Toward a Reference Architecture for Economy Mechanics in Digital Games

Wilson Kazuo Mizutani and Fabio Kon
Computer Science Department
Mathematics and Statistics Institute from University of São Paulo
São Paulo, Brazil
{kazuo,kon}@ime.usp.br

***Abstract*—Economy mechanics in computer games provide unpredictable, unstable, and complex requirements to software design. Thus, a single, generic software cannot be reused for different games. Instead, reference architectures grant knowledge reuse, a more abstract but flexible approach. In our ongoing research, we are systematically designing and evaluating a reference architecture for economy mechanics in games. This study presents the findings we have gathered in this particular domain, as well as the current state of the reference architecture and its preliminary evaluations.**

***Keywords*-computer games; software architecture; economy mechanics**

I. Introduction

Software reuse is a common practice to reduce production costs in expensive processes such as game development. Game engines, middleware tools, libraries, and data-driven design are examples of this practice [1]–[3]. However, not all parts implemented in games can be easily reused.

Game mechanics, in particular, can be divided according to the system requirements they share in common. Based on Adams and Dormans' classification [**?**], we found three main groups of mechanics with distinct implementation requirements: physics (bodies, shapes, and movement), internal economy (interactions between currencies, life, score, and any other in-game resource), and progression mechanisms (stages, win conditions, story flags, etc.). Of these three types of mechanics, internal economy is the hardest to reuse, as we explain next.

Since physics mechanics are based on the physics of the real world, their implementation specifications do not vary much from game to game, so a single physics library can be reused in multiple games. *Havok*[1] and *Bullet*[2] are examples of this. Progression mechanisms also benefit directly from software reuse through the practice of data-driven design [1], [2], making dialogues, goal conditions, stage geometries, and other game content be stored in files outside the source code of the game. Engines such as *Unity*[3] and *Godot*[4] support this in the form of editable scenes representing stages of the game. Economy mechanics, on the other hand, vary widely from game to game and, sometimes, even inside the same game. The economy mechanics of *Hearthstone* (*Blizzard Entertainment*, 2014), for instance, have practically no intersection with the economy mechanics of *Factorio* (*Wube Software*, 2012), though both titles are economy-focused games.

We identified three core challenges in the development of economy mechanics that reduce the benefits of software reuse. First, economy mechanics are unpredictable, because they do not try to simulate realistically the real world like physics mechanics do. Second, they are unstable, i.e., their specifications change constantly during the iterative design process of game development [5]. Third, economy-focused games are designed with considerably complex mechanics, including economy mechanics that structurally change other economy mechanics at runtime. For instance, rule-changing cards are one of the most popular features of *Hearthstone*.

In our research, we are studying how the discipline of *software architecture* [6] can provide *knowledge reuse* in the context of internal economy mechanics in games, since *software reuse* is a less practical approach. For that, we are designing a *reference architecture* [7]. Its objective is to minimize the development cost incurred by the three challenges described above, i.e. the unpredictability, instability, and complexity of economy mechanics.

This text is organized as follows. Section II describes our research methodology. Section III analyzes the architectural requirements for our reference architecture. In Section IV, we present the current state of the reference architecture as well as the results of our initial validations. Lastly, Section V discusses future work of this research.

II. Methodology

The process we are using to design and evaluate our reference architecture is an extension of the the ProSA-RA process developed by Nakagawa *et al.* [8]. We cycle through four steps, iteratively. First, we investigate information sources such as the literature and domain expertise. Second, we analyze the architectural practices for economy

[1] havok.com
[2] pybullet.org/wordpress
[3] unity.com
[4] godotengine.org

mechanics to extract and update the architectural requirements and domain concepts our reference architecture should satisfy. Third, we synthesize and improve the reference architecture using the findings of the previous step. Fourth, we evaluate the current state of the reference architecture to find out how well it fulfills the objectives of the research. Then we go back to the first step to keep iterating over the design.

Our information sources include related work from the literature, knowledge from industry authors, expert knowledge from active developers, recurrent solutions from commercial implementations of games and engines, and reference models from different sources. To investigate the literature, we performed a systematic literature review, which we submitted to a major journal on computer entertainment. For expert knowledge, we consulted a student special interest group from a prestigious public university, which also provided us with contacts to Brazilian studios whose developers we plan to invite for structured interviews.

