CS2043 Assignment 2
Design and Implementation Principles
Interface Segregation Principle

Swati Mehta & Jatin Vig

Introduction

The Interface Segregation Principle (ISP) is one of the core principles in the SOLID framework for object-oriented design. It emphasizes designing interfaces to be as small and specific as necessary, aiming to minimize side effects and reduce the need for frequent changes. Put simply, it means that a class shouldn't be required to implement methods it doesn't actually need or use. This principle helps prevent the creation of "fat interfaces"—interfaces overloaded with unrelated or unnecessary methods—which can lead to tight coupling, increased complexity, poor separation of concerns, and brittle code that's harder to maintain and extend. When a class is forced to implement methods it doesn't use, it introduces unnecessary dependencies and violates the spirit of modular design. Instead, the Interface Segregation Principle encourages developers to split large, unwieldy interfaces into smaller, more focused ones tailored to specific client needs. This way, each implementing class only deals with the functionality it actually requires, leading to cleaner, more maintainable, and flexible code. It becomes simpler to test, update, and understand. More importantly, it reduces the chances that a small change in one place will accidentally cause problems elsewhere. In the bigger picture, the Interface Segregation Principle encourages us to respect that different objects have different jobs, and our code should reflect that clearly and cleanly.

Advantages Of ISP

1. **Less Developer Confusion**: When interfaces are small and specific, it's easier for developers to understand what each one does. You're not searching through unrelated methods trying to figure out which one's matter—everything is relevant and to the point.
2. **Flexibility to Mix and Match**: We can combine just the pieces we need without dragging along extra stuff. This leads to more customizable and adaptable code structures.
3. **Faster Debugging and Maintenance:** It easier to pinpoint where the issue is and fix it without fear of breaking something unrelated.
4. **Easier Collaboration on Teams:** In team projects, developers often work on different parts of a system. ISP helps each team member focus on a clear set of responsibilities without overlapping into different areas

