JOURNAL OF INFORMATION TECHNOLOGY AND ITS UTILIZATION, VOLUME 9, ISSUE 1, MAY 2026
EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

ScriptableObject-Based Data Architecture for 2D Management Games: A Case Study of "BUTIKKU"

Aka Shakthi Avri Afika[1], Dwi Setiawan[2*], Juniana Husna[3]

[1,2,3]Game Technology Study Program, School of Multi Media, Indonesia
akashakthi0@gmail.com[1], dwis0609@gmail.com[2*], juniana.husna@mmtc.ac.id[3]
*Corresponding author

***Abstract*--Hard-coded data management in small-scale Unity game development introduces three fundamental problems: data-logic entanglement, high inter-system coupling, and restricted content scalability. This study implements a ScriptableObject-based data management system in the 2D boutique management game "BUTIKKU" as a realization of the Type Object Pattern and the Open/Closed Principle in game software engineering. The ScriptableObject approach was selected based on three technical considerations: native integration with the Unity editor, type-safe object referencing, and documented memory efficiency. The development process followed the Game Development Life Cycle (GDLC) framework comprising three stages: pre-production (requirements analysis and data schema design), production (ScriptableObject class creation, data asset population, and gameplay system integration), and post-production (black-box functional testing through combined static code review and Inspector-based verification). Five ScriptableObject classes were implemented: MaterialSO, ItemSO, CatalogSO, OrderSO, and OrderLibrarySO, forming a hierarchical data ecosystem fully separated from program logic. All four functional test scenarios yielded a pass status, confirming data-logic separation, asset structural completeness, inter-class object reference integrity, and data-driven order selection logic. The OrderLibrarySO weighted random selection mechanism demonstrates that ScriptableObject is capable of functioning as a data-driven decision engine, in which runtime behavioral variation is achieved through data configuration rather than code modification.**

***Key words*: 2D game; Data management; Management game; ScriptableObject; Type Object Pattern.**

I. INTRODUCTION

The digital game industry has experienced significant growth, functioning not only as an entertainment medium but also as an interactive learning platform. The global game market reached a value of over 184 billion USD in 2023, with mobile and casual game segments as the primary growth drivers [1]. This development has increased the complexity of systems used in game development, including the need for data management that is efficient, well-structured, and easily maintainable [2].

Small-to-medium scale game development commonly faces three fundamental problems in data management. The first problem is the widespread use of hard-coded approaches, where values such as item names, prices, or order configurations are stored directly within programming scripts. While practical in the early stages of development, this approach introduces difficulty in code maintenance and an increased risk of bugs caused by duplicated values scattered across multiple scripts [3]. The second problem is high inter-system coupling, where a change to a single data value can simultaneously affect multiple parts of the system, making the codebase fragile and difficult to scale [4]. The third problem is the lack of separation between data and program logic, which prevents content designers from modifying game parameters independently without direct involvement from programmers, thereby slowing down the development iteration cycle [5].

Several methods have been proposed to address these data management problems in game development. One common approach is storing game data in external file formats such as JSON or XML, which separates data from code but requires additional parsing logic and offers no native integration with the game engine's editor environment. Another approach involves using relational databases or spreadsheet-driven pipelines, which provide structured data storage but introduce significant overhead in setup and runtime access [6]. A third approach is the application of software design patterns, particularly the Type Object Pattern [7], [8], which enables the definition of multiple object types through a single class where each instance represents a different type without requiring code

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

modification.

Unity's ScriptableObject feature was selected based on three technical considerations: it provides native editor integration that allows designers to manage data directly through the Unity Inspector without writing additional parsing code; it leverages the Type Object Pattern natively within the Unity ecosystem; and it has been documented by Unity Technologies [9] to significantly reduce memory usage compared to MonoBehaviour-based data storage, since static data is stored only once regardless of how many objects reference it.

In software engineering, the three problems identified above are addressed through the principle of separating data from behavior [5]. The Open/Closed Principle (OCP) introduced by Martin [10] states that a software system should be open for extension but closed for modification. Applied to game engine architecture, the separation of data from logic serves as a foundational design pattern for scalable systems [11]. Unity's ScriptableObject feature serves as a practical implementation of both the Type Object Pattern and OCP within the Unity engine, enabling data to be stored as independent assets that are referenced across multiple scenes and managed directly through the Unity Inspector without recompilation [9]. Furthermore, ScriptableObject facilitates a data-driven decision engine, where runtime system behavior is dictated by externally stored data parameters rather than compiled code logic.

In this study, the OrderLibrarySO class implements a weighted random selection mechanism entirely through data parameters weight, minStage, and maxStage stored within ScriptableObject assets. Consequently, the selection logic governing which customer order appears at any given game stage is controlled entirely by data values modifiable through the Inspector, achieving behavioral variation through data configuration rather than code branching [11].

This context is directly relevant to the development of "BUTIKKU", a Unity-based 2D boutique management game that simultaneously serves as an educational medium for introducing traditional Indonesian clothing. Players take on the role of a boutique manager who fulfills customer orders by matching combinations of traditional garments such as Kebaya Kartini, Kebaya Kutubaru, Jarik Kawung, and Jarik Sogan. Kim [12] demonstrated that game-based approaches are effective in increasing user engagement in educational contexts, making "BUTIKKU" relevant not only from a technical standpoint but also in terms of its learning benefits. The data complexity of this game encompassing material data, clothing item specifications, active outfit catalogs, and weighted randomly selected customer orders makes ScriptableObject particularly suitable as the foundation of its data management system, addressing all three problems identified above within a single unified architecture.

