Comprehensive Exercise Report

Team SoftSolutions of Section 392

Lorin Rammo : 250ADB067

Sorin Rammo : 250ADB068

Héctor Hernández Sánchez : 250ADB056

Nazarkevych Andrii 250ADB006

Yaseen Almahayshi 250AIM008

Georgi Beev : 250ADM009

NOTE: You will replace all placeholders that are given in <<>>

Requirements/Analysis

Week 2

Journal

The following prompts are meant to aid your thought process as you complete the requirements/analysis portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- After reading the client's brief (possibly incomplete description), write one sentence that describes the project (expected software) and list the already known requirements.
    - Project description: The project is to develop a simple inventory tool within Unreal Engine 5.5 that allows users to pick up, store, and discard 3D assets in a first-person level.

Known Requirements:

- The tool must be developed using Unreal Engine 5.5.
- The inventory system should allow players to:
    - Pick up 3D assets.
    - Store 3D assets.
    - Discard 3D assets.
- The tool can be implemented using Unreal Engine's Blueprints or C++.
- 3D assets should be sourced from Quixel Megascans
- Assets must be in .FBX or .uasset format with .png or .exr textures.
- The final product should be a playable first-person level.

- After reading the client's brief (possibly incomplete description), what questions do you have for the client? Are there any pieces that are unclear? After you have a list of questions, raise your hand and ask the client (your instructor) the questions; make sure to document his/her answers.
    - **Questions for the Client:**

1. Should there be a limit to the number of items that can be stored? - 18
2. Should there be a limit of the stackable items at a time? - 20
3. How should item interactions work? (e.g., key bindings, drag-and-drop, etc.) - Click to drop
4. Should discarded items remain in the world or be deleted permanently? - Remain on the map
5. Should items have additional properties, such as weight or rarity? - No
6. Do assets need to be categorized (e.g., furniture, tools, weapons)? - No
7. Is multiplayer functionality required? - No

- Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.
    - We are familiar with the topic, because we all have used at least once an inventory management system. Some examples of these are: Minecraft, Elden Ring.
- Describe the users of this software (e.g., small child, high school teacher who is taking attendance).
    - The primary users of this software are, first and foremost, game developers who will integrate it into their games. Once implemented, the end users will be the players, who will interact with the inventory system as part of their gameplay experience.
- Describe how each user would interact with the software

    - Game Developers: They Import assets into the tool and test the inventory interaction(Picking, storing, and discarding).
      Players: Use in-game controls to pick up, store and discard items.
- What features must the software have? What should the users be able to do?
    - Store items within an inventory.
    - Support stackable and non-stackable items..
    - Allow item pickup and dropping.
    - Implement drop functionality in the UI.
    - Save and load inventory data persistently.
- Other notes:
    - Add, remove, and organize items in the inventory.
    - Equip, use, or consume items.
    - Interact with inventory via keyboard/mouser.

Software Requirements

Project Overview

The project involves the development of a simple inventory tool within Unreal Engine 5.5, designed for a first-person game environment. This tool will allow players to interact with 3D assets by picking them up, storing them in an inventory, and discarding them when necessary. The system will support drop functionality and will use assets sourced from Quixel Megascans. It will be implemented using either Unreal Engine's Blueprints or C++. The primary users of the tool are game developers, who will integrate the inventory system into their games, and players, who will interact with the inventory as part of their gameplay experience.

Requirements

Functional Requirements

1. **Inventory System Implementation**
    - The inventory system must allow players to pick up, store, and discard 3D assets.
    - Drop functionality should be implemented for item interaction.
    - Items discarded by the player should remain in the world.
2. **Storage and Item Management**
    - The inventory should support a maximum of 18 items.
    - Stackable and non-stackable items should be supported.
    - Items should not have additional properties such as weight or rarity.
3. **User Interaction**
    - The inventory should be accessible through keyboard or mouse inputs.
4. **Persistence and Data Handling**
    - The inventory system should support saving and loading inventory data to ensure persistence between game sessions.
    - 3D assets must be sourced from Quixel Megascans and stored in.FBX or .uasset format with .png or .exr textures.

