

The Software Modeling Paradox in Videogames: An Empirical Evaluation of Unreal Blueprints vs. Unity Visual Scripting

Jose Ignacio Trasobares
SVIT Research Group, Universidad San Jorge
Zaragoza, Spain
VRAIN, Universitat Politècnica de València
Valencia, Spain
jtrasobares@usj.es

Lorena Arcega
SVIT Research Group, Universidad San Jorge
Zaragoza, Spain
larcega@usj.es

África Domingo
SVIT Research Group, Universidad San Jorge
Zaragoza, Spain
adomingo@usj.es

Carlos Cetina
VRAIN, Universitat Politècnica de València
Valencia, Spain
cetina@upv.es

Abstract

Despite the fact that the leading video game engines include Model-Driven Development (MDD) principles, a paradox exists in the video game industry: while Unreal Engine's Blueprints (BP) enjoy widespread industrial adoption, Unity Visual Scripting (UVS) remains largely marginalized. To investigate the underlying causes of this phenomenon, we present a controlled empirical experiment with 31 participants. Using a mixed-methods approach, we compare both software models (BP and UVS) across quantitative performance metrics (Correctness and Efficiency) and qualitative assessments (Method Evaluation Model and focus group discussions). Results show that BP significantly outperforms UVS across all metrics. Qualitative analysis reveals that while BP provides a tailored abstraction layer and model-based debugging, UVS functions primarily as a visual wrapper for C#, inheriting the cognitive load of traditional coding without its efficiency. Additionally, the seamless bidirectional integration between Blueprint models and C++ code in Unreal mitigates the "loss of control" barrier to MDD adoption. Our results may not only inform the video game domain by identifying effective directions for key factors such as abstraction and debugging, but may also generalize to more traditional software domains where the "loss of control" adoption barrier can also emerge.

CCS Concepts

• Software and its engineering → Empirical software validation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

Keywords

Model-Driven Development, Game Software Engineering, Empirical evaluation, Experiment, Unreal, Unity, Software Modeling, Blueprints.

ACM Reference Format:

Jose Ignacio Trasobares, África Domingo, Lorena Arcega, and Carlos Cetina. 2026. The Software Modeling Paradox in Videogames: An Empirical Evaluation of Unreal Blueprints vs. Unity Visual Scripting. In . ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Over the past few decades, the video game industry has evolved from a niche market into the largest sector within the global entertainment economy. In 2025, the market's projected annual revenues exceeded \$223 billion, generating more revenue than the film and music industries combined [43]. This sustained growth has driven the formalization of Game Software Engineering (GSE), a specialized subfield at the intersection of traditional software engineering and video game development [11].

Unlike traditional software, video games require the tight integration of artistic assets and complex programming logic within a unified, iterative development pipeline shared by artists and programmers [39]. To manage this complexity and facilitate large-scale production, modern game engines like Unreal Engine and Unity have become foundational tools, adopted by approximately 82% of software developers in the video game industry [26]. The current versions of these engines incorporate the principles of Model-Driven Development (MDD) through Software Modeling. The primary goal of the software models of these engines is to democratize game development, bridging the gap between core programmers and creative profiles (game designers and animators). By translating complex syntax into visual data flows, software modeling allows multidisciplinary teams to rapidly prototype mechanics, iterate on gameplay loops, and implement behaviors.

However, despite the theoretical benefits of MDD in GSE, a paradox exists in the current industry landscape. On one hand, Unreal Engine's Blueprints (BP) have achieved widespread success, evolving into a standard practice for implementing core gameplay logic, user interfaces, and interdisciplinary collaboration in triple-A

commercial projects [23, 45]. On the other hand, Unity Visual Scripting (UVS), despite relying on the same fundamental node-based paradigm, visual connection of execution pins, and underlying computational logic, has failed to gain similar traction. In professional Unity environments, classical code-driven development utilizing traditional C# remains the dominant approach, with UVS frequently discarded during the scale-up of commercial projects [37].

A knowledge gap currently exists in the literature regarding empirical evidence to explain why Unity's visual models struggle with industry adoption while Unreal's thrive. To investigate the underlying causes of this gap, we present a controlled empirical experiment involving 31 game engine users in video game development. We compare BP and UVS using a mixed-methods approach to investigate the factors contributing to this gap. Quantitatively, we measure correctness and efficiency; qualitatively, we evaluate user satisfaction based on the Method Evaluation Model (MEM), complemented by a thematic analysis of focus groups.

Our findings show that the success of software modeling in GSE requires appropriate abstraction levels and tight engine integration, rather than merely replacing text with graphical nodes. Visual wrappers that fail to abstract underlying complexities are actively rejected by the participants. One might think that Unity's visual node-based programming approach would appeal to programmers, because of the programming aspect, and to artists because of the nodes, but our results show that the opposite is true. Furthermore, thorough debugging at the software model level has been identified as a key factor in the success of Blueprints. Finally, previous studies on MDD in video games [14, 15, 37] had suggested that increasing the level of abstraction reduced developer control compared to programming in a language like C++, but our results show that in the case of Blueprints, developers do not report this loss of control in any instance. Developers can expose C++ methods and variables for use within BP, while C++ code can seamlessly interact with elements defined in BP.

In the video game domain, our results can help reorient the UVS models of the Unity engine (the most widely used in the industry [50]) and prevent other engines from making the same mistake of assuming that a graphical wrapper around code is sufficient. Beyond video games, our results highlight the importance of seamless integration between models and code, as exemplified by BP, in mitigating the "loss of control" adoption barrier, which may also arise in other software domains.

The rest of this paper is structured as follows. Section 2 presents the related work. Section 3 provides essential background information on the evaluated engines. Section 4 details the experiment design. Section 5 presents the quantitative and qualitative results of our empirical study, followed by a discussion in Section 6. Section 7 outlines the threats to validity and our mitigation strategies. Finally, Section 8 concludes the paper and discusses future work.

2 Related Work

Model-Driven Development (MDD) in Classic Software Engineering (CSE) has been analyzed through empirical studies for many years, revealing numerous benefits. However, its application in Game Software Engineering (GSE) remains underexplored, representing a vast field yet to be fully investigated. For this reason, this section is

divided into two parts. In the first part (Subsection 2.1), we present the results of previous works that state the benefits and challenges of MDD in CSE. The second part (Subsection 2.2) presents the works on evaluating the adoption of MDD in GSE.

2.1 Model Driven Development in Classic Software Engineering

Empirical evidence in CSE demonstrates that MDD can significantly reduce development effort and increase productivity. For instance, Krogmann and Becker [34] found MDD required only 11% of the time compared to Code-Centric Development (CCD), while Kapteijns et al. [32] reported a 2.6-fold productivity increase. Tolvanen and Kelly [53] synthesized 25 years of research, arguing that Domain-Specific Modeling Languages (DSMLs) offer productivity improvements ranging from +500% to +1000%. Furthermore, studies by Heijstek and Chaudron [29], Panach et al. [40], and Domingo et al. [17] consistently highlight that MDD boosts productivity, reduces complexity, and ensures superior correctness and efficiency. Research also shows that junior developers and students feel particularly comfortable using models [17, 36, 41].

Despite these advantages, MDD adoption in real-world projects remains limited [27]. Large-scale surveys [2–4, 9, 16, 27, 28, 35, 51, 52] suggest that models are frequently used informally for documentation or communication—a practice described as "modeling as sketch" [24] rather than as integral development artifacts [6]. Grossman et al. [28], Chaudron et al. [9], and Störle [52] confirmed that modeling is primarily used to achieve consensus within teams, with development efforts still relying heavily on manual coding. This informal usage is also prevalent in specific domains like embedded systems [2, 3, 35]. Hutchinson et al. [30, 60] and Sánchez-Cuadrado et al. [13] emphasize that successful MDD adoption requires gradual integration, stakeholder commitment, and overcoming tool-related usability challenges [31].

