Entertainment Computing 38 (2021) 100421

Contents lists available at ScienceDirect

Entertainment Computing

journal homepage: www.elsevier.com/locate/entcom

Software architecture for digital game mechanics: A systematic literature review

Wilson K. Mizutani [a,*], Vinícius K. Daros [b], Fabio Kon [a]

[a] Department of Computer Science, University of São Paulo, R. do Matão 1010, São Paulo, Brazil
[b] Wildlife – R. Dr. Renato Paes de Barros 1017, Conj. 81, São Paulo, Brazil

ARTICLE INFO

Keywords:
Systematic literature review
Digital games
Software architecture

ABSTRACT

Game mechanics, the rules that simulate the virtual world inside a game, take a great part in what makes a game unique. For *digital* games, this uniqueness reduces the opportunity for software reuse. A high-level software architecture for game mechanics, however, can still be reused where a single, specific implementation cannot. Despite that potential, existing research on game development lacks a comprehensive analysis of how game mechanics could benefit from the field of software architecture. This limits the opportunities for developers and researchers alike to benefit from findings on the subject. To help guide future research on game development, we analyzed the state-of-the-art architectures in game mechanics through a systematic literature review. This work carefully documents data from 36 studies, analyzing the reflections and compromises between design requirements, practices, and restrictions, as well as how they contribute to different types of mechanics. The main findings are that researchers favor reduced development complexity, but often tailor their solutions to specific games or genres. We conclude that a valuable avenue for future research in the field is the generalization of architectural solutions around specific types of mechanics and formalizing the use of software engineering for game mechanics.

1. Introduction

Digital games are pieces of software developed for the entertainment of users. From the perspective of computer science, these applications are *real-time interactive computer simulations* most of the time [1]. On one side, they simulate a virtual world with an initial state, a number of state-changing rules, and possible end states. On the other side, users interact with that simulation by perceiving its real-time state while also simultaneously affecting it through real-time input devices. For instance, computer graphics studies how to render virtual worlds onto displays, allowing the user to visually read the state of the game and, thus, perform an informed interaction with it. Physics engines such as Bullet[1] or Havok,[2] on the other hand, are tools developed to simulate the laws of physics inside the game.

However, while interaction is often restricted by the peripherals available to computers, simulation is only limited by the computing resources it requires. In practice, this means that games can differ widely one to the other in terms of the nature of the virtual world they simulate, as the industry has demonstrated over the many decades since digital games came to be. It is, arguably, part of the intrinsic value of games. Some might focus on real-time action dynamics where the player is tested for their coordination and reflexes, while other games prefer simulations centered on strategic or puzzle-solving thinking. Compared to computer graphics, where there are algorithms and structures known to provide the interaction features required for most games – to the point where most successful game engines support them – there are no standard or completely generic implementations for simulating the virtual worlds of games. Instead, every game at least partly implements its own simulation. Given the variety of possible virtual worlds, unless some of their aspects are considerably common – such as conventional physics simulation – individual simulation implementations offer little reuse to developers in general.

On the other hand, knowledge about how to design software solutions in a specific domain – such as simulation in games – *can* be reused [1,2]. For that, developers rely on the discipline of software architecture, which allows them to think "*in terms of computational components*

* Corresponding author.
E-mail addresses: kazuo@ime.usp.br (W.K. Mizutani), vinicius.daros@wildlifestudios.com (V. K. Daros), kon@ime.usp.br (F. Kon).

[1] pybullet.org/wordpress.
[2] www.havok.com.

https://doi.org/10.1016/j.entcom.2021.100421
Received 4 June 2019; Received in revised form 23 April 2020; Accepted 20 February 2021
Available online 4 March 2021

and interactions among those components" [3], instead of in terms of specific implementations. Knowledge of software architecture can take different forms. For instance, *data-driven design* is an architectural *practice* where developers implement their games without specifying all of its simulation parameters through code, providing them as data inputs to the runtime application instead [4]. *Entity-Component-System*, on the other hand, is an architectural *pattern* where the properties of game entities – both inside and outside the simulation – are determined by the aggregation of objects (components) as opposed to class inheritance [2,5]. Both are examples of reusable solutions in the architecture level of implementing simulation in games.

In this systematic literature review, we analyze the state-of-the-art of software architecture in the context of digital game mechanics – a concept in the development of games that is closely tied to their virtual worlds and how they are simulated. We explain the relation between them in Section 1.1. After that, Section 1.2 explains how this review contributes to the field of software architecture in digital game mechanics and to game development in general. Section 1.3 shows how game mechanics fit into the software architecture of games. With that, Section 1.4 states the formal goals of this review. Finally, Section 1.5 describes the organization of the remaining sections of this paper, which comprise the review itself.

1.1. Game mechanics

There is no consensus on what "game mechanics" means, despite its widespread use by the game community. Sicart [6] defines them as "methods invoked by agents, designed for interaction with the game state" and Järvinen [7, page 254] as "means to guide the player into particular behavior by constraining the space of possible plans to attain goals". Both these definitions consider mechanics as affordances available to players (or AI agents) when interacting with the virtual world simulated in the game. Other authors with similar approaches to this notion of game mechanics include Osborn et al. [8] and Dubbelman [9].

These definitions contrast, for instance, with the definition given by Schell, where mechanics "are the procedures and rules" of the game [10, page 51], Adams and Dormans' definition that "game mechanics are the rules, processes, and data at the heart of a game" [11, page 1], and the definition used by Hunicke et al., where mechanics "describe the particular components of the game, at the level of data representation and algorithms" [12]. These last definitions assume that mechanics cover the general inner workings of the virtual worlds simulated in games, whether these rules afford "outside" interaction or not. Other definitions that follow this general approach include the work from Larsen and Schoenau-Fog [13].

A key distinction between these definitions is that Sicart's and Järvinen's definitions allow game input – an interaction feature – to be part of the mechanics while the definitions used in the works of Schell, Adams and Dormans, and Hunicke et al. frame the scope of mechanics predominantly around the simulation features of games. In fact, Sicart and Järvinen argue that the rules of a game simulation should be clearly distinguished from game mechanics. That approach has many benefits in the design process of games – especially due to the formality Sicart and Järvinen achieve – but, as software architects, we favor definitions that tie mechanics to simulation rules for the following reasons.

As we mentioned in the beginning of Section 1, games are software applications designed to entertain users. Schell explains how not only the mechanics, but also the technology, the aesthetics, and the story of a game all equally contribute to the final player experience and each "powerfully influences each of the others" [10, pages 53]. From the perspective of this *elemental tetrad* – as Schell calls it – the piece of software composing the digital copy of a game is part of the technology element. It instrumentalizes the mechanics, the aesthetics and the story of the game through its simulation and interaction features.

This is a useful framework because, as software architects, we can now understand that each component of the game software promotes one or more of the tetrad elements to produce the intended experience. For instance, an interaction feature such as rendering the game state onto a digital display explicitly exposes both the aesthetics and the story of the game to the user. When a programmer implements the rendering pipeline behind this feature, they must make sure that the avatars of characters and that the text of their speeches are properly conveyed to the users. That is, the elemental tetrad provides us a lens to evaluate the impact of the implementation of a given software component over each other tetrad element and, therefore, the game experience.

In Schell's definition, mechanics are the processes and rules of the game and what "make[s] a game a game" [10, page 52] compared to more linear entertainment experiences such as books or movies. Computationally speaking, what mainly enables game software as non-linear experiences is the fact that its internal simulation is a state machine, allowing multiple state "paths" from the initial state to one of the many end states. Adams and Dormans support this notion when they claim that "rules define games" [11, page 1] based on further definitions from the literature. They also argue that a key aspect of what makes games fun is that their outcome is unpredictable due to, among other reasons, the presence of complex rules. Adams, in a different work, further claims that game designers turn "the general rules of the game into a symbolic and mathematical model that can be implemented algorithmically" and that this model consists of the game mechanics [14, page 35]. In this characterization of game mechanics, the simulation features of a game are the ones mainly responsible for computing mechanics and handling mechanical complexities, which in turn makes us more explicitly aware of how these software features contribute to the game experience.

As such, we have seen that there are many definitions and interpretations to what "game mechanics" means in game design or game programming. However, we find that equating game mechanics to simulation rules is a more useful convention for studying the software architecture of games because it exposes the role of the software components that simulate the virtual world of a game – the role of implementing the rules that make a game a game. Furthermore, as we will see in Section 1.2, the field of software architecture in games would benefit from a standard terminology for game mechanics, which we believe the following definition provides. Thus, in this systematic literature review, we define game mechanics as *the set of states of a game virtual world simulation, the definition of the initial and end states, and the corresponding state-changing rules*. This review presents the state-of-the-art of the software architecture of digital game mechanics.

1.2. Motivation

Game developers and researchers have used software architecture to improve knowledge reuse in the implementation of mechanics in games for years. Several publications study this practice in the industry, mostly from developers describing the solutions they used in their games [15–17]. It is also often present when developers discuss broader topics of software architecture in games [1,2,5,4]. Although most do not adopt open source licenses, many game engines make at least part of their source code available on-line – such as *Godot*,[3] *CRYENGINE*,[4] *Unity3D*,[5] *Lumberyard*,[6] among others – widely providing reusable knowledge about game architecture.

However, we see that, outside of physics mechanics, general-purpose game engines do not offer any precise architecture for implementing the mechanics of games developed with them. In fact, more genre-specific engines such as *RPG Maker*[7] do provide reusable frameworks for the

[3] github.com/godotengine/godot.
[4] github.com/CRYTEK/CRYENGINE.
[5] github.com/Unity-Technologies/UnityCsReference.
[6] github.com/aws/lumberyard.
[7] rpgmakerweb.com.

commonly expected mechanics of role-playing games, such as combat, character progression, inventory management, etc. We consider this an evidence that architectural knowledge, including when and how to apply it, promotes the reuse of past efforts in the particularly open-ended implementations of mechanics in games.

Notwithstanding, academic publications, differently from the entertainment- and commercial-oriented goals of game companies, have other reasons to turn to digital games. Edutainment, pervasive games and artificial intelligence are common examples of fields that study digital games but do not necessarily share interests – and thus, experience – with the industry. Because of this diversity, depending on the background, authors unknowingly use different terms to refer to the mechanics of digital game architectures. For instance, they call it the "domain model" [18], "game logic" [19], "game rules" [20], and even "G-factor" [21]. Sometimes, the terms cover wider or stricter aspects of the definition for "mechanics" we proposed in Section 1.1.

This lack of standard terminology among computer science researchers prevents authors from reliably finding previous works for reference, which in turn may lead to duplicate efforts and redundant studies. This also makes unclear what the current advances in the software architecture of digital game mechanics are. Thus, future research in this area – which improves software reuse in front of the endless design space of games for both developers and researchers – requires first a structured understanding of the state-of-the-art to reconcile its diverse backgrounds and motivations. In other words, despite the numerous contributions to the study of software architectures of digital game mechanics, it is just not clear how much is covered compared to the industry or if there are any organized intents of doing so and how they are characterized.

1.3. Software architecture in games

To study the architecture of game mechanics it is useful to first determine our expectations and assumptions over the general architecture of games. Though the diversity of platforms, genres, and scale of games is vast, there is one architectural pattern most, if not all, game software abide by: the *Game Loop* [1,2]. In this pattern, an endless loop is responsible for synchronizing the interaction and the simulation computations of a given game. By keeping track of user time and alternating between interaction queries and simulation steps at sufficiently high frequencies (usually 30 or 60 cycles per second), the *Game Loop* ensures that user input is translated into its corresponding mechanics as soon as possible and that what the user sees and hears in real-time corresponds to the most up-to-date state of the simulation.

