niche genre in games, but that grew in popularity over the last decade as many games tried to incorporate roguelike mechanics.

III. The Unlimited Rulebook Architecture

Through the ProSA-RA process explained in Section I-C, we determined all the *domain concepts* of economy mechanics. These concepts form our *reference model*, described in Section III-A. At the same time, there are a number of architectural patterns known to both industry and academia that reduce the cost of economy mechanics. Section III-B explains them. Afterward, Section III-C describes the URB architecture as part of the ProSA-RA method.

A. Reference Model

Gregory defines game applications as *real-time interactive simulations* [5]. They are interactive because user input and application output are exchanged multiple times, instead of receiving all data first then producing all the results as output afterward. They are simulations because, with every user input, the game changes its internal state to *simulate* a virtual world, then presents the new state to the player. Both interaction and simulation happen in real-time, which constricts how much processor time the game can use between each interaction cycle. Gregory also explains that games are divided into runtime subsystems, each responsible for an aspect of the game, be it user interaction or simulation.

The purpose of our reference model is to organize the architectural requirements of the game subsystem responsible for processing economy mechanics. These requirements aim to reduce the cost of developing and maintaining this subsystem, given the unpredictable, unstable, and complex nature of the domain. In our previous work [6] we describe the complete list of architectural requirements we collected from our information sources and how we found six **domain concepts** that summarize these requirements: *Game Loop integration*, *simulation timeline*, *object model*, *behavior model*, *iterative development*, and *data-driven design*. Since then we have further re-organized the domain concepts into three simpler concepts given their shared concerns: Subsystem Integration (Game Loop integration and simulation timeline), Mechanics Model (object and behavior model), and Iterative Development (which includes data-driven design). When designing the architecture of an economy subsystem, the specific economy mechanics of the game must be mapped into a model fitting this reference model, which in turn will allow it to use the URB architecture to specialize the architecture for that specific game.

1) Subsystem Integration: Based on the principles of decoupling and encapsulation, the economy subsystem of a game must regulate how much access other subsystems have to its data and behavior. Taking the rest of the application as a "client" of this subsystem, the services it must provide include queries about the state of the economy, requests to change that state, and notifications about events of interest. This basic interface and separation of concerns compose the subsystem integration concept and limit what parts of the code are subject to changes due to new decisions in the design of economy mechanics. To meet the requirements of this domain concept, architects have to determine where the economy subsystem begins and ends, and how it services other subsystems.

2) Mechanics Model: The economy subsystem stores and manipulates data that represents virtual resources (player lives, equipped items, cards in a deck, etc.). Part of developing the game involves implementing the logic about what types of resources can exist and what operations can be done with them. That is the Mechanics Model of the economy subsystem architecture. With an understanding of the design space desired for these mechanics, architects must separate what mechanics might be stored as data and what mechanics are dynamic and must be computed at runtime. The URB architecture maps these two aspects of the economy (data and computations) into separate parts of the final architecture of the game.

3) Iterative Development: As the development of a game progresses, its design changes constantly and economy mechanics are no exception. The economy subsystem has to often change to meet new design decisions, but without a proper architecture, change can be expensive. Thus, both the Subsystem Integration and the Mechanics Model should support the requirements for Iterative Development. More specifically, this concept expects architects to identify what kind of changes are more common or important, like adding new characters and spells to a role-playing game or new cards to a card game. The URB architecture considers this when specifying the dynamic and static parts of the final architecture.

B. Architectural Patterns and Practices

Game developers have dealt with the costs of economy mechanics for decades and a number of solutions were found to help in different aspects of this field. Understanding the architecture and practices used allows us to design an architecture that not only relies on state-of-the-art and state-of-the-practice solutions, but that also fits appropriately with other game subsystems. We discuss first patterns that help understand how an economy subsystem fits and interacts with other parts of a game; then, we discuss practices and patterns used specifically to implement economy mechanics. All the practices and patterns were collected from the information sources defined in the first step of the ProSA-RA methodology. More specifically but not comprehensively, we decided on relevant items using the most frequently observed in our systematic literature review about the architecture of game mechanics [11] among other academic studies [19], [20], [26], the patterns and practices industry authors focus on in their published works [5], [22], and other grey literature references [25], [27].

1) Used outside the economy subsystem: The most basic and important architectural pattern in games is the *Game Loop* [5], [22], [26], which controls the execution flow of the game to guarantee real-time interactivity and simulation. In this pattern, there is an endless loop that periodically services the many subsystems in a game while keeping track of real-world time. Each loop is called a *frame* and performance is measured in *frames per second* (FPS). Typically desired values range from 30 FPS to 60 FPS. The economy subsystem could

either be one of the subsystems directly serviced by the Game Loop or a subsystem serviced on demand by other subsystems (or both).