2.2 Model Driven Development in Game Software Engineering

In the context of GSE, existing research examines how MDD is implemented and evaluates it against other methodologies. Initial studies evaluating Software Product Lines (SPLs) in video game modeling [54] demonstrated that SPL-based developments achieved higher correctness. Chueca et al. [10] extended this, showing that SPL differences were more pronounced in CCD than in MDD. From a different perspective, Roca et al. [44] examined bug localization in the software models of a video game, finding that developers reported greater satisfaction using MDD, although performance varied by experience level.

Recent research has evaluated node-based software models directly. do Prado et al. [15] and Derakhshandi et al. [14] proposed frameworks where less experienced developers could create games faster using visual models, though this approach reduces detailed control over the code. This trade-off is central to current game engines. Martinez et al. [37] compared MDD with CCD for extended reality environments using BOLT. They concluded that while CCD provides detailed control, MDD offers a more intuitive approach, with model-based programming having limitations in expressing complex algorithms. Conversely, Trasobares et al. [56]

conducted a controlled experiment with 31 industry professionals comparing Unreal Engine's BP versus C++. They demonstrated that constrained visual structures significantly increased software correctness, consolidating models as effective development artifacts rather than just documentation.

As evidenced in the literature, Unreal Engine has established itself as an industrial success story in the adoption of MDD. Unity incorporates analogous frameworks, but these approaches have not managed to replicate Unreal BP's success. Consequently, there is a knowledge gap regarding sufficient empirical evidence to explain why Unity's implementation has failed to establish itself as an industry approach. From a complementary, sociological standpoint, Foxman [25] analyzes Unity as a platform whose tooling fosters technological lock-in among its user base, helping explain why Unity's adoption became so widespread despite the limitations of tools such as the one this study examines empirically. To bridge this gap, this paper presents a controlled empirical experiment involving 31 game engine users to directly compare Unreal's Blueprints (BP) and Unity Visual Scripting (UVS). Unlike previous works that either evaluate academic MDD frameworks or compare models versus code within a single environment, our study uses a mixed-methods approach to empirically evaluate the two leading industry engines against each other.

3 Background

GSE requires the seamless integration of artistic assets and programming logic within a unified development pipeline [39]. To manage this, approximately 82% of software developers utilize game engines [26]. A game engine streamlines production by providing essential functionalities such as rendering graphics, simulating physics, managing memory, and abstracting platform-specific complexities, making them a compelling choice for multi-platform projects. Understanding how these engines facilitate large-scale production requires examining both the underlying development paradigms they employ and the specific multidisciplinary tasks that practitioners execute daily.

Currently, Unreal Engine and Unity stand as the two most widely adopted engines in professional environments [19, 57], both integrating Model-Driven Development (MDD) principles through Software Modeling.

Unreal Engine, developed by Epic Games, operates on a core architecture written in C++ [19]. Structurally, it utilizes an object-oriented model based on the Actor class hierarchy, where behavior is modularized through ActorComponents. Unreal Engine incorporates Blueprint (BP), a Domain-Specific Modeling Language (DSML) that enables the implementation of game logic through a node-based interface rather than traditional text-based code [21]. Evolving from the linear "Kismet" in Unreal Engine 3, Epic Games redesigned the software model for Unreal Engine 4 into a comprehensive object-oriented programming approach [47]. According to Epic's official documentation, this node-based workflow was conceived to provide "designers with a wide range of scripting concepts and tools that are generally only available to programmers", while Blueprint-specific markup in the C++ implementation lets "programmers create baseline systems that designers can extend" [22]. Operationally, developers construct behaviors within an

Event Graph by connecting nodes that represent events, functions, and variables. The engine's compiler translates these visual graphs into virtual machine (VM) bytecode that executes natively during runtime [21].

Within the industry, BP have achieved widespread adoption and success, functioning as a primary approach for defining core software architecture and logic in commercial projects [45]. Its market presence is extensive, powering nearly 80 of the most anticipated games released in 2023 [23], supporting massive titles like Fortnite with over 235 million monthly users [1], and expanding into industries such as film, architecture, and simulation [20].

Conversely, Unity Technologies developed the Unity engine utilizing a component-based architectural paradigm centered around GameObjects and MonoBehaviour components, with C# serving as the primary programming language [57]. The engine utilizes Unity Visual Scripting (UVS). Originally emerging as a third-party asset named "Bolt," Unity acquired and integrated it natively into the engine starting with the 2021 release [37]. Operationally, UVS allows developers to structure logic using "Script Graphs" to govern execution flow and "State Graphs" for finite state machines. To execute this logic, UVS dynamically generates nodes from existing C# assemblies and employs runtime reflection to invoke C# methods, or utilizes Ahead-Of-Time (AOT) compilation to translate visual connections into the underlying engine API [59]. Unity states an analogous goal for UVS, describing it as a way to "bring the power of Unity scripting to non-coders and future coders" that "empowers non-coders to extend or modify C# scripts" and grants them "access to the entire Unity Scripting API" [58].

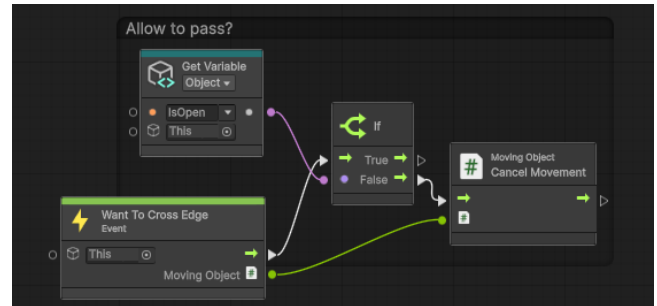


Figure 1: UVS example screenshot.

Figure 1 illustrates the core mechanics of node-based software modeling using Unity Visual Scripting (UVS) as a representative example. The system relies on a visual interface where developers construct logic by connecting nodes through execution wires (white lines) and data wires (colored lines). In this specific scenario, a "Want To Cross Edge" event initiates the execution flow. The flow proceeds to an "If" control node, which evaluates a boolean variable ("IsOpen"), retrieved via a "Get Variable" node. If the condition evaluates to false, the execution branches to a "Cancel Movement" node, halting the specific "Moving Object" whose reference is passed as data directly from the initial event. If the condition is true, the execution simply terminates. This structural paradigm, relying on the visual connection of execution pins for control flow and data pins for variable passing, is visually analogous to Unreal Engine's Blueprints.

Despite UVS and BP sharing the same node-based approach, their industry adoption diverges. In professional Unity environments, classical code-driven development utilizing C# remains the dominant standard for implementing core architecture [37]. Unlike BP in Unreal, UVS has not achieved the same level of reliance or success as a primary development artifact. Nevertheless, its market penetration is similarly significant, serving as the foundation for complex titles like *Genshin Impact* (with over 60 million monthly active players) [48] and powering 71% of the top 100 highest-grossing mobile games in 2023 [42]. Thus, both vendors state essentially the same design intent, lowering the barrier for non-programmers while preserving a bridge to the underlying language, yet, as our results show, only Blueprints realizes it in practice.

4 Experiment design

4.1 Objectives

To define objectives, our research follows the Goal Question Metric (GQM) model, initially introduced by Basili and Rombach [7]. Our goal is to **analyze** different Software Models, **for the purpose of** comparison, **with respect to** the correctness of the requested tasks, efficiency, and user satisfaction, **from the point of view of** game engines users, **in the context of** developing software for a video game.

4.2 Variables

In this study, the factor under investigation is the **Software Model** (SM), with two alternatives: using UVS or using BP to complete a representative task. The goal of this experiment is to evaluate the effects of different SMs when developing a representative task using two different game engines, Unity and Unreal. To assess this, we selected *Correctness* and *Efficiency* as the objective dependent variables, which are directly related to performance.