The many interaction and simulation features of a game are usually spread across different subsystems [1, chapter 5] which the *Game Loop* services periodically. Each subsystem usually has a main routine for updating its corresponding part of the general game state (both simulation and interaction states) [2, chapter 10]. Despite our didactic separation between interaction and simulation features, game implementations do not necessarily divide them explicitly when assigning roles to each subsystem. However, software architecture studies on interactive software not only claim that isolating interaction modules from domain-specific modules is a good separation of concerns but also propose an architectural pattern known as the *Model-View-Controller* (MVC) [22,23] to formalize this architectural separation. This pattern assigns high-level control flow of an application to Controllers, which mediates user interface features in the View modules with domain-specific features in the Model modules.

In particular, Olsson et al. [24] argue that game development very much benefits from the use of this pattern. They highlight how a clean separation between the View (interaction) and the Model (simulation) improves the cross-platform support of a game (since the same game mechanics can be reused by simply changing the View modules) among other enhancements to software quality. Theoretically, then, game software uses the *Game Loop* pattern as the Controller in a Model-View-Controller architecture, dividing its subsystems between View subsystems and Model subsystems. When collecting data from the studies in this review, we are interested in the architecture used in "Model" subsystems. When such a division is not explicit, we investigate the architecture of any subsystem that computes at least part of the game mechanics.

We are also interested in what types of mechanics different architectures are more useful for. To measure that, we propose a categorization of game mechanics *from the perspective of their implementation characteristics*, though we use Adams and Dormans's [14,25] game-design-based (and non-exhaustive) classification as a starting point. We chose their classification since our definition of game mechanics is aligned with theirs. They divide game mechanics into physics, internal economy, progression mechanisms, tactical maneuvering, and social interaction.

Physics mechanics involve time and space-based mechanics in the virtual world of the game. They involve the simulation of space, bodies, and motion. This typically includes representing the position and shape of elements in the virtual world and simulating how they would physically collide if they were in the real world. However, not all games follow conventional physics, so this category includes movement and collision mechanics in discrete grid-based spaces, for instance. Computationally, these mechanics require specialized data structures and algorithms that allow efficient queries about simulation positions, collisions, and force integrations. In particular, spatial-partitioning techniques that allow finding simulation objects based on their in-game location are necessary for guaranteeing good performance in larger games. As mentioned before, "real-world" physics is so common in games that a number of software libraries and engines have been developed for physics reuse in games. As such, subsystems that implement these structures and algorithms are categorized as supporting *physics mechanics* in this review.

Internal economy, as Adams and Dormans explain [14,25], relate to the manipulation of virtual in-game resources such as currency, health, skills, items, cards, units, etc. A straightforward example is a store where the player's character can exchange a collectible currency for useful supplies, but economy mechanics cover a broader range of features. When a mage character spends magical energy to conjure a huge rock into existence, the conversion from energy into creation is part of the economy mechanics. When the defensive system of the player's spaceship is powered, the rules and calculations that determine that this act will cause the oxygen supply to stop working for a lack of power also belong to the economy mechanics. In card games, paying to play a card by discarding or tapping other cards is one of the many possible economy mechanics in the genre.

In terms of their implementation, economy mechanics involve more scalar data as compared to the predominantly vectorial data of physics. By exclusion from physics mechanics as well, economy mechanics do not have any locality by themselves, which means a procedure implementing economy mechanics can potentially access and write to any part of the economy state of the simulation unless there are physics mechanics that say otherwise. For instance, in a platform game, a player may only collect coins if they touch them, but a card in a card game might change the state of any other card in the entire board. Due to this contrast in the implementation between physics and economy, we consider internal economy a category of interest in our review.

Progression mechanisms are mechanics related to the progression of the simulation state of a game from start to finish, such as the transition between stages, story "flags" that indicate advances in the plot, win and loss conditions, etc. These mechanics operate over the "narrative state" of the game, usually moving the simulation towards a certain conclusion. A progression mechanism can range from a simple timed trigger that prevents the player from meeting a certain goal, to how the mood of non-playable characters toward the player affect which plotline the game will follow.

In terms of their implementation, progression mechanisms

complement physics and economy features by tying them to higher-level simulation states, such as what stage the player is currently on or how many points they need to beat a challenge and proceed with the game story. Progression mechanics are also often associated with the interaction modes of a game. In role-playing games, for instance, interaction modes alternate between world exploration (mostly physics-based) and combat resolution (mostly economy-based). The sequence of world sections players can explore and the battles they fight are part of the progression mechanisms as well. Many engines provide a "scene" abstraction to represent each interaction mode or progression state inside the simulation (stages), often supported by data-driven decoupling of scene data and engine code. As such, we consider progression mechanisms to be a relevant category for mechanics in our review under the revised name of "narrative progression".

Tactical maneuvering consists of mechanics that associate the placement of simulation entities with some form of in-game advantage or disadvantage. In a strategic war game, units might protect themselves better if they are within a mountain range, for instance. Computationally, this means that certain physical states cause changes in the economy of the game. In the war game example, entities within the area of the mountain range (a physics collision check) gain a boost to their defenses (a change to a combat resource, which is part of the economy). For this reason, our proposed categories do not include tactical maneuvering, which we argue *are implemented in terms of* direct interactions between physics and economy states of the simulation.

Social interaction mechanics make two or more players interact through the game. These interactions can be competitive or cooperative, such as player-versus-player challenges in on-line games or team coordination in real-time tactical games. Although the simulation implementation might be aware of what entities belong to players, it does not need to treat interactions with them any differently. As its name suggests, we believe that interaction features are what makes social interaction possible in games, be them through the sharing of a video game screen or by connecting multiple devices via the Internet. Thus, we do not consider social interaction a pertinent mechanics category for this review. This is one of the points where our definition of mechanics diverges from the one used by Adams and Dormans [14,25].

1.4. Objective

We discussed the importance of game mechanics in the development of entertaining experiences and how software architecture knowledge provides reuse that complements the reuse of actual implementations in that context. However, academic studies in this field have yet to uniformize their terminology and efforts, making unclear what the current state-of-the-art is and what research venues demand further investigation. In general terms, we aim to determine this state-of-the-art as well as the current challenges and research opportunities of software architecture solutions for digital game mechanics. More specifically, based on all uses of software architecture applied digital game mechanics across the reviewed studies, we want to analyze, present, and discuss:

1. what system requirements motivate them;
2. what types of game mechanics they support;
3. what are their design limitations; and
4. what future research they would benefit from.

1.5. Text organization

Section 2 describes works that are similar or related to our systematic literature review, mostly other literature surveys whose scope intersects with ours. Next, in Section 3, we detail the methodology used, presenting the review protocol we designed and followed, including our research questions. In Section 4, we present the data collected from the reviewed studies, an overall categorization, and quantitative results regarding each research question. Section 5 analyzes the results of the review, highlighting any relevant patterns in the characteristics and interactions between requirements, practices, restrictions, and types of mechanics among the reviewed studies. It also discusses the opportunities for future research we found from the analysis of the literature. Finally, Section 6 draws conclusions about the state-of-the-art of software architecture in digital game mechanics, based on the findings from previous sections.

2. Related work

Morelli and Nakagawa's systematic mapping [26] identifies the most common software architecture topics (reference architectures, frameworks, design patterns, etc.) in game development research, as well as the most commonly investigated game subsystems (graphics, audio, network, etc.). Different from our proposal, their study considered applications of software architecture in all areas of digital game development, while we focus on the more specific field of mechanics, excluding subsystems designed for either interaction features (graphics, audio, input), infrastructural features (networking, hardware, control), or artificial intelligence. In fact, the only mechanics-related subsystem Morelli and Nakagawa consider in their mapping is physics, while we also account for economy and narrative mechanics. Additionally, since our study is a systematic literature review instead of a systematic mapping, our research questions target more specific topics and the analysis of the studies goes into deeper considerations. However, since our field of interest is a subset of the field mapped in their study, we reuse some of their search and selection parameters, as described in Section 3.

Ampatzoglou and Stamelos' systematic literature review [27] studies the applications of software engineering to game development and determines the most commonly addressed topics and research methodologies in that field. As such, their study addresses an even broader field of which software architecture is only a part. In particular, only 3% of studies they reviewed proposed software architectures; also 3% of studies investigated software design in games, and 2% of them regarded software reuse. That is, Ampatzoglou and Stamelos' work motivates our study on software architecture in games since the topic is shown to be underexploited. Their review also characterizes the literature surrounding the field of software architecture of digital game mechanics. We borrow from their methodology too, as we explain in Section 3.

There are a few other secondary studies (studies about studies, such as literature reviews) regarding different aspects of software engineering in game development. Zhu et al.'s systematic literature review [28] investigates the network architecture of games using the *Game World Graph* framework to identify research opportunities. Shi and Hsu [29] present the state-of-the-art of interactive remote rendering systems, which is the research field behind cloud gaming. Yahyavi and Kemme [30] survey peer-to-peer architectures of massively multiplayer online

games. None of these studies specifically address the architecture of digital game mechanics.

3. Methodology

We conducted this study in the form of a systematic literature review (SLR) [31,32]. Like any other academic survey, SLRs aim to answer a set of research questions inside a given field of knowledge through the investigation of studies previously performed on it. Authors commonly refer to the latter as **primary studies** and, as **secondary study**, the review itself [31,26]. The characterizing feature of SLRs compared to surveys, in general, is the formal structure of the methodology, which reduces bias in the selection and analysis of primary studies while also promoting reproducibility of the review itself.

An SLR is performed in three phases: first, the authors design the protocol they will use in advance, and review it as necessary; second, they conduct the review itself, following the protocol established in the previous step; third, they report the results of the study [31,32]. The foundation of an SLR, which guides all of its phases, is the research questions it addresses.

The protocol makes clear the process that led the authors to the results and conclusions they claim. It typically contains (1) the research questions, (2) a study selection strategy, and (3) a data extraction method. Additional steps are included according to the needs of a particular SLR. In our work, the protocol designed during the planning phase is composed of the following steps:

(1) Research Questions – Section 3.1;
(2) Search Strategy – Section 3.2;
(3) Selection Criteria – Section 3.3;
(4) Fitness Assessment – Section 3.4; and
(5) Data Extraction – Section 3.5.

The particular protocol used in this review adapts Kitchenham and Charters' technical report on systematic reviews for software engineering [31] – originally aimed at reviews of formally empirical studies – to the specific context of software architecture. For our review, we also broadened the analysis to consider studies that propose architectures with varying degrees of validation, not only those with formal verifications. To compensate for this, we include an assessment of the level of validation of the reviewed studies – step (4) above – as part of the data extraction and analysis.

3.1. Research questions

Since the aim of this study is to characterize the relation between software architecture and the implementation of digital game mechanics in academic research, our review plan required the following research questions to be addressed:

RQ_1. What software design challenges do researchers face when implementing game mechanics?
RQ_2. What software architecture practices and patterns do researchers use for the implementation of game mechanics?
RQ_3. What types of mechanics most often require, in research, the use of architectural practices and patterns, and which are they?

Software architecture is a discipline researchers rely on to understand and express the "*elements from which systems are built, interactions among these elements, patterns that guide their composition, and restraints on those patterns*" [3]. The software design of a system might target one or more of these structural qualities for a number of practical reasons. The purpose of RQ_1 is to identify what these reasons are in the context of digital game mechanics, making explicit the areas that require the most attention from software architecture research. RQ_2 complements that information by inquiring what software architecture researchers use to develop game mechanics. That provides us with an understanding of what is the state-of-the-art of this field of knowledge, as well as what are the predominant approaches and their eventual shortcomings. Finally, RQ_3 aims to determine what is the demand for software architecture solutions from each category of game mechanics – i.e., physics, progression, and economy mechanics (as per Section 3.5 – as a means of measuring the impact further research could have on each of them.

3.2. Search strategy

To select primary studies, an SLR must first determine one or more sources – i.e. digital libraries – from which to pool those studies, and what search terms and filters to apply [31]. However, since the subject of game mechanics is quite specific, we found that wording the search terms too precisely resulted in a very limited set of studies. We attribute this to the problem of lack of terminology conventions in game-related research discussed in Section 1.2, which means many studies do regard game mechanics but use different names to refer to them, or do not name them at all. Because of this, we opted for broader search terms and delegated the responsibility of removing irrelevant studies to the inclusion and exclusion criteria (Section 3.3).