Two other patterns might go along with the Game Loop. The first is the MVC pattern [18]–[20], which can be applied in different ways depending on the game and its platform. This pattern explicits a separation of concerns between domain logic (model), user interface (view), and high-level control flow of a system (controller), and was, in fact, originally designed for graphical interactive applications [18]. In games, the general idea is that the Game Loop works as the game controller: every frame, it queries input data from the interface, feeds it into the model subsystems (physics, economy, etc.), then uses the resulting state to request a new rendering of the interface graphics and sound, which the view subsystems are responsible for. A few best practices for MVC in games from Olsson's study [19] include:

- "Model classes should only carry functionality and state related to the game rules";
- "Use two interfaces to separate communication between model and user interface: one for game events and one for input"; and
- "Let controllers and view read information in the model but all requests to change data should go via the input interface".

The second pattern that works hand-in-hand with the Game Loop is the classic *State* pattern [5], [22], [28]. Games can have different interaction modes: a title menu, exploration, combat, inventory management, etc. Each mode has specific needs for user input, simulation operations, and output rendering. Game engines usually refer to each of these as *scenes* [5] (which is the case for *Unity3D* and *Godot*[7], for instance). The use of the State pattern might mean that the interface used to access game subsystems changes throughout the application execution or that subsystems themselves might require a state-machine-like behavior.

2) Used inside the economy subsystem: The Game Loop, the MVC, and the State pattern help us understand how the economy subsystem architecture integrates with other parts of the game. As for specifically implementing economy mechanics, there are a few notable patterns we used to design the URB architecture. First, the ECS pattern [5], [22] is widely applied in the industry, including in *Unity3D*, due to its flexibility and extensibility. In this pattern, game objects (such as characters, items, timers, scenery props, etc.) have *components* that each define a characteristic of that object. The player's avatar could consist of an object with a "Sprite", a "Collision", a "Jump", and a "Life" component. This way, the "type" of an object is the sum of its components (which can even change at runtime).

Being able to define entity types and behavior through composition allows a wider variety of entities and behaviors at a much lower cost than through inheritance-based typing [5], [22]. It also simplifies adding new behavior, which amounts to implementing a new component and adding it to relevant entities. When taken to the extreme, in a "pure" ECS pattern, entities can be reduced to an identifier value (e.g. an integer) used to index their corresponding components in separate lookup tables. When strictly used to implement gameplay mechanics, Gregory refers to this pattern as a property-centric object model [5].

Another pattern that flexibilizes object types and behavior but focuses on runtime adaptability is the *Adaptive Object-Model* (AOM) [29]. This pattern can be used in different levels, scaling from the *Type Object* pattern to a full *Type Square* (Figure 2) with an *Interpreter* pattern for dynamic rules [22], [28], [29]. It is especially useful for statically typed languages and to allow changes to software features without changing its source code. Because of that, we considered this pattern to reduce the cost of changing economy mechanics.

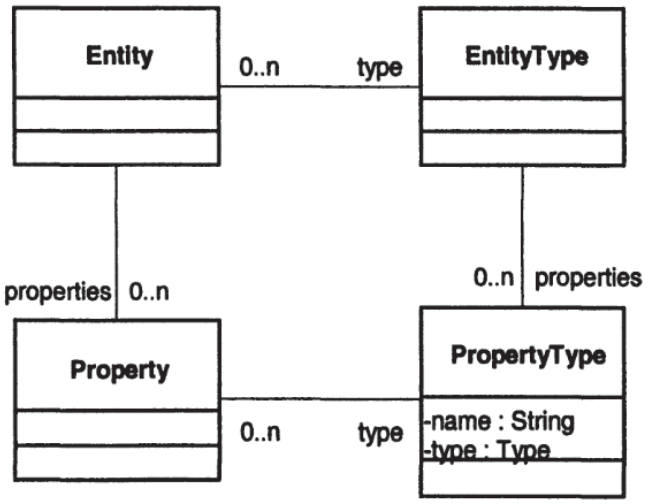

Fig. 2. The *Type Square* pattern [29]. The types of objects are defined dynamically by the relation between instances of entity types, properties, and property types. When we use it in the URB architecture, though, the names are different: entities are properties and properties are fields. The figure comes from [29].

Plotkin [25] argues that interactive fiction and roguelike games, known for their deep and complex mechanics, have many more exceptions than general rules in their gameplay. He defends that tying behavior to objects fails to handle the complexity of these games; instead, he proposes a language for expressing mechanics based on rules. While similar, Plotkin's proposed language and its hypothetical execution do not conform to a rule-based system.

