

MET CS665

Class Project

a weapon shop system for a game

Github repo:

<https://github.com/metcs/met-cs665-assignment-project-primnp>

note this presentation is a simplified version of the readme. Please refer to my Github repo readme for more detailed information.

By: Nuwapa Promchotichai (Prim), U45776029, pnuwapa@bu.edu

Table of Contents.

01 Project Concept

Overall application description

02 Project Features

Diving deeper to understand what the application offers

03 New Pattern

Bridge pattern implementation

04 Other Patterns Used

Abstract factory, singleton, and observer pattern implementations

05 The code

Code walkthrough and demonstration of the application

06 UML diagram

UML diagram of the project





Project Concept

Application Description

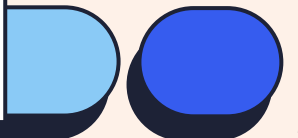
- A weapon shop system application written in Java
- The application consists of 2 main parts:

1. *Weapon Customization*

Allows user to pick the weapon type, material, and the enhancement for the weapon. Note that there is material inventory and user will be unable to place an order for a weapon if the inventory is out of stock.

2. *Order Processing*

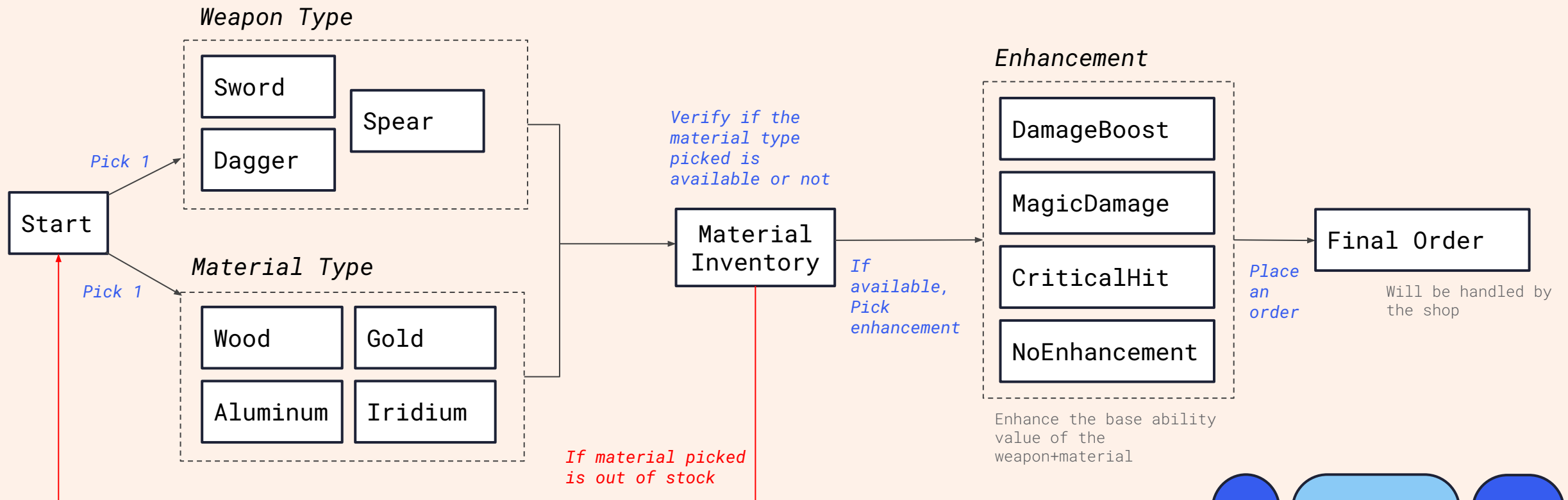
After user finalized the order from 1., the shop will process the order request. The shop will send out the new order request to all its available Blacksmith. Blacksmith will pick up the order request on the first come, first served basis. Blacksmith which completed crafting the order will return to the queue.