The final list of primary study sources consisted of digital libraries considered dominant publishers in the field of entertainment computing, and was based on the list used in Ampatzoglou and Stamelos' systematic literature review [27] (from Section 2): ACM Digital Library (with default parameters), IEEE Xplore (with default parameters), and Springer Link (filtered for the "Computer Science" discipline).

As for the search terms, we chose the string used by Morelli and Nakagawa's systematic mapping [26] (from Section 2), which prioritized studies which regard both games and software architecture, but may or may not discuss the game mechanics: `("computer game" OR "video game" OR "digital game") AND "software architecture"`

3.3. Selection criteria

Once the starting pool of primary studies is defined, the next step in the protocol of an SLR specifies how to select relevant studies from among them. This is done by deciding on **inclusion and exclusion criteria** [31,32,27,26]: a primary study is selected only if it meets all inclusion criteria and does not match any exclusion criteria. We chose a single inclusion criterion, which effectively attests whether a study contributes to the discussion brought about by our research questions:

IC_1. The study describes the investigation of one or more software architecture practices or patterns for implementing gameplay mechanics in digital games.

As for exclusion, we elected five criteria:

EC_1. The study is not in English
EC_2. The study does not have an abstract
EC_3. The study is in abstract-only format
EC_4. The study is not peer-reviewed
EC_5. The study is a duplicate

EC_1 to EC_4 exclude studies that do not meet the minimum requirements for our SLR, usually because its publication is inaccessible to us or its contribution is not formally recognized by peers. EC_5 removes studies that are shorter or previous versions of other studies to avoid duplicate data. Finally, IC_1 detects whether a given study does not address our research questions. Even if the context of a study lies outside the field of software architecture, this inclusion criterion accepts the study if it contains relevant discussions around the software architecture of game mechanics.

3.4. Fitness assessment

Since we impose no restrictions on the methodologies the collected studies adopt, our protocol instead assigns them a *fitness assessment*. Its purpose is to measure how relevant each selected study is towards answering the research questions. For instance, studies that propose a certain software architecture but do not detail its design nor present a reference implementation receive a low fitness assessment since we cannot fully benefit from their contributions. That is, we wish to measure how helpful a study is for our research. Each study was thus assigned a *fitness assessment value* in the range $[0, 1]$ – 0 means the study has no substantial contributions towards answering the research questions and 1 means the study addresses everything we need to answer the research questions. To formalize the criteria used to assign a fitness assessment value, we designed five *fitness scores* (described ahead) whose values are chosen from $\{0, 0.25, 0.5, 0.75, 1\}$ for each study. The fitness assessment value of a study is thus computed by averaging the scores together. The fitness scores measure different aspects of how a study could contribute to our review. To reduce bias, we had two of the authors assign scores independently to selected studies, then discuss discrepancies and find a consensus, as recommended in the literature [31]. We list and explain each fitness score in the rest of this section.

3.4.1. Study documentation

This score measures the level of documentation in a study, that is, how accessible are its resulting artifacts. One parameter we use to establish a value for this score is whether studies provide access to source code and data produced in the authors' research. We also consider how clear the text presents any proposed architecture, in the sense of how much it enables the reader to reproduce the architecture on their own.

3.4.2. Discussion of architectural choices

This score indicates how much the authors of a study elaborate on the motivation for the software architectures they chose. We found that many of the selected studies are not about software architecture, but about game-related technologies or games themselves. In these cases, they might not fully explain why a certain architecture was used since it falls out of scope for their study.

3.4.3. Description of architecture design

This score points out how detailed are the descriptions of proposed software architectures in a study. Here, we are especially interested in understanding the design patterns used, as well as architecture diagrams and other illustrations that clarify the relations between different subsystems of game mechanics.

3.4.4. Presence of reference implementation

This score accounts for how much a study explains the process of turning its proposed design into practical implementation. This is usually done through a reference implementation, but any implementation that illustrates the use of an architecture counts for the purpose of this assessment.

3.4.5. Study validation

Finally, the last score measures the formality with which the authors validated their study. For instance, proposals which have been successfully used in case studies are considered to have a partial validation. Studies with quasi-experiments, experiments, or other formal empirical methods, rank a higher score.

3.5. Data extraction

After studies have been collected and selected, the next step towards answering the research questions is extracting relevant data from them. First, we need basic information such as venue and year of publication. Besides that, we need the fitness assessment value, which comes from the five fitness scores as explained in Section 3.4. Last and most importantly, we are interested in how each study addresses our research questions. To collect this information, we designed a form with seven *data fields* that we filled for each selected study. Every study is assigned a unique identifier and, as with the fitness assessment, we had two of the authors fill the forms separately, then converge the analysis later. The data fields in the form are:

F_1. Source venue (conference proceedings or journal)
F_2. Year of publication
F_3. Fitness assessment
F_4. Software Design requirements
F_5. Software architecture practices and patterns
F_6. Type of game mechanics supported
F_7. Design restrictions

F_1 to F_3 are simple metadata from the study (including the fitness scores and assessment value calculus). F_4 describes what structural qualities a study aimed for when proposing its software architecture (e.g. scalability, portability, etc.). F_5 enumerates practices the study used from the discipline of software architecture (e.g. design patterns, model-driven development, etc.) as well as architectural patterns. F_6 specifies what types of mechanics are supported by the proposed architecture of a study. Here we use the classification discussed in Section 1.4, i.e. an architecture supports any combination of physics, economy and progression mechanics, but not all three at the same time. When a study supports all three types of mechanics, we assign it a special code, as shown in Sections 4.1 and 4.4. Lastly, F_7 points out the shortcomings of the architecture proposed by the study.

For the qualitative fields F_4 to F_7, we used the coding method from

Table 1
Number of studies by source after each selection pass.

Source	Keyword search	Abstract pass	Full-text pass
ACM Library	39	16	9
IEEE Xplore	106	29	7
Springer Link	368	51	20
Total	512	96	36
Inclusion rate	100%	19%	7%

grounded theory [33] to represent the data: each of these fields is assigned a list of codes indicating all the corresponding topics, approaches, techniques, and issues that we identified in each study. A code assigned to a study may not necessarily appear explicitly on its text, because authors may rely on a concept without formally stating they did. For instance, an architecture found in a study can make evident use of the *Entity-Component-System* pattern even if its authors do not mention it, in which case we *will* assign the corresponding code.

4. Results

We performed the search strategy described in Section 3.2, collecting all matching studies as of April 4th, 2018, then later again on August 12th, 2019.[8] Then, using the inclusion and exclusion criteria from Section 3.3 we selected relevant studies. The selection comprised two passes, which one of the authors carried out. During the first pass, we applied the criteria only to the title and abstract of the studies, while in the second pass the remaining studies were selected based on their full text. The initial search resulted in 512 studies; after the two selection passes, we found 36 of them were relevant to our research questions. Table 1 shows how many studies we had after each selection pass, separated by source. Appendix A lists all selected studies.

4.1. General results

Once we established the pool of selected studies, we read each of the studies to assign their fitness scores (Section 3.4) and collect the data needed to answer the research questions (Section 3.5). We had two authors independently read the studies, evaluate their fitness, and collect data using this process, then merged the results. The authors discussed evaluations that differed by 0.5 score points or more, increasing or decreasing the assigned values accordingly whenever the discussions led to reevaluations of the studies. After resolving all of these cases, for each study, we used the average of the fitness assessment values produced by each author as the final value used for analysis. As for the data, we used the union of codes assigned to a given study to know *what data* it contained and used the sum of their counts when analyzing the *frequency of the data*. All the data and fitness assessment values (together with the originating fitness scores) of the studies can be found in a separate document.[9] Table 2 shows the codes found for the qualitative data fields F_4 through F_7. Since F_5 regards well-established treatments used in software development and studied in software engineering, we chose to use references to indicate explicitly where they come from. Additionally, as we better explain in Section 4.4 when we analyze the codes of field F_6, the *All* code is a special case.

Since we do not filter studies by their research field, a significant part of them are not direct proposals in the disciplines of software engineering or software architecture. They were rather from fields such as *serious games* [42], where authors use games for purposes beyond entertainment, such as education or health care. In these cases, the studies often approached the subject of software architecture only tangentially, thus receiving a lower fitness assessment. For instance, selected study S_{23} proposes a story-based exercise game system for children which relies on light and audio devices to provide immersion for students. The focus of that study is in the novel use of these devices to encourage the participation of kids and their evaluation measures how much they succeed in that regard. The software architecture of that system is only discussed in a brief section (Section 4.1) and is of no further relevance to the interests of study S_{23}. As such, we cannot know what were the benefits and restrictions of the architecture they designed. In other words, some studies did not fully contribute towards answering our research questions, despite the contributions they provide for other research fields.

We found this to be a relevant factor to consider in our analysis and classified the studies accordingly, depending on whether they (1) explicitly propose a *software architecture* for game mechanics, (2) incidentally discuss it as part of a wider investigation of *software engineering*, or (3) regard some other area of game development but had at least a minor part of their texts discuss architecture of game mechanics. These other areas often included different combinations of types of serious games, like educational games (edugames) [42], *pervasive games* [43], and *exergames* (games for physical exercise and health care) [44]. Fig. 1 shows the distributions of studies according to these categories (the exact studies of each category can be found in Table B.1 in Appendix B).

We processed data gathered from the studies to answer the research questions stated in Section 3.1. Sections 4.2, 4.3, and 4.4 respectively address RQ_1, RQ_2, and RQ_3. In those sections, we provide a broad view of the data, highlighting immediately notable occurrences for reference in Sections 5 and 6. We present the results from fields F_4 to F_7, using a bar chart for each of them (Figs. 2, 3, 4, and 8). These charts show how many selected studies (in percentage) contain each code for that field, from the most common to the least common. We call this metric the *frequency of a code*. For the precise list of codes assigned to each study, see

Table 2
Data codes found in the reviewed studies for F_4 through F_7.

F_4. Software design requirements	
Accessibility	Context-awareness
Emergent behavior	Didactic development
Distributed-development friendly	Domain model mapping
End-user development	Extensibility
Flexibility	Heterogeneous hardware
Integration with third-party tools	Pattern conformance
Performance	Portability
Minimization of brittle code	Minimization of coupled code
Minimization of fragile code	Rapid prototyping
Reusability	Run-time adaptability
Security	Testability
Type consistency	Unstable network tolerance
Verifiability	
F_5. Architectural practices and patterns	
Adaptive Object-Model [34]	Custom Architecture
Data-driven design [1,4]	Design patterns [35]
Event-driven [3]	Entity-Component [1,2]
Inheritance-based [2]	Layered subsystems [3]
Model-driven development [36]	Model-View-Controller [22,23]
Ontology-driven architecture [37]	Modularization
Reference architecture [38]	Requirements analysis [39]
Reuse of components [40]	Test-driven development [41]
F_6. Types of game mechanics supported	
Economy	Narrative progression
Physics	All
F_7. Design restrictions	
Complex usage	Does not scale well
Encourages hardcoding	Game-specific
Genre-specific	Incomplete
No proposed design	No direct support for complex mechanics
Outdated requirements	Outside-code support only
Requires cost/benefit analysis	Requires domain-specific knowledge
Requires end-user expertise	Technology-dependent

[8] The second time we collected studies, we limited the search to studies published since the previous collection.

[9] https://www.ime.usp.br/kazuo/slr2020/data.xlsx.