Non-Functional Requirements

1. **Performance and Optimization**
    - The inventory system should operate smoothly without causing significant performance degradation.
    - Items discarded should remain on the map.
2. **Scalability**
    - The system should be designed to allow future expansion, such as additional item interactions or multiplayer functionality if needed.
3. **Usability**
    - The drop system should be intuitive and easy to use for both developers and players.
    - UI should provide clear feedback for item pickup, storage, and removal.

User Stories

1. **Game Developer:**
    - As a game developer, I want to import and test 3D assets within the inventory system to ensure proper functionality.
    - As a game developer, I want to integrate the inventory system into my game without additional modifications.
2. **Player:**
    - As a player, I want to pick up items in the game world and store them in my inventory.
    - As a player, I want to drag and drop items within my inventory for easy organization.

- As a player, I want to discard items and see them remain in the world so I can pick them up later if I need them.

Additional Notes

- The inventory system does not require multiplayer support at this stage.
- No categorization of items (such as weapons or tools) is required.
- The system should be developed using Unreal Engine 5.5, with the choice of Blueprints or C++ as the implementation method.

Black-Box Testing

Instructions: Week 4

Journal

Remember: Black box tests should only be based on your requirements and should work independent of design.

The following prompts are meant to aid your thought process as you complete the black box testing portion of this exercise. Please review your list of requirements and respond to each of the prompts below. Feel free to add additional notes.

- What does input for the software look like (e.g., what type of data, how many pieces of data)?
    - **Type of Data:**
        - Player actions (picking up, storing, and discarding items).
        - Click and drop interactions for inventory management.
        - Keyboard/mouse inputs.
    - **Number of Data Pieces:**
        - 3D assets with textures (Quixel Megascans, .FBX/.uasset, .PNG/.EXR).
        - Up to **18 items** stored in the inventory at a time.

- What does output for the software look like (e.g., what type of data, how many pieces of data)?
    - **Output Data**:
    - Items successfully added to the inventory. Items removed from the world upon pickup. Items successfully dropped and remain in the world.
    - Inventory UI updates reflecting changes (adding/removing items). Inventory is correctly saved and loaded across sessions.

- What equivalence classes can the input be broken into?

1. **Valid Inputs:**

   Picking up a valid item. Storing an item within the inventory. Dropping an item within the world. Loading a saved inventory.

2. **Invalid Inputs:**

   Attempting to pick up an item when the inventory is full. Trying to store an unsupported file format. Trying to save the game state when no inventory changes have been made.

- What boundary values exist for the input?
    - **Inventory Limit:**
        - 0 items (empty inventory scenario).
        - 1 item (minimum item count).
        - 17 items (almost full inventory scenario).
        - 18 items (full inventory scenario, testing if an additional item is rejected).
        - 19 items (attempting to exceed the limit, should fail).
    - **Item Interaction:**
        - Attempting to discard an item when no items are present.

    - **Persistence Testing:**
        - Saving an empty inventory.
        - Saving an inventory with items.
        - Reloading a saved inventory with varying item counts.
        - Reloading a saved game where items were discarded.

- Are there other cases that must be tested to test all requirements?
    - Ensuring items are visually displayed correctly in the inventory UI.
    - Verifying that discarded items remain in the world.
    - Checking whether inventory interactions work with both keyboard/mouse inputs.
    - Ensuring stackable items combine correctly and non-stackable items remain separate.
    - Testing game save/load functionality to ensure persistence.

- Other notes:
    - Testing should include various game resolutions to verify UI responsiveness.
    - Performance testing to ensure large inventories do not degrade game performance.
    - Ensure drop functions smoothly across different system configurations.