Correctness was measured using a marking rubric, which was applied to the tasks developed by participants after completing the experiment. The scores range from 0 to 100, representing the percentage of points achieved according to the marking rubric.

Efficiency was calculated as the ratio of the achieved *Correctness* to the time spent completing the task. Time was recorded as the total duration (in minutes) each participant required to complete the assigned task. However, time is not analyzed as an independent variable because *Efficiency* is a direct derivative of it, effectively encapsulating the temporal aspect of performance. *Efficiency* represents the rate of accurate work produced per minute and is defined as: $\text{Efficiency} = \text{Correctness} / \text{time}$

To prevent mathematical distortions (e.g., artificially high efficiency scores for subjects who abandoned the task early with near-zero correctness), the time was standardized to the maximum limit of 30 minutes for participants who failed to complete the core mechanics.

Additionally, we examined the SMs in terms of *Satisfaction*, using a 5-point Likert-scale questionnaire (1 = Strongly Disagree, 5 = Strongly Agree) based on the (MEM) [38], an adaptation of the Technology Acceptance Model (TAM) for software engineering methods. We broke down *Satisfaction* into three subjective dependent variables as follows:

- **Perceived Usefulness (PU)**: The degree to which a person believes that using a specific SM would enhance performance.
- **Perceived Ease of Use (PEOU)**: The degree to which a person believes that learning and using a specific SM would require minimal effort.
- **Intention to Use (ITU)**: The degree to which a person intends to use a specific SM.

Each of these variables corresponds to specific items in the questionnaire. The scores for these items were averaged to obtain the final value for each variable.

4.3 Design

The experimental design is a between-subjects design with one factor and two treatments [18, 61]. The independent factor is the SM, and the two treatments are UVS and BP. Each subject only uses one treatment of the factor, as participants were assigned to one of two groups based on their prior engine expertise (Unity users or Unreal Engine users). This expertise-based grouping ensured engine-specific data collection and minimized the confounding variable of learning a completely new game engine interface, allowing participants to focus solely on the software modeling mechanics [33].

All subjects performed a single, standardized experimental task, which consisted of implementing a specific set of mechanics provided by the researchers. By keeping this main task, the execution time limit (30 minutes), and the evaluation method (15-minute guided focus group and 10 minutes to complete the questionnaire) strictly constant across all sessions, we established a controlled environment [61]. This allowed us to qualitatively compare the two treatments.

This between-subjects design enhances the external validity and industrial relevance of the experiment [49]. Observing developers working within their domain of expertise provides a realistic assessment of how these tools are perceived by actual industry practitioners, a crucial aspect when evaluating software engineering techniques and models in real-world industrial contexts [18, 49]. Furthermore, by matching the treatment to the user's background, the design prevents severe fatigue and frustration effects that could act as confounding variables if participants were forced to complete a time-bound programming task in an unfamiliar engine [33].

4.4 Research questions and hypotheses

The research questions and null hypotheses are formulated as follows:

RQ1 - Does the **Software Model** used impact the *Correctness* of software developed for video games? The corresponding null hypothesis is $H_{0,C}$: The **Software Model** used does not have an effect on *Correctness*. **RQ2** - Does the **Software Model** used in Unity influence the *Efficiency* of developers when creating software for video games? The null hypothesis for *Efficiency* is $H_{0,E}$: The **Software Model** used does not have an effect on *Efficiency*. **RQ3** - Does user satisfaction differ when video game software is developed using different **Software Models**? To answer this question, we formulated three hypotheses based on the variables *Perceived*

Table 1: Participant demographics categorized by assigned environment (Unreal vs. Unity).

Environment	Age Group						Main Role		Dev. Hours per Day ($\mu \pm \sigma$)			Prior Knowledge (1–5, $\mu \pm \sigma$)		
	<25	25-29	30-34	35-39	40-44	>44	Dev.	Artist	Game	Modeling	Coding	Coding	Modeling	Mod. Lang.
Unreal (N=16)	0	9	5	0	0	2	12	4	5.7 $\pm$ 2.8	4.1 $\pm$ 3.0	3.1 $\pm$ 3.2	3.44 $\pm$ 1.31	2.94 $\pm$ 1.18	3.50 $\pm$ 1.32
Unity (N=15)	7	5	1	1	0	1	12	3	2.2 $\pm$ 2.4	0.3 $\pm$ 0.6	4.2 $\pm$ 2.2	4.13 $\pm$ 0.74	3.07 $\pm$ 0.96	2.07 $\pm$ 0.96
Total (N=31)	7	14	6	1	0	3	24	7	4.0 $\pm$ 3.1	2.3 $\pm$ 2.9	3.6 $\pm$ 2.8	3.77 $\pm$ 1.12	3.00 $\pm$ 1.06	2.81 $\pm$ 1.35

Usefulness, Perceived Ease of Use, and Intention to Use, with their corresponding null hypotheses:

- $H_{0,PU}$: The **Software Model** used does not have an effect on *Perceived Usefulness*.
- $H_{0,PEOU}$: The **Software Model** used does not have an effect on *Perceived Ease of Use*.
- $H_{0,ITU}$: The **Software Model** used does not have an effect on *Intention to Use*.

4.5 Participants

The population of this study consists of 31 participants with experience in video game development. Given that modern game development relies heavily on comprehensive game engines, evaluating subjects who actively use these environments ensures that our comparative findings between Unity and Unreal are grounded in real-world usage patterns. Furthermore, according to established systematic reviews of controlled experiments in software engineering, such as the seminal work by Sjöberg et al. [49], a sample size of around 30 participants aligns with approach practices and provides adequate statistical power.

Before the experiment, subjects completed a demographic questionnaire designed to characterize the sample and capture potential confounding variables [33]. The results are summarized in Table 1. A total of 16 subjects were assigned to the Unreal group and 15 to the Unity group. To ensure a comprehensive characterization, subjects were classified by their primary roles, yielding 24 developers and 7 artists. Documenting this multidisciplinary nature is relevant in GSE studies, as the interplay between technical and creative profiles differentiates game development from traditional software engineering [39, 56].

Importantly, the distribution of age and roles is well-balanced across both the Unity and Unreal groups, thereby mitigating potential selection threats.

Furthermore, the questionnaire captured the participants' average dedication (in hours) to various technical tasks as a proxy for their practical proficiency.

Analyzing the data by environment reveals that the Unreal group reported a notably higher average dedication to Software Modeling (4.1 $\pm$ 3.0) compared to the Unity group (0.3 $\pm$ 0.6). Conversely, the Unity group exhibited a higher mean in pure Coding hours (4.2 $\pm$ 2.2 versus 3.1 $\pm$ 3.2 for Unreal). These differences align with the typical workflows inherent to each engine's ecosystem.

4.6 Experimental objects

Video game development is a highly multidisciplinary process that bridges the gap between traditional software engineering and digital arts. In Game Software Engineering (GSE), the development

pipeline involves specialized daily tasks that can be broadly grouped into three categories [39]: (1) *Game Development (Logic and Mechanics)*, which encompasses gameplay rules, player controls, and behaviors; (2) *Coding (Core Programming)*, focusing on lower-level architecture, performance optimization, and memory management using traditional text-based languages; and (3) *3D Modeling and Art Integration*, which involves configuring materials, physical collisions, and integrating visual assets into the engine ecosystem.

To properly evaluate and compare the effectiveness of software modeling in modern game engines, the experimental task must align with the category where these visual tools are primarily intended to be utilized: *Game Development (Logic and Mechanics)*. Therefore, the experimental object of this study consists of a single, highly representative task extracted from previous literature on software models in video games [56]: the implementation of an input processing component that enables character movement based on user inputs from a keyboard or controller.