Based on this background, this study aims to design and implement a ScriptableObject-based data management system in the "BUTIKKU" game as a realization of the Type Object Pattern and the Open/Closed Principle in game software engineering and to evaluate its effectiveness through technical black-box testing. The findings of this study are expected to contribute a practical data architecture reference for small-to-medium scale Unity game developers facing similar data management challenges.

II. METHOD

This study adopts an applied research approach through structured system design and implementation process. The development lifecycle is adapted from the game production framework outlined by Chandler [13], encompassing three core production phases: pre-production, production, and post-production with a dedicated functional testing framework embedded within the post-production phase. The complete methodological workflow, detailing the sequential tasks, tools, and technical outputs across these phases, is depicted in Fig. 1.

A. Pre-Production: Requirements Analysis and Data Schema Design

The pre-production stage begins with defining the game concept of "BUTIKKU" as the basis for identifying system data requirements. "BUTIKKU" is a Unity-based 2D boutique management game developed in portrait orientation, targeting two platforms: Android

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

(APK) as the primary output and WebGL as a browser-based demonstration alternative. Both builds are generated from a single Unity repository to ensure asset and logic consistency. Player interaction is tap-based, where players select outfit combinations and confirm orders by tapping or clicking UI elements.

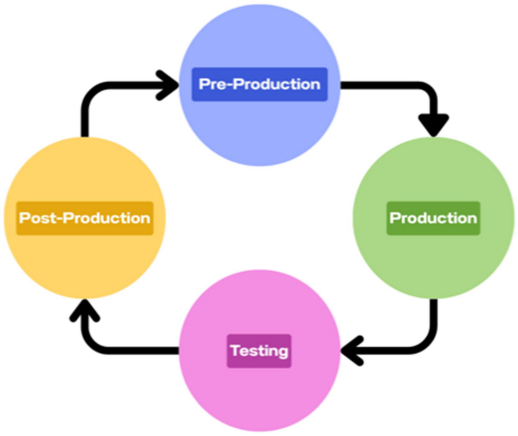

Fig. 1. Overall methodological flow of the study following the GDLC framework

The core gameplay of "BUTIKKU" follows a fixed loop consisting of four phases, as illustrated in Fig. 2. The four phases are as follows: (1) material management, in which the player monitors and manages raw material stock; (2) clothing item creation, in which the player selects and produces a clothing item; (3) income acquisition, in which the player receives payment upon successful order fulfillment; and (4) customer order fulfillment, in which the player matches outfit combinations to customer requests. This loop repeats continuously, forming a consistent game cycle.

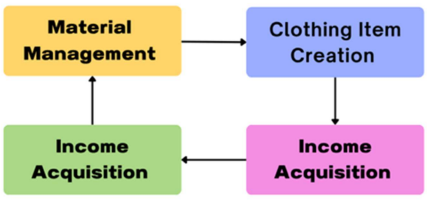

Fig. 2. Core gameplay loop of "BUTIKKU"

All data displayed across these four phases, including item names, icons, material requirements, and order combinations, are read directly from ScriptableObject assets. This ensures that any content change made through the Unity Inspector is automatically reflected in the game interface without requiring script modification, as illustrated in the gameplay and interface flow shown in Fig. 3.

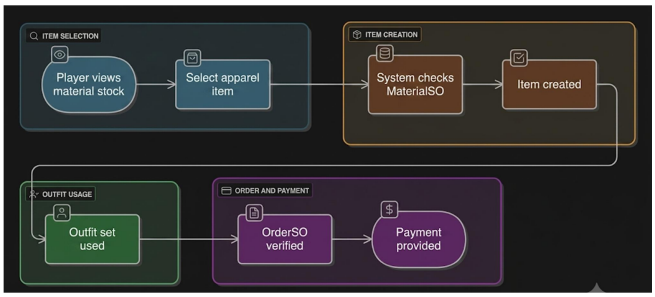

Fig. 3. Gameplay and interface flow of "BUTIKKU"

Fig. 3 illustrates how the gameplay flow is divided into four interface sections: (A) item selection, where the player views material stock and selects a clothing item; (B) item creation, where the system verifies MaterialSO availability and produces the item; (C) outfit usage, where the selected outfit is assigned to the player's boutique; and (D) order and payment, where the active OrderSO is verified and payment is processed. Each section reads its data directly from the corresponding ScriptableObject asset, demonstrating that the interface is fully driven by the SO-based data architecture.

The pre-production stage then proceeds to formally identify all data requirements and design the ScriptableObject architecture schema. This process consists of two main steps.

The first step is data requirements analysis. From the gameplay analysis above, four primary data categories were established: raw material data, clothing item data, active outfit catalog data, and customer order data. Each category maps directly to one ScriptableObject class, as summarized in Table I. This categorization follows the Single Responsibility Principle in software engineering, whereby each ScriptableObject class is designed to be responsible for exactly one type of data.

The second step is designing the relational schema between ScriptableObject classes. Based on the requirements analysis, five ScriptableObject classes were designed along with their referential relationships. This design follows the Type Object Pattern and the Open/Closed Principle, in which each class acts as a type definition that can be instantiated into different data assets without requiring code modification. Inter-class relationships are

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

designed using direct object references rather than strings to guarantee data consistency at the Unity engine level. The referential relationships between classes are as follows: MaterialSO is referenced by ItemSO through the materialCosts field; ItemSO is referenced by both CatalogSO (through its item list) and OrderSO (through requiredTop and requiredBottom); and OrderSO is referenced by OrderLibrarySO through its entries list.