Black-box Test Cases

Use your notes from above to complete the black-box test plan section of the formal documentation by writing black box test cases (other than actual results since no program currently exists). Remember to test each equivalence class, boundary value, and requirement.

Test ID	**Description**	**Expected Results**	**Actual Results**
TC01	Pick up a valid item	Item is added to inventory, removed from world	TBD
TC02	Pick up an item when inventory is full	Item pickup fails, the item remains in the world	TBD
TC03	Store an item in inventory	Item is stored successfully	TBD
TC04	Drop an item from inventory	Item appears in world and remains	TBD
TC05	Save inventory with items	Inventory state is saved	TBD
TC06	Load saved inventory	Previously saved items appears correctly	TBD
TC07	Test stackable items	Stackable items combine properly	TBD
TC8	Test non-stackable items	Items remain separate	TBD
TC9	Ensure inventory UI updates on change	UI accurately reflects inventory state	TBD
TC10	Performance test with full inventory	No significant lag or frame drops	TBD

Design

Instructions: Week 6

Journal

Remember: You still will not be writing code at this point in the process.

The following prompts are meant to aid your thought process as you complete the design portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- List the nouns from your requirements/analysis documentation.
    - **Project** (or Inventory Tool)
    - **Unreal Engine 5.5**
    - **First-person level**
    - **3D Assets** (or Assets)
    - **Item**
    - **Inventory**
    - **Game Developer**
    - **Player**
    - **UI**
    - **Click and drop Functionality**
    - **Keyboard/Mouse Inputs**
    - **File Formats** (e.g., .FBX, .uasset, .png, .exr)
    - **Persistence Data** (Save/Load)
    - **Interaction**

- Which nouns potentially may represent a class in your design?
    - **Inventory**: This class would encapsulate the collection of items, its capacity, and methods to add or remove items.
    - **Item**: This class represents a 3D asset and would hold properties such as the asset's file format, texture type, and whether it is stackable.
    - **Player**: Representing the user interacting with the system (though this might be minimal if the focus is on the tool itself).
    - **UI**: A class (or set of classes) to manage the click and drop functionality, display the inventory contents, and handle input.
    - **Persistence Manager**: (Optional) A class responsible for saving and loading inventory data.

- Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.

    - **Inventory Class**
- **Capacity**: Maximum number of items (e.g., 18).
- **Items List**: A collection (e.g., array or list) of Item objects.
- **Current Count**: Could be maintained as an attribute to quickly check fullness.

    - **Item Class**
- **Name/ID**: Identifier for the asset.

- **File Format**: For example, ".FBX".
- **Texture Format**: For example, ".png" or ".exr".
- **Stackable Flag**: Boolean indicating if the item can be stacked.
- **Thumbnail/Image**: For UI representation.

    - **UI Class**
- **Display Elements**: Components to render the inventory.
- **Click-and-drop Handlers**: Methods or fields related to user input for dropping items.
- **Feedback Messages**: For actions like successful pickup or errors.

    - **Player Class**
- **Input Methods**: To handle keyboard/mouse events.
- **Reference to Inventory**: Linking the player to their inventory instance.

    - **Persistence Manager Class** *(if used)*
- **Save Method**: To serialize the current state of the inventory.
- **Load Method**: To deserialize and restore inventory data.

- Now that you have a list of possible classes, consider different design options (***lists of classes and attributes***) along with the pros and cons of each. We often do not come up with the best design on our first attempt. Also consider whether any needed classes are missing. These two design options should not be GUI vs. non-GUI; instead you need to include the classes and attributes for each design. Reminder: Each design must include at least two classes that define object types.

<u>Option 1: Minimal Two-Class Design (Inventory and Item)</u>

- **Classes Included:**
    - **Inventory**
    - **Item**

- **Pros:**
    - **Simplicity:** Fewer classes make the design straightforward and easier to implement initially.
    - **Direct Mapping:** Directly reflects the core functionality (managing items in an inventory).