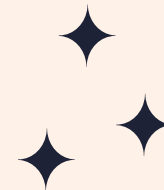
Project Features: Weapon Customization

Diving deeper to understand what the application offers



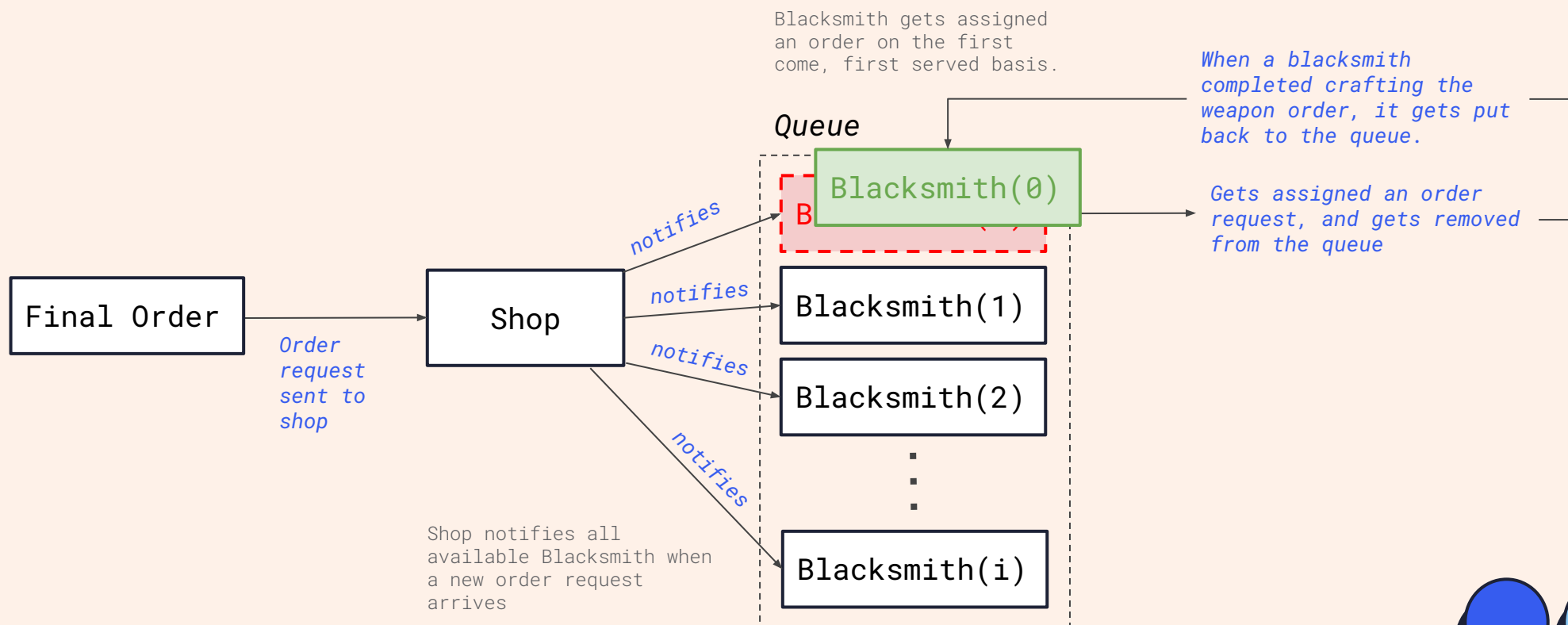
note each weapon type and material type has the base ability value of: damageDealt, MagicPierce, agility, and accuracy.