TABLE I
Mapping of Data Categories to ScriptableObject Classes

Data Category	SO Class	Primary Fields
Raw material data	MaterialSO	-id, -displayName, -icon (Sprite)
Clothing item data	ItemSO	-id, -displayName, -slot (Top/Bottom), -sprite, -materialCosts
Active outfit catalog	CatalogSO	items: List<ItemSO> (11 elements)
Customer order data	OrderSO	-requiredTop (ItemSO ref.), -requiredBottom (ItemSO ref.)
Order library	OrderLibrarySO	-order (OrderSO ref.), -weight, -minStage, -maxStage

B. Production: ScriptableObject Implementation

The production stage is the realization of the designed schema into the Unity project. Implementation was carried out in three sequential sub-stages. The first sub-stage is the creation of ScriptableObject classes. Each class was written in C# using the [CreateAssetMenu] attribute, which enables asset creation through the Unity Inspector context menu. The five classes implemented are MaterialSO, ItemSO, CatalogSO, OrderSO, and OrderLibrarySO. The structure of each class was kept minimal in accordance with the principle of data encapsulation [10], storing only the data strictly required by the systems that access it.

The second sub-stage is the data asset population. Once the classes were compiled, concrete ScriptableObject assets were created through the Unity Inspector. At this stage, all game data, including 11 traditional clothing items, 3 types of raw materials, and 6 customer order combinations, were entered into the assets without touching the program code. This process directly confirms that the designed architecture enables content management to be performed independently from the codebase.

The third sub-stage is gameplay system integration. The C# gameplay scripts comprising StockService, CustomerController, and OrderService were connected to the ScriptableObject assets through object references in the Inspector. Order validation logic was implemented using direct ItemSO object reference comparison rather than string comparison, following Gregory's recommendation [11] to avoid the risk of string mismatch in game data management systems.

C. Post-Production: Functional Testing

The functional testing stage was designed to evaluate the ScriptableObject-based data management system through a combined approach of static code review and Inspector-based verification, adapted as a black-box testing methodology (Myers et al. [14]). This combined approach was selected due to its suitability for validating ScriptableObject-based architectures without requiring the game to be executed at runtime. The correctness of the system can be verified by examining the structure of the code and the configuration of assets directly.

In this approach, black-box testing is applied in two complementary ways. First, static code review (black box at the logic level) examines C# gameplay scripts StockService, CustomerController, and OrderService to verify that no literal data values are embedded in the code, confirming that all game data is read exclusively from ScriptableObject assets. Second, Inspector-based verification (black box at the data level) examines the structure and referential relationships of all five ScriptableObject assets through Unity Inspector screenshots, confirming that the data architecture is correctly implemented as designed.

This combined approach specifically targets properties unique to ScriptableObject-based systems, derived from the Type Object Pattern and the Open/Closed Principle: data separation from logic, asset structure completeness, object reference integrity, and data-driven selection

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

logic. The pass or fail status of each scenario was determined based on the criteria defined in Table II.

TABLE II
Black-Box Functional Testing Scenario Design

No	Test Scenario	Verification Method	Pass Criteria	Fail Criteria
1	Data separation from logic	Static code review of StockService, CustomerController, and OrderService	No literal data values identified in any C# gameplay script	Literal data values found embedded within game play scripts
2	SO asset structure completeness	*Inspector*-based verification of all 5 SO classes	All 5 SO classes contain correct fields; 11 items registered in CatalogSO; 6 orders registered in OrderLibrary SO with *weight*, *minStage*, and *maxStage* parameters.	One or more SO classes contain incorrect, incomplete, or missing fields.
3	Object reference integrity	*Inspector*-based verification of inter-SO references in OrderSO and OrderLibrarySO	All *requiredTop* and *requiredBottom* fields in OrderSO reference valid ItemSO objects; no *string* values were employed.	Any field contains a *string* value or an unresolved object reference
4	Data-driven order selection logic	Static code review of OrderService combined with *Inspector*-based verification of OrderLibrary SO	Order selection logic retrieves parameters exclusively from OrderLibrary SO; no *hard-coded* selection criteria are present in OrderService.	Hard-coded selection criteria identified within the OrderService script.

III. Result and Discussion

The implementation of a ScriptableObject-based data management system in the 2D boutique management game "BUTIKKU" was successfully realized through five ScriptableObject classes forming an integrated data ecosystem. All game data is fully separated from program logic following the Type Object Pattern and the Open/Closed Principle. The discussion is structured in a top-down manner, beginning with the overall system architecture, followed by the system runtime flow, technical implementation of each ScriptableObject class, code analysis, functional testing results, and technical implications.

A. ScriptableObject-Based Data Architecture Overview

The data architecture of "BUTIKKU" comprises five ScriptableObject classes arranged in a hierarchical referential structure. Each class is assigned a single, distinct responsibility within the system, following the Single Responsibility Principle. The hierarchical organization of these classes and their referential relationships are summarized in Table III.

TABLE III
ScriptableObject-Based Data Architecture of "BUTIKKU"

SO Class	Role in Architecture	References	Referenced By
MaterialSO	Lowest-level data unit; stores raw material attributes	None	ItemSO (via materialCosts)
ItemSO	Mid-level blueprint; defines clothing item type and material requirements.	MaterialSO	CatalogSO, OrderSO
CatalogSO	Central collection; manages all active item entries available to the player.	ItemSO	UI Manager
OrderSO	Order definition: specifies required outfit combination per customer request.	ItemSO	OrderLibrarySO
OrderLibrarySO	Top-level controller; manages weighted random order selection by stage	OrderSO	OrderService

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

The hierarchical structure ensures unidirectional data flow from the lowest-level assets (MaterialSO) to the highest-level controller (OrderLibrarySO). All inter-class relationships are implemented as direct object references rather than string values, eliminating the risk of string mismatch and guaranteeing data consistency at the Unity engine level. This architecture realizes the Type Object Pattern, where each ScriptableObject class functions as a reusable type definition that can be instantiated into multiple distinct asset instances without modifying any code.