- **Cons:**
    - **Limited Modularity:** UI interactions, persistence (save/load), and player input handling are not clearly separated.
    - **Scalability:** Future enhancements (e.g., adding a dedicated UI system or additional item interactions) might require significant refactoring.

<u>Option 2: Extended Modular Design</u>

- **Classes Included:**

    - **Inventory**: Manages a collection of Item objects.
    - **Item**: Represents each 3D asset with its properties.
    - **Player**: Handles user interactions and links to the Inventory.
- **Pros:**
    - **Separation of Concerns:** Each class handles a specific aspect of the system (e.g., business logic vs. UI).
    - **Modularity:** Easier to update or extend specific functionality (like UI improvements or additional input methods) without affecting the core inventory logic.
    - **Scalability:** Better suited for future requirements or additional features (like multiplayer support or more complex item interactions).

- **Cons:**
    - **Increased Complexity:** More classes mean more interfaces and potential overhead when integrating the components.
    - **Implementation Overhead:** For a simple inventory tool, a modular design may initially seem like overengineering.

- Which design do you plan to use? Explain why you have chosen this design.

We have chosen Option 2 because it provides a **modular and scalable architecture** that cleanly separates the logic, user interface, and interaction layers. Each class (Inventory, Item and Player) has a **single responsibility**, which improves code maintainability and makes future enhancements easier to implement. While it introduces some complexity, the long-term benefits in terms of flexibility and extensibility justify this design, especially for a game development environment like Unreal Engine 5.5.

- List the verbs from your requirements/analysis documentation.
    - **Pick up**
    - **Store**
    - **Drop**
    - **Save**
    - **Load**
    - **Display**
    - **Stack**
    - **Update**
    - **Drag (optional)**

- Which verbs potentially may represent a method in your design? Also list the class each method would be part of.

  Inventory Class:
    - pickUp(Item item): Adds an item to the inventory if space is available.
    - store(Item item): Stores the item in the inventory.
    - drop(Item item): Removes the item from the inventory.
    - save(): Saves the current inventory state.
    - load(): Loads a saved inventory state.

  Item Class:
    - getName(): Returns the name or ID of the item.
    - getTextureFormat(): Returns the texture format (e.g., .PNG, .EXR).

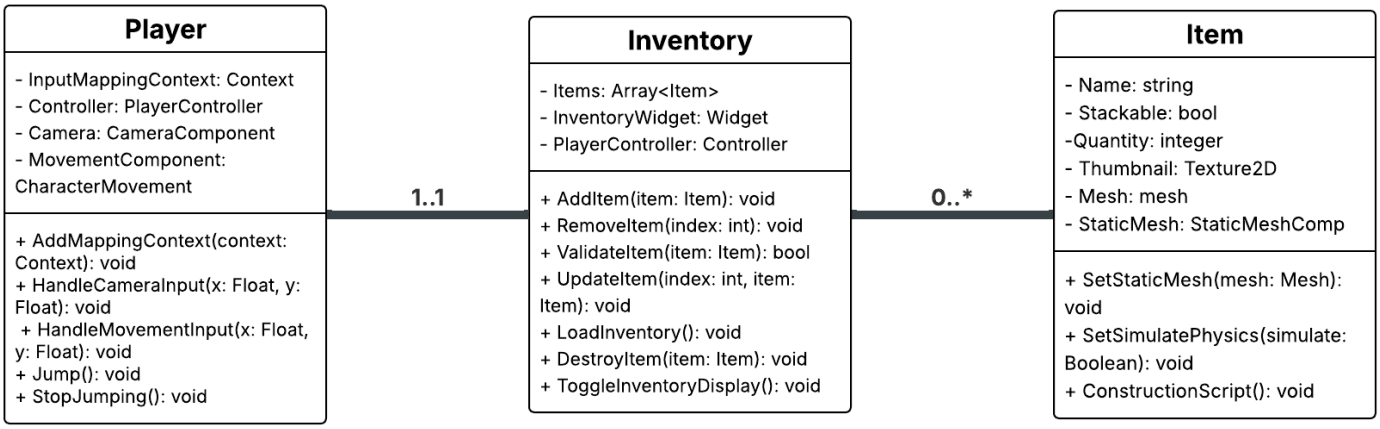
- isStackable(): Returns a boolean indicating if the item is stackable.
- getThumbnail(): Returns the thumbnail image for the UI.