Rule-based systems are systems where domain experts write if-then rules in a domain-specific language, which are then processed by a rule engine given an initial input, applying rules with valid preconditions until an end state is reached. In Plotkin's hypothetical language, however, rules are not applied in sequence. They have names that can be invoked, and only the corresponding rule with valid preconditions is executed. When two or more rules are valid, it is called a **conflict**, which must be solved with consistent criteria. Plotkin's proposal, in fact, amounts to a predicate dispatching

[7]godotengine.org

system [30], where method dispatching is determined by arbitrary predicates instead of limiting it to types, pattern matching and other dispatching mechanisms.

Predicate dispatching is valuable to economy mechanics because they allow rules to change at runtime (because their preconditions changed) and they allow new rules to be added to the game without interfering with preexisting ones since their preconditions differ. That is, it reduces the cost of economy mechanics being complex and unstable. Since it makes very little assumption over what the rules actually do, depending on the implementation specifics, it can overcome their unpredictability too.

C. Reference Architecture

A reference architecture "is a reference model mapped onto software elements" [9] and it provides "guidance for the development, standardization, and evolution of system architectures of a specific domain" [10]. Thus, we describe the URB architecture by explaining how each domain concept corresponds to components and interactions in a possible architecture for economy mechanics in games. We also constantly refer to the challenges stated in Section I-A to guide our design.

The basic design of the URB architecture follows a predicate dispatching approach similar to Plotkin's [25]. As he suggests, we begin by decoupling behavior from data. Economy simulation state is entirely kept in a `Record` object and computations are defined by `Rule` objects, following the Mechanics Model. This allows all computations to read from and write to any part of the economy state, which might seem counter-intuitive but, in fact, accounts for the unpredictable nature of economy mechanics [27]. After all, we are under the assumption that we cannot confidently estimate what information future computations will need or affect.

Also as in Plotkin's proposal, each rule defines how a computation should be done given certain preconditions. That is, each rule is composed of three parts: an identifier, which we will call a **keyword**; a procedure (e.g. a function, lambda, block, etc.) for evaluating whether the preconditions are true, called the **when-block**, and a procedure defined for the computation in such a case called the **apply-block**. Computations are requested by invoking a keyword, which works much like calling a function: you can pass arguments to it and receive the value it returns. Whenever a given keyword is invoked in the economy subsystem, it will infer what rule applies using their when-blocks and then execute the corresponding apply-block. Both when-block and apply-block can access everything in the `Record`, but only the apply-block can write to it. For instance, a hypothetical rule for the effect of the *Mayor Noggenfogger* card in *Hearthstone* could be:

```
rule for "get_target":
  when:
     "Mayor Noggenfogger" is in play
  apply:
     return a random valid target
```

With this rule, anything that tried to invoke the "`get_target`" keyword while *Mayor Noggenfogger* is in play would result in a random target. We call the part of the subsystem responsible for inferring rules the `RuleSolver`. When more than one rule is applicable, the `RuleSolver` must solve the conflict. The simplest approach is to always consider the newest rule loaded onto the subsystem, but extremely complex games can use the rule system itself to define the precedence between rules [25].

The protocol for defining new rules in the economy subsystem varies according to the level of dynamic behavior required by the game. For simpler cases, hardcoded rules might be sufficient. For sufficiently complex games, rules are defined by `RulePatch` objects, which can be loaded by the subsystem on demand. Each `RulePatch` defines rules for a very specific aspect of the game mechanics, like damage formulas, a new magical effect that changes visibility, or a card and its corresponding rules. This allows for Iterative Development since each specific feature in the subsystem should be, if well-designed, in a single `RulePatch`, limiting the costs of the change. Card games have the added benefit of implementing cards as `RulePatch` instances that can be loaded and unloaded when the cards they describe enter or leave play. Figure 3 illustrates how each class interacts inside the economy subsystem using the URB architecture (though specific implementations do not necessarily map one-to-one to this diagram).

As for the `Record`, where the economy state is kept, the URB architecture relies on the "pure" version of the ECS pattern. That is, a property-centric object model: the `Record` has a pool for each type of property that entities can have in the game while entities themselves are simple identifiers that can be indexed into these pools to retrieve its corresponding properties. If an entity has or does not have a property, rules can check for that and define associated computations accordingly. The `Record` should provide an application programming interface (API) to allow the creation, insertion, retrieval, listing, advanced querying, and removal of properties.