B. System Flow Overview

The "BUTIKKU" data management system operates through a runtime flow driven entirely by ScriptableObject assets. Upon game initialization, CatalogSO is loaded to populate the player's outfit selection panel. OrderService subsequently processes OrderLibrarySO through a weighted random selection algorithm, determining one active OrderSO based on the weight, minStage, and maxStage parameters stored within the asset.

Fig. 4 presents the complete system runtime flow, comprising four sequential phases: (1) game initialization and item catalog loading from CatalogSO; (2) OrderLibrarySO processing by OrderService to determine the active order; (3) player interaction through the outfit selection interface; and (4) order validation through comparison of the player's selected ItemSO references against those stored in the active OrderSO. The flow operates cyclically upon completion of each order, and OrderService re-initiates phase two to select the subsequent order. All system parameters across each phase are read exclusively from ScriptableObject assets, with no hard-coded values present in any gameplay script, as verified through the static code review conducted in the functional testing stage.

C. Implementation of the Five ScriptableObject Classes

Five ScriptableObject classes were successfully implemented based on the schema designed during the pre-production stage. The Inspector configuration of each class is presented as Inspector-based verification evidence within the black-box testing framework.

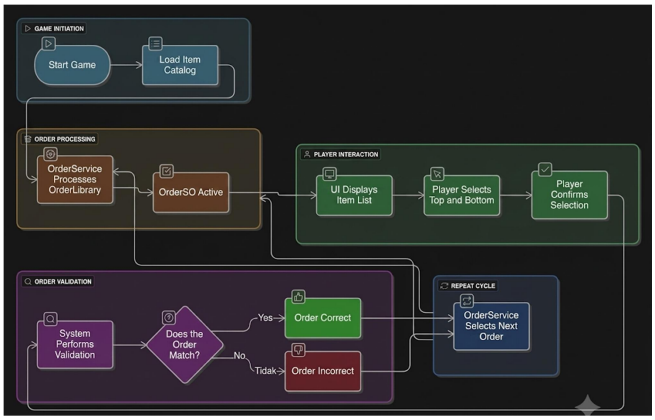

Fig. 4. ScriptableObject-based customer service system flow

1. MaterialSO

MaterialSO constitutes the most fundamental class in the system hierarchy and is responsible for storing raw material data used in clothing production. Each MaterialSO asset represents a single material type, such as thread (Benang) or fabric (Kain), through three primary attributes: id (unique identifier), displayName (display name), and icon (visual sprite). This structure implements the Type Object Pattern [8], in which a single class definition is instantiated into multiple distinct material types without code duplication.

The Inspector configuration presented in Fig. 5 confirms that all material data, including identity and economy attributes, are stored entirely within the SO asset. No material values are embedded in any C# gameplay script. This configuration constitutes the Inspector-based verification evidence for Test Scenario 1 (data separation from logic), confirming that data-logic separation has been correctly implemented across the material data layer [13]. The attribute grouping within the Inspector further enables content designers to modify material parameters independently, without programmer involvement.

2. ItemSO

ItemSO serves as the blueprint for each clothing item in the game, covering both the Top (upper garment) and Bottom (lower garment) categories. This class contains four attribute groups: Identity (id and displayName), Slot & Visual (slot OutfitSlot. Top/Bottom and sprite), and Crafting Requirements (a materialCosts list consisting of MaterialSO object reference and quantity pairs). The materialCosts structure, which uses direct MaterialSO object references

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

rather than material name strings, guarantees data consistency at the Unity engine level and eliminates the risk of string mismatch.

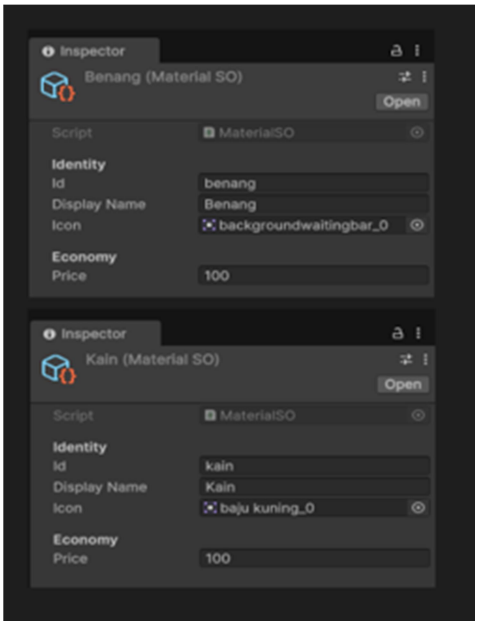

Fig. 5. MaterialSO assets in Unity Inspector: Benang and Kain

Fig. 6 – 9 shows four examples of successfully implemented ItemSO assets. Adding a new clothing item, such as a kebaya variation from another region, requires only the creation of a new ItemSO asset through the Create menu in the Unity Inspector, without modifying any UI scripts, crafting scripts, or order scripts. This demonstrates that the Open/Closed Principle in software engineering has been fulfilled, whereby the system is open for content extension but closed to code modification.