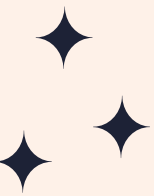




Project Features: Order Processing

Diving deeper to understand what the application offers



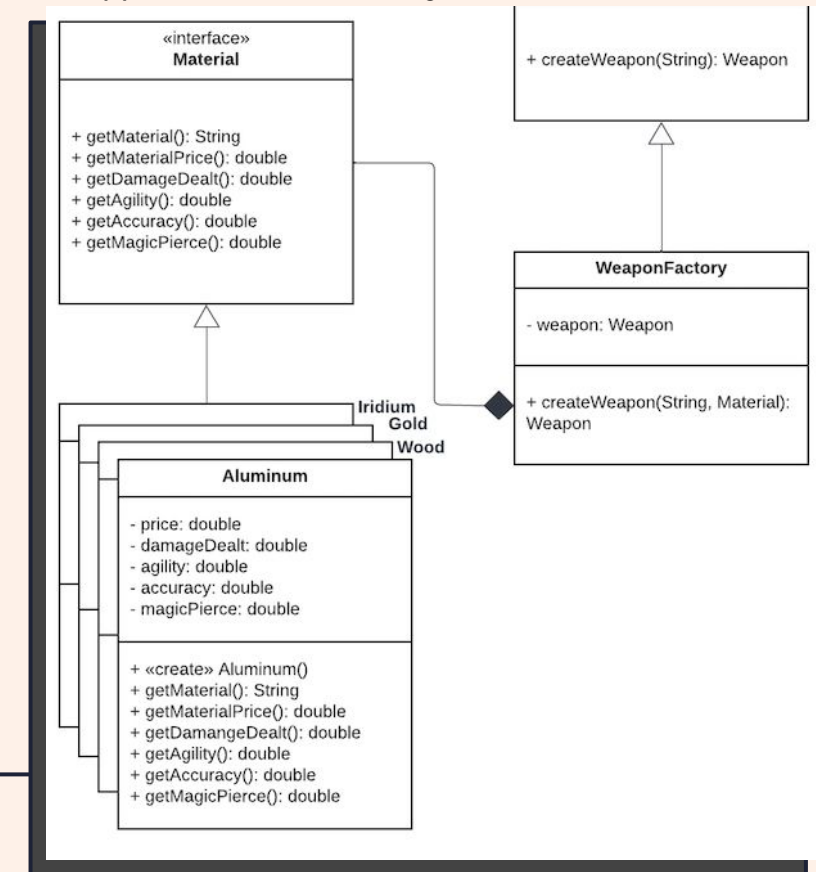


New Pattern: Bridge Pattern

Bridge pattern implementation inside the application

- Bridge pattern was implemented for the creation of the weapon process.
- WeaponFactory has Material composition. **Material** also has its subclasses of **Wood**, **Aluminum**, **Gold**, and **Iridium**.
- Bridge pattern decouples abstraction from implementation.
 - When a user call **createWeapon(String, Material)** from WeaponFactory, the system will create the specified weapon with the specified material.
- Developers can add more material type by adding subclasses to Material class instead of creating more inheritance from Weapon.

Snippet from UML diagram



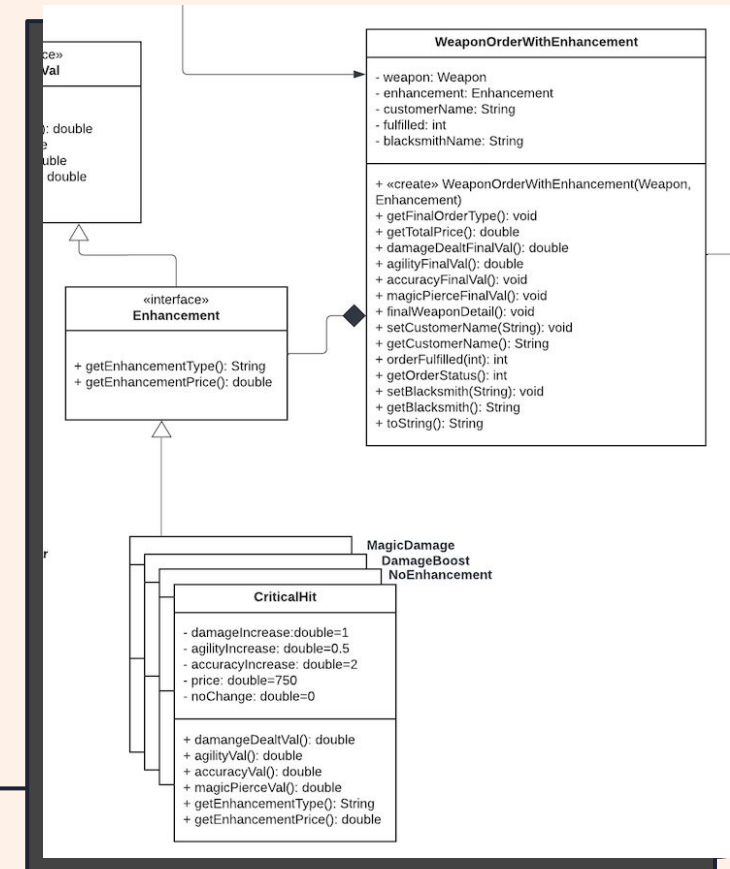


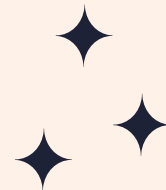
New Pattern: Bridge Pattern (2)

Bridge pattern implementation inside the application

- WeaponOrderWithEnhancement class has an Enhancement composition.
- The **Enhancement** class has subclasses of **MagicDamage**, **DamageBoost**, **CriticalHit**, and **NoEnhancement**.
- WeaponOrderWithEnhancement(Weapon, Enhancement) constructor has parameters of weapon object and enhancement object.
- The final order of specified weapon type and specified enhancement is created using the bridge pattern to avoid duplicated inheritance.
- **Note:** there are 2 bridge patterns instead of one because the user will be checked for material inventory before he/she is allowed to pick enhancement for the weapon.