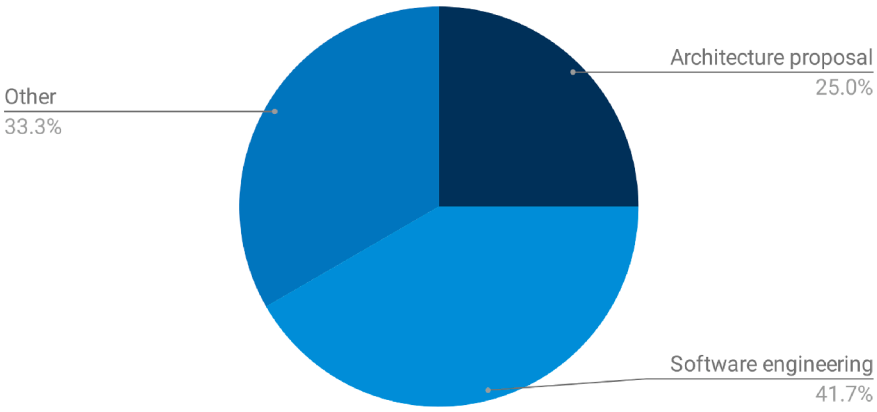


Fig. 1. Studies divided by their major research field.

Tables B.2–B.5 in Appendix B.

4.2. Software design challenges (RQ_1)

Fig. 2 shows the frequency of selected studies containing codes that express software design requirements (F_4). We can see that *Reusability*, *Flexibility*, and *Extensibility* are, by a significant margin, the most common requirements when designing or choosing a software architecture for game mechanics, present in approximately 40% of studies. Other common codes for design requirements include *Performance*, *Integration with Third-Party Tools*, *Heterogeneous Hardware*, *Portability*, *Domain Model Mapping*, *Didactic Development*, and *Rapid Prototyping*, present in between 18% and 23% of studies. Among the less frequent codes, present in less than 10% of studies, we find many codes regarding software stability, such as *Pattern Conformance*, *Minimization of Fragile Code*, *Minimization of Brittle Code*, *Verifiability*, and *Type Consistency*.

Fig. 3 illustrates the frequency of codes for each design restriction we found in the reviewed architectural solutions. The most common restriction among selected studies was the *Technology-Dependent* code. This code was used to indicate studies where the design of its proposed architecture is *coupled* enough to third-party technology that it is not self-evident how it could generalize to games and engines that did not use such technology, either because they chose another implementation or developed their own. The two most common cases of this were studies that used *Unity3D*, proposing architectures that relied on the particularities of that engine, and studies about pervasive games which required the use of very specific hardware and/or software to integrate their games and tools with special peripherical devices.

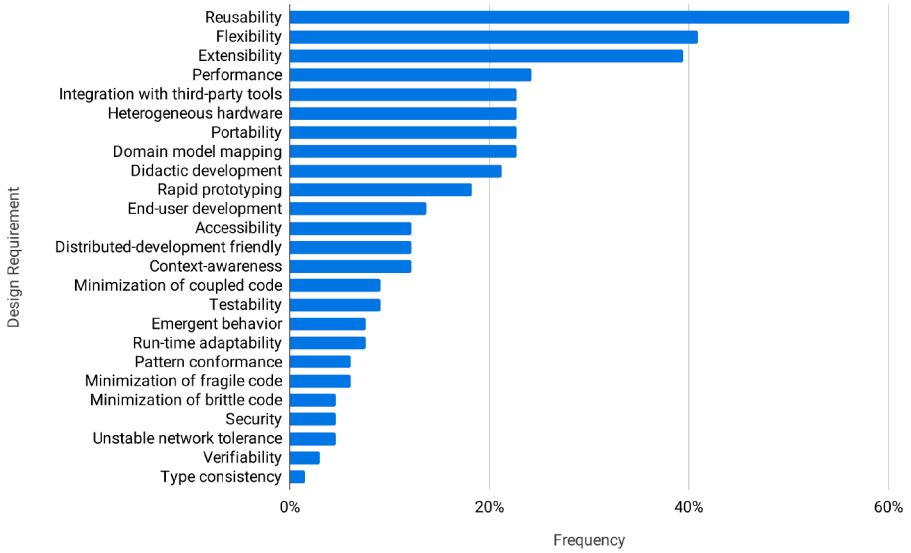


Fig. 2. Frequency of selected studies coded with each software design requirement found in the review. For the full list of studies with each code, see Table B.2 in Appendix B.

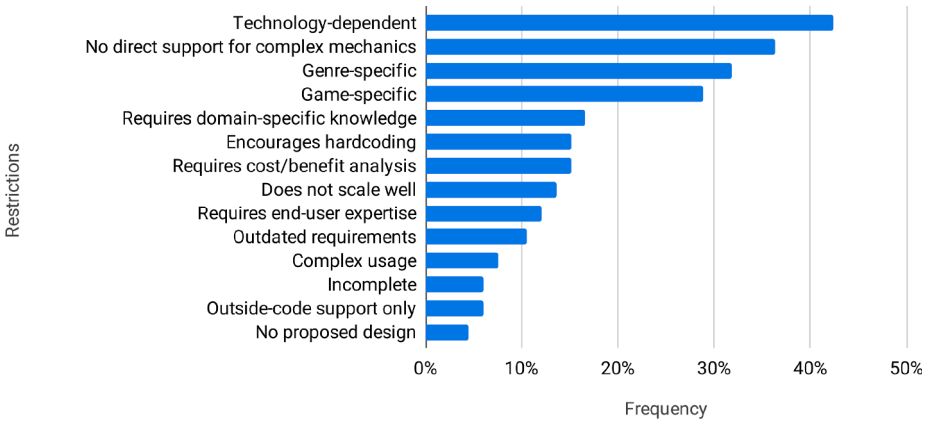


Fig. 3. Frequency of selected studies with codes for different design restrictions. For the full list of studies with each code, see Table B.5 in Appendix B.

Other common restrictions found were *No Direct Support for Complex Mechanics*, *Genre-Specific*, and *Game-Specific*, present in 29% to 45% of selected studies. *No Direct Support for Complex Mechanics* is a code for proposed architectures that do not formalize a way to increase the interaction between mechanics – which is what we considered "complex mechanics". An example would be studies that use inheritance to model the types of entities simulated in a game or engine because it is usually hard to mix-and-match types in a hierarchical architecture [2, chapter 14]. That is, once two different types of entities are defined in different branches of the hierarchy, it is hard to implement a new type with characteristics from both without incurring in either repeated code or dangerous multiple inheritance. Another example of complex interaction is when mechanics can dynamically affect the behavior of other mechanics, like when different equipment completely changes the way a character moves (walking, jumping, flying, swimming, etc.). Architectures that do not explicitly address any possibility of this kind were thus coded with *No Direct Support for Complex Mechanics*.

4.3. Architecture patterns (RQ_2)

Fig. 4 illustrates the frequency of selected studies that contain codes related to architectural practices (F_5). We can see that *Data-Driven Design* is, by a considerable margin, the most common code, present in 45% of studies. Other common practices are *Inheritance-Based Entities*, *Layered Systems*, *Design patterns*, *Modularization*, and the *Entity-Component* pattern, present in between 20% and 30% of selected studies. However, other practices with active research fields in software engineering and architecture are less prominent among selected studies, such as *Reference architecture*, *Model-Driven Development*, *Model-View-Controller*, *Adaptive Object-Model*, and *Test-Driven Development*, present in less than 10% of studies.

Next, we complement the presentation of results by inspecting the fitness assessment (from Section 3.4) assigned to the selected studies and comparing them to codes in data field F_5 (architectural practices and patterns). That is, instead of counting how many studies contain each code like in Fig. 4, we now consider the *total fitness assessment* of the codes. To do so, we summarize how much contribution there is for a given code across all reviewed studies by considering the fitness assessment value of each study with that code. Basically, we ponder the code count by the fitness assessment values of the corresponding studies:

$$A(code) = \frac{\sum_{i=1}^{36} a_i c_i(code)}{\sum_{i=1}^{36} c_i(code)} \tag{1}$$

Where $A(code)$ is the *total fitness assessment* of the code, $c_i(code) \in \{0, 1\}$ counts whether the i-th study contains the code, and $a_i \in [0, 1]$ is the fitness assessment value of the i-th study. This means each count adds at most one, making the maximum theoretical value for the numerator be the total count of that code. We are interested in the percentage of that total that the code actually scored, hence the denominator. Fig. 5a shows that metric.

We observe that all practices have fitness greater than 50% and there is no clear correlation between the total fitness assessment and the frequency of codes. For instance, *Adaptive Object-Model* and *Model-View-Controller* exhibit simultaneously the greatest fitness and the lowest frequencies. To further explore the relationship between fitness and

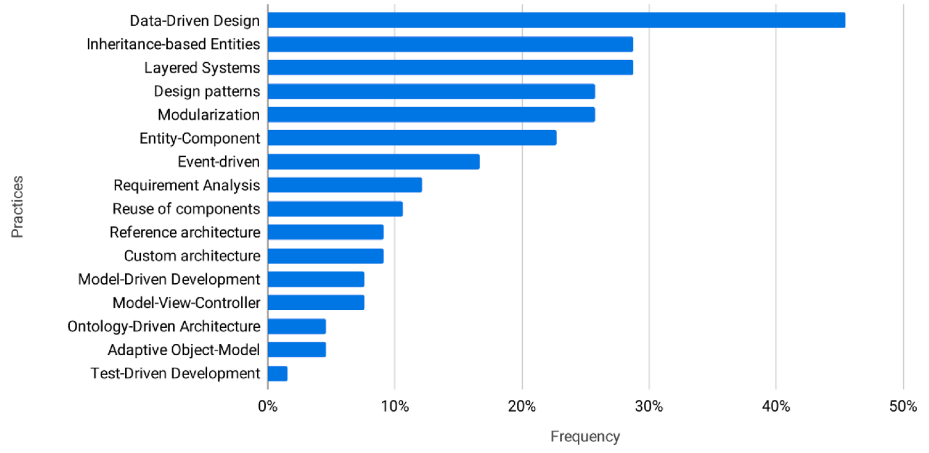
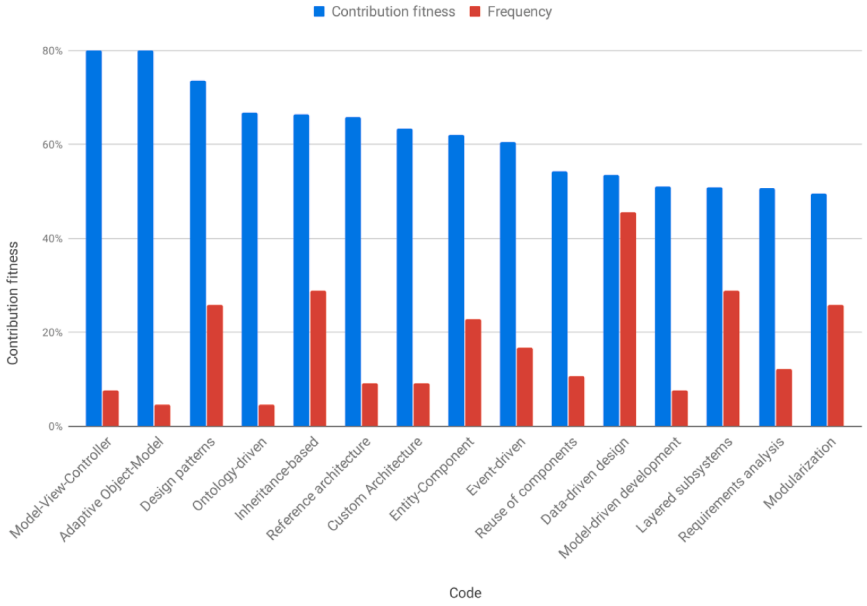
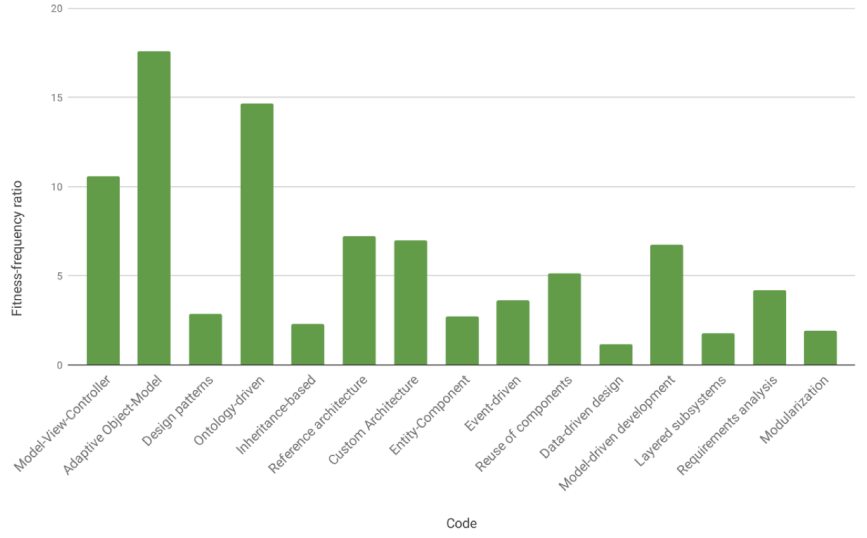


Fig. 4. Frequency of selected studies with codes for different architectural practices. For the full list of studies with each code, see Table B.3 in Appendix B.