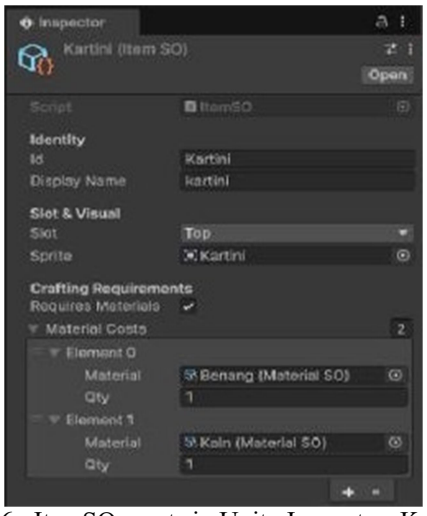

Fig. 6. ItemSO assets in Unity Inspector: Kartini

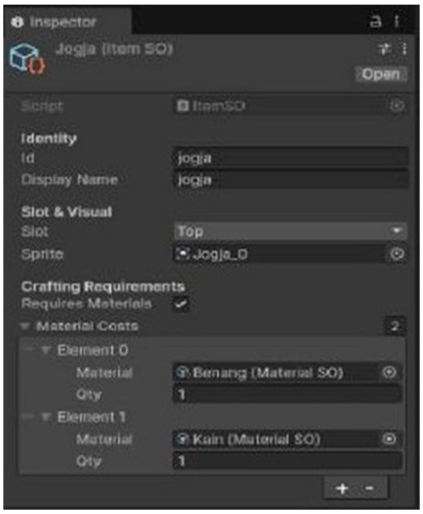

Fig. 7. ItemSO assets in Unity Inspector: Jogja

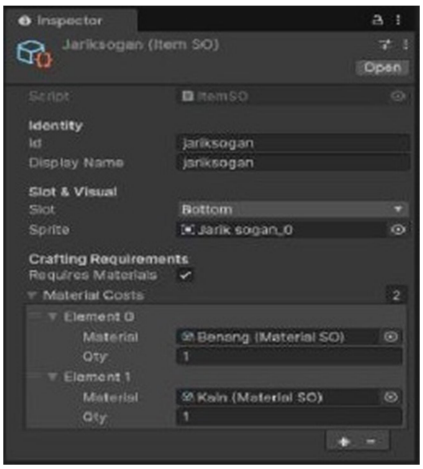

Fig. 8. ItemSO assets in Unity Inspector: Jarik Sogan

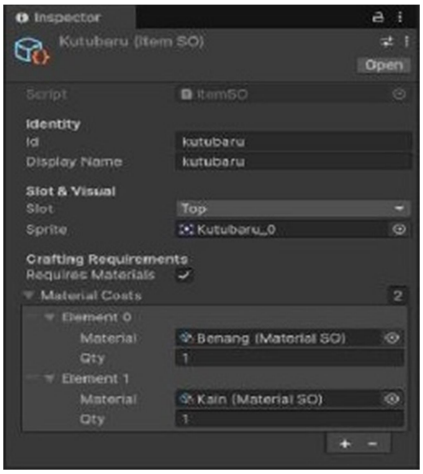

Fig. 9. ItemSO assets in Unity Inspector: Kutubaru

3. CatalogSO

CatalogSO serves as a centralized collection that stores the list of all active ItemSO entries in the game. This class has one primary attribute in the form of a List<ItemSO>, which is read by the interface system to dynamically generate the outfit selection panel. This approach ensures that any change to the CatalogSO, such as adding or

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

removing an item, is immediately reflected in the user interface without manual modification of any prefab or UI script.

Fig. 10 shows the CatalogSO successfully holding 11 active ItemSO entries, comprising encim, jarikkartini, jarkawung, jarkondel, jarksogan, jarktanjidor, jogja, kartini, kutubaru, modern3, and solo. This entire list is managed within a single asset that can be modified at any time without recompilation, demonstrating the scalability of the designed architecture.

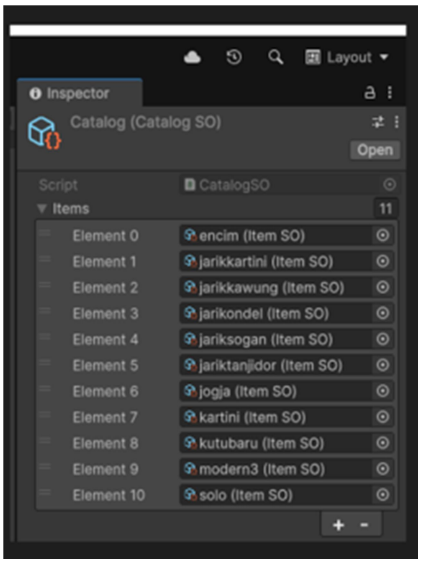

Fig. 10. CatalogSO asset containing 11 active ItemSO entries in the Unity Inspector

To provide a complete and transparent overview of the game's data layer as requested by the evaluation profile, Table IV details the exact configuration of the 11 traditional Indonesian clothing assets implemented within CatalogSO. This matrix demonstrates how the architecture successfully maps structural metadata across different clothing slots without hard-coded scripts.

4. OrderSO

OrderSO defines a single customer order scenario through two primary attributes: requiredTop and requiredBottom, both implemented as direct ItemSO object references. The use of object references rather than string names ensures that order verification operates at the Unity engine's object identity level, which is structurally immune to string comparison errors such as capitalization inconsistencies, extraneous whitespace, or typographical errors.