Snippet from UML diagram



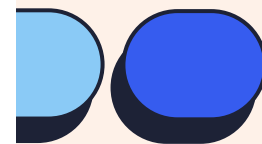
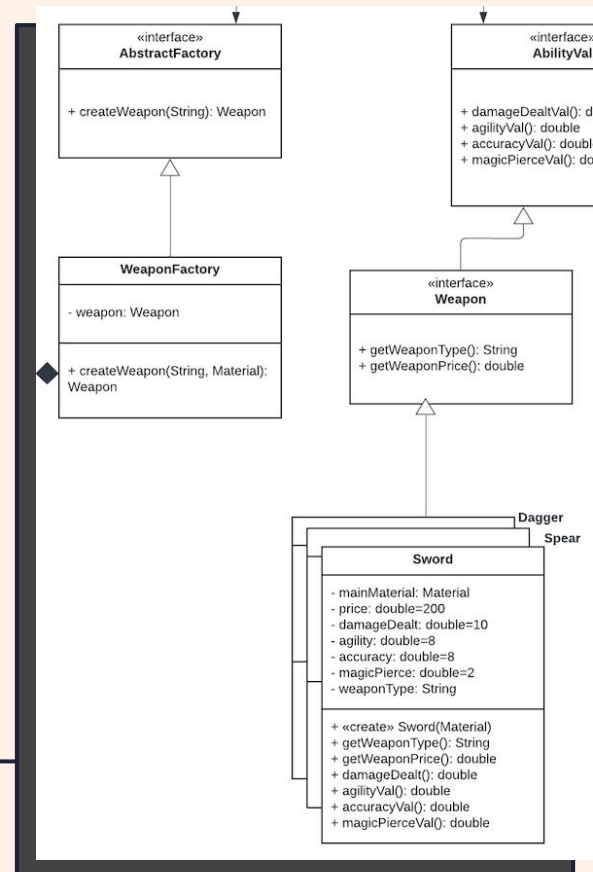


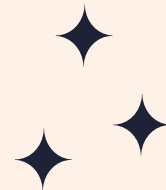
Other Pattern: Abstract Factory

Abstract factory pattern implementation inside the application

- Abstract factory was implemented to create a specific type of weapon object.
- **WeaponFactory** is the concrete factory implementation which is used to create a sword, dagger, or spear based on the input from the user.
- Concrete product class implementations are **Sword**, **Dagger**, and **Spear** class.
 - The concrete product implementation consists of methods to `getWeaponType()` and `getWeaponPrice()` and also other methods to get ability values of the weapon.
- Developers can add more subclasses of weapon type to the weapon class, and specify creation methods inside `WeaponFactory`.