Player Class:

- handleInput(): Responds to player input for inventory actions (picking up, storing, discarding).
- accessInventory(): Allows the player to access their inventory for management.
- UI Manager Class (Widget):
- displayInventory(): Displays the inventory in the UI.
- updateInventoryUI(): Updates the UI to reflect changes in the inventory.
- showFeedbackMessage(): Displays success or error messages related to inventory actions.

- Other notes:
  - **Player Input:** The Player class will handle all keyboard and mouse inputs for interacting with the inventory. This will include handling the drag-and-drop feature via mouse and keyboard shortcuts for picking up, storing, and discarding items.
  - **UI Design:** Since Unreal Engine uses UMG (Unreal Motion Graphics) for UI elements, the UI Manager will integrate with this system for a seamless experience. It will need to be responsive to different screen resolutions to ensure that the UI scales appropriately.
  - **Persistence:** The Persistence Manager (optional) could be a future extension that handles saving/loading data from disk to preserve the player's inventory across game sessions.

Software Design



Implementation

Instructions: Week 8

Journal

The following prompts are meant to aid your thought process as you complete the implementation portion of this exercise. Please respond to each of the prompt below and feel free to add additional notes.

- What programming concepts from the course will you need to implement your design? Briefly explain how each will be used during implementation.

To implement the InventoryTool project in Unreal Engine 5.5, several key programming and software development concepts from the course materials ("SDLC introduction.pdf," "W3.pdf," "Introduction to Software Development.pdf," "Software Architecture, Design, and Patterns.pdf," and "user interface.pdf") will be applied. These concepts are critical for developing a robust, user-friendly inventory system that manages game items, presents them through Unreal's UMG interface, and potentially integrates item meshes via the ModelingToolsEditorMode plugin.

1. **Object-Oriented Programming (OOP)**

Source: Software Architecture, Design, and Patterns.pdf (pp. 21–25)
Application: OOP is fundamental to Unreal Engine's architecture. Key classes will include:

- UInventoryItem: Stores item data (e.g., name, quantity)
- UInventoryComponent: Manages the collection of items

Inheritance will allow for specialized item types (e.g., UWeaponItem), and encapsulation will expose only necessary functions such as AddItem via UFUNCTION. UML class diagrams will help visualize the system architecture, and Blueprints will support OOP concepts visually for item behaviors and UI interactions.

2. **Coding for Quality**

Source: SDLC introduction.pdf (pp. 38–40)
Application: High-quality code will be ensured through clear naming conventions (e.g., AddItem, ItemName), thorough documentation, and adherence to Unreal coding standards such as using UPROPERTY for serializable variables. Blueprint debuggers and linters will aid in early bug detection, promoting maintainability and testability.

3. **Requirements Gathering and Design**

Sources: SDLC introduction.pdf (pp. 35–37, 45–58), Software Architecture, Design, and Patterns.pdf (pp. 2–11)
Application: A Software Requirements Specification (SRS) will define core functionality: item management, display, and data persistence. Design components include:

- UInventoryComponent (logic)
- UInventoryItem (data)
- UMG widgets (UI)

Class diagrams and other UML tools will map relationships, while a Software Design Document (SDD) will outline dependencies, design assumptions, and architectural decisions.