(a) *Total fitness assessment* of practice codes.



(b) *Fitness-frequency ratio* of practice codes, following the same order from 5a.

Fig. 5. .

frequency, we computed the ratio between them for each code in F_5. We called this metric the *fitness-frequency ratio* of a code, and the result is presented in Fig. 5b. High values indicate practices with a high fitness score but present in few studies. In addition to *Adaptive Object-Model* and *Model-View-Controller*, we see *Ontology-driven*, *Custom architecture*, and *Reference architecture* with high ratios, while *Data-Driven Design* shows a considerably low ratio. We interpret these results in Section 5.

In addition to the frequency of architectural practices exhibited in Fig. 4, we wish to understand what motivated studies to adopt each practice. For that, we analyze the connection between practices and the design requirements from Fig. 2. This is shown in Fig. 6, where each circle is associated with a practice-requirement pair by its position, and the percentage of studies that contain both corresponding codes (we show only pairs with 10% or more in this case). To extract significant information from this chart (and the others shown after it), it is important to consider the original frequency of each code, since very common codes are more likely to have larger bubbles even if there is no real association with the other codes. This means we are interested in pairs that have a high percentage but one of the codes is not as frequent, or the opposite. The first thing to notice is the high density in the columns of *Extensibility*, *Flexibility*, and *Reusability* as a consequence of the high number of studies listing these characteristics as requirements – which we know from Fig. 2. Moreover, *Data-Driven Design* is the most common practice associated with these three requirements.

We are also interested in understanding the design restrictions associated with noteworthy practices. This relationship is pictured in Fig. 7, as the frequency of each practice-restriction pair found in studies. As we can see, the most recurrent restriction independently of practice adopted is *No Direct Support for Complex Mechanics*, which was also the second most common design restriction in Fig. 3. Surprisingly, *Data-driven design* is the practice with the most associated restrictions. The practice with the second-highest increase of restrictions is *Inheritance-based*, which, in particular, increases the frequency of the *No Direct Support for Complex Mechanics* code from 35% in Fig. 3 to over 50%: an

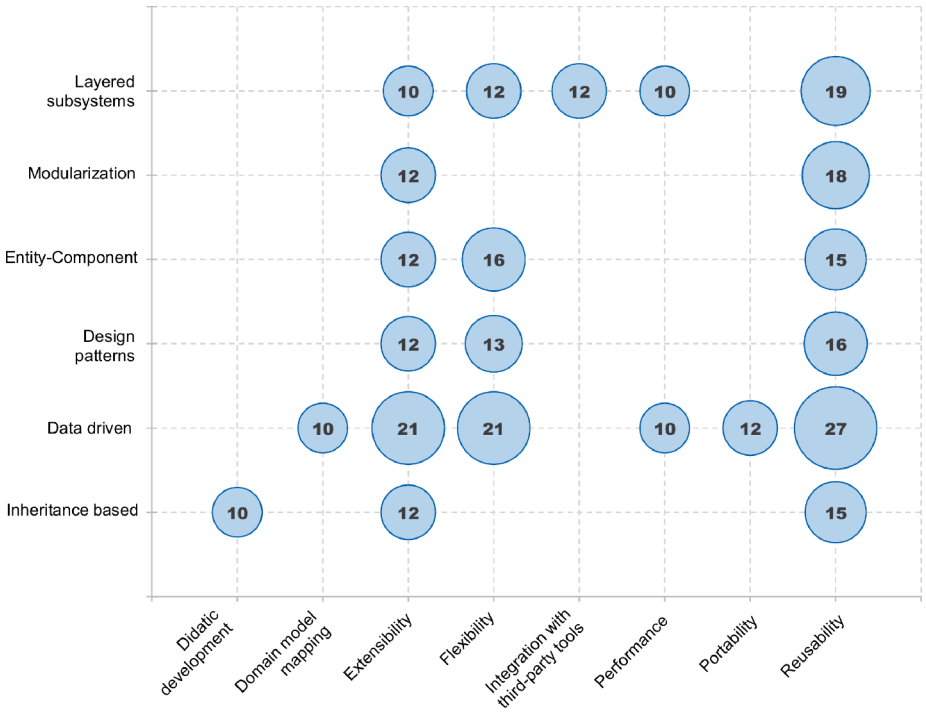


Fig. 6. Frequency (in percentage) of intersection between requirements (horizontal axis) and architectural practices (vertical axis). We show only intersections with three or more studies.

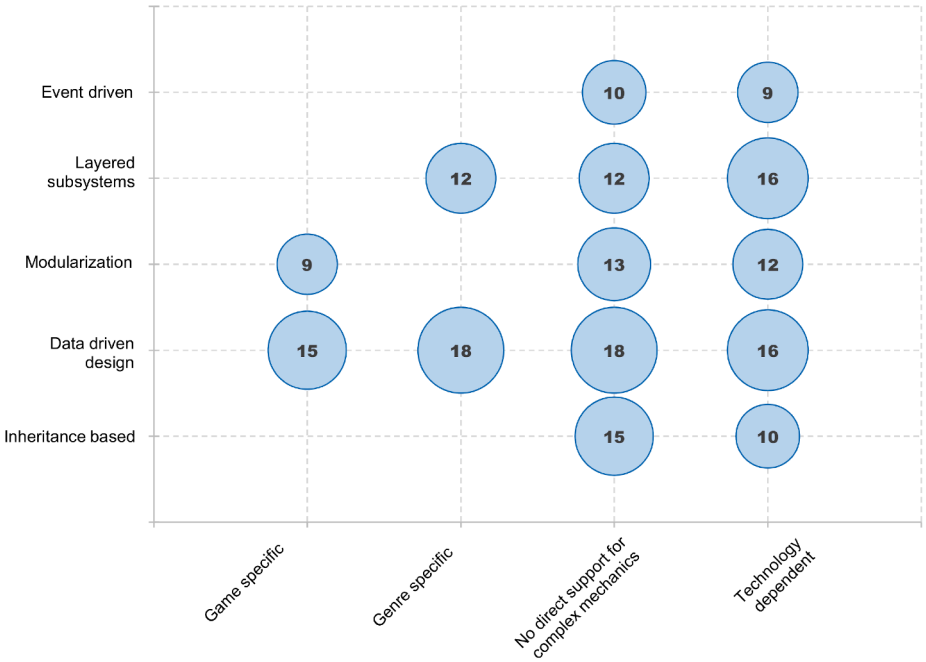


Fig. 7. Frequency (in percentage) of intersection between restrictions (horizontal axis) and architectural practices (vertical axis). We show only intersections with three or more studies.

intersection of 15% in Fig. 7 of the original 29% *Inheritance-based* codes in Fig. 4. On the other hand, even though it is a strongly adopted practice as seen in Fig. 4, *Entity-Component* has little association with any restriction (it does not even appear in Fig. 7 due to the cutoff). A similar lack of restrictions is also remarkable with the *Design patterns* code.

4.4. Game mechanics (RQ_3)

Fig. 8 shows the frequency of selected studies bearing codes for each type of game mechanics as categorized in Section 1.4. We added a fourth code, *All*, for studies whose contributions support all types of mechanics, or rather, are not designed to support any specific type of game mechanics in particular. That is, we assume that if an architecture contributes to all kinds of mechanics, we cannot conclude whether it would

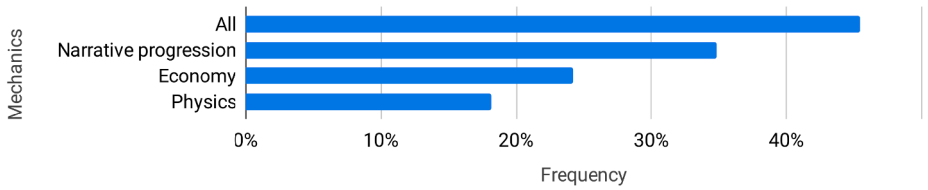


Fig. 8. Frequency of selected studies with codes for each type of game mechanics. For the full list of studies with each code, see Table B.4 in Appendix B.

be more particularly useful for some mechanics but not others. Thus, we code studies with such architectures with *All* so we can analyze them separately. If they regard one or two specific types of mechanics, we assigned the corresponding codes. In other words, studies could be assigned either one or two codes from *Economy*, *Physics*, and *Narrative Progression*, or be only assigned with *All* and no other codes.

It is important to remember that, as explained in Section 3.5, we assigned codes according to the features identified in the studies, even if the study itself does not declare that feature in its text. For instance, even if a study does not mention "economy mechanics" (or even just the word "mechanics"), we analyze the game or engine described to assess whether it supports economy mechanics or not despite what the authors say or do not say.

In the data we obtained, the frequencies of codes for *Economy* and *Physics* mechanics are evenly distributed, with respectively 24% and 19% presence in selected studies. *Narrative Progression* mechanics is above both of them with a 35% presence in selected studies, while 45% of the studies provided solutions that support all types of mechanics.

The relationship between types of mechanics and architectural practices is detailed in Fig. 9. As expected, there is a clear dominance of the *All* code since it is, by a considerable margin, the most common code regarding types of mechanics as shown in Fig. 8. On the other hand, we see that the *Economy* code is only strongly related to *Inheritance-based* and *Design patterns* but with a proportionally reduced intersection with the commonly adopted *Data-driven design*: its frequency decreases significantly from the 45% in Fig. 4 to less than 38% (9% in Fig. 9 out of 24% studies coded with *Economy* in Fig. 8). In contrast, we see *Narrative progression* highly associated with *Data-Driven Design*. *Physics* and *Narrative progression* are both significantly related to the *Event-driven* code, the former remarkably so: the frequency of *Event-driven* codes increases from 16% in Fig. 4 to approximately 40% (7% in Fig. 9 out of 18% studies coded with *Physics*).

By analyzing the relation between requirements and mechanics in Fig. 10, we see that *Performance*, followed by *Heterogeneous hardware* and *Reusability* are the most desired features from *Physics*. *Didactic development* and *Flexibility* are the only codes that significantly increase in frequency among studies coded with *All*.

5. Analysis

This section discusses the results from Section 4. We address each research questions at a time in Sections 5.1, 5.2, and 5.3. Then, we consider promising research topics for future research based on our findings in Section 5.4.

5.1. Software design challenges (RQ_1)

The first research question our review addresses is RQ_1: *What software design challenges do researchers face when implementing game mechanics?* The goal of this question is to identify the main concerns researchers take into account while designing game mechanics architecture in general. To analyze this topic, we evaluate field F_4, *Software Design requirements*.