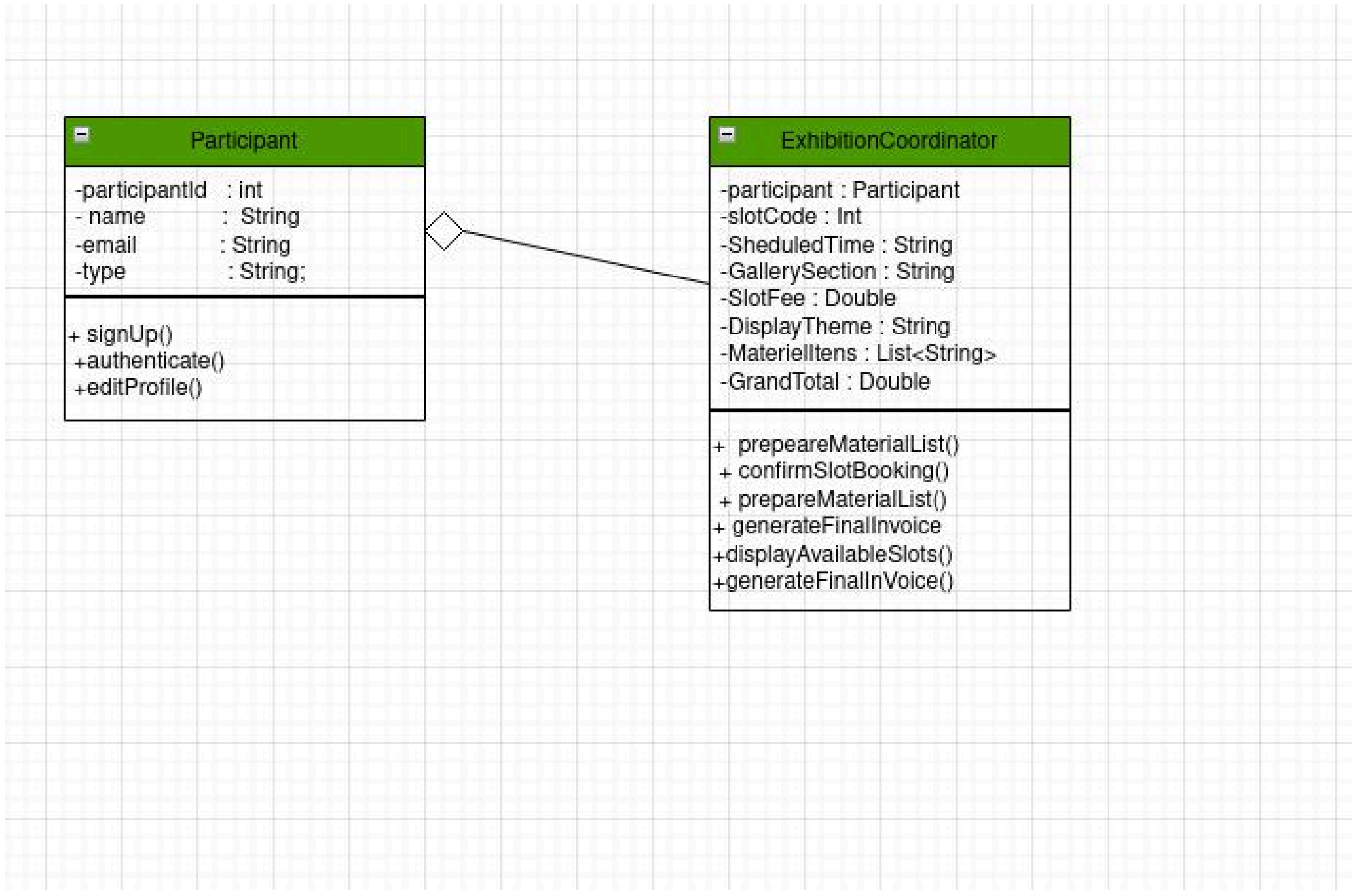
How to Apply this ISP?

Search for the Overloaded Interfaces ➡ Divide and Define Roles ➡ Give Classes Only What They Need ➡ Revisit and Refine ➡ Name with Purpose