4. **User Interface Design**

Source: user interface.pdf (pp. 1–36)
Application: UI design will follow key principles:

- **User in Control**: Inventory opens with a keypress (e.g., "I"), allows drag-and-drop, and includes undo options.
- **Reduce Memory Load**: Icons, organized categories, and default layouts simplify navigation.
- **Consistency**: UI components (fonts, buttons, layouts) will be uniform across the game.

The design will also account for user profiles (gamers), tasks (viewing, using, sorting items), content layout, and display environments. Usability testing will validate interaction design, and improvements will be based on player feedback.

5. **Software Development Methodologies**

Source: W3.pdf (pp. 3–11)
Application: An Agile methodology will be adopted for its adaptability. Development will proceed in sprints, each focusing on specific aspects such as data structures or UI functionality. Iterative testing and feedback incorporation will ensure continuous improvement. While Waterfall may be referenced for documentation phases, Agile's flexibility is more suitable for evolving game requirements.

Implementation Details

The inventory system was implemented in **Unreal Engine 5.5** using **Blueprints**, Unreal visual scripting system. The system allows players in a first-person game environment to:

- **Pick up** 3D assets from the environment
- **Store** assets in a visual inventory interface
- **Discard** assets when no longer needed

Assets are sourced from **Quixel Megascans**, imported into Unreal Engine in .FBX or .uasset format with .PNG or .EXR textures. Shaders and materials are applied as needed before integration into the gameplay environment.

The inventory tool supports both:

- **Blueprint-based development** (easier to maintain and modify visually)
- **C++ implementation** (for future optimization if desired)

README: Inventory Tool User Guide

Prerequisites

To use this tool, you'll need **Unreal Engine 5.5.1** installed.

Installing Unreal Engine 5.5.1

1. **Go to the Epic Games Website**
   https://www.unrealengine.com

2. **Sign In or Create an Epic Games Account**
   Click **Sign In** in the top-right corner, or **Sign Up** if you don't have an account.

3. **Download the Epic Games Launcher**
   - Hover over your profile icon and click **Download**
   - Run the installer and follow the on-screen instructions

4. **Install Unreal Engine 5.5.1**
   - Open the Epic Games Launcher and sign in
   - Click on the **Unreal Engine** tab > **Library**
   - Click the **+** icon next to "Engine Versions"
   - Choose **5.5.1** (or select "More Versions…" if it's not listed)
   - Click **Install** and wait (30–50 GB download)

Running the Inventory Tool

1. **Download the Project**
    - Clone or download the repository from GitHub
    - Locate the `src` folder
    - Double-click on `InventoryTool.uproject` to launch the project

2. **Start the Game**
    - Press **F5** to play the game
    - Click into the game window to activate player controls

How to Interact with the Inventory

Action	Keybind / Method
Pick up item	Press **E** when near an item
Open/Close inventory	Press **I**
Interact with UI	Use **mouse** when inventory is open
Discard an item	Click on the item's image in the UI
Exit game	Press **ESC**

- When inventory is open, the **mouse cursor** becomes visible
- Items remain in the game world when dropped

Exploring the Blueprints

To explore how the inventory system was implemented:

1. Open the **Content Drawer** in the lower-left corner of Unreal Editor

2. Navigate to:
    - `All > Content > FirstPerson > Blueprints > BP_InventoryComponent`
    - `All > Content > Inventory > BP_Item`
    - `All > Content > FirstPerson > Blueprints > BP_FirstPersonCharacter`

3. Double-click each to view their Blueprint graphs

These Blueprints represent the core logic of the inventory system, including item handling, player interaction, and UI updates.

Testing

Instructions: Week 10

Journal

The following prompts are meant to aid your thought process as you complete the testing portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

Important: Since our project is developed using Blueprints programming, traditional unit testing is not applicable. Therefore, testing will be conducted through playtesting, following practices commonly used in the video game development industry.