This task is highly representative for several reasons. First, handling player input and translating it into logical character movement is a foundational mechanic required in almost all video games, making it a standard daily task designed to allow rapid prototyping and quick iteration by both software developers and game designers. Second, it forces developers to translate physical human interaction into continuous mathematical transformations, interacting directly with the engine's core physics components. Rather than a trivial logical check, it reveals how each environment handles data flow and vector math under continuous execution, exposing the potential for visual clutter (spaghetti code) in a foundational scenario.

To ensure a balanced difficulty level and strict relevance to real-world game development, the task was explicitly selected by an expert panel (comprising two independent senior industry developers and two academic researchers). The implementations of this task across the different Software Models were carried out in collaboration with professional video game developers and subsequently reviewed by external professional developers to guarantee consistency and correctness. To maintain strict experimental control, all evaluated methods were designed to be functionally identical, executing the exact same sequence of operations. This task is logically divided into two main parts: first, obtaining the rotation, which involves calculating the forward and right directions based on the controller's current rotation; and second, moving the object, which applies the user's axis input values to move the character along those calculated directions. The complete set of experimental objects used in this study is available in the replication package at <https://doi.org/10.5281/zenodo.19254651>.

4.7 Instrumentation

To accurately measure the dependent variables and collect participant feedback, several instruments were designed and utilized during the experiment. First, to evaluate the *Correctness* of the submitted tasks, a standardized marking rubric was established by the expert panel. This rubric allowed researchers to objectively assign a score from 0 to 100 based on the presence and proper configuration of the required logic nodes and connections. For qualitative data collection and subjective variable measurement, we developed a structured questionnaire using Microsoft Forms, which was divided into the following sections:

- I 'Patient Information Sheet' that subjects must review and voluntarily accept. It clearly explains what the experiment consists of and how personal data will be handled. All questionnaire responses and task solutions were anonymized
- II Demographic questionnaire to characterize the sample, capture their primary roles (Developer or Artist), and record their average dedication to various development tasks.
- III Specific questionnaire based on the Method Evaluation Model to collect the observations of the subjects and their satisfaction (*PU*, *PEOU*, and *ITU*) with the approach and the method used in the experiment.
- IV Final open-ended section to collect any additional qualitative observations or feedback that the subjects may wish to provide regarding their experience.

All instruments utilized in this study, including the detailed task descriptions, the marking rubric, the forms used for the questionnaires, the raw demographic and statistical results, and the analysis of the subjects' opinions from the focus groups, are fully available in the same replication package referenced in Section 4.6.

4.8 Experimental procedure

The experiment was conducted across seven sessions: six were held online via Microsoft Teams, and one was conducted in person.

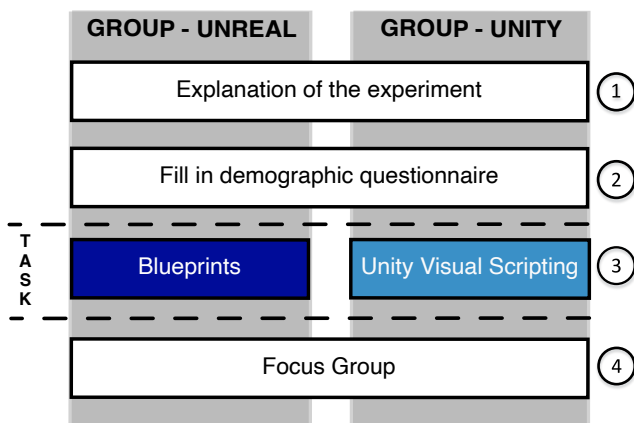


Figure 2: Experimental procedure flowchart.

To ensure engine-specific data collection, the sessions were organized by expertise, consisting of four sessions for Unity developers and three sessions for Unreal Engine developers. Participants in

the online sessions joined via Microsoft Teams, where the chat feature was used to share information, distribute links, and answer questions. Each session was scheduled to last for 70 minutes and followed the experimental procedure outlined in Figure 2, described as follows:

- (1) **Explanation of the experiment:** (15 minutes) The experiment began with a general explanation of the study, introducing subjects to its objectives, structure, and expectations. An instructor explained the different stages and explicitly clarified that the experiment was not an evaluation of individual programming abilities. Subjects received detailed instructions on the task sequence and the focus group dynamics.
- (2) **Fill in demographic questionnaire:** (10 minutes) All subjects completed a form designed to collect demographic and background information. This questionnaire assessed their age, specific industry roles, and their average dedication to specific game development tasks.
- (3) **Task:** (30 minutes) Following the questionnaire, subjects proceeded to solve the main development task. As illustrated in the experimental design workflow, the sample was split into two independent tracks based on their engine expertise: **GROUP - UNREAL** performed the task using **Blueprints**, while **GROUP - UNITY** completed the exact same task using **Unity Visual Scripting**. Instructors were available throughout the stage to resolve any technical blockers or environment-related doubts.
- (4) **Focus Group:** (15 minutes) Finally, once the task was submitted, the participants of the session took part in a guided focus group discussion. This allowed them to openly discuss their experiences with the respective software modeling approach used. The discussion covered key aspects such as ease of use, logic flow clarity, efficiency, and the perceived applicability of the method for real-world game development scenarios. During these sessions, at least two researchers acted as independent observers, systematically taking notes on the participants' comments and reactions. Following the sessions, these independent notes were aggregated and synthesized using a basic thematic analysis approach to identify recurrent pain points and workflow differences.

4.9 Analysis Procedure

The statistical data analysis was conducted to evaluate the effects of the **Software Model** (BP vs. UVS) on the dependent variables: *Correctness*, *Efficiency*, *PU*, *PEOU*, and *ITU*. To ensure mathematical rigor and replicability, the analysis was followed a structured four-step procedure [61], which considers all underlying statistical assumptions:

- (1) **Exploratory Data Analysis:** Descriptive statistics, including the mean (μ) and standard deviation (σ), were calculated to summarize the metrics across the two experimental groups.
- (2) **Normality Check:** To determine the appropriate inferential statistical approach, the assumption of normality for the continuous dependent variables was evaluated. Because the

sample sizes for the independent groups were small ($n_1 = 16$ and $n_2 = 15$), the Shapiro-Wilk test was employed [61].

- (3) **Effect Size Calculation:** To quantify the practical significance and the magnitude of the differences observed between the two methods, standardized effect sizes were computed. We calculated Cohen's d using the pooled standard deviation (σ_p) [12]. The results were interpreted following standard statistical guidelines: values around 0.2 denote a small effect, values around 0.5 indicate a medium effect, and values ≥ 0.8 represent a large effect.
- (4) **Hypothesis Testing Selection & p-value Calculation:** Based on the outcomes of the normality checks, the appropriate comparative statistical method was selected to perform the formal hypothesis testing. Since the data significantly deviated from a normal distribution ($p \leq 0.05$), non-parametric alternatives were required [5]. Consequently, we utilized the Mann-Whitney U test. The U statistic for each group is calculated by first ranking all observations combined, and then computing the sum of ranks for each group. Each group's rank sum produces one U value, and the final U statistic is the minimum of these two values. To compute the p -value given our sample size ($N \geq 20$), the U statistic is approximated to a standard normal distribution. This involves calculating a Z -score, which represents the deviation of our observed U statistic from the expected mean of U , divided by the expected standard deviation. Finally, the corresponding two-tailed p -value is derived from this Z -score using the standard normal cumulative distribution function.

5 Results

This section presents the quantitative results obtained from the empirical evaluation comparing the two **Software Models**: BP and UVS. The statistical analysis is structured into four main phases: an initial exploratory data analysis to describe the sample and variables, an assessment of data normality to select the appropriate statistical tests, the calculation of effect sizes to quantify the magnitude of the observed differences, and finally, the non-parametric hypothesis testing conducted to address the formulated Research Questions (RQs). Figure 3 presents, for each dependent variable (Correctness, Efficiency, PU, PEOU, and ITU), the per-group histograms with overlaid normal-distribution curves and the corresponding box-plots.