In statically-typed languages, implementing a `Record` that can dynamically register new types of properties is not straightforward. One way to do this is to rely on metaprogramming techniques such as macros, generics, or templates. Another way is to use the Type Object or the Type Square from the AOM pattern, so that the type of the data structure used to represent each property type can be generalized and dynamically defined by its *fields* and *field types*. This essentially makes properties behave like dynamically-typed objects, adding to the flexibility and extensibility of the URB architecture.

An important invariant of the URB architecture is that only the economy subsystem can access the `Record` directly while all external access *has* to go through the rule engine. This guarantees not only a clean separation of concerns but also prohibits other subsystems of bypassing the rules, which would couple them to their property-centric representation, making

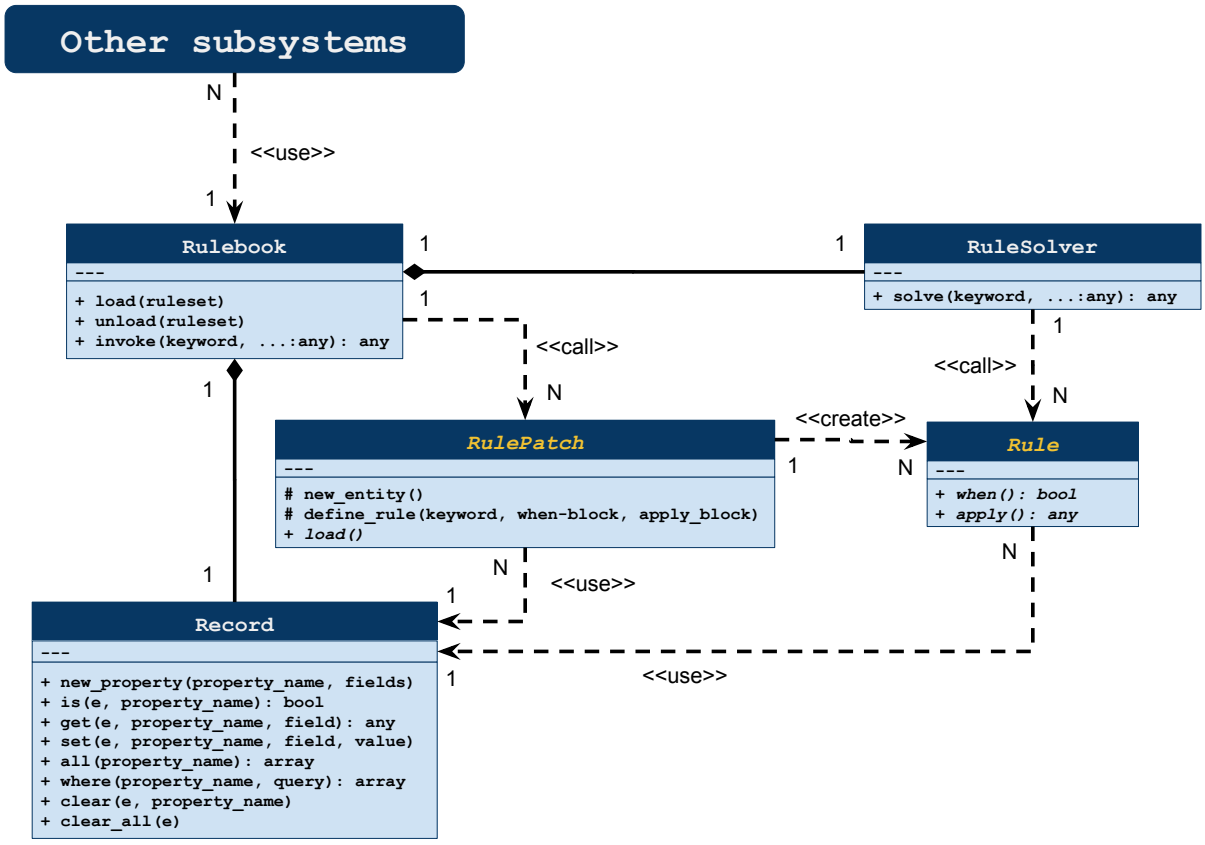

Fig. 3. Module view (UML class diagram) of the URB architecture showing the relations between each key class in the reference architecture. The highlighted abstract classes, `Rule` and `RulePatch`, are the only ones that need to be implemented into child classes to modify and add economy mechanics to the economy subsystem.

future changes more expensive. For similar reasons, the ECS used in the `Record` can be the same used by the game engine in general, but it is recommended that it be kept as a separate entity system.