Snippet from UML diagram



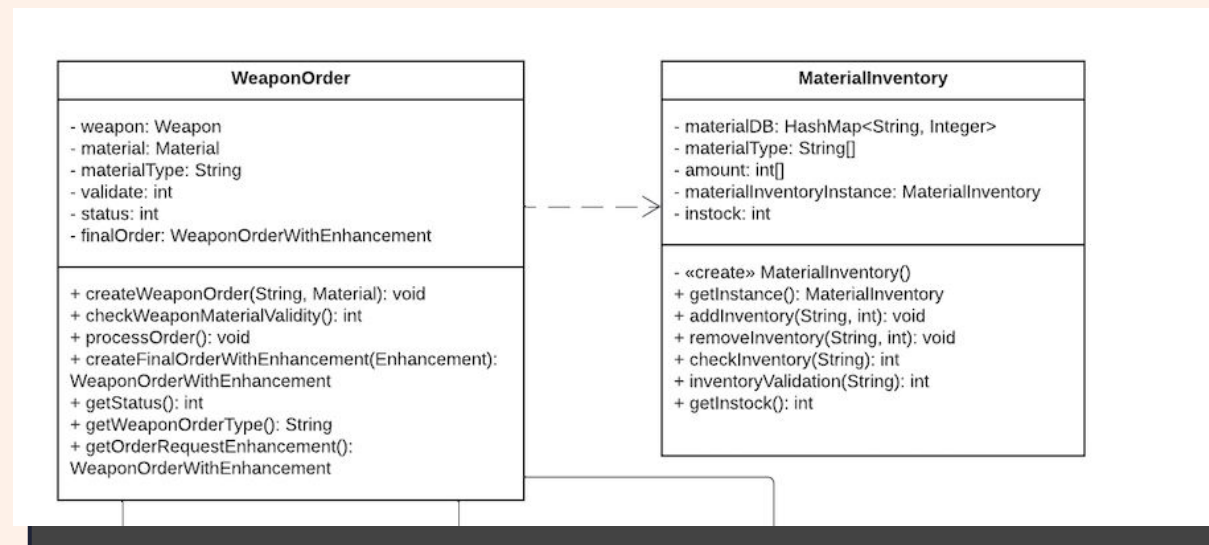


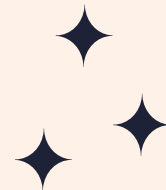
Other Pattern: Singleton

singleton pattern implementation inside the application

- Singleton pattern was implemented inside **MaterialInventory** class
- Singleton ensures a class only have one instance.
- In my case, I implemented **MaterialInventory** class as a database to store materials inventory.
- I only need ONE instance of a database to be used throughout the application.
- Therefore, Singleton pattern was implemented.

Snippet from UML diagram



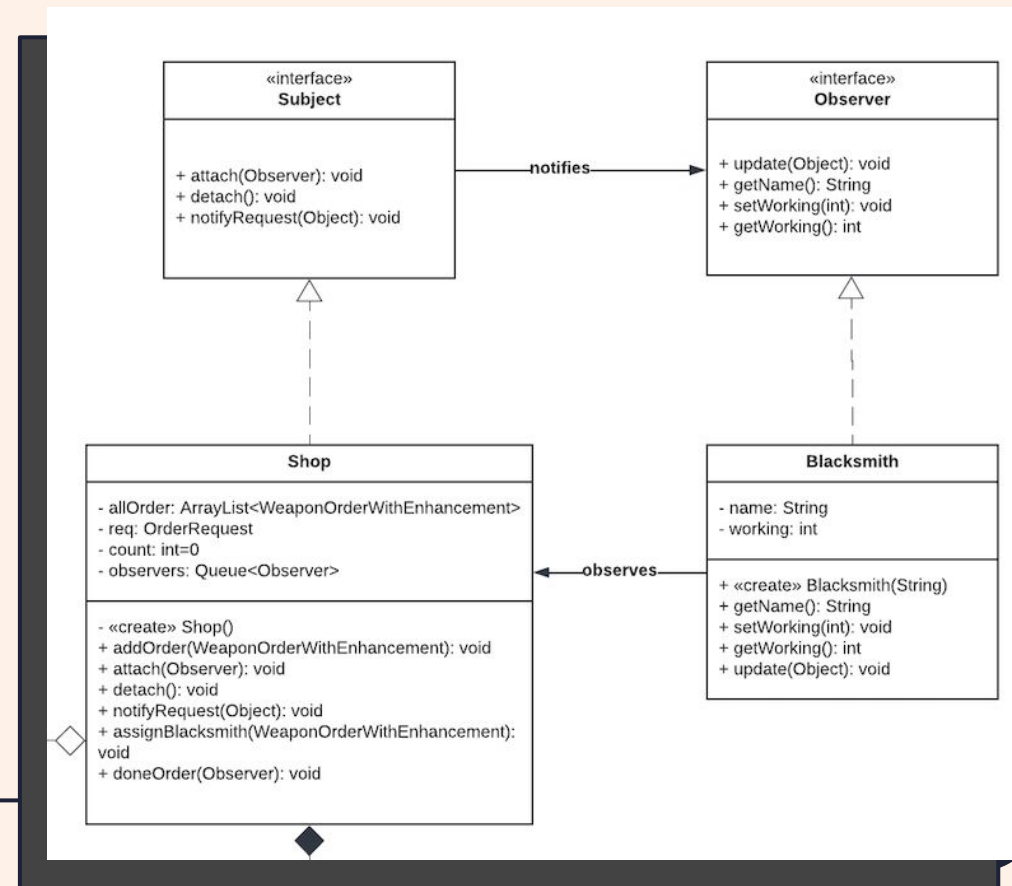


Other Pattern: Observer

observer pattern implementation inside the application

- Observer pattern was implemented for *order processing* part of the application.
- The **Shop** class is a concrete subject implementation.
 - It contains method to attach, detach, and notify observers (in this case, blacksmiths).
- **Blacksmith** is a concrete observer implementation which receives order request update from the shop.
- The implementation allows shop to notify blacksmiths of order request.
- Managing order request such as assign blacksmith to an order can be done via Shop class.