The most commonly coded requirements (shown in Fig. 2) are, by a

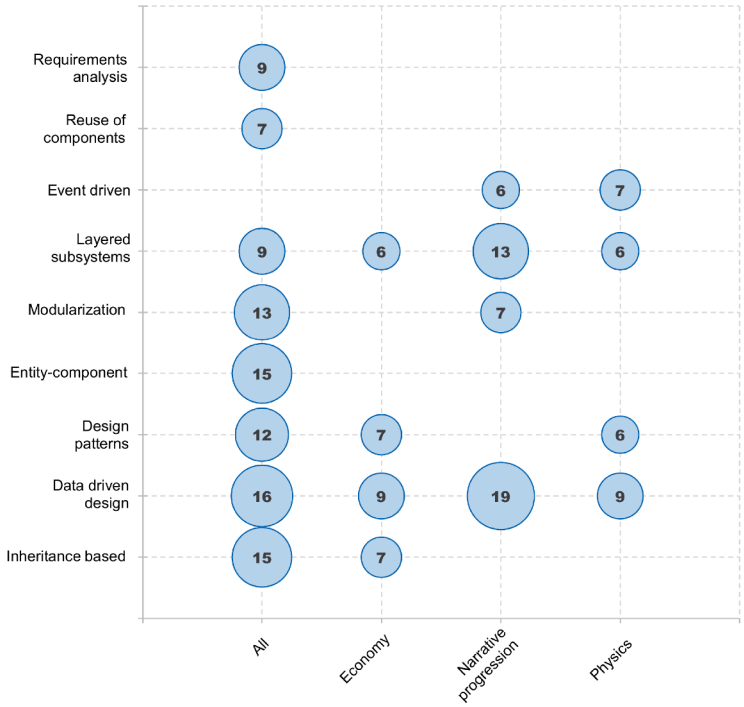


Fig. 9. Frequency (in percentage) of intersection between game mechanics (horizontal axis) and architectural practices (vertical axis). We show only intersections with two or more studies.

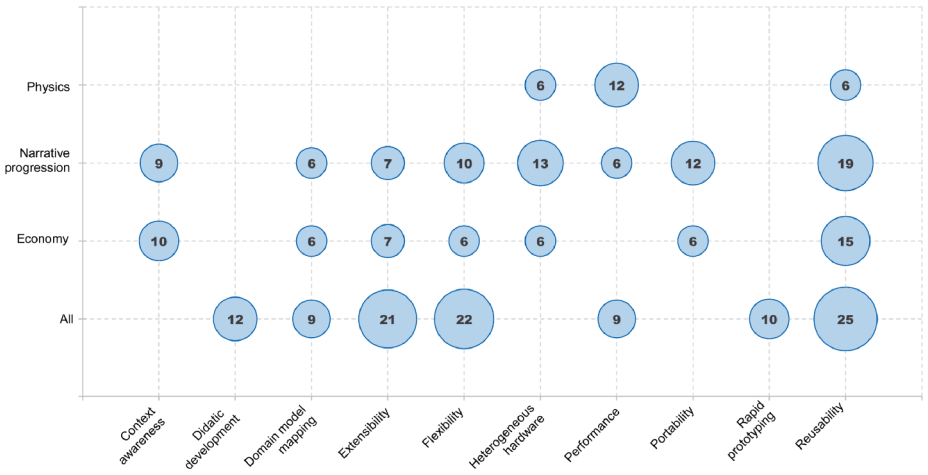


Fig. 10. Frequency (in percentage) of intersection between game mechanics (vertical axis) and requirements (horizontal axis). We show only intersections with two or more studies.

significant margin, *Reusability*, *Flexibility*, and *Extensibility*, present in more or less 40% of studies. This indicates that the foremost goal in research when designing the architecture of game mechanics is favoring change and evolution of the code, either by re-purposing parts of it (reusability and flexibility) or by reducing the effort needed to add parts to it (extensibility). If these are recurrent requirements, then the studies agree that developing game mechanics involves a great number of changes to the implementation, or that each change is individually expensive, or both [1,45]. It is, in particular, expected that games demand a great number of changes to their mechanics as part of balancing its gameplay [10].

At the same time, the cost of each addition to the game mechanics can increase for a number of reasons. For instance, the architecture could inadvertently require writing code to many source files for every change, or the programmers (as can be the case in some academic contexts) might still be inexperienced game developers and navigating the code base of a game could be a slow process for them. Additionally, even when costs are not high, limitations such as deadlines, budget, and team size may further motivate a research development team to seek reusability, flexibility, and extensibility in their software design. For instance, study S_{17} says that the "*creative team can demand changes to the software architecture during development, but this decision depends on how far the project has progressed and the cost and benefit of making the change*", and study S_{21} defends that "*one way to reduce the cost of games is to reuse particular game components. Rather than reinventing the wheel when developing a 3d engine, a physics engine or a network component, game developers can choose to use an existing Commercial of the Shelf (COTS) Component*". Study S_5 also clearly corroborates with this idea, saying that "*flexibility in software architecture is especially important in game development, when the object model needs to be rebuilt so often that refactoring conventional architectures becomes prohibitively expensive*".

Regarding other common requirements, the high frequency for the *Performance* code is an expected result since games require real-time processing and because game research often studies implementation on less traditional platforms such as mobile or pervasive games, which often demand a more optimal use of computational resources. It is notable, though, that *Performance* is not among the most frequent codes among the studies, indicating that studies prioritized *Reusability*, *Flexibility*, and *Extensibility* over it.

Another notably frequent code, *Didactic Development*, demonstrates the interest of software architecture research in games with educational environments, either via educational games or via game development courses. For instance, study S_{15} states that "*because of [games'] wonderful characteristics and its popularity in children, more and more peoples believe that computer games are powerful educational tools, if used appropriately*", and proposes a common software architecture for educational games. At the same time, study S_2 used game development as a project to teach software engineering for undergraduate students, concluding that the students "*developed fundamental understanding of game engine architecture through design and implementation of complex game systems*" and they "*saw in detail how the use of design patterns gave rise to a software architecture that was decoupled, scalable and data-driven*".

Close in frequency to *Didactic Development*, the *Integration with Third-Party Tools* code reinforces the preference for software reuse and is closely related to the next most frequent codes, *Heterogeneous Hardware* – which we mainly attribute to the research field of pervasive games – and *Portability* – which indicates the interest in making games more accessible, even if only to increase the user base and, perhaps, the sales of the final product. Incidentally, the most common restriction among selected studies was the *Technology-Dependent* code, which means the solutions provided by many studies were coupled to some third-party technology or platform, and their contributions might not be generalizable to other implementation contexts. Lastly, the *Rapid Prototyping* code, with a frequency of 18% highlights the demand for technology that promotes the quick experimentation and evaluation of game concepts, a common practice in the iterative development of games [10].

Comparing the least common codes and the preference for *Flexibility*, we see a tendency of studies favoring versatility over stability when developing game mechanics. On the other hand, the low frequency of codes such as *Security* and *Unstable Network Tolerance* might be simply due to the studies being selected for their contributions to software architecture and game mechanics, making requirements of other aspects of game development less present in our particular sample.

When we consider the design restrictions found in the selected studies, we see that *Technology-Dependent*, *Game-specific*, and *Genre-specific* are among the most common codes. This is explained by the fact that the majority of the selected studies do not aim for general purpose and reusable solutions, such as game engines. Instead, they focus on specific games and therefore have limited applicability outside their well-defined scopes. Beyond that, a significant number of studies deal with game prototypes. By doing so, not only do they need to restrict the project's complexity, but they also need to reduce implementation effort by extensively using libraries and frameworks previously built. It is, thus, to be expected that *Technology-Dependent* is the most common restriction in Fig. 3, and the second most common restriction independently of practice in Fig. 7.

There seems to be a curious dichotomy between the most common architectural requirements and the most common design restrictions. Studies seem to favor architectures that are reusable, flexible, and extensible, but end up proposing architectures that can only be applied

in relatively narrow contexts, usually involving a certain genre or technology. Though apparently contradictory, we understand that these conflicting features occur together because, while studies do focus on specific and limited contexts, they still intend to provide reusable solutions in that particular context. For instance, study S_7 proposes a framework for narrative-based audio games, which are restricted both by genre and technology. Nevertheless, the framework they propose is still reusable because, through data-driven design, it accepts any number of "game worlds" as input to simulate and present to players. We hypothesize that this practice is actually very effective since, by controlling the scope of architectures, developers and software designers can use more assumptions about its requirements. This, in turn, allows systems to be parameterized and, thus, become more reusable.

To summarize, our analysis suggests that the main concern of researchers when implementing game mechanics is to reduce implementation effort and cost through software reusability, flexibility, and extensibility. That is, the implementation of game mechanics is an organic process where writing code can easily become a bottleneck, thus requiring a versatile software design. Ideally, studies seek this benefit without compromising runtime performance but, as we explain in Section 5.3, this depends on the type of mechanics involved. Finally, we believe there is evidence that restricting the scope of an architecture for game mechanics may improve its opportunities for reusability.

5.2. Architecture patterns (RQ_2)

Given the design requirements discussed in Section 5.1, it is interesting to explore the methods applied to meet these goals. Hence, our second research question, RQ_2, is *What software architecture practices and patterns do researchers use for the implementation of game mechanics?* Since there were many interesting results for this question, we divided this section to produce more focused discussions.

5.2.1. Data-Driven Design

The high frequency of the *Data-Driven Design* code in Fig. 4 corroborates with industry experts that defend this practice as a fundamental part of game development [1,4]. There is a significant presence of studies that involve or target researchers without a programming background, given the multidisciplinary nature of games. This brings an increased demand for tools and practices that promote more accessible game development, making contributions with a data-driven approach a reliable and, thus, common solution. When compared to the three most common requirement codes – *Reusability*, *Flexibility*, and *Extensibility* – in Fig. 6, reusability, flexibility and extensibility have a substantial relation to studies that use data-driven design. This can be explained by a series of factors:

- Data-driven design offers a practical way of changing software behavior without modifying the codebase;
- Besides behavior, data-driven design also makes it easy to change and expand game content, helping non-programmers contribute to projects;
- The reduced need for refactoring and recompiling promotes faster iteration during development, fine-tuning, and post-release maintenance.

It is also notable that *Data-driven Design* has a low fitness-frequency ratio (Fig. 5b). Given the high relation to *Reusability*, *Flexibility* and *Extensibility*, and the fact that data-driven design is one of the standard practices in the industry to fulfill these requirements [1,4] our review suggests that investigations exclusively directed at this practice in particular (as opposed to incidentally related studies) are sparse and would thus be welcome contributions. One extreme example of how well data-driven design can separate coding from the actual production of a game is *Doom*, a computer game released in 1993. The first-person shooter architecture was developed in a way that the game engine, responsible for the core mechanics, could support customized additions via data packages known as *WAD* files [1]. Through Doom WAD files it was possible to modify game rules, as well as create custom levels, enemies, and weapons. As a result, the engine could be considered a black box and many players were able to create modified versions of the game – also known as MODs – without actual access to its source code.

However, when we look at Fig. 7, we also note the strong relation between *Data-Driven Design* and a few restrictions. This is unexpected because this technique is known for its broadly generic suitability, which should not relate to codes such as *Game-specific* and *Genre-specific*, in particular. Further investigation of this peculiar result showed that, although data-driven design is not restrictive in nature, the described instantiations of the practice by studies are themselves very specific and difficult to reuse in other contexts. It is a similar phenomenon to what we discussed in Section 5.1 about the dichotomy between versatility requirements and restrictions due to limited scope. Again, study S_7 provides an example of how an architecture can use data-driven design but still be limited in its application regarding genre and technology.

5.2.2. Inheritance-based and entity-component architectures

A notable observation in Fig. 4 is the presence of both *Inheritance-Based Entities* and the *Entity-Component* patterns among the most common practices since they are usually mutually exclusive architectural patterns. In particular, practitioners and game engines in the last decades have favored entity-component systems over inheritance-based ones [2,1], especially when implementing game mechanics, while the selected studies showed a slight preference for Inheritance-Based approaches, though both methods ranked high in the review. Some studies openly defend entity-component-system architectures over inheritance-based ones, such as study S_1 that states that game development is "*moving from inheritance to component-based software architecture in game engines*", and study S_5, that explains how "*the reliance on implementation inheritance can cause many problems*" and that "*an entity system architecture is delegation-based alternative to an inheritance-based architecture for implementing game engines*".