The last domain concept is Subsystem Integration. The economy subsystem uses the *Façade* pattern [28] to isolate its rules and properties from other subsystems. Its Façade component works as an API to the subsystem and is called the `Rulebook`. It has two to three methods: one for invoking keywords, on for loading `RulePatch` objects, and optionally one for unloading them. Invoking a keyword can use any number of arguments and may or may not return a value. This design creates a clear definition of what does or does not concern the economy subsystem and how it interfaces with other subsystems and higher-level control patterns (e.g. the Game Loop, MVC, or State-based scene system).

IV. An Example with *Magic: the Gathering*

To illustrate how the URB architecture is used to define the architecture of a game, we use a very small and oversimplified subset of the mechanics of an existing game – *Magic: the Gathering* (MtG), a table-top card game (that also has a few digital implementations). The choice is due to MtG being not only an economy-centered game but a remarkably complex one while also being notably successful and popular. In Section V we point out an actual implementation we wrote for this example, using slightly more complex mechanics.

In this example, we assume the game is divided into subsystems for graphics rendering, user input, etc. that already provide the interaction needed to play a digital card game. We are interested in the architecture of the economy simulation subsystem that essentially manages the state of the entities of the game, most of which are the cards. It also provides operations that other subsystems can invoke to read or write to the economy state. For instance, the graphics rendering subsystem needs to read the state of cards to draw them and the input subsystem needs to invoke economy operations whenever the players try to do something in the game. Calls to these operations go through the `Rulebook` object, which encapsulates access to the inner representation of the economy

state. Once this is all in place, most of the evolution of the game revolves around new card sets which, in turn, bring new mechanics into the fold. We emulate this by presenting mechanics piece by piece, beginning with the basic definition of creature cards:

- Some cards are creatures, having *power* and *toughness* values written on them
- Creatures can be *removed*, which means they are taken out of the game and placed into the discard pile (but deleting their data from memory will suffice)

This means that some entities in the economy simulation must be registered as "creatures" with power and toughness values. Thus, the first `RulePatch` in this hypothetical implementation of MtG needs to define a new property type in the `Record`. In pseudo-code, the *`load`* method of a specialized implementation of the `RulePatch` abstraction would do something akin to:

```
record.new_property "creature":
  power: 1
  toughness: 1

rule for "new_creature" (power, toughness):
  when:
    return true
  apply:
    id <- new_entity()
    record.set(id, "creature"):
      power: power
      toughness: toughness

rule for "remove_creature" (id):
  when:
    return record.is(id, "creature")
  apply:
    record.clear(id, "creature")

rule for "get_power" (id):
  when:
    return record.is(id, "creature")
  apply:
    return record.get(id, "creature",
                          "power")

rule for "get_toughness" (id):
  when:
    return record.is(id, "creature")
  apply:
    return record.get(id, "creature",
                          "toughness")
```

That is, it has to define four new rules, each with an appropriate keyword: creation of creatures (`new_creature`), removal of creatures (`remove_creature`), read-access to a creature's power (`get_power`), and read-access to a creature's toughness (`get_toughness`). Next, we consider the basic combat mechanics of MtG:

- Creatures can receive a *damage* value, which begins at zero and increases by the amount received each time damage is taken
- Creatures can *fight* each other, meaning each of them gains an amount of damage equal to the power of the other
- If a creature ever has as much damage as it has toughness, it has *lethal damage* and dies

The second specialized implementation of `RulePatch` loaded into the economy subsystem must then define a new property, `damaged`, which carries a single unsigned integer field. Additionally, it defines rules for the keywords `take_damage`, `fight`, and `has_lethal_damage`:

```
record.new_property "damaged":
  amount: 0

rule for "take_damage" (id, amount):
  when:
    return record.is(id, "creature")
  apply:
    record.set(id, "damaged"):
      amount: amount

rule for "has_lethal_damage" (id):
  when:
    return record.is(id, "creature")
       and record.is(id, "damaged")
  apply:
    damage <- record.get(id, "damaged",
                             "amount")
    tough <- record.get(id, "creature",
                            "toughness")
    return damage >= tough
```

This code assumes some simplifications: that creatures only take damage once and that it only makes sense to ask for lethality about a creature that has been damaged. So far, we have only added types and operations that did not require change to previous patches. Now, we introduce a new set of mechanics that are an "exception" to these basic rules:

- Creatures can be *indestructible*, which means no amount of damage is lethal to them

These mechanics require not only a new property type (`indestructible`) but a change to how `has_lethal_damage` works. The predicate dispatching system of the URB architecture allows us to define how an old operation works for a new type of data. Since this new patch will be loaded after the one containing the initial rule for `has_lethal_damage`, the `RuleSolver` will defer to this new implementation in case of conflicts:

```
record.new_property "indestructible"

rule for "has_lethal_damage" (id):
  when:
    return record.is(id, "creature")
```

```
      and record.is(id, "damaged")
      and record.is(id, "indestructible")
  apply:
    return false
```