III. Domain Analysis

From the domain knowledge the information sources provided us, we found seventeen architectural requirements for our reference architecture so far. Following the ProSA-RA process, we group these requirements into *domain concepts* [8] that better reflect the roles of the reference architecture in the development of economy mechanics. Table I presents this relation, and we briefly explain the requirements in the rest of this section.

Table I
Architectural requirements and domain concepts

Code	Architectural Requirement	Domain Concept
(R_1)	Decouple interaction code from simulation code	*Game Loop* integration
(R_2)	Per-frame updates of the simulation state economy	*Game Loop* integration
(R_3)	Synchronize player actions with simulation time	*Game Loop* integration
(R_4)	Decouple simulation time from execution time and user interaction	Simulation timeline
(R_5)	Support both real-time and turn-based simulations	Simulation timeline
(R_6)	Economy resources in the object model	Object model
(R_7)	Economy resources modifiers in the object model	Object model
(R_8)	Simulation of triggered behaviors	Behavior model
(R_9)	Simulation of action behaviors	Behavior model
(R_{10})	Simulation of behavior modifiers	Behavior model
(R_{11})	Short and incremental development iterations	Iterative development
(R_{12})	Extensible object model	Iterative development
(R_{13})	Flexible object model	Iterative development
(R_{14})	Extensible behavior model	Iterative development
(R_{15})	Flexible behavior model	Iterative development
(R_{16})	Data-driven object specifications	Data-driven design
(R_{17})	Data-driven behavior specifications	Data-driven design

Every computer game relies on the *Game Loop* pattern to alternate the execution flow between user interaction and simulation [1], [9]. This imposes a series of restrictions on the architecture of the game, especially regarding the control of execution flow. Requirements (R_1), (R_2), and (R_3) reflect how these restrictions should be handled by the architecture of the economy simulation subsystem. At the same time, the economy simulation has a *timeline* that dictates the sequence of changes to the state of the economy. This timeline is independent of the timeline following the *Game Loop* and fits into a spectrum that ranges from real-time simulations to turn-based simulations. The reference architecture contemplates this through requirements (R_4) and (R_5).

The simulation of economy mechanics has a state that changes over time. That state is composed of the individual states of each resource in the game economy. These resources are grouped and organized into objects for implementation purposes and the available types of objects are dictated by the *object model* of the simulation. Resources might be transferred from object to object, or they might have their state modified by the state of other resources [10]. Requirements (R_6) and (R_7) regard these issues. The changes incurred in the state of the simulation comprise a sequence of *behaviors* invoked by the simulation timeline when appropriate. The nature of the timing of the behavior determines whether it is a triggered behavior (invoked when a certain event occurs in the simulation) or an action behavior (invoked as an action by something outside the simulation, e.g., the player). Like simulation objects, simulation behaviors are limited by a *behavior model* and may also be modified by the state of resources. These conditions are all contained within requirements (R_8), (R_9), and (R_{10}).

The game development process is often based on trial-and-error approaches to determine what makes it fun and engaging for users [5]. To keep the software design from eroding due to the constant change of specifications, its architecture needs to support *iterative development*. In particular, extending or changing the object model and the behavior model of the simulation should cost the least possible to the team. That is what requirements (R_{11}), (R_{12}), (R_{13}), (R_{14}), and (R_{15}) cover. There is also another technique used to reduce this cost, called *data-driven design*, where object and behavior specifications (in the case of economy mechanics) can be loaded at runtime from data stored in the file system, outside the game executable [1], [2]. The last requirements, (R_{16}) and (R_{17}), bring this aspect to the reference architecture.