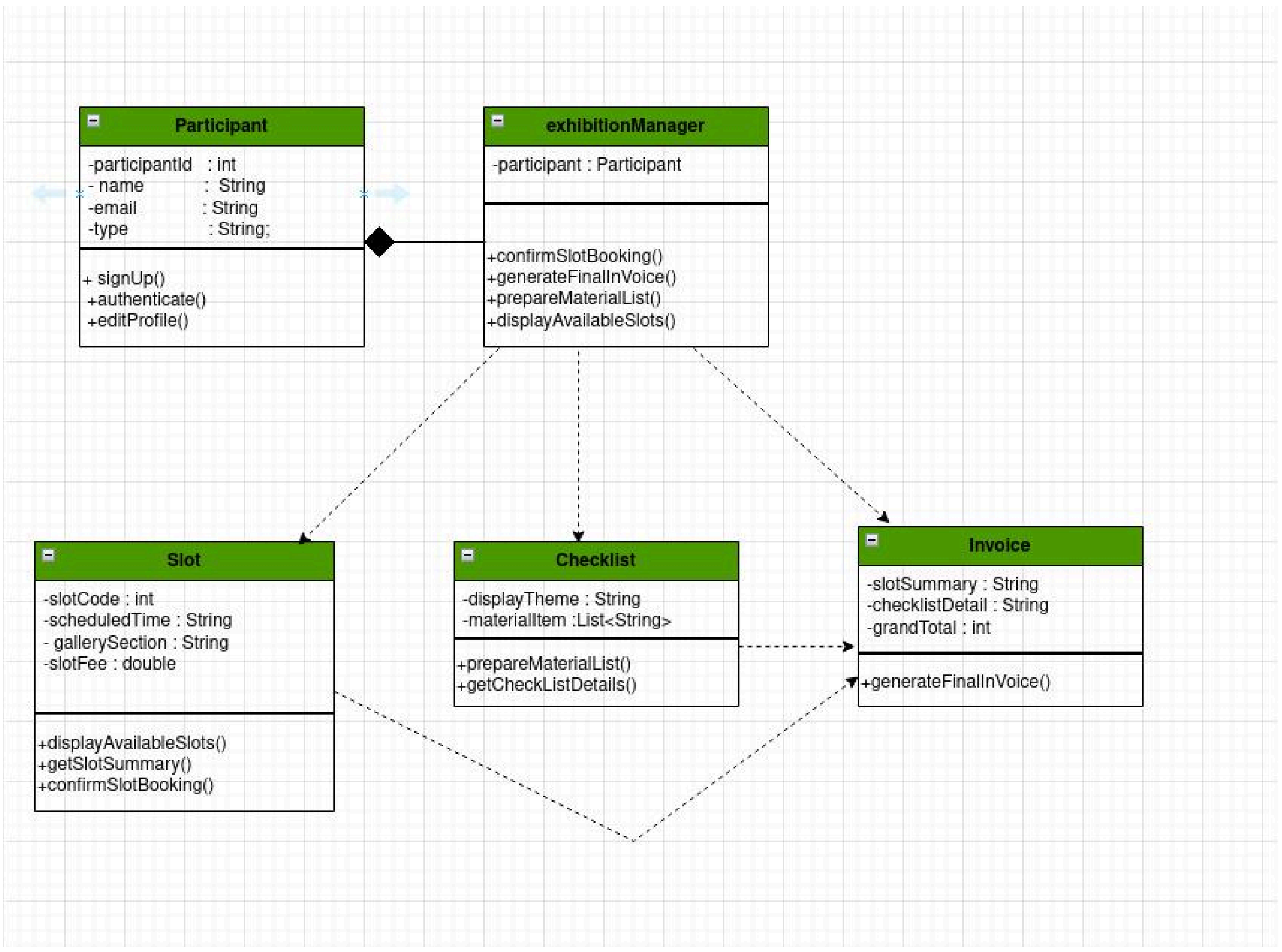
<u>Example of ISP using one our project we did in the course CS1083</u>