5.1 Exploratory Data Analysis

Table 2 shows the values for the mean (μ) and standard deviation (σ) of all the variables considered for each of the methods compared: BP and UVS. The total sample size was $N = 31$, divided into subjects using BP ($n = 16$) and subjects using UVS ($n = 15$).

The data reveals substantial differences in the means and standard deviations of all dependent variables depending on the **SM** used, consistently favoring BP. In terms of performance, the box-plots for *Correctness* (%) and *Efficiency* demonstrate that BP users achieved higher median scores, despite a broader variability in *Efficiency* that includes a noticeable outlier. Similarly, the subjective satisfaction metrics display a clear upward shift for the BP group,

Table 2: Values for the Mean and Standard Deviation of the Response Variables by Specification Method

Dependent Variable	Blueprints $\mu \pm \sigma$	Unity Visual Scripting $\mu \pm \sigma$
<i>Correctness</i>	61.61 ± 36.23	32.94 ± 23.51
<i>Efficiency</i>	6.52 ± 8.37	1.05 ± 0.72
<i>PU</i>	3.50 ± 1.42	2.00 ± 0.86
<i>PEOU</i>	3.70 ± 1.13	2.25 ± 0.76
<i>ITU</i>	3.66 ± 1.16	1.57 ± 1.00

whose medians are consistently closer to the maximum score of 5.0. In contrast, the UVS group's scores are concentrated toward the lower end of the scale, with isolated upper outliers visible in the *PEOU* and *ITU* distributions. The inclusion of theoretical normal distribution curves over the histograms in Figure 3 highlights the data density and its deviation from normality, visually emphasizing the stark contrast in how the two engines were received.

5.2 Normality Check

To determine the appropriate statistical tests for comparing the two alternatives of the **SM**, we verified the normality of the distributions. The results are detailed in Table 3 (see Blueprints and Unity Visual Scripting p -value columns).

Table 3: Shapiro-Wilk Test Results for Normality by Software Model

Dependent Variable	Blueprints p -value	Unity Visual Scripting p -value	Cohen's d
<i>Correctness</i>	0.020	0.034	0.932
<i>Efficiency</i>	< 0.001	0.172	0.907
<i>PU</i>	0.027	0.070	1.269
<i>PEOU</i>	0.046	0.819	1.496
<i>ITU</i>	0.06	< 0.001	1.908

The test indicated that all variables significantly deviated from the normality assumption ($p < 0.05$) in at least one of the experimental groups. Consequently, to maintain statistical consistency across the entire analysis, non-parametric tests were uniformly selected to conduct the hypothesis testing.

5.3 Effect Size (Cohen's d)

To quantify the differences in the dependent variables due to the **SM** considered, we calculated the Cohen's d values (using the pooled standard deviation). Conceptually, the effect size is inversely related to the distributional overlap between two groups: a larger d -value indicates less overlap and a clearer separation of their data.

As presented in Table 3 (see Cohen's d column), the effect size of the differences due to the **SM** is large across all dependent variables. Large effects in favor of BP were found for *Correctness* ($d = 0.932$), *Efficiency* ($d = 0.907$), *PU* ($d = 1.269$), *PEOU* ($d = 1.496$), and *ITU* ($d = 1.908$).

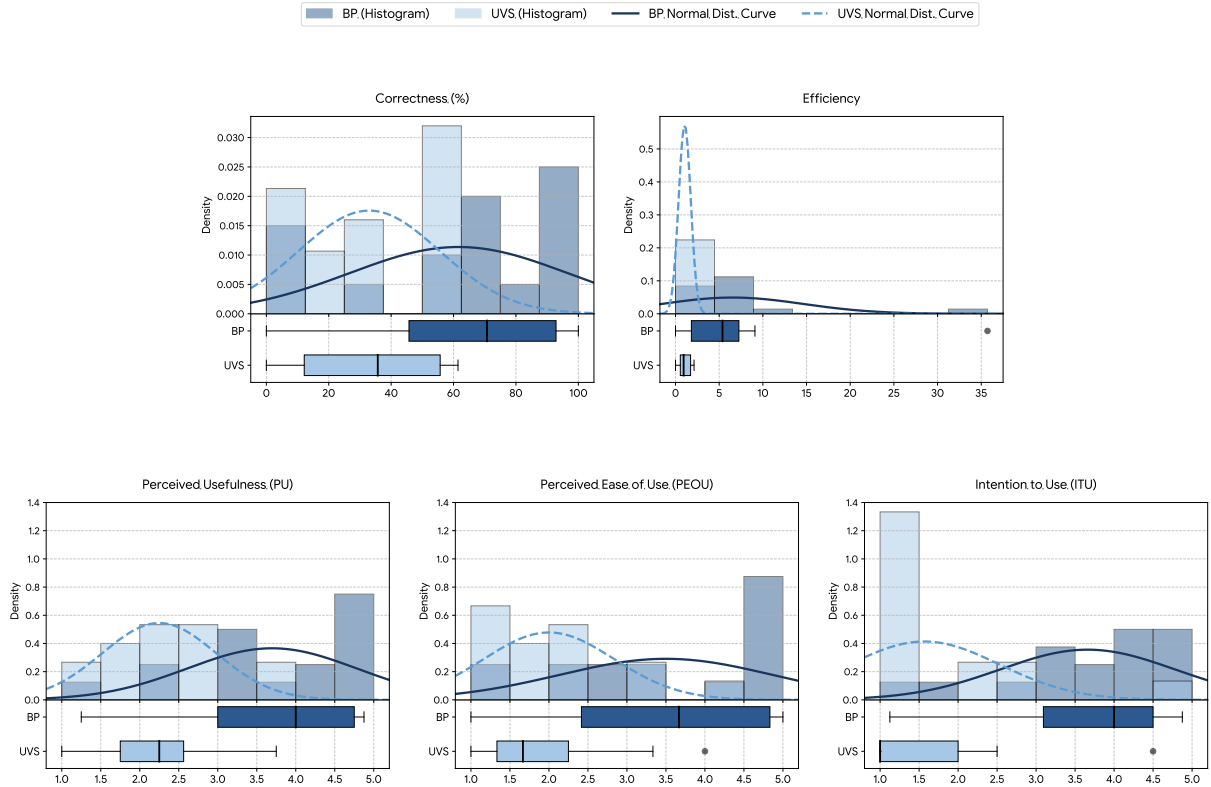


Figure 3: Histograms and boxplots illustrating the differences in *Correctness*, *Efficiency*, and *Satisfaction* between BP and UVS.

5.4 Non-Parametric Model Statistics and Hypothesis Testing

To formally test the hypotheses and answer the formulated research questions, the Mann-Whitney U test for independent samples was employed. The summary of the test statistics, including the standardized Z-scores, is provided in Table 4.

Regarding **RQ1**, the statistical analysis reveals significant differences in *Correctness* ($U = 63.00$, $Z = -2.259$, $p = 0.024$). Consequently, we reject the null hypothesis ($H_{0,C}$), confirming that the **Software Model** impacts the correctness of the developed software, with Blueprints demonstrating superior performance.

To address **RQ2**, the analysis indicates that the differences in *Efficiency* are also statistically significant ($U = 49.00$, $Z = -2.812$, $p = 0.005$). We therefore reject the null hypothesis ($H_{0,E}$), concluding that the choice of **Software Model** significantly influences developer efficiency, again favoring Blueprints.

Finally, for **RQ3**, we analyzed the three variables related to user satisfaction. The tests yielded p -values below the 0.05 threshold across all metrics: *Perceived Usefulness* ($U = 47.00$, $Z = -2.895$, $p = 0.004$), *Perceived Ease of Use* ($U = 37.50$, $Z = -3.267$, $p = 0.001$), and *Intention to Use* ($U = 20.00$, $Z = -4.023$, $p < 0.001$). Consequently, we reject the null hypotheses $H_{0,PU}$, $H_{0,PEOU}$, and $H_{0,ITU}$. These results confirm that user satisfaction differs significantly between the two approaches, with subjects reporting higher satisfaction and technological acceptance when using BP compared to UVS.