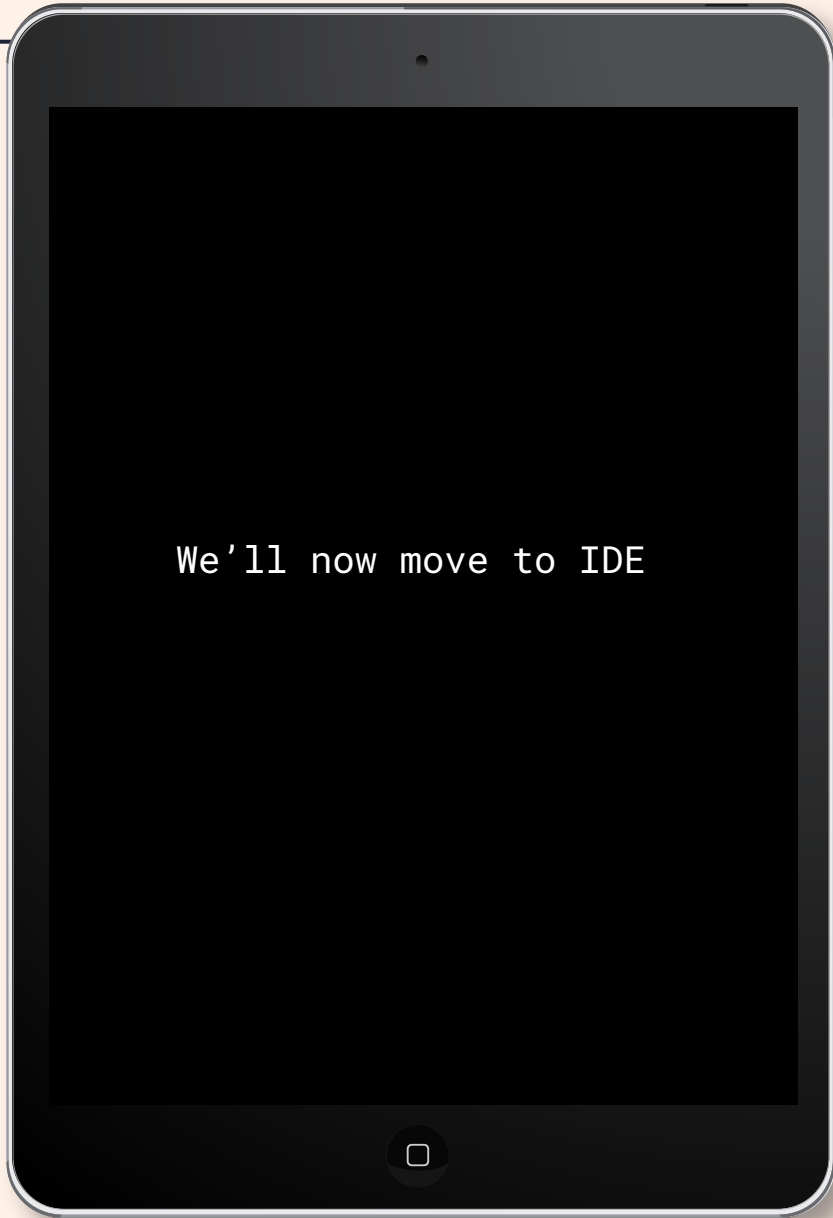
Snippet from UML diagram



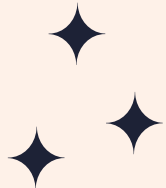
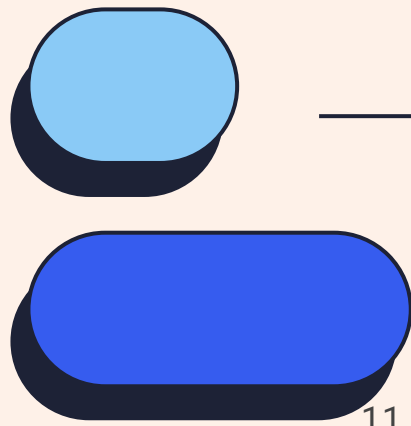