TABLE IV
Structural Configuration of Implemented ItemSO Assets

Asset Name (ItemSO)	Outfit Slot Category	Visual Representation (Sprite)	Crafting Dependency (MaterialSO)
encim	OutfitSlot.Top	baju_encim	Kain, Benang
jarikkartini	OutfitSlot.Bottom	jarik_kartini	Kain, Benang
jarkawung	OutfitSlot.Bottom	jarik_kawung	Kain, Benang
jarkondel	OutfitSlot.Bottom	jarik_kondel	Kain, Benang
jarksogan	OutfitSlot.Bottom	jarik_sogan	Kain, Benang
jarktanjidor	OutfitSlot.Bottom	jarik_tanjidor	Kain, Benang
jogja	OutfitSlot.Top	baju_jogja	Kain, Benang
kartini	OutfitSlot.Top	baju_kartini	Kain, Benang
kutubaru	OutfitSlot.Top	baju_kutubaru	Kain, Benang
modern3	OutfitSlot.Top	baju_modern3	Kain, Benang
solo	OutfitSlot.Top	baju_solo	Kain, Benang

Fig. 11 presents six implemented OrderSO assets, each defining a distinct order combination such as kutubaru+jarksogan, jogja+jarkawung, and encim+jarktanjidor. The Inspector configuration confirms that requiredTop and requiredBottom fields display ItemSO asset identifiers such as "kutubaru (Item SO)" and "jarksogan (Item SO)" rather than plain string values, providing Inspector-based verification evidence for Test Scenario 3 (object reference integrity) across all six order definitions

5. OrderLibrarySO

OrderLibrarySO is the most complex ScriptableObject class in the system, functioning as an order library with a weighted random selection mechanism. Each entry in the OrderLibrarySO stores four parameters: an OrderSO reference, a weight value (appearance probability), minStage, and maxStage. These parameters are used by OrderService at runtime to proportionally select orders according to the current game stage.

Fig. 12 presents the OrderLibrarySO Inspector configuration, confirming that all order selection parameters weight, minStage, and maxStage are stored as configurable data fields within the SO asset. No selection criteria are embedded within

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

the OrderService script. This configuration provides Inspector-based verification evidence for Test Scenario 4 (data-driven order selection logic) and constitutes the implementation of a data-driven decision engine as defined by Gregory [12]: a system in which runtime behavioral variation is achieved entirely through data configuration rather than code branching. Order appearance frequency and stage range parameters are adjustable exclusively through the Inspector, without programmer involvement.

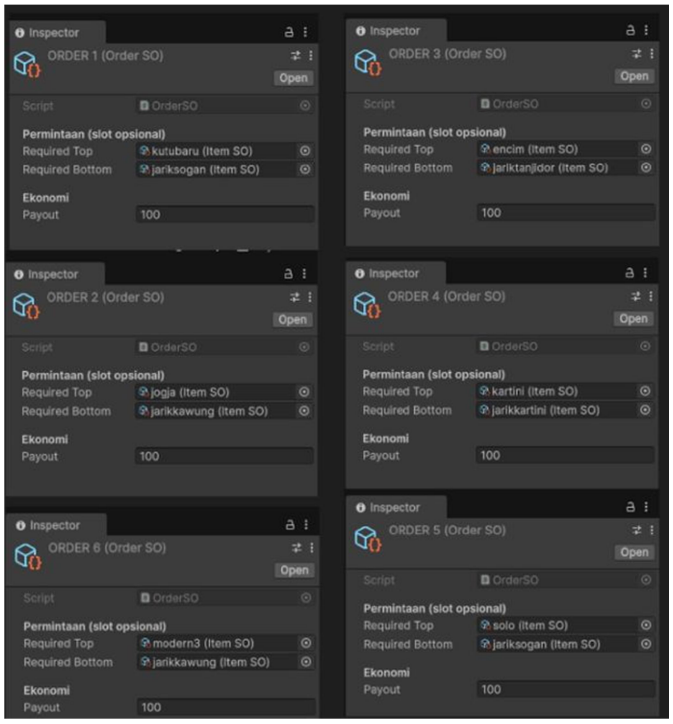

Fig. 11. Six OrderSO assets with various customer order combinations

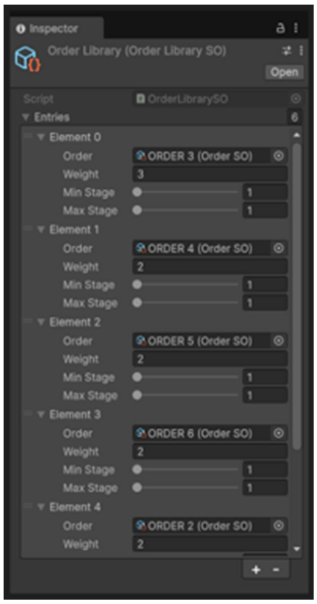

Fig. 12. OrderLibrarySO asset with weight and stage parameters

D. Code Analysis: Object Reference-Based Order Validation

The following code analysis examines three representative code segments that collectively demonstrate the consistent application of the ScriptableObject-based data architecture across the system from data structure definition through logic execution to order validation.

1. ScriptableObject Class Structure: ItemSO

The ItemSO class definition illustrates the structural implementation of the Type Object Pattern within the Unity ScriptableObject framework:

```
[CreateAssetMenu(menuName = "MMDress/Item")]
public class ItemSO : ScriptableObject
{
 public string id;
 public string displayName;
 public OutfitSlot slot;
 public Sprite sprite;
 public List<MaterialCost> materialCosts = new();
}
```

The [CreateAssetMenu] attribute enables asset instantiation directly through the Unity Inspector context menu, fulfilling the Open/Closed Principle: the class definition remains unmodified regardless of how many item assets are created. Each field within the class corresponds directly to one functional attribute id and displayName for identity, slot for outfit categorization, sprite for visual representation, and materialCosts for crafting requirements. The materialCosts field stores a list of MaterialCost objects, each containing a direct MaterialSO reference and a quantity value, rather than string-based material identifiers. This structural decision ensures that material references are validated at the Unity engine level rather than at runtime string comparison.

