**Storage Management System**

Contents

[**1.** **Introduction** 2](#_Toc35641985)

[**1.1 Storage System Structure** 2](#_Toc35641986)

[**1.2 UML Diagram** 2](#_Toc35641987)

[**2.** **Storage Model** 3](#_Toc35641988)

[**2.1 Implementation** 3](#_Toc35641989)

[**2.2 Core Functionality** 4](#_Toc35641990)

[**3.** **Storage Controller** 6](#_Toc35641991)

[**3.1** **Implementation** 6](#_Toc35641992)

## **Introduction**

This documentation will lay out the planning and design decision for a storage management system, the goal is to create a storage system that can handle all types of storage.

There are two main types of storage:

* Collectable storage such as loot boxes that allow users to withdraw items but are unable to store new items
* Depositable storage such as inventories and banks that allow users to deposit and withdraw items

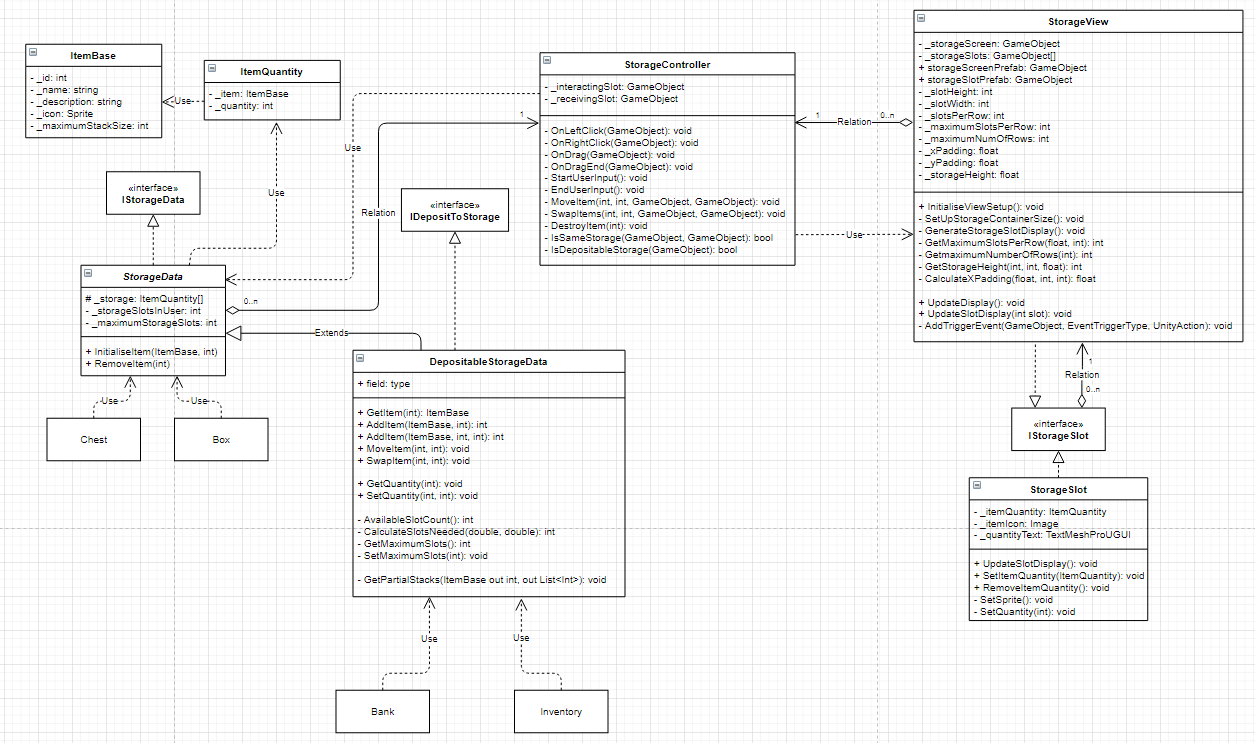
### **1.1 Storage System Structure**

The storage system will be decoupled and use inheritance for DRY purposes. I felt that a traditional MVC pattern would be a logical approach for a storage system as it can be naturally be separated into three elements:

* Storage data – populate and manage the data
* Storage control – Handle input and pass correct instructions to data
* Storage view – handle output for display

Unity also allows event systems to be used, this closely mimics the observer pattern and will allow a single storage controller to listen for input on all storage devices input, process the instructions and pass to the correct storage models and views involved.