- Have you changed any requirements since you completed the black box test plan? If so, list changes below and update your black-box test plan appropriately.
    - At this stage, **no fundamental changes** have been made to the **core functionality requirements** defined in the original black-box test plan.

- List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.

    - **Item**
        - Creation of items with valid/invalid formats
        - Whether stackable flag works as intended
        - Handling of null or missing thumbnails
        - Getter methods return correct data
    - TC11 - Create item with valid file and texture format
        - Input:
        - Expected output: Functional item
    - TC12 - Create item with unsupported file format
        - Input: FileFormat = “.exe”
        - Expected output: Item fails validation or logs error (if validateItem is used)
    - TC13 - isStackable returns correct value
        - Input: Item with Stackable = false
        - Expected output: isStackable() = false
    - TC14 - Null thumbnail image
        - Input: Thumbnail = null
        - Expected output: getThumbnail() handles without crashing

    - **Inventory**
        - Adding/removing items at various inventory states (empty, full)
        - Handling stackable items
        - Saving and loading inventory state
        - Validating item before adding
    - TC15 - Add item when inventory has space
        - Input: Inventory has 17 items, Capacity = 18, AddItem(Item)
        - Expected output: Item added, Inventory count = 18
    - TC16 - Add item when inventory is full
        - Input: Inventory has 18 items, AddItem(Item)

        - Expected output: Addition rejected, Inventory count unchanged
    - TC17 - Drop item from inventory
        - Input: Inventory has item at index 3, drop(index = 3)
        - Expected output: Item removed from inventory, count decreases
    - TC18 - Save and Load inventory
        - Input: Save current state, Load it again
        - Expected output: Loaded inventory matches saved state exactly

    - **Player**
        - Handling input to open inventory
        - Input to pick or drop items
        - Access to inventory
    - TC19 - Open inventory using key input
        - Input: Press "I" key
        - Expected output: Inventory UI toggles to visible
    - TC20- Attempt to pick up item near player
        - Input: Player clicks on item in range
        - Expected output: Item added to inventory
    - TC21 - Try dropping item when inventory is empty
        - Input: dropItem()
        - Expected output: Error or feedback shown, no crash
    - TC22- Access inventory object
        - Input: player.accessInventory()
        - Expected output: Returns valid Inventory instance linked to player

Testing Details

- **Black box tests**

Test ID	Description	Expected Results	Actual Results
TC01	Pick up a valid item	Item is added to inventory, removed from world	Success
TC02	Pick up an item when inventory is full	Item pickup fails, message displayed	Success
TC03	Store an item in inventory	Item is stored successfully	Success
TC04	Drop an item from inventory	Item appears in world and remains	Success
TC05	Save inventory with items	Inventory state is saved	Success
TC06	Load saved inventory	Previously saved items appears correctly	Success
TC07	Test stackable items	Stackable items combine properly	Success
TC8	Test non-stackable items	Items remain separate	Success
TC9	Ensure inventory UI updates on change	UI accurately reflects inventory state	Success
TC10	Performance test with full inventory	No significant lag or frame drops	Success

- **Test programs**

Test ID	Description
TC11	Create item with valid file and texture format
TC12	Create item with unsupported file format
TC13	isStackable returns correct value
TC14	Null thumbnail image
TC15	Add item when inventory has space
TC16	Add item when inventory is full
TC17	Drop item from inventory

TC18	Save and Load inventory
TC19	Open inventory using key input
TC20	Attempt to pick up item near player
TC21	Try dropping item when inventory is empty
TC22	Access inventory object

Presentation

Instructions:Week 12

Preparation

The following prompts are meant to aid your thought process as you complete the presentation portion of this exercise. It is recommended that you examine the previous sections of the journal and your reflections as you work on the presentation as it is likely that you have already answered some of the following prompts elsewhere. Please respond to each of the prompts below and feel free to add additional notes.