Table 4: Mann-Whitney U Test Results for the Specification Method Factor

Dependent Variable	Mann Whitney U	Z	p-value
<i>Correctness</i>	63.00	-2.259	0.024
<i>Efficiency</i>	49.00	-2.812	0.005
<i>PU</i>	47.000	-2.895	0.004
<i>PEOU</i>	37.50	-3.267	0.001
<i>ITU</i>	20.00	-4.023	< 0.001

6 Discussion

The results of our empirical evaluation reveal a clear difference between the two software modeling tools. Despite sharing the same fundamental node-based paradigm, Unreal Engine's Blueprints (BP) outperformed Unity Visual Scripting (UVS) across all objective and subjective metrics. Quantitatively, BP yielded much higher *Correctness* ($p = 0.024$, $d = 0.932$) and *Efficiency* ($p = 0.005$, $d = 0.907$), alongside superior user satisfaction in terms of *Perceived Ease of Use*, *Perceived Usefulness*, and *Intention to Use* (all yielding large effect sizes, $d > 1.2$). The qualitative data collected from the focus groups provides critical insights into why this discrepancy exists, pointing to three fundamental differences: the level of abstraction, engine integration and debugging, and target audience alignment

(supporting quotes in the Focus Group Report of the replication package).

6.1 The Abstraction Level and Cognitive Load

One of the primary reasons UVS fails to match BP's efficiency is its approach to abstraction. The quantitative data reflects a strong "floor effect" for Unity users, with a mean *Correctness* of merely 2.94%. This indicates that within the 30-minute timeframe, the majority of UVS participants were entirely blocked and failed to establish even the foundational logic.

Qualitative feedback suggests that this occurs because UVS acts largely as a 1-to-1 visual wrapper for C#, rather than providing higher-level macro functionalities. Participants explicitly reported that simple operations, such as basic mathematics or conditional statements (IF), require an excessive number of nodes in UVS, leading to visual fatigue and "spaghetti" graphs. Developers noted that what takes a single line of code in C# becomes a complex web of graphs in UVS, making it slower and more frustrating to use. This outcome directly contradicts Unity's stated design intent of granting non-coders access to the full Scripting API: the very breadth of that low-level access is what overwhelms the non-programmer audience UVS was meant to serve.

Furthermore, this forces an unnatural cognitive process for experienced developers. The focus groups revealed that programmers approach UVS by first thinking of the C# solution and then attempting to translate it into nodes. In contrast, BP provides a higher level of abstraction tailored for gameplay mechanics, which reduces visual clutter.

6.2 The Debugging Disconnect

Our findings show that UVS presents major debugging issues, rooted in how each engine processes the models. Participants reported that when an error occurs in UVS, the engine's console does not point to the problematic visual node. Instead, it redirects the user to the generated C# code. This is like writing in a high-level language but debugging in assembly code. Because developers created the logic using visual models, being forced to locate and fix a bug within auto-generated C# code, which they did not manually write and which does not map one-to-one with their visual mental model, breaks their mental model. This destroys the abstraction of software modeling and makes bug localization much harder. Additionally, users noted a disconnect between the UVS hierarchy and Unity's approach scene hierarchy, making navigation confusing.

Conversely, BP is natively integrated into Unreal. When a BP fails, the exact visual node itself is highlighted, keeping the developer within the model-driven context and making bug localization straightforward. Participants acknowledged that this native integration provides a reliable "sandbox" environment, where artists and designers feel safe prototyping without the fear of crashing the engine, a security that C++ lacks but BP successfully provides.

6.3 The Illusion of Accessibility: Programmers and nonprogrammers

Finally, qualitative observations suggest a mismatch between UVS and its intended target audience. Software modeling in video games is theoretically designed to bridge the gap between programmers

and non-programmers (such as artists and game designers). According to the thematic analysis of the focus groups, less technical profiles, such as the artists in our sample, expressed that UVS still heavily assumes deep programming knowledge, requiring users to understand C# concepts, variable types, and object-oriented architectures ("it assumes you know something about programming").

While our sample size of artists ($n = 7$) requires cautious interpretation, the consensus during discussions was that because UVS requires programming knowledge but removes the speed and efficiency of writing raw C#, it alienates both ends of the spectrum: programmers reject it because writing C# is fundamentally faster and more readable, and artists report finding it too technically demanding and poorly documented compared to the readily available C# resources.

Unreal, conversely, has managed to position BP as a highly effective middle ground. While C++ is reserved for core architecture and performance-critical systems, BP is a standard as the approach for gameplay logic, UI, and asset integration, facilitating genuine interdisciplinary collaboration ("UI? Always in BP", "Shaders and AI are easier in BP", whereas "State machines are better in UVS"). A crucial factor enabling this synergy is the seamless bidirectional integration between Blueprints and C++ [21, 45]. Developers can readily expose C++ methods and variables to be accessed and modified within BP, and conversely, C++ code can easily interact with elements defined in BP.

This architecture allows non-programmers to focus strictly on visual scripting within BP, while programmers can comfortably alternate between both paradigms depending on the specific needs of the task. As a result, none of the participants using Unreal expressed feeling a "loss of control" over the system. This finding is significant because previous literature studying software modeling in video games has frequently identified this perceived loss of control as a primary barrier to the adoption of Model-Driven approaches [11, 44]. However, those prior studies did not specifically evaluate Unreal's Blueprints. Therefore, this high degree of integration and retained developer control appears to be a defining factor in the widespread success of BP modeling in the industry.

7 Threats to Validity

To present the threats to validity, we use the classification proposed by Runeson and Höst [46]. This classification describes the following four aspects:

Construct Validity: This aspect defines the extent to which the operational measures under study truly represent the researcher's objective. A potential threat is *mono-operation bias*, which occurs when the experiment relies on a single treatment or task. This study was affected by this threat since all subjects performed a single, standardized experimental task consisting of an input processing mechanic. While generalizing to all types of video game mechanics (e.g., complex AI behavior trees) requires caution, this specific task provides a robust, representative baseline for evaluating continuous data flow and math operations in engines. To mitigate *mono-method bias*, we combined objective performance metrics (*Correctness* and *Efficiency*) with validated subjective metrics (MEM questionnaire) to capture a comprehensive view of the Software Models [38].

Finally, to mitigate *evaluation apprehension*, where participants alter their behavior out of fear of being judged, instructors explicitly clarified at the beginning of each session that the experiment was not an evaluation of their individual programming abilities. Furthermore, to mitigate *social desirability bias* and post-frustration rationalization during the focus groups, particularly for UVS users who experienced a floor effect, moderators maintained a strictly neutral stance, requiring participants to anchor their complaints to specific interface interactions rather than emotional assessments.

Internal Validity: This aspect is important when causal relationships are examined, ensuring that the investigated factor is not affected by confounding variables. The *selection threat* arises when experimental results depend on the specific characteristics of the subjects. To mitigate this, participants were assigned to groups based on their prior engine expertise (Unity users or Unreal Engine users). This minimized the confounding variable of learning a completely new interface. As noted in the demographic data, the Unity group showed a higher propensity for pure coding, whereas the Unreal group leaned towards modeling and hybrid tasks. Rather than a pure bias, this reflects the ecological reality of these engines' respective communities. Forcing pure C# developers to use visual nodes in Unity naturally induces friction, but it precisely highlights UVS's failure to adapt to or seduce its own native user base. While there is an inherent asymmetry in approach familiarity, Unreal users are naturally more experienced with BP than Unity users are with UVS. This reflects the exact ecological reality and paradox that our study investigates. Finding professional Unity developers with thousands of hours in UVS is practically impossible because the industry inherently rejects it.