Fig. 6 provides us with further information. First, we see that *Inheritance-Based Entities* rarely appears together with *Flexibility*, despite the high frequency of both codes, which indicates that studies that require flexible architectural solutions seldom rely on inheritance-based designs. On the other hand, the *Entity-Component* code is present in approximately 40% of studies that require flexibility, even though it is present in only a little over 20% of the selected studies in general. We understand from this that the literature considers the entity-component-system pattern to be more flexible than its older alternative, inheritance-based entities, which agrees with industry knowledge [2,1]. However, the *Inheritance-Based Entities* code, with a frequency of more or less 30%, is present in over half the studies coded with *Didactic Development*, suggesting that studies that target didactic designs might prefer this approach – but further investigation is required to confirm this. Additionally, Section 4.3 highlighted many evidences in Fig. 7 indicating advantages of entity-component systems over inheritance-based entities due their reduced design restrictions.

5.2.3. Traditional research topics and architectures

Another noteworthy observation is the low incidence of codes that are otherwise active research topics, such as *Reference architecture* and *Test-Driven Development*. There are a few reasons that could explain this. One of them is that the review protocol selected only studies that regarded the implementation of at least one game mechanic, often leaving out studies that contributed to other parts of game development with more active research fields (e.g. computer graphics, networking) which were thus more likely to rely on more formal practices like the ones mentioned. This is the case, for instance, of the *Model-View-Controller* code, which refers to a pattern used in a number of studies not included in the review because they did not pass one of the steps of the selection process described in Section 3.3. That usually happened when

the study did not investigate the architecture inside the Model, which is our main interest in this review (as explained Section 1.4). Such is the case, for instance, of the works of Olsson et al.'s [24] and Caltagirone et al.'s [46].

Other reasons for the low frequency of active research fields applied to game mechanics could be either that these applications of software engineering and software architecture are still emergent fields of research, or that they are indeed ill-suited for the development of game mechanics in general. However, as we see in Fig. 5b, *Adaptive Object-Model*, *Model-View-Controller*, *Ontology-Driven*, and *Reference architecture* codes are all topics related to software architecture, but their fitness-frequency ratios rank among the highest found in our review. This means that studies discussing them are usually very relevant to our research questions, but there are only a few of them, which suggests there is room for further, significant contributions in these fields.

5.2.4. Layered subsystems

The *Layered Subsystems* and *Design patterns* codes are common practices shown in Fig. 4, and reflect strategies known in game development for their practicality and/or effectiveness [1,2]. By comparing both Figs. 6 and 7, we follow an interesting pattern regarding the *Layered Subsystems* code. Although it is present in 29% of studies, roughly 55% of studies that listed *Integration with Third-Party Tools* as a requirement chose this software architecture. This supports the general software architecture the industry employs in large-scale games [1], where the layers abstract lower-level APIs to allow a more technology-agnostic design of higher-level code. For instance, study S_7 proposed a layered architecture with which they "*can design and test audio games using PC hardware and later transfer our work to a mobile platform by rewriting the necessary code*" on a single layer in the architecture. Additionally, more than 50% of studies employing the layered architecture were also coded *Technology-dependent*, a code present in only 42% of studies in general, and the intersection between these two codes is also among the largest in Fig. 7, which reinforces this pattern.

We also notice in Fig. 6 dense code intersections of the *Reusability* code with *Layered Subsystems* and *Modularization*. Both practices are traditional approaches to organize features and allow code reuse inside a project as well as among different projects. Examples of features commonly written as separate modules are physics, artificial intelligence, network. Recurrent examples of layered systems are game engines and platform-specific interfaces, such as file system access.

5.2.5. Answering RQ_2

To sum up, the predominant approach used by reviewed studies to implement extensible, flexible, and reusable game mechanics is *Data-driven design*, with *Entity-Component*, *Layered subsystems*, *Modularization*, and *Design patterns* further supporting the software design of games. *Inheritance-based* is also a reliable practice, but with more restrictions. At the same time, underused practices from more traditional research fields, such as *Reference architecture*, *Test-Driven Development*, and the *Model-View-Controller* pattern, presented more substantial contributions. The opposite happened with data-driven design, which is investigated only superficially by most studies, and its applications are often restricted to the scope of the corresponding game or game development tool.

5.3. Game mechanics (RQ_3)

Finally, the third research question, RQ_3, is *What types of mechanics most often require, in research, the use of architectural practices and patterns, and which are they?* The data pictured in Fig. 8 is directly aligned with that question. From there we see *All* as the most frequent code. This shows that most studies do not guide their software design towards types of mechanics separately. At the same time, the high frequency of the *Game-specific* and *Genre-specific* codes in Fig. 3 indicates that a significant part of studies focus on solutions to particular sets of game mechanics instead of more general approaches, because this allows authors to design architectures that best fit their situation, at the cost of implementing less flexible systems. That is, solutions are often tailored to the particular needs of each study, but that does not mean they necessarily differentiate the mechanics between narrative progression, physics, and economy.

Despite that, *Narrative Progression* is the most isolatedly present type of mechanic, which indicates a preference in the selected studies for games focused on story-telling. That, in turn, concurs with the relevant amount of researches related to *Edugames* within the *Other* category reported in Fig. 1, since exposition is a straightforward method of providing knowledge. There is also a strong relation between *Narrative Progression* and *Data-Driven Design*, reinforcing that the practice is very suitable for projects highly dependent on content production and customization (in this case, usually stages, dialogues, scene scripts, etc.). This also matches our observation that games based on narrative mechanics often require *Reusability*, promoted by data-driven design.

As for the other types of mechanics, a few patterns are notable. *Economy* mechanics relate to *Inheritance-based* and *Design patterns*, but (proportionally) not enough to *Data-driven design*, despite it being a common practice in the reviewed studies. This combination suggests that economy mechanics in the literature are usually designed for specific scenarios without flexibility in mind. *Physics* mechanics present a relation to *Event-driven* architectures, which regards a practice appropriate for decoupling code when entities interact with one another by triggering multiple events (*e.g.* collision handling). Our data also showed that researchers implementing physics of mechanics seek *Performance*, *Integration with third-party tools*, and *Reusability*. Indeed, for efficient ways of implementing laws of physics there are well-known and optimized libraries widely available. Because of that, most authors opt to save time and work by using third-party libraries.

On further inspection, we also noticed that studies that employed mechanics of narrative progression had an increased chance of being coded with *Layered Subsystems*. As we saw in Section 5.2.4, this architectural style is tied with the need for integrating third-party tools in the context of software architecture for game mechanics. Additionally, Fig. 10 confirms that studies with narrative mechanics were more often coded with *Heterogeneous Hardware* Thus, we understand that research involving narrative-oriented mechanics is usually interested in the use of innovative technology, to promote player immersion into the narrative through, for instance, pervasive games. Similarly, studies that separately target either narrative or – more notably – economy mechanics had a greater incidence of the *Context-Awareness* requirement code, indicating that pervasiveness might be a key incentive in the use of economy mechanics, which is otherwise weakly related to most design requirements among selected studies.

Answering the third and last research question, we see that most architectural practices and requirements involving game mechanics in the literature are not associated with specific types of mechanics. Studies usually focus on more general software design aspects of games, with both interaction and simulation features in mind, and whether they frame their solution design by a certain type of mechanics depends entirely on whether the kind of game they want to develop is limited to that type of mechanics. As a consequence of that, there are only a few conclusive observations we can make about the architectural needs of each type of game mechanics separately. First, narrative mechanics receive support from the practices of data-driven design and, when they involve some technological innovation, the layered subsystems pattern. Next, physics mechanics benefit from event-driven designs and require the most performance. Finally, economy mechanics are less tied to data-driven designs despite it being the most common practice among studies, with context-aware game applications being a particular case where studies favor economy mechanics.

5.4. Opportunities, challenges, and future trends

The reviewed studies often discussed games in contexts outside the topic of software architecture. The use of games as immersive experiences that engage users into learning or exercising, for instance, is a recurrent field of interest. In that context, the software is designed to facilitate the management of game content, often by game designers without the background needed to venture into the game code by themselves. Another use of games outside the discipline of software architecture is when exploring experimental technologies, such as pervasive games or mobile games when the technology was still a novelty. At the same time, there is educational use for game software architecture in computer science courses where students are tasked with implementing games themselves [47–49]. In these cases, the architecture might simply serve a didactic purpose, instead of a practical one, like in study S_2, where the students "*were required to use one or more Gang-of-Four design patterns in their implementation*". This also means many studies focus on prototyping, likely bearing different design requirements when compared to commercial games. The review reinforces that understanding the context and scope of a game project is thus an important step in determining what requirements and practices game mechanics demand.

An aspect that caught our attention is the relative absence of applications of common topics in software architecture to digital game mechanics. Popular software engineering topics such as reference architectures, the *Model-View-Controller* pattern, test-driven development, and the *Adaptive Object-Model* pattern are only rarely seen in studies about the development of digital game mechanics. This could be caused either by a lack of interest from the software engineering community to apply their knowledge to the context of game development or by lack of an explicit description of how game development researchers rely on software engineering techniques. It is also possible that they simply do not adopt any formal software engineering approach. Since the benefits of software engineering practices are well-known and could bring a number of opportunities to game development, we see many research possibilities in more formal applications of software engineering to design implementations of digital game mechanics.

In particular, the *Model-View-Controller* is a widely studied architectural pattern and there is little to no use of it among the selected studies. The few studies that do discuss it scored high in our review regarding their contribution towards the field of software architecture in digital game mechanics, which means that it is a relevant topic to investigate more. It would be especially interesting to further analyze the role of the Model part since it hosts the implementation (and corresponding architecture) of the game mechanics. Similar opportunities exist for the *Adaptive Object-Model*, a pattern designed with maximum runtime type flexibility in mind [34] since it also scored a high fitness to our review but was discussed only by very few studies.

In contrast, we found data-driven design to be the most common practice, even though its total fitness assessment ranked proportionally low. As explained by authors in the industry, data-driven design is a commonplace solution for reusability and extensibility, and yet studies only superficially describe its use. This means that a more in-depth investigation of this practice is needed. Furthermore, studies provide only game-specific uses of data-driven design, which supports little reusability across titles; thus, more general-purpose applications would provide greater contributions to the field. At the same time, however, the tendency for specific usages of data-driven design (and other practices that promote reusability, flexibility, and extensibility) suggests that this is an effective and pragmatic approach to the software architecture of game mechanics.

Another practice that presented opportunities for future research was the entity-component system, a very common pattern in the game industry [1,2]. Despite there being more studies approaching its alternative – inheritance-based entities – we saw that entity-component solutions were more flexible, presenting fewer design restrictions. The analysis suggests that inheritance is a more didactic solution, so we understand there is room for further research in both practices.

The study of design patterns, such as the ones described by Gamma et al. [35] and Nystrom [2], showed no evident support towards the many reviewed design requirements. Notwithstanding, studies that relied on them had fewer design restrictions. This means that understanding good software design practices in the development of game mechanics is a reliable approach, and would also benefit from further study.

Finally, a critical observation in this review is that types of mechanics are seldom approached independently. An inconvenient consequence of that is that studies do not present solutions that satisfy the specific needs of each type. This points to the research opportunity of investigating the architectural requirements of game mechanics separately. However, there are a few other tendencies our review highlighted. While physics mechanics have a long history of research activity, we see that narrative progression is the most commonly studied when software architecture is involved. There is a particular interest in designing implementations that support incremental narrative content, especially in a way that non-programmers are empowered to contribute content by themselves. On the other hand, there are very few practices and requirements tied with economy mechanics, which means it would benefit from further investigation. In particular, we see there is a relative lack of data-driven design applications towards economy mechanics. A more focused investigation could verify whether that is characteristic of this type of mechanic, or if there are still unexplored ways of applying data-driven design to economy mechanics in games.