- Give a brief description of your final project

    - The Inventory Tool is a Blueprint-based system developed in Unreal Engine 5.5 for a first-person game environment. It enables players to pick up, store, and discard 3D assets sourced from Quixel Megascans (.FBX format with .png/.exr textures) in a playable level. The system supports a 18-item inventory with stackable and non-stackable items, a click-and-drop UI, and persistent data via Unreal's SaveGame system. The tool is designed for game developers to integrate into their projects and for players to interact with during gameplay.

- Describe your requirement assumptions/additions.

    - The customer's brief required picking up, storing, and discarding 3D assets in a first-person level using Blueprints or C++. We assumed a click-and-drop UI for intuitive inventory management, as interaction details were unspecified. We added support for stackable items to enhance functionality, allowing identical assets to combine in one slot. Additional assumptions included:

    - 18-item inventory limit.

    - Discarded items remain in the world.

    - No multiplayer, item categorization, or additional properties (e.g., weight). The optional freeform rotation preview was not implemented due to time constraints and lack of client prioritization.

- Describe your design options and decisions. How did you weigh the pros and cons of the different designs to make your decision?
    - **Minimal Two-Class Design** (Inventory_BP, Item_BP):
        - **Pros**: Simple, fast to implement, directly supports core inventory functions (pickup, store, discard).
        - **Cons**: Limited modularity, challenging to add UI or future features (e.g., multiplayer), poor separation of concerns.
    - **Modular Four-Class Design** (Inventory_BP, Item_BP, Player_BP, UI_Manager_BP):
        - **Pros**: Separates inventory logic, player input, and UI, scalable for extensions, aligns with Unreal's UMG for drag-and-drop, supports material instance setup.
        - **Cons**: Higher complexity, increased development time for Blueprint setup. We chose the modular design for its maintainability and scalability, critical for a reusable game development tool.

- How did the extension affect your design?
    - No mandatory extensions were required, as the core requirements (pickup, store, discard, 18-item limit) remained consistent. The optional freeform rotation preview was not implemented due to time constraints and lack of client emphasis. However, the modular design was structured to support potential extensions.
- Describe your tests (e.g., what you tested, equivalence classes).
    - Since Blueprints don't support traditional unit testing, we used black-box playtesting based on requirements. We tested boundary values—0 and 18 items—and equivalence classes for valid/invalid inputs, like supported file formats. Tests covered picking up items, saving/loading inventories, and UI updates, ensuring smooth performance. I helped validate UI functionality. Andrii will share our test results.
    - All 10 test cases passed successfully: picking up valid items added them to the inventory, full inventory rejected new items, and saving/loading preserved states. Stackable items combined correctly, and the UI updated accurately with no lag, as shown in our report. These results confirmed our tool meets all requirements. Andrii will discuss challenges and lessons learned.
- What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
    - We faced challenges debugging Blueprints, which was slower than scripted testing, and integrating dropping items by clicking UI with persistent saves was complex. We learned modular design's value for maintainability and the importance of UX-driven testing. I worked on UI design.
- What functionalities are you going to demo?
    - For the demo, we'll show our first-person level in Unreal Engine 5.5. We will demonstrate picking up an asset with 'E', toggling the inventory with 'I', using click and drop, and discarding an item, which stays in the world. The demo highlights our core functionality and persistence.
- Who is going to speak about each portion of your presentation? (Recall: Each group will have ten minutes to present their work; minimum length of group presentation is seven minutes. Each student must present for at least two minutes.
- Each member presents about 2 Minutes.
    - **Lorin:** Title Slide – Team introduction and Information, Project Overview and Motivation.
    - **Sorin:** Chosen SDLC Methodology, Requirements Gathering & Analysis, Software Requirements Specification.
    - **Hector:** System and Software Design, Implementation Phase.
    - **Andrii:** Testing Strategy, Test Results and Validation, Process Challenges and Lessons Learned.
    - **Yaseen** and **Georgi:** Demo Overview.
    - **Georgi:** Conclusion and Future Work.