The **CODE**

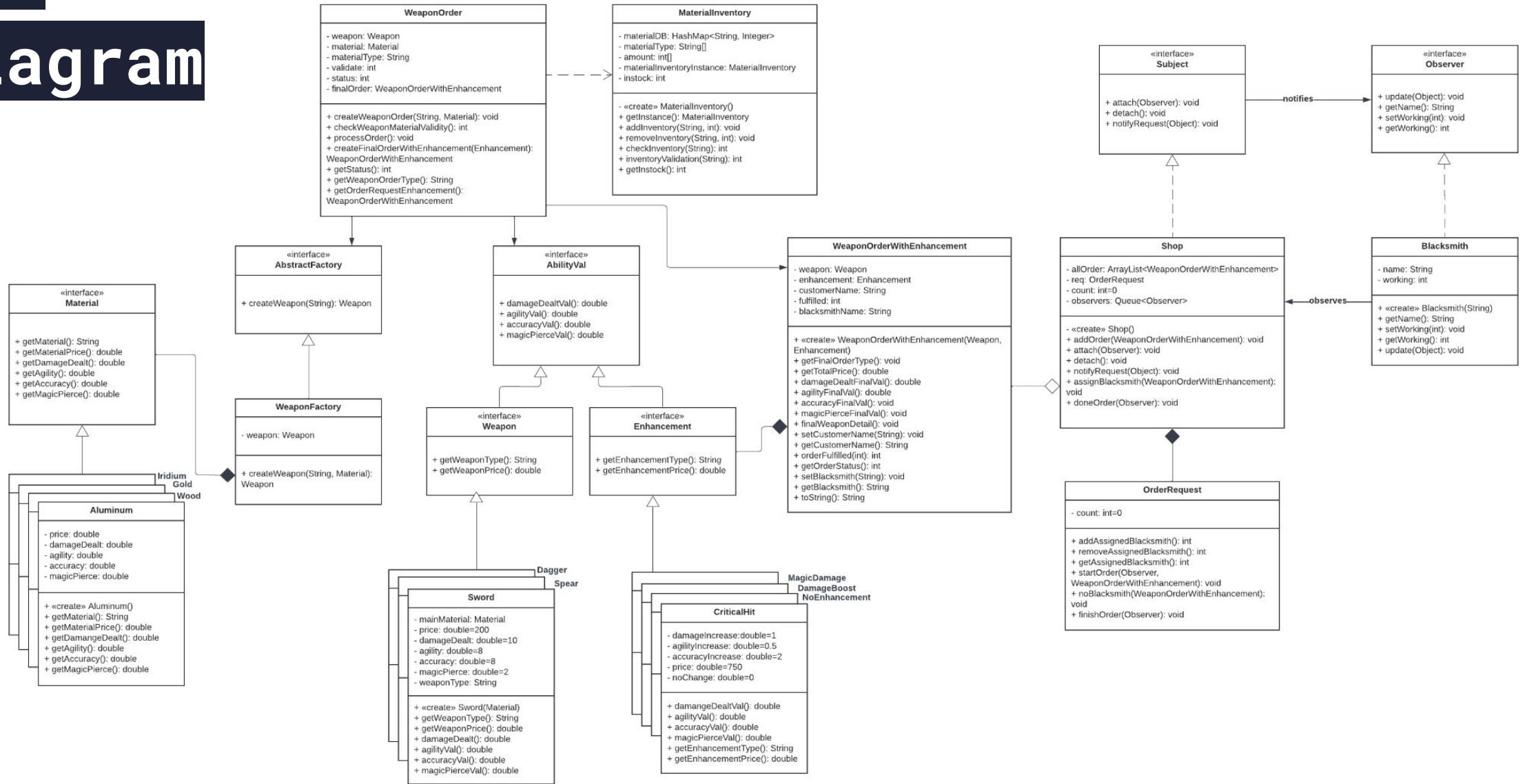
Code walkthrough and demonstration are provided via the video recording.



We'll now move to IDE



Diagram





Thank you!

Please email me if you have any questions.

pnuwapa@bu.edu

T
H
A
N
K

Y
O
U