2. ScriptableObject Data Reading: TryCraft Method

The TryCraft method in StockService demonstrates how gameplay logic reads data exclusively from ScriptableObject assets without embedding any literal values within the script:

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

```
public bool TryCraft(ItemSO item, int qty)
{
 foreach (var cost in item.materialCosts)
  if (GetMaterial(cost.material) < cost.qty * qty)
  return false;
 AddGarment(item, qty);
 return true;
}
```

The method receives an ItemSO object reference as its parameter and iterates through the materialCosts list to verify material availability. No material names, quantities, or item identifiers are hard-coded within the method all values are retrieved dynamically from the ItemSO asset at runtime. This constitutes the static code review evidence for Test Scenario 1 (data separation from logic): the crafting logic operates identically regardless of which ItemSO is passed as an argument, demonstrating that the system remains functionally correct when new ItemSO assets are added without script modification.

3. ScriptableObject Reference Validation: Order Verification

The order validation logic in CustomerController demonstrates the application of object reference-based comparison as the final layer of the ScriptableObject data architecture:

```
bool isCorrect =
  chosenTop == currentOrder.requiredTop &&
  chosenBottom == currentOrder.requiredBottom;
```

The comparison operates at the Unity object reference level rather than the string value level. The system evaluates whether the player's selected ItemSO references (chosenTop, chosenBottom) are identical to those specified in the active OrderSO (requiredTop, requiredBottom) at the memory address level. This mechanism eliminates the class of errors associated with string-based comparison, including capitalization inconsistencies, extraneous whitespace, and typographical errors. The validation logic contains no literal item names or order specifications. All comparison values are retrieved from the active OrderSO asset at runtime, consistent with the data-logic separation verified in Test Scenario 1.

Fig. 13 presents the complete order confirmation flow, structured into three functional sections. The first section (Input and Data Retrieval) depicts the system retrieving ItemSO references from the player's selection (chosenTop, chosenBottom) and from the active OrderSO (requiredTop, requiredBottom). The second section (Validation and Decision) depicts the ItemSO reference validation step, represented by the decision node evaluating whether the player's selected references match those specified in the active OrderSO. This constitutes the ItemSO reference validation component of the confirmation system. The third section (Result and Cycle Continuation) depicts the two resulting execution paths: upon a match, the CustomerServed event is dispatched, and OrderService initiates the selection of the subsequent order; upon a mismatch, failure feedback is rendered, and the current order remains available for retry. The entire flow operates without hard-coded values, as all data is sourced exclusively from ScriptableObject assets.

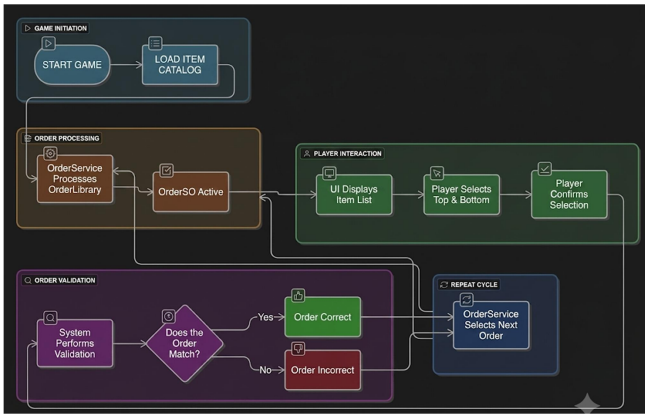

Fig. 13. Order confirmation flow and ItemSO reference validation

E. Functional Testing Results

Black-box functional testing was conducted against four main test scenarios as designed during the post-production stage. Table V summarizes the testing results. All four test scenarios yielded a PASS status, demonstrating that the implemented ScriptableObject architecture fulfills three primary criteria: data separation from logic (Open/Closed Principle), inter-system reference integrity (Type Object Pattern), and data-driven behavioral control (data-driven system definition).

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

TABLE V
Black-Box Functional Testing Results of the ScriptableObject System

No.	Test Scenario	Evidence	Actual Result	Status
1.	Data separation from logic	Static code review of StockService, CustomerController, and OrderService confirmed the absence of literal data values in all gameplay scripts.	All game data were retrieved exclusively from SO assets at runtime; no *hard-coded* values were identified in any C# gameplay script.	**PASS**
2.	SO asset structure completeness	Fig. 5 - 12 (*Inspector* screenshots of MaterialSO, ItemSO, CatalogSO, OrderSO, and OrderLibrary SO)	All 5 SO classes were confirmed to contain correct fields; CatalogSO registered 11 active ItemSO entries; OrderLibrarySO registered 6 orders with *weight*, *minStage*, and *maxStage* parameters.	**PASS**
3	Object reference integrity	Fig. 11 - 12 (OrderSO and OrderLibrary SO *Inspector* screenshots displaying ItemSO object references)	All *requiredTop* and *requiredBottom* fields in OrderSO were confirmed to reference valid ItemSO objects; no *string* values were employed across all 6 order definitions	**PASS**
4	Data-driven order selection logic	Static code review of OrderService combined with Fig. 12 (OrderLibrary SO *Inspector* screenshot confirming *the weight*, *minStage*, and *maxStage* fields)	Order selection logic confirmed to retrieve all parameters exclusively from OrderLibrarySO ; no *hard-coded* selection criteria identified in OrderService.	**PASS**

F. Technical Implications and Limitations

The implementation and testing results demonstrate three significant technical implications of applying the Type Object Pattern and the Open/Closed Principle through ScriptableObject in "BUTIKKU".

First, the consistent separation of data from logic verified through static code review in Test Scenario 1 enables an effective division of development responsibilities. Programmers are required to construct the ScriptableObject class infrastructure once, after which content designers may manage all game data independently through the Unity Inspector. This finding is consistent with Ampatzoglou & Stamelos [15], who identified architectural modularity as a primary factor in development efficiency for small-to-medium scale game projects.