Art Exhibition Management System

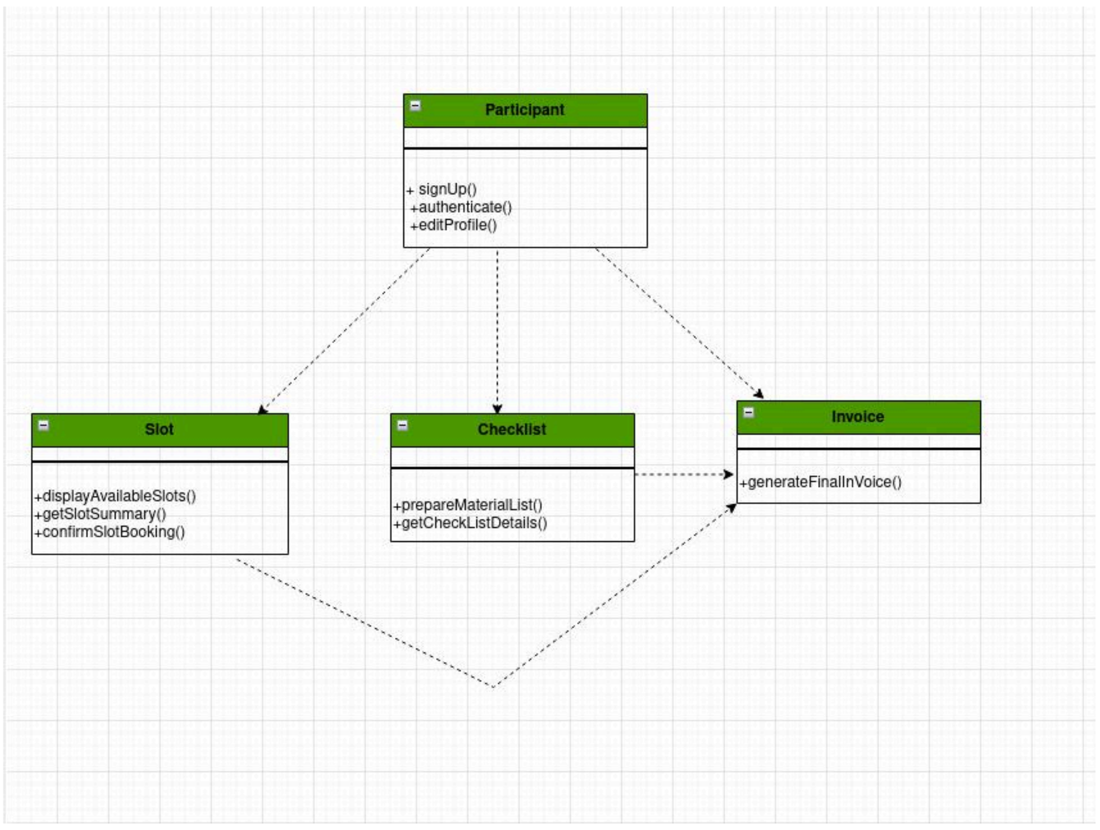
UMl Diagram



Performing ISP:


Participant
-participantId : int
- name : String
-email : String
-type : String;
+ signUp()
+authenticate()
+editProfile()
exhibitionManager
-participant : Participant
+confirmSlotBooking()
+generateFinalInVoice()
+prepareMaterialList()
+displayAvailableSlots()
Slot
-slotCode : int
-scheduledTime : String
- gallerySection : String
-slotFee : double
+displayAvailableSlots()
+getSlotSummary()
+confirmSlotBooking()
Checklist
-displayTheme : String
-materialItem :List<String>
+prepareMaterialList()
+getCheckListDetails()
Invoice
-slotSummary : String
-checklistDetail : String
-grandTotal : int
+generateFinalInVoice()

After ISP:


Participant
+ signUp()
+authenticate()
+editProfile()
Slot
+displayAvailableSlots()
+getSlotSummary()
+confirmSlotBooking()
Checklist
+prepareMaterialList()
+getCheckListDetails()
Invoice
+generateFinalInVoice()

How your example applies the principle?

Before applying ISP, our Art Exhibition Management System had classes that implemented methods they didn’t necessarily need. The ExhibitionManager class, for instance, was responsible for finding slot options, booking slots, generating invoice and preparing material lists, making it a "fat" class with multiple responsibilities.

After applying ISP, we broke these responsibilities into smaller, more specific classes:

1. slot: Handles slot-related functionalities (displayAvailableSlots(), confirmSlotBooking(). getSlotSummary).
2. Checklist: Responsible only for material list preparation (prepareMaterialList(), getChecklistSummary()).
3. Invoice: Dedicated to invoice-related operations (generateFinalInvoice()).

By doing this, each class now only depends on the methods it actually requires, preventing unnecessary dependencies and reducing tight coupling.

Why ISP is Useful in Your Example

- **Better Maintainability -** Changes to slot management logic won’t affect invoice generation or material checklist functionalities. The system is easier to update and expand with minimal risk of breaking unrelated parts.
- **Higher Flexibility -** If new functionalities need to be added (e.g., sending email confirmations for bookings), they can be implemented in a separate, relevant class without affecting the rest of the system.
- **Improved Readability & Collaboration -** The system is now structured in a way that developers can focus on specific components without unnecessary complexity. Teams working on different functionalities can do so independently, improving productivity.
- **Avoids Unnecessary Dependencies -** Each class only interacts with what it truly needs, making unit testing more efficient and reducing unintended side effects.