6. Conclusion

In this paper, we conducted a systematic literature review to determine the state-of-the-art, current challenges, and research opportunities in the field of software architecture for digital game mechanics. We collected 512 studies from three dominant digital libraries. After a filter of inclusion and exclusion criteria, the final sample was a set of 36 representatives. Each of them was assessed by two different authors and received fitness scores in five different aspects. The data extracted from the studies were analyzed to answer the research questions stated in Section 3.1: what are the design challenges of implementing game mechanics, what software architectures are used to implement them, and what is their relation to each specific type of game mechanics?

Regarding RQ_1, our findings show that research involving software architecture of game mechanics favor design versatility and, to a lesser but still considerable extent, performance. The analysis suggests that support for accessible and immediate change to the mechanics of a game is a valuable feature for researchers, who usually lack the team size and experience of game companies. Even in the industry, authors concur with the importance of iterating faster in game development through such means. The key difference between academia and industry we found was that research often interacts more with education and innovation, which weighs on the software design of game mechanics. In particular, research solutions often lead at most to a prototype, possibly opting for less stability and robustness as opposed to production-ready games from the industry.

To achieve the aforementioned versatility, we saw in RQ_2 that a number of established techniques (from both academia and industry) remain the practice of choice in studies, such as data-driven development, modularization, and layered architectures. Data-driven design was especially prevalent, though most of the time only superficially investigated, i.e., applied to fit specific scenarios but not studied as a more general technique. At the same time, some practices emerged from the interaction with innovative technology, as was the case of layered architectures. When considering the restrictions imposed by each approach, the review revealed the entity-component-system pattern and the use of design patterns in general to be reliable solutions. The most common restriction was that practices were often coupled to the scope in

which they were used, both in terms of mechanics support and of technology dependency, which hampers extrapolating contributions to other contexts. General-purpose software designs for game mechanics were a minority, the case of data-driven design being the most noteworthy example.

Concerning RQ_3, the review also confirmed what we stated in Section 1.2, that academic studies have no clear terminology for referring to the parts of implementing a game that regards its mechanics. The great majority of studies generalized the developed mechanics, and only a few relations could be found between the types of mechanics and the corresponding software design requirements and practices. We see that physics mechanics have little intersection with the field of software architecture in general, and requires the most performance. On the other hand, narrative and economy mechanics are separately investigated mostly as a means of supporting education or technological innovation. Moreover, economy mechanics is the only type of mechanics that displayed a reduced use of data-driven design, despite its benefits.

Motivated by the seemingly discrepant practices between the industry and academia regarding the architecture of game mechanics, this study analyzed how and why such differences exist, and what possibilities are there to increase the synergy between these two contexts. We established that, while divergences in prospect and circumstances have an impact on the nature of research into this field, some evidence exists that both academy and industry could benefit from exchanging knowledge and experience. Studies have much to investigate in techniques used by game companies, and a number of long-standing research topics in software architecture and engineering are barely untapped sources of valuable contributions that professional game developers would benefit from.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Funding. This work is supported by the Sao Paulo State Research Foundation (FAPESP) under Grant No.: 2017/18359–6.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at https://doi.org/10.1016/j.entcom.2021.100421.

References

[1] J. Gregory, Game engine architecture, second edition, AK Peters/CRC Press, Boca Raton, FL, 2014.
[2] R. Nystrom, Game programming patterns, Genever Benning (2014).
[3] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.
[4] S. Rabin, The magic of data-driven design, in: M. DeLoura (Ed.), Game Programming Gems, Charles River Media, 2000, pp. 3–7.
[5] M. West, Evolve your hierarchy (Jan. 2007) [cited Sep 10, 2018]. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy.
[6] M. Sicart, Defining Game Mechanics, Int. J. Comput. Game Res. 8 (2). http://gamestudies.org/0802/articles/sicart.
[7] A. Järvinen, Games without Frontiers: Theories and Methods for Game Studies and Design, Ph.D. thesis, University of Tampere, 2008.
[8] J.C. Osborn, N. Wardrip-Fruin, M. Mateas, Refining operational logics, ACM Int. Conf. Proc. Series Part F1301. doi:10.1145/3102071.3102107.
[9] T. Dubbelman, Narrative Game Mechanics, in: International Conference on Interactive Digital Storytelling, 2016, pp. 39–50, https://doi.org/10.1007/978-3-319-48279-8_4.
[10] J. Schell, The Art of Game Design: A book of lenses, second edition, AK Peters/CRC Press, Boca Raton, FL, 2014.
[11] E. Adams, J. Dormans, Game mechanics: advanced game design, New Riders, Berkeley, CA, 2012.
[12] R. Hunicke, M. LeBlanc, R. Zubek, Mda: A formal approach to game design and game research, in: Proceedings of the AAAI Workshop on Challenges in Game AI, Vol. 4, 2004, p. 1722.
[13] B.A. Larsen, H. Schoenau-Fog, The Narrative Quality of Game Mechanics, in: International Conference on Interactive Digital Storytelling, 2016, pp. 61–72, https://doi.org/10.1007/978-3-319-48279-8.
[14] E. Adams, Fundamentals of game design, Pearson Education, 2014.
[15] S. Bilas, A data-driven game object system (2002) [cited Sep 10, 2018]. https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf.
[16] T. Leonard, Postmortem: Thief: The dark project (1999) [cited Sep 10, 2018]. http://www.gamasutra.com/view/feature/3355/postmortem_thief_the_dark_project.php.
[17] H. Fujibayashi, S. Takizawa, T. Dohta, Breaking conventions with the legend of zelda: Breath of the wild (2017) [cited Sep 10, 2018]. https://www.youtube.com/watch?v=QyMsF31NdNc.
[18] D. Llansó, M.A. Gómez-Martín, P.P. Gómez-Martín, P.A. González-Calero, Explicit domain modelling in video games, in: Proceedings of the 6th International Conference on Foundations of Digital Games, FDG '11, ACM, New York, NY, USA, 2011, pp. 99–106. doi:10.1145/2159365.2159379.
[19] E. Folmer, Component based game development – a solution to escalating costs and expanding deadlines?, in: Component-Based Software Engineering Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 66–73.
[20] S. George, É. Lavoué, B. Monterrat, An environment to support collaborative learning by modding, in: Scaling up Learning for Sustained Impact, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 111–124.
[21] A. BinSubaih, S. Maddock, D. Romano, A survey of 'game' portability.
[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, Wiley, Chichester, UK, 1996.
[23] G.E. Krasner, S.T. Pope, et al., A description of the model-view-controller user interface paradigm in the smalltalk-80 system, J. Object Orient. Program. 1 (3) (1988) 26–49.
[24] T. Olsson, D. Toll, A. Wingkvist, M. Ericsson, Evolution and Evaluation of the Model-View-Controller Architecture in Games, in: International IEEE/ACM Workshop on Games and Software Engineering, 2015, https://doi.org/10.1109/GAS.2015.10.
[25] J. Dormans, Engineering Emergence - Applied Theory for Game Design, Ph.D. thesis, University of Amsterdam, 2012.
[26] L.B. Morelli, E.Y. Nakagawa, A panorama of software architectures in game development, in: Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering, SEKE'2011, Eden Roc Renaissance, Miami Beach, USA, 2011.
[27] A. Ampatzoglou, I. Stamelos, Software engineering research for computer games: A systematic review, Inf. Softw. Technol. 52 (9) (2010) 888–901, https://doi.org/10.1016/j.infsof.2010.05.004.
[28] M. Zhu, A.I. Wang, H. Guo, From 101 to nnn: A review and a classification of computer game architectures, Multimedia Syst. 19 (3) (2013) 183–197, https://doi.org/10.1007/s00530-012-0274-0.
[29] S. Shi, C.-H. Hsu, A survey of interactive remote rendering systems, ACM Comput. Surv. 47 (4) (2015) 57:1–57:29. doi:10.1145/2719921. http://doi-acm-org.ez67.periodicos.capes.gov.br/10.1145/2719921.
[30] A. Yahyavi, B. Kemme, Peer-to-peer architectures for massively multiplayer online games: A survey, ACM Comput. Surv. 46 (1) (2013) 9:1–9:51. doi:10.1145/2522968.2522977.
[31] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, EBSE Technical Report EBSE-2007-01, School of Computer Science and Mathematics, Keele University, Keele - Staffs, UK, 2007.
[32] P. Brereton, B. Kitchenham, D. Budgen, M. Turner, M. Khalil, Lessons from applying the systematic literature review process within the software engineering domain, J. Syst. Softw. 80 (4) (2007) 571–583, https://doi.org/10.1016/j.jss.2006.07.009.
[33] K. Charmaz, Grounded Theory as an Emergent Method, in: Handbook of Emergent Methods, 2008, Ch. 7, pp. 155–170. doi:10.1002/9781405165518.wbeosg070.pub2.
[34] J.W. Yoder, R. Johnson, The adaptive object-model architectural style, in: Software Architecture, Springer, 2002, pp. 3–27.
[35] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.
[36] O. Pastor, J.C. Molina, Model-driven architecture in practice: a software production environment based on conceptual modeling, Springer Science & Business Media, 2007.
[37] P. Tetlow, J.Z. Pan, D. Oberle, E. Wallace, M. Uschold, E. Kendall, Ontology driven architectures and potential uses of the semantic web in systems and software engineering, W3C Working Draft.
[38] E.Y. Nakagawa, P.O. Antonino, M. Becker, Reference architecture and product line architecture: A subtle but critical difference, in: European Conference on Software Architecture, Springer, Berlin, Heidelbrg, 2011, pp. 207–211.
[39] G. Kotonya, I. Sommerville, Requirements engineering: processes and techniques, Wiley Publishing, 1998.
[40] C. Szyperski, D. Gruntz, S. Murer, Component software: Beyond object-oriented programming, Addison-Westley, Boston MA, USA, 1998.
[41] K. Beck, Test-driven development: by example, Addison-Wesley Professional, 2003.
[42] T.M. Connolly, E.A. Boyle, E. MacArthur, T. Hainey, J.M. Boyle, A systematic literature review of empirical evidence on computer games and serious games, Comput. Educ. 59 (2) (2012) 661–686, https://doi.org/10.1016/j.

compedu.2012.03.004, http://www.sciencedirect.com/science/article/pii/S0360131512000619.
[43] C. Magerkurth, A.D. Cheok, R.L. Mandryk, T. Nilsen, Pervasive games: bringing computer entertainment back to the real world, Comput. Entert. (CIE) 3 (3) (2005) 4.
[44] E. Brox, L. Fernandez-Luque, T. Tøllefsen, Healthy gaming–video game design to promote health, Appl. Clin. Inform. 2 (02) (2011) 128–142.
[45] A.I. Wang, N. Nordmark, Software Architectures and the Creative Processes in Game Development, in: K. Chorianopoulos, M. Divitini, J. Baalsrud Hauge, L. Jaccheri, R. Malaka (Eds.), Entertainment Computing - ICEC 2015, Springer International Publishing, Cham, 2015, pp. 272–285.
[46] S. Caltagirone, M. Keys, B. Schlief, M.J. Willshire, Architecture for a massively multiplayer online role playing game engine, J. Comput. Sci. Colleges 18 (2) (2002) 105–116.
[47] A.I. Wang, Extensive evaluation of using a game project in a software architecture course, ACM Trans. Comput. Educ. 11 (1) (2011) 5:1–5:28. doi:10.1145/1921607.1921612. http://doi.acm.org/10.1145/1921607.1921612.
[48] P. Gestwicki, F.-S. Sun, Teaching design patterns through computer game development, J. Educ. Resour. Comput. 8 (1) (2008) 2:1–2:22. doi:10.1145/1348713.1348715. http://doi.acm.org/10.1145/1348713.1348715.
[49] K. Claypool, M. Claypool, Teaching software engineering through game design, in: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE '05, ACM, New York, NY, USA, 2005, pp. 123–127. doi:10.1145/1067445.1067482. URL http://doi.acm.org/10.1145/1067445.1067482.