Later on in development, though, we create a new type of in-game effect:

- Some cards have the effect of *destroying* a creature card
- Creatures that are destroyed are removed
- If a creature is indestructible, destroying it does nothing instead

This new set of mechanics defines a new operation (`destroy`), which must be implemented for a number of previously defined types (`creature` and `indestructible`). We can do this in a single new `RulePatch` without having to modify any previous patches:

```
rule for "destroy" (id):
  when:
    return record.is(id, "creature")
  apply:
    invoke "remove_creature" (id)

rule for "destroy" (id):
  when:
    return record.is(id, "creature")
       and record.is(id, "indestructible")
  apply:
    do nothing
```

There are many things left out of this example. For example, it does not demonstrate how keywords can have compound implementations or how to integrate data-driven design in the architecture. Some evaluation projects presented in Section V contain more interesting uses of the URB architecture.

V. Evaluation

We evaluated the URB architecture using two quasi-experiments. Both followed the same design. Students in a game programming course were tasked with making two games: a turn-based role-playing game and a real-time tower defense. Each student team was assigned one first then the other – half the teams the role-playing game and the other half the tower defense. The first time around, students were free to design their game architecture. The second time, they had to use the URB architecture, for which we provided a sample implementation. That is, the experiments followed a *Latin Square* design [31] (Table I) where we measure how much does the cost of implementing economy mechanics change between a game developed without and with the URB architecture.

TABLE I
Latin square design of our quasi-experiments.

	First project	**Second project**
Group 1	real-time game, free design	turn-based game, URB
Group 2	turn-based game, free design	real-time game, URB

The first quasi-experiment was a pilot carried out in a summer course and with a previous iteration of the URB architecture. Games were written in Lua[8] using the LÖVE engine[9]. There were 6 students divided into two teams. The results showed that the architecture proved useful but had a steep learning curve. Since it was only a pilot, we used these observations to prepare for the next quasi-experiment.

The second quasi-experiment took place during a college computer science course for game programming. The games were also written in Lua using LÖVE and the most recent iteration of the URB architecture as described in Section III. This implementation is available in a public repository[10]. We had 25 undergraduates and 3 graduate students, most from a computer science background. Our preliminary analysis shows that:

- Implementing economy mechanics was neither harder nor easier using the URB architecture;
- The URB architecture proved itself neither useful nor useless;
- Using the URB architecture slightly reduced the amount of changes needed to the architecture of the game;
- Almost all students would use URB again if possible.

As for threats to validity, we gave both projects the same amount of time, but the students struggled more with the second one due to finals and other deadlines in the same period. This overload on the students outweighed even the fact that by the second project they had more experience, as we observed an evident decrease in their grades. Additionally, though the games used were designed to be economy-centered, we believe they still were rather small in scope for there to be notable differences in the use of the URB architecture. Yet, it would not be possible to assign students greater projects due to time limitations. For this reason, as we explain in Section VI, evaluations of future iterations of the URB architecture will focus on case studies and proofs of concept.

The learning curve still needs improvement as well. We found that code examples are particularly helpful. However, understanding the rationale, purpose, and potential of the rule system still seems to demand a firmer grasp of software design. Unless there are simpler and more intuitive ways to present and use the URB architecture, we risk limiting its accessibility. Similarly, though the URB is designed to reduce costs of an evolving codebase, it requires the extra initial effort of implementing the bare minimum features. We could not measure this aspect in the quasi-experiments because students were already provided an initial implementation of the economy simulation subsystem.

Besides the quasi-experiments, the design of the URB architecture was always accompanied by minimal proof-of-concept implementations. We have a sample project[11] that

[8] lua.org
[9] love2d.org
[10] gitlab.com/unlimited-rulebook/ur-proto
[11] gitlab.com/unlimited-rulebook/prototype-example

extends the *Magic: the Gathering* example from Section IV using the same URB implementation as the quasi-experiment and an ongoing *Dungeons & Dragons 5th-edition* simulator (*Wizards of the Coast*, 2014) using a Ruby[12] implementation[13]. In the *Magic: the Gathering* example, we evaluated whether the architecture could easily deal with mechanics that override each other, with positive results.

Though the size of these proofs-of-concept and even the games developed during the quasi-experiments was still too small to draw any conclusive evidence, one of the main concerns we have with the URB architecture is its performance. In the current straightforward implementations, invoking rules with many conflicts takes exponentially more time each time a nested rule is invoked. Such cases are rare but, given the importance of performance in the real-time experience of games, further evaluation is required to assess the processing limits of this approach and whether more optimized implementations are possible.