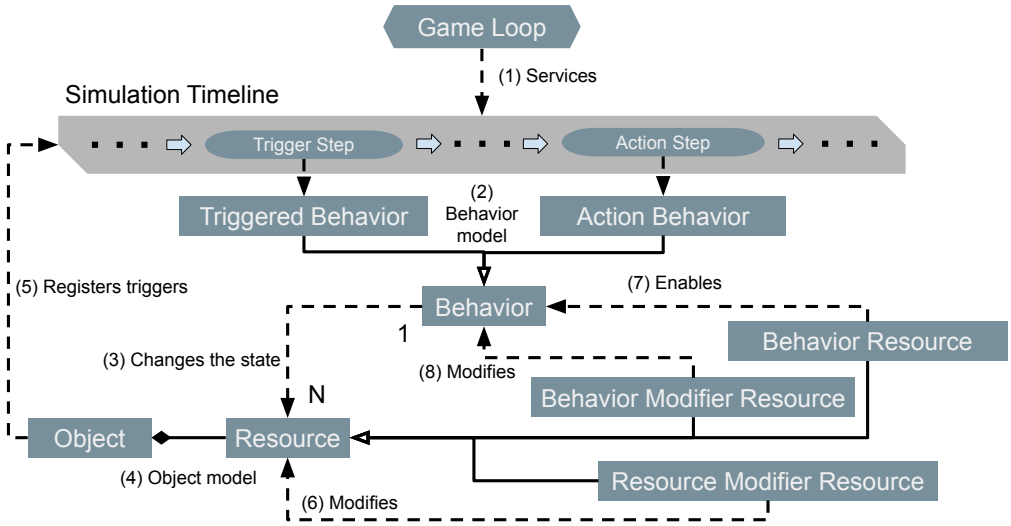

Figure 1. Conceptual view of the reference architecture.

IV. Preliminary Results

There are two main results our research yielded until now. First, we have reached a work-in-progress but functional stage of the reference architecture. Second, we partially evaluated the current state of the reference architecture with case studies and a quasi-experiment pilot. These results are described in the following sections.

A. Current State of the Reference Architecture

Based on the domain concepts and architectural requirements, we are designing the reference architecture and documenting it using a number of *views*, i.e. diagrams that represent different architectural perspectives. Due to space limitations, here we only show the *conceptual view* (Figure 1) and the *module view* (Figure 2).

In the conceptual view, we illustrate the relations between four out of six of the domain concepts (*Game loop integration*, *Simulation timeline*, *Object model*, and *Behavior model*). The *Game Loop* interacts (1) with the simulation timeline, which invokes behaviors from the behavior model (2). Each behavior changes the state of the simulation resources (3) organized according to the object model (4). These changes might trigger further behaviors, so they have to be registered in the timeline for future reference (5). Among resources, there are three special types that deserve special attention: resource modifier resources, which change the perceived state of other resources (6); behavior resources, which enable one or more behaviors in the simulation (7); and behavior modifier resources, which change the nature of existing behaviors (8).

In the module view, we present the division of responsibilities among modules of the reference architecture. The *Game Loop* requests an update of the state of the economy simulation subsystem every frame through the *timeline tracker* module. It runs the simulation from the step where it stopped the previous frame, activating behaviors in the *behavior engine*. This engine changes the state of the simulation by manipulating the resources kept in the *object pool*, which in turn register triggers back into the timeline tracker. Other subsystems of the game (e.g. graphics or sound) may query the state of resources in the pool at any time the economy subsystem is not under control of the execution flow.

The *behavior model* determines what are the possible operations the behavior engine can apply on the object pool through its behaviors. At the same time, the *object model* dictates the possible types and states of resources in the object pool. Both these models can be either source code definitions (such as classes) or runtime structures. For instance, resource types can be determined by having each resource reference a "resource type" instance from the object model, which is a variation of the *Adaptive Object-Model* pattern [11] also known as the *Type Object* pattern [9]. Both the behavior model and the object model are modules designed to grow and change with reduced cost for developers to satisfy the *Iterative development* concept.

Lastly, to meet the *Data-driven design* concept, the specifications of behaviors and resources should be loaded from file system data, which is the responsibility of the *Content Loader* module. To further reduce development costs, the behavior model and the object model could also be optionally loaded from the file system.

B. Architectural Evaluations

The first part of evaluating our reference architecture consists of applying it to games in development and studying the benefits and costs of doing so. We are carrying two of these case studies at the moment, and we plan to perform two more in the future. The first case study involves a