### **1.2 UML Diagram**



[**https://drive.google.com/file/d/1zBt-hDv-DElLtyQaZiK4rOPdamKtmzWP/view?usp=sharing**](https://drive.google.com/file/d/1zBt-hDv-DElLtyQaZiK4rOPdamKtmzWP/view?usp=sharing)

## **Storage Model**

### **2.1 Implementation**

There are typically two types of storage devices, those that contain items for the users to collect such as chests and boxes and those that act as fully fledged storage facilities such as inventories and bank systems.

Collectable storage devices would typically only allow the storing and withdrawal of items that the user can collect.

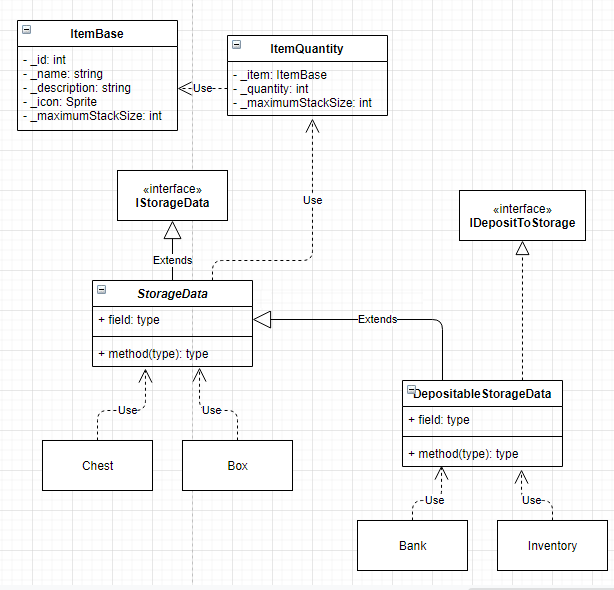
Depositable storage devices would allow storing as well as both withdrawal and depositing of items.

When interacting with a storage device, this differentiation would need to be made to understand what user interaction functionality should be allowed.

As multiple interfaces can be inherited by a class, using an IStorageData interface and an IDepositToStorage interface would allow decoupling between the storage models and the interaction system.

The interaction system can differentiate between the two types of storage devices and explicitly know the core methods that each has without being affected by the implementation.

This also allows the depositable storage devices to inherit the core functionality of all storage devices while having its own additional implementations.



### **2.2 Core Functionality**

#### **2.2.1 Storage Data Core Functionality**

* Storage data structure
* Initialise starting items
* Remove item

#### **2.2.2 Depositable Storage Data Core Functionality**

* Add item
* Move item
* Swap item
* Get item
* Add quantity
* Remove quantity
* Set quantity
* Remove all quantity
* Get quantity
* Slot empty check
* Slots available check
* Get incomplete slots
* Get maximum storage slots
* Set maximum storage slots

Add item

Two methods required to allow for specific slot selection and generic fill/find new slots. This seems like a better solution instead of using an optional slot argument and overloading the method with redundant code within if statements.

Specific slot selection:

As user input is validated on the controller side, this method will only be called if the slot is empty or the existing item matches, otherwise a method such as SwapItem would be called so additional validation checks are not required.

IF slot empty:

Add item

Return 0

ELSE:

Update quantity up to maximum stack size

Return remaining quantity

Generic fill/find validation checks:

* Are there any partial stacks containing the same item or available slots
* Are there any remaining quantity after slots have been filled

Indexes 🡨 list of indexes with partial stacks  
existingSpaceInSlots 🡨 total space available inside partial stacks  
availableSlots 🡨 number of free slots in storage device  
remainingSpaceNeeded 🡨 calculation of quantity remaining after killing partial stacks  
slotsNeeded 🡨calculation of slots required after filling partial stacks

IF indexes is greater than 0:

FOREACH index:

Fill the slot and update remaining quantity

IF quantity remaining equals 0:

Return 0

ENDIF

ENDIF

FOREACH slot in storage

IF slot is empty:

Fill slot up to maximum stack size

Update remaining quantity

ENDIF

IF quantity remaining equals 0:

Return 0

ENDIF

Return remaining quantity

## **Storage Controller**

### **Implementation**

1 controller that handles the input from all storage views (multiple storage devices) using the EventSystem to call methods within the controller class to interpret the input and update the model accordingly, this would essentially provide an observer pattern between the controller and the multiple views without the need for referencing and lower overhead with only 1 listener.

Right clicking an item should move it from the current active storage device to any other active storage device, this means that a special event needs to be created for the controller that passes the two storage devices active in the scene.

MoveItem and SwapItem can use optional parameters implemented in .NET 4.0 to take in references to different storage devices if needed.