A qualitative analysis of the design patterns used by the URB architecture points to a few other considerations besides the cost reduction over the lifetime of the codebase. The ECS pattern, especially in the property-centric variation, provides a very flexible, dynamic and expressive type system of game entities [5], [22], [23], ensuring that the URB architecture supports a wide variety of game-specific object models. The use of the Façade pattern to encapsulate the economy subsystem decouples it from the rest of the game, increasing the opportunity for refactoring, testing, and other maintenance practices. The Adaptive Object-Model architectural style significantly promotes the use of Data-Driven Design, since additional types of entities can be defined as data and loaded at runtime, which allows non-programmers to contribute to the economy mechanics of the game directly.

VI. Future Work

Though the current version of the URB architecture is ready for use, we believe there are still improvements to be made as we begin a new cycle in the ProSA-RA process. Our future plans include interviews with professional game developers as a new information source, as well as investigation of open-source economy-centered games, especially roguelikes. We will continue our longitudinal studies as well. That includes the proofs-of-concept evaluations with the *Dungeons & Dragons* simulator and a roguelike of our own. Additionally, we will perform a case study where we refactor a preexisting open source game to use the URB architecture, measuring the benefits and drawbacks.

There are also a few concerns that require further research. The current implementations of the URB architecture solve rule conflicts by ordering all applicable rules but this will likely prove a performance bottleneck with larger games. We plan to study when this solution stops scaling and how we can speed-up the rule-solving mechanism then, especially through caching strategies. Additionally, the predicate dispatching design of the URB architecture is very generic, so we plan on mining recurrent patterns for mechanics implementation through rules. We also invite other game developers and software architects to use the URB architecture and add to the knowledge pool of this still new field of research.

[12]ruby-lang.org

[13]gitlab.com/unlimited-rulebook/ur-ruby

VII. Conclusions

Game mechanics are a fundamental part of game design; but not all mechanics are equally easy to implement and, more importantly, easy to reuse across games. Economy mechanics are unpredictable, unstable, and complex; thus, we designed, evaluated and proposed the Unlimited Rulebook reference architecture to reduce the costs of writing and maintaining economy mechanics through knowledge reuse as well. Our methodology followed the ProSA-RA process to guarantee the efficacy of our reference architecture, which involved a thorough analysis of architectural requirements and existing solutions and empirical evaluations of our architecture.

We found that many practices and architectural patterns are already used, especially in economy-centered games such as role-playing games, roguelikes, strategy games, and card games. Patterns like the Game Loop, MVC, and State help understand the role of economy subsystems in game applications, while ECS, AOM, and predicate dispatching provide flexibility and extensibility for economy mechanics to grow while reducing development costs. We used all these state-of-the-art and state-of-the-practice techniques in the design of the Unlimited Rulebook architecture.

Though our empirical evaluations are still incomplete, our preliminary results show that the URB architecture reduces the amount of changes needed to extend the economy mechanics of games. Our proofs-of-concept evaluations demonstrate how the architecture simplifies the implementation of complex interaction between economy mechanics, such as mechanics that change dynamically at runtime. By focusing on dynamic rules that isolate specific game interactions from other aspects of the economy, our proposal streamlines how changes are made and minimizes the impact they have on previous changes. The URB architecture changes the way developers and architects think about the implementation of economy mechanics, making it an innovative reference architecture that aims at making economy-centered games more flexible, more scalable, and cheaper to develop and maintain.

Acknowledgment

This research is supported by the São Paulo Research Foundation, FAPESP proc. 2017/18359-6.

References

[1] R. Hunicke, M. LeBlanc, and R. Zubek, "MDA: A formal approach to game design and game research," in *Proc. of the AAAI Workshop on Challenges in Games AI*, San Jose, CA, Jul. 2004, p. 1722.

[2] J. Schell, *The Art of Game Design: a Book of Lenses, 2nd Edition*. Boca Raton, FL: CRC Press, 2014.

[3] E. Adams and J. Dormans, *Game Mechanics: Advanced Game Design*. Berkeley, CA: New Riders Games, 2012.

[4] J. Dormans, "Engineering emergence – applied theory for game design," Ph.D. dissertation, Univ. of Amsterdam, Amstardam, Jan. 2012. [Online]. Available: https://dare.uva.nl/search?identifier=40b1a42a-4291-48a3-80a1-c85dfe927f50

[5] J. Gregory, *Game Engine Architecture*. Boca Raton, FL: CRC Press, 2014.

[6] W. K. Mizutani and F. Kon, "Toward a reference architecture for economy mechanics in digital games," in *Proc. of the Brazilian Symposium on Games and Digital Entertainment (SBGames 2019)*, Rio de Janeiro, Brazil, 2019.