Second, the application of direct object references verified through Inspector-based verification in Test Scenarios 2 and 3 demonstrates greater structural reliability than string-based data management approaches. In string-based systems, missing or incorrectly specified values may remain undetected until runtime execution. In contrast, Unity's object reference mechanism provides structural validation at the asset configuration level: an unresolved reference is immediately visible in the Inspector as a "None (Item SO)" or "Missing (Item SO)" field, enabling identification and correction of referential errors prior to game execution.

Third, the OrderLibrarySO weighted random selection mechanism verified through Test Scenario 4 demonstrates that ScriptableObject is capable of functioning as a data-driven decision engine. Relative to alternative data management approaches discussed in the introduction, including JSON/XML file-based storage and relational database pipelines, the ScriptableObject-based approach provides a distinct technical advantage through its native integration between data configuration and runtime behavioral control within the Unity editor environment, without requiring additional parsing code or external tooling.

The primary limitation of this system concerns the management of dynamic runtime data. ScriptableObject assets do not persist in runtime modifications after application closure. Consequently, dynamic player progress data, including in-game currency, completed level

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

records, and boutique reputation necessitate integration with an external serialization mechanism such as JSON or PlayerPrefs. This constraint represents an inherent boundary of ScriptableObject-based architecture that requires a hybrid data management approach in subsequent development iterations.

IV. Conclusion

A ScriptableObject-based data architecture successfully addresses the three fundamental data management problems identified in small-scale Unity game development: hard-coded data embedding, high inter-system coupling, and the absence of separation between data and program logic. The implementation in "BUTIKKU" demonstrates that the five ScriptableObject classes MaterialSO, ItemSO, CatalogSO, OrderSO, and OrderLibrarySO collectively form an integrated data ecosystem that realizes the Type Object Pattern and the Open/Closed Principle within a single cohesive architecture.

Black-box functional testing through combined static code review and Inspector-based verification across four test scenarios yielded a pass status for all cases. Static code review confirmed the complete absence of hard-coded data values in all gameplay scripts, while Inspector-based verification confirmed the structural correctness of all five SO asset configurations and the integrity of all inter-class object references. A notable finding of this implementation is the capability of OrderLibrarySO as a data-driven decision engine, wherein order selection behavior is governed entirely by configurable data parameters weight, minStage, and maxStage without code modification. This finding demonstrates that ScriptableObject extends beyond its conventional role as a passive data container to function as an active behavioral controller within the Unity architecture.

The primary limitation of this system concerns the persistence of dynamic runtime data. ScriptableObject assets do not retain runtime modifications upon application closure, necessitating integration with external serialization mechanisms such as JSON or PlayerPrefs for dynamic player progress data. Three directions are recommended for subsequent development: first, the integration of a JSON-based serialization layer for cross-session data persistence; second, the development of a Unity custom editor for batch content management as item and order volumes increase; and third, the extension of the SO architecture through the addition of level configuration and customer type classes, enabling the established framework to serve as a reusable data architecture foundation for Unity-based management and educational game projects.

V. Acknowledgment

The authors gratefully acknowledge the support and facilities provided by the Game Technology Study Program, Sekolah Tinggi Multi Media "MMTC" Yogyakarta, throughout the research and development process of this work. The authors also wish to thank the 2D artist contributor for the visual assets used in "BUTIKKU", as well as all respondents who participated in the functional testing of the game.

VI. References

[1] Newzoo, “Global Games Market Report 2023,” Newzoo, Amsterdam, Netherlands, 2023. [Online]. Available: https://newzoo.com/resources/trend-reports/newzoo-global-games-market-report-2023-free-version

[2] S. Scott, “Can the learning mechanic–game mechanic framework be utilised to innovate serious game mechanics?,” University of the West of England, Bristol, UK, 2023.

[3] Q. K. Fannoun, “Escape the planet: Revolutionizing game design with novel OOP techniques,” Mankato, MN, USA, 2024.

[4] A. Ruiz Rabasseda, “Creating a Unity framework for Scriptable Object Driven Development (SODD),” Mataró, Spain, 2024.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[6] Alan. Thorn, *Mastering Unity Scripting*. Birmingham B3 2PB, UK: Packt Publishing, 2015.

[7] R. Nystrom, *Game Programming Patterns*. San Francisco, CA, USA: Genever Benning, 2014. [Online]. Available: https://gameprogrammingpatterns.com/type-object.html

[8] Janine. Suvak, *Learn Unity 3D programming with UnityScript : Unity’s JavaScript for beginners*. Berkeley, CA, USA: Apress, 2014.

[9] Unity Technologies, “ScriptableObject,” Unity User Manual 2022.3 (LTS). Accessed: Apr. 25, 2026. [Online]. Available:

EISSN 2654-802X ; PISSN 2985-4067
DOI: https://doi.org/10.56873/jitu.6.2.5259. SUBMITTED: APRIL 26, 2026; REVISED: JUNE 20, 2026; ACCEPTED: JULY 23, 2026

https://docs.unity3d.com/Manual/class-ScriptableObject.html

[10] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall, 2003. doi: 10.1002/pfi.21408.

[11] J. Gregory, *Game Engine Architecture*, 3rd ed. Boca Raton, FL, USA: AK Peters/CRC Press, 2018.

[12] J. T. Kim, "A study on visualization of digital twin electric vehicles based on gamification," *Journal of Information Technology Utilization*, vol. 19, no. 3, pp. 220–226, 2025.

[13] H. Maxwell. Chandler and Rafael. Chandler, *Fundamentals of game development*. Jones and Bartlett Learning, 2011.

[14] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, 2011. doi: 10.1002/9781119202486.

[15] A. Ampatzoglou and I. Stamelos, "Software engineering research for computer games: A systematic review," *Inf. Softw. Technol.*, vol. 52, no. 9, pp. 888–901, 2010, doi: 10.1016/j.infsof.2010.05.004.