Maturation and fatigue effects were mitigated by setting a strict execution time limit of 30 minutes and matching the treatment to the user's background, preventing severe frustration. Furthermore, an *instrumentation threat* could arise from poorly designed tasks. We mitigated this by having an expert panel select the task and external professional developers review the implementations to guarantee consistency and correctness across both engines.

External Validity: This aspect refers to the extent to which the results can be generalized outside the experiment's specific setting. A *statistical power threat* arises when the sample size limits generalization. Although a sample size of $N = 31$ requires cautious generalization, our demographic accurately reflects the current landscape of active practitioners in Game Software Engineering, aligning with standard empirical practices in software engineering.

An *interaction of setting and treatment threat* was mitigated by conducting the experiment within actual, widely adopted industrial environments (Unity and Unreal Engine 5) rather than toy environments, and by selecting a foundational, highly representative mechanic (input processing) that is required in almost all interactive software. However, a *domain threat* remains, as our findings are strictly contextualized within the video game development domain.

Reliability: This aspect ensures that the results are independent of the specific researchers conducting the study, meaning that the study could be replicated with the same outcomes. To mitigate *researcher bias* during the evaluation of the tasks, Correctness was not assessed subjectively; instead, a standardized correction template (rubric) established by the expert panel was used to objectively assign scores from 0 to 100.

The *data collection threat* was mitigated by following a strict, standardized experimental procedure across all seven sessions (both online and in-person) to establish a controlled environment.

Finally, to ensure full reproducibility and mitigate any *incomplete data threats*, all instruments utilized in this study, including task descriptions, correction templates, raw demographic data, and focus group analyses, are fully available in our replication package.

8 Conclusion

This paper shows that merely replacing text with visual nodes is insufficient for video games; raising the level of abstraction is required to hide underlying technical complexities. Ultimately, this study provides empirical evidence regarding software models in game engines as follows.

Our results show that UVS fails because it functions primarily as a visual wrapper for C#, inheriting the cognitive load of traditional coding without its efficiency. Furthermore, its disconnected debugging paradigm, where errors are flagged in the generated code rather than the visual models used to create them, breaks the model-driven abstraction, making bug localization frustrating and alienating its intended audience.

Conversely, Unreal Engine's Blueprints (BP) succeed by offering a tailored level of abstraction, robust visual debugging that points directly to the failing nodes, and a safe sandbox environment. Crucially, BP allows for seamless integration with C++, enabling bidirectional access where BP can invoke C++ methods and variables, and C++ can interact with BP elements [45]. This architecture allows non-programmers to concentrate entirely on BP, while programmers can leverage both BP and C++ depending on the task. Consequently, no participant reported the "loss of control" typically associated with software modeling in video games a common limitation identified in previous studies that had not evaluated BP [11, 44]. This combination of abstraction with control and the model-based debugging are defining factors, empowering both non-programmers and developers to build collaboratively.

As future work, we will apply phylogenetics, the reconstruction of ancestor, descendant lineages, recently used in search-based software engineering [8] and for reuse in video games [55], to the modeling concepts of BP and UVS, tracing how high-level nodes derive from primitives to learn why concepts succeed or fail and to guide the design of more effective modeling languages.

Acknowledgments

This work was supported in part by the project PHYLOVAR (PID2024-162114OB-C21, PID2024-162114OA-C22), through funds from MICIU/ AEI / 10.13039/501100011033 / FEDER, UE and the Project VARNETICA (CNS2023-145422), funded by MICIU/AEI / 10.13039 / 501100011033 and by the European Union NextGenerationEU/PRTR. Additional support was provided by Industrias Químicas del Ebro through a pre-doctoral fellowship granted to José Ignacio Trasobares.