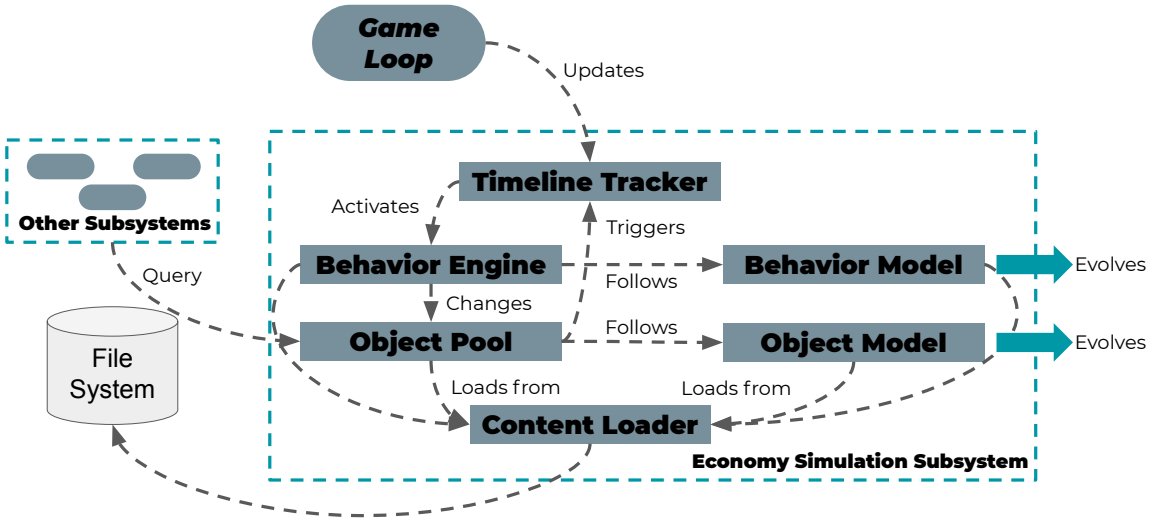

Figure 2. Module view of the reference architecture.

deck-building[5] rogue-like[6]. The study demonstrated that the current reference architecture can handle games with the very unpredictable and unstable economy mechanics of a card game. The second regards a base-building survival rogue-like. Its study verified that the object model of our reference architecture is indeed flexible and extensible.

The second part of the architectural evaluation consists of quasi-experiments [12] to measure how much the reference architecture reduces development costs for student projects in game programming courses. We performed a quasi-experiment pilot with students from a Summer course in a prestigious public university, which shows positive results but points out that the learning curve of the reference architecture is too steep for beginning programmers. We will carry out a complete version of the quasi-experiment during an undergraduate and graduate course at the same university during the second term of 2019.

V. Future Work

The next steps of our research involve performing structured interviews with active professional game developers to add more domain knowledge to our analysis, finishing the design of the reference architecture, then proceeding with its evaluations. Two more case studies will be developed, and we will collect more data on the performance of the architecture during our quasi-experiment. The software architecture and an associated reference implementation will be released as open source software.

Acknowledgment

This work is supported by the São Paulo State Research Support Foundation (FAPESP) under Grant No.: 2017/18359-6.

[5]en.wikipedia.org/wiki/Deck-building_game
[6]en.wikipedia.org/wiki/Roguelike

References

[1] J. Gregory, “Game engine architecture,” 3rd ed. Boca Raton, USA: CRC Press, 2017.

[2] S. Rabin, “The magic of data-driven design,” in Game programming gems, M. DeLoura, Ed. Newton, USA: Charles River Media, 2000, pp. 3–7.

[3] W. Scacchi, “Practices and technologies in computer game software engineering,” IEEE Software, vol. 34, Jan. 2017, pp. 110–116.

[4] E. Adams and J. Dormans, “Game Mechanics: Advanced Game Design”. San Francisco, USA: New Riders, 20.4.

[5] J. Schell, “The art of game design: a book of lenses,” 2nd edition. Boca Raton, USA: CRC Press, 2014.

[6] M. Shaw and D. Garlan, “Software Architecture: Perspectives on an Emerging Discipline”. New Jersey, USA: Prentice Hall, 1996.

[7] E. Y. Nakagawa, P. Oliveira, and A. M. Becker, “Reference Architecture and Product Line Architecture: A Comparison,” Proc. European Conference on Software Architecture (ECSA 11), Sep. 2011, pp. 2–5.

[8] 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,” Proc. Working IEEE/IFIP Conference on Software Architecture (WICSA 14), Apr. 2014, 143–152.

[9] R. Nystrom, “Game programming patterns”. Genever Benning, 2014.

[10] J. Dormans, “Engineering Emergence: Applied Theory for Game Design”. PhD Thesis, University of Amsterdam, 2012.

[11] J. Yoder and R. Johnson. “The Adaptive Object-Model Architectural Style”. Proc. Working Conference on Software Architecture (WICSA), Aug. 2002, pp; 3–27.

[12] D. T. Campbell and J. C. Stanley, “Experimental and Quasi-Experimental Designs for Research,” in Handbook of research on teaching. Boston, USA: Houghton Mifflin Company, pages 1–71, 1963.