[7] M. Torgersen, "The expression problem revisited," in *European Conference on Object-Oriented Programming (ECOOP2 2004)*, M. Odersky, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 123–146.

[8] A. I. Wang and N. Nordmark, "Software Architectures and the Creative Processes in Game Development," in *International Conference on Entertainment Computing*, Trondheim, Norway, Dec. 2015.

[9] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Vancouver, Canada: Addison Wesley, 2003.

[10] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. Oquendo, "Consolidating a process for the design, representation, and evaluation of reference architecture," in *Proc. of Working IEEE/IFIP Conference on Software Architecture (WICSA 2014)*, Sydney, Australia, Apr. 2014.

[11] W. K. Mizutani, V. K. Daros, and F. Kon, "Software architecture for digital game mechanics: Systematic literature review," unpublished.

[12] L. B. Morelli and E. Y. Nakagawa, "A panorama of software architectures in game development," in *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, ser. SEKE'2011, Eden Roc Renaissance, Miami Beach, USA, July 2011.

[13] 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, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2010.05.004

[14] W. Scacchi, "Practices and Technologies in Computer Game Software Engineering," *IEEE Software*, vol. 34, no. 1, pp. 110—-116, 2017.

[15] M. Zhu, A. I. Wang, and H. Guo, "From 101 to nnn: A review and a classification of computer game architectures," *Multimedia Systems*, vol. 19, no. 3, pp. 183–197, Jun. 2013.

[16] M. Zhu, "Model-driven game development addressing architectural diversity and game engine-integration," Ph.D. dissertation, Norwegian University of Science and Technology, Trondheim, Norway, Mar. 2014. [Online]. Available: https://pdfs.semanticscholar.org/195e/55c90fd109642116ee51f7205c106f341111.pdf

[17] A. BinSubaih, S. Maddock, and D. Romano, "A survey of 'game' portability," University of Sheffield, Sheffield, UK, Tech. Rep. CS-07-05, May 2007.

[18] G. E. Krasner and S. T. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[19] T. Olsson, D. Toll, A. Wingkvist, and M. Ericsson, "Evolution and evaluation of the model-view-controller architecture in games," in *Proc. of the International ACM/IEEE Workshop on Games and Software Engineering (GAS 15)*, Florence, Italy, May 2015.

[20] V.Sarinho, G. D. Azevedo, and F. Boaventura, "AsKME: A feature-based approach to develop multiplatform quiz games," in *Proc. of the Brazilian Symposium on Games and Digital Entertainment (SBGames 2018)*, Foz do Iguaçu, Brazil, 2018.

[21] S. Rabin, *The Magic of Data-Driven Design*. Newton, MA: Charles River Media, 2000, ch. 1.

[22] R. Nystrom, *Game Programming Patterns*. Genever Benning, 2014.

[23] ——. (2018) Is there more to game architecture than ECS? Event talk. Roguelike Celebration 2018. San Francisco, CA. [Online]. Available: https://www.youtube.com/watch?v=JxI3Eu5DPwE

[24] D. Llansó, M. A. Gómez-Martín, P. P. Gómez-Martín, and P. A. González-Calero, "Explicit domain modelling in video games," *Proceedings of the 6th International Conference on the Foundations of Digital Games, FDG 2011*, pp. 99–106, 2011.

[25] A. Plotkin. (2009) Rule-based programming in interactive fiction. Event talk. [Online]. Available: https://eblong.com/zarf/essays/rule-based-if/

[26] M. Zamith, L. Valente, B. Feijo, and E. Clua, "Game loop model properties and characteristics on multi-core CPU and GPU games," pp. 100–109, 2016.

[27] C. West. (2018) Using rust for game development. Event talk. RustConf 2018. Portlan, OR. [Online]. Available: https://kyren.github.io/2018/09/14/rustconf-talk.html

[28] E. Gamma, J. Vlissides, R. Helm, and R. Johnson, *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley, 1995.

[29] J. W. Yoder and R. Johnson, "The Adaptive Object-Model Architectural Style," in *Working Conference on Software Architecture (WICSA 2002)*, Montreal, Canada, Aug. 2002, pp. 3–27.

[30] M. Ernst, C. Kaplan, and C. Chambers, "Predicate dispatching: A unified theory of dispatch," in *European Conference on Object-Oriented Programming (ECOOP 98)*, Jul. 1998, pp. 186–211.

[31] D. T. Campbell and J. C. Stanley, *Handbook of research on teaching*, 1963, ch. Experimental and Quasi-Experimental Designs for Research, pp. 1–76.