Replication Package

<https://doi.org/10.5281/zenodo.19254651>

References

- [1] activeplayer.io. 2023. Fortnite live player count and statistics. <https://activeplayer.io/fortnite/> Accessed: 1-March-2025.
- [2] Luciane Telinski Wiedermann Agner, Inali Wisniewski Soares, Paulo Cézar Stadisz, and Jean Marcelo Simão. 2013. A Brazilian survey on UML and model-driven practices for embedded software development. *Journal of Systems and Software* 86, 4 (2013), 997–1005.
- [3] Deniz Akdur, Vahid Garousi, and Onur Demirörs. 2018. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture* 91 (2018), 62–82.
- [4] Bente Anda, Kai Hansen, Ingolf Gulleßen, and Hanne Kristin Thorsen. 2006. Experiences from introducing UML-based development in a large safety-critical project. *Empirical Software Engineering* 11, 4 (2006), 555–581.
- [5] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 1–10. doi:10.1145/1985793.1985795
- [6] Simonetta Balsamo, Antinisa Di Marco, Paola Inverardi, and Marta Simeoni. 2004. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering* 30, 5 (2004), 295–310.
- [7] Victor R. Basili and H. Dieter Rombach. 1988. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering* (1988).
- [8] Daniel Blasco, Antonio Iglesias, Jorge Echeverría, Francisca Pérez, and Carlos Cetina. 2025. Introducing phylogenetics in search-based software engineering: Phylogenetics-aware SBSE. *ACM Transactions on Software Engineering and Methodology* (2025). doi:10.1145/3715002
- [9] Michel RV Chaudron, Werner Heijstek, and Ariadi Nugroho. 2012. How effective is UML modeling? *Software & Systems Modeling* 11, 4 (2012), 571–580.
- [10] Jorge Chueca, Jose Ignacio Trasobares, África Domingo, Lorena Arcega, Carlos Cetina, and Jaime Font. 2023. Comparing software product lines and Clone and Own for game software engineering under two paradigms: Model-driven development and code-driven development. *Journal of Systems and Software* 205 (2023), 111824.
- [11] Jorge Chueca, Javier Verón, Jaime Font, Francisca Pérez, and Carlos Cetina. 2024. The consolidation of game software engineering: A systematic literature review of software engineering for industry-scale computer games. *Information and Software Technology* 165 (2024), 107330.
- [12] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. routledge.
- [13] Jesús Sánchez Cuadrado, Javier Luis Cánovas Izquierdo, and Jesús García Molina. 2014. Applying model-driven engineering in small software enterprises. *Science of Computer Programming* 89 (2014), 176–198.
- [14] Mohammad Derakhshandi, Shekoufeh Kollahdouz-Rahimi, Javier Troya, and Kevin Lano. 2021. A model-driven framework for developing android-based classic multiplayer 2D board games. *Automated Software Engineering* 28, 2 (2021), 7.
- [15] Ely Fernando do Prado and Daniel Lucredio. 2015. A Flexible Model-Driven Game Development Approach. In *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software*. 130–139. doi:10.1109/SBCARS.2015.24
- [16] Brian Dobing and Jeffrey Parsons. 2006. How UML is used. *Commun. ACM* 49, 5 (2006), 109–113.
- [17] África Domingo, Jorge Echeverría, Óscar Pastor, and Carlos Cetina. 2020. Evaluating the Benefits of Model-Driven Development. In *International Conference on Advanced Information Systems Engineering*. Springer, 353–367.
- [18] Jorge Echeverría, Jaime Font, Francisca Pérez, and Carlos Cetina. 2021. Comparison of Search Strategies for Feature Location in Software Models. *Journal of Systems and Software* (2021).
- [19] Epic Games. 1998. Unreal Engine: The most powerful real-time 3D creation tool. <https://www.unrealengine.com>. [Online; accessed 27-March-2025].
- [20] Epic Games. 2024. About Epic Games. <https://www.epicgames.com/site/en-US/about> Accessed: 13-March-2026.
- [21] Epic Games. 2024. *Blueprint Compiler Overview*. Unreal Engine Documentation. https://dev.epicgames.com/documentation/en-us/unreal-engine/compiler-overview-for-blueprints-visual-scripting-in-unreal-engine?application_version=5.3 [Online; accessed 25-March-2026].
- [22] Epic Games. 2024. *Introduction to Blueprints Visual Scripting in Unreal Engine*. Unreal Engine Documentation. <https://dev.epicgames.com/documentation/en-us/unreal-engine/introduction-to-blueprints-visual-scripting-in-unreal-engine> [Online; accessed 22-June-2026].
- [23] Epic Games, Inc. 2023. Real-time round-up: the state of interactive 3D. <https://www.unrealengine.com/en-US/blog/real-time-round-up-the-state-of-interactive-3d> [Online; accessed 25-March-2026].
- [24] Martin Fowler. 2004. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional.
- [25] Maxwell Foxman. 2019. United We Stand: Platforms, Tools and Innovation With the Unity Game Engine. *Social Media + Society* 5, 4 (2019). doi:10.1177/2056305119880177
- [26] Game Developers Conference. 2025. *2025 State of the Game Industry Report*. Technical Report. Informa Tech. <https://reg.gdconf.com/state-of-game-industry-2025>
- [27] Tony Gorschek, Ewan Tempero, and Lefteris Angelis. 2014. On the use of software design models in software development practice: An empirical investigation. *Journal of Systems and Software* 95 (2014), 176–193.
- [28] Martin Grossman, Jay E Aronson, and Richard V McCarthy. 2005. Does UML make the grade? Insights from the software development community. *Information and Software Technology* 47, 6 (2005), 383–397.
- [29] Werner Heijstek and Michel R V Chaudron. 2009. Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process. In *Conference Proceedings of the EUROMICRO*.
- [30] John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming* 89 (2014), 144–161.
- [31] Rodi Jolák, Truong Ho-Quang, Michel RV Chaudron, and Ramon RH Schifferers. 2018. Model-based software engineering: A multiple-case study on challenges and development efforts. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 213–223.
- [32] T Kapteijns, S Jansen, S Brinkkemper, H Houet, and R Barendse. 2009. A Comparative Case Study of Model Driven Development vs Traditional Development: The Tortoise or the Hare. *From code centric to model centric software engineering Practices Implications and ROI* (2009).
- [33] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734.
- [34] K Krogmann and S Becker. 2007. A Case Study on Model-Driven and Conventional Software Development : The Palladio Editor. *Software Engineering* (2007).
- [35] Nadja Christiane Marko, Grischa Liebel, Daniel Sauter, Aleksander Lodwich, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. 2014. Model-based engineering for embedded systems in practice. *Research reports in software engineering and management* (2014), 1–48.
- [36] Yulkeidi Martínez, Cristina Cachero, and Santiago Meliá. 2013. MDD vs. traditional software development: A practitioner’s subjective perspective. In *Information and Software Technology*.
- [37] Sara Martínez-González, Francisco-Javier Azcondo-San-Segundo, José-Andrés Díaz-Severiano, and Valentin Gomez-Jauregui. 2025. A Comparative Analysis of Code-Based Versus Visual Programming Approaches for Extended Reality (XR) and BIM. In *Advances on Mechanics, Design Engineering and Manufacturing V*. Springer Nature Switzerland, 543–555.
- [38] Daniel L Moody. 2003. The method evaluation model: a theoretical model for validating information systems design methods. *ECIS 2003 proceedings* (2003), 79.
- [39] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. 2014. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM/IEEE, 1–11.
- [40] Jose Ignacio Panach Navarrete, Oscar Dieste, Beatriz Marín, Sergio España, Sira Vegas, Oscar Pastor, and Natalia Juristo. 2018. Evaluating Model-Driven Development Claims with respect to Quality: A Family of Experiments. *IEEE Transactions on Software Engineering* (2018).
- [41] Paulo Eduardo Papotti, Antonio Francisco Do Prado, Wanderley Lopes de Souza, Carlos Eduardo Cirilo, and Luís Ferreira Pires. 2013. A quantitative analysis of model -driven code generation through software experimentation. In *International Conference on Advanced Information Systems Engineering*. Springer, 321–337.
- [42] PocketGamer. 2023. 71% of the biggest mobile games use a Unity SDK. <https://www.pocketgamer.biz/71-of-the-biggest-mobile-games-use-a-unity-sdk/> [Online; accessed 10-March-2026].
- [43] PwC. 2025. *Global Entertainment and Media Outlook 2025-2029*. Technical Report. PricewaterhouseCoopers. <https://www.pwc.com/gx/en/industries/tmt/media/outlook.html>
- [44] Isis Roca, África Domingo, Óscar Pastor, Carlos Cetina, and Lorena Arcega. 2024. Comparing MDD and CdD in the Bug Localization Context: An Empirical Evaluation in Video Games. In *International Conference on Advanced Information Systems Engineering*. Springer, 581–595.
- [45] Marcos Romero and Brenden Sewell. 2022. *Blueprints Visual Scripting for Unreal Engine 5: Unleash the true power of Blueprints to create impressive games and applications in UE5*. Packt Publishing Ltd.
- [46] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* 14, 2 (2009), 131–164. doi:10.1007/s10664-008-9102-8
- [47] B. Sewell. 2015. *Blueprints Visual Scripting for Unreal Engine*. Packt Publishing Ltd.
- [48] Aakrit Sharma. 2024. How many people play Genshin Impact? Player count. <https://www.charlieintel.com/games/how-many-people-play-genshin>

- impact-player-count-190314/ [Online; accessed 10-March-2026].
- [49] Dag IK Sjøberg, Jo E Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N-K Liborg, and Anette C Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering* 31, 9 (2005), 733–753.
- [50] SlashData Team. 2024. Did you know that 60% of game developers use game engines? <https://www.slashdata.co/post/did-you-know-that-60-of-game-developers-use-game-engines>. Accessed: 2026-02-17.
- [51] Mirosław Staron. 2006. Adopting model driven software development in industry—a case study at two companies. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 57–72.
- [52] Harald Störrle. 2017. How are Conceptual Models used in Industrial Software Development? A Descriptive Survey. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. 160–169.
- [53] Juha-Pekka Tolvanen and Steven Kelly. 2016. Model-driven development challenges and solutions: Experiences with domain-specific modelling in industry. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 711–719.
- [54] Jose Ignacio Trasobares, África Domingo, Lorena Arcega, and Carlos Cetina. 2022. Evaluating the benefits of software product lines in game software engineering. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*. 120–130.
- [55] Jose Ignacio Trasobares, África Domingo, Rodrigo Casamayor, Daniel Blasco, and Carlos Cetina. 2025. A multiple case study on reuse in Game Software Engineering. *Information and Software Technology* 185 (2025), 107781. doi:10.1016/j.infsof.2025.107781
- [56] Jose Ignacio Trasobares, África Domingo, Jorge Echeverría, Lorena Arcega, and Carlos Cetina. 2025. Deepening our Understanding on the use of Models and Code in Game Software Engineering: A Controlled Experiment in Unreal Engine. In *Proceedings of the 2025 MODELS Conference*.
- [57] Unity Technologies. 2005. Unity Real-Time Development Platform | 3D, 2D VR & AR Engine. <https://unity.com>. [Online; accessed 27-March-2026].
- [58] Unity Technologies. 2021. About Unity Visual Scripting. <https://learn.unity.com/course/introduction-to-visual-scripting/tutorial/about-unity-visual-scripting>. <https://learn.unity.com/course/introduction-to-visual-scripting/tutorial/about-unity-visual-scripting> [Online; accessed 22-June-2026].
- [59] Unity Technologies. 2021. *Unity Visual Scripting Documentation*. Unity User Manual. <https://docs.unity3d.com/Packages/com.unity.visualscripting@1.9/manual/index.html> [Online; accessed 22-June-2026].
- [60] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2013. The state of practice in model-driven engineering. *IEEE software* 31, 3 (2013), 79–85.
- [61] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.