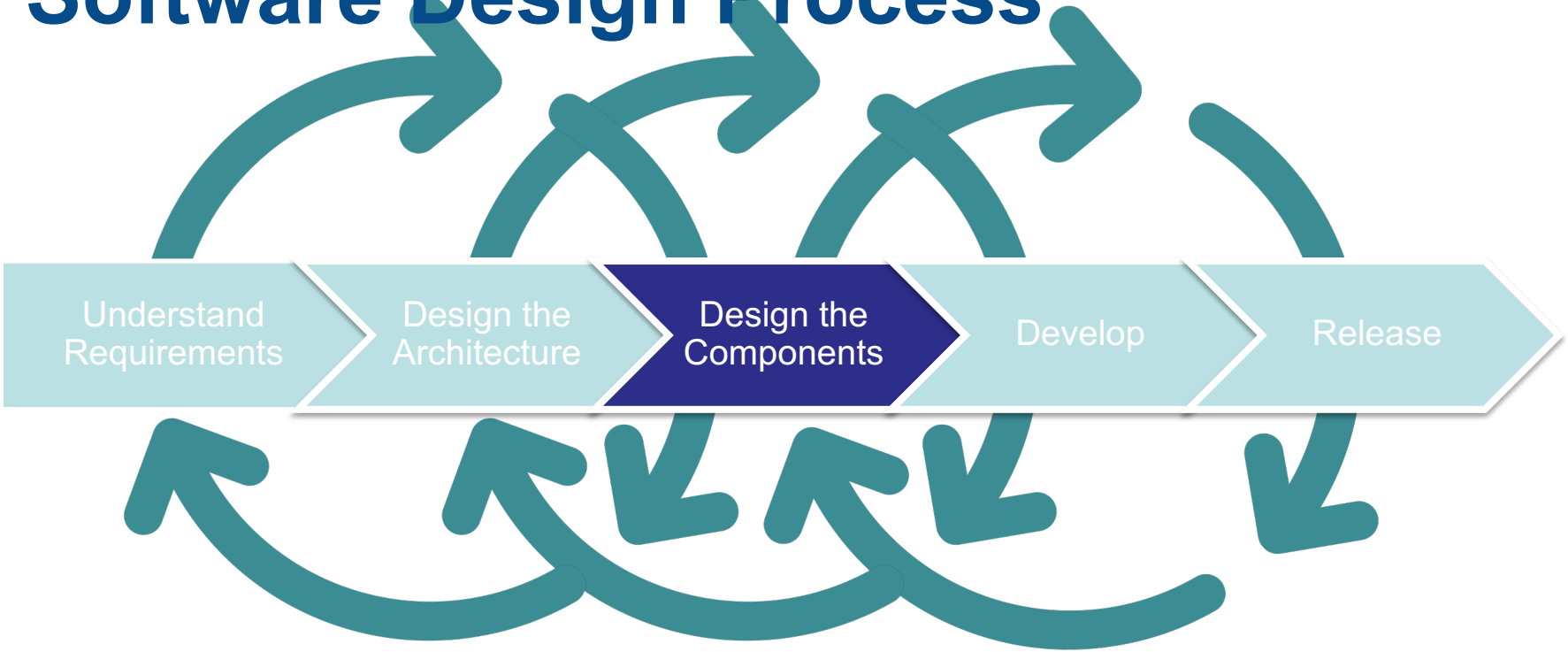
University of
South Australia

INFS 2044

Week 4
Implementation Design

Software Design Process

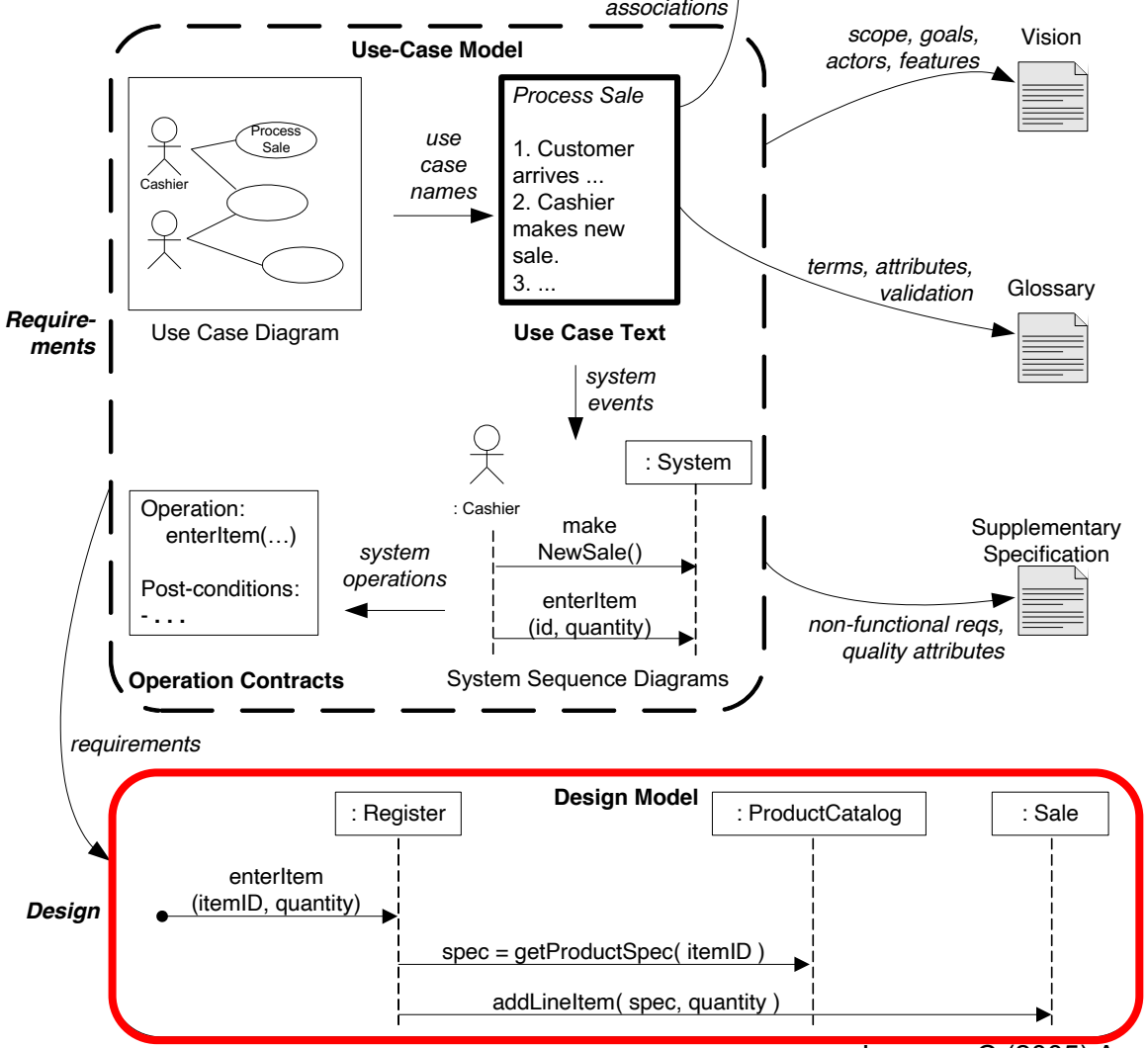


Learning Objectives

- Explain the implementation design process
- Design use case realisations using responsibilities and design principles
- Document implementation designs using UML Class- and UML Interaction diagrams

Module Design Recap

- Choosing the right abstraction
- Information hiding
- Defining and validating interfaces



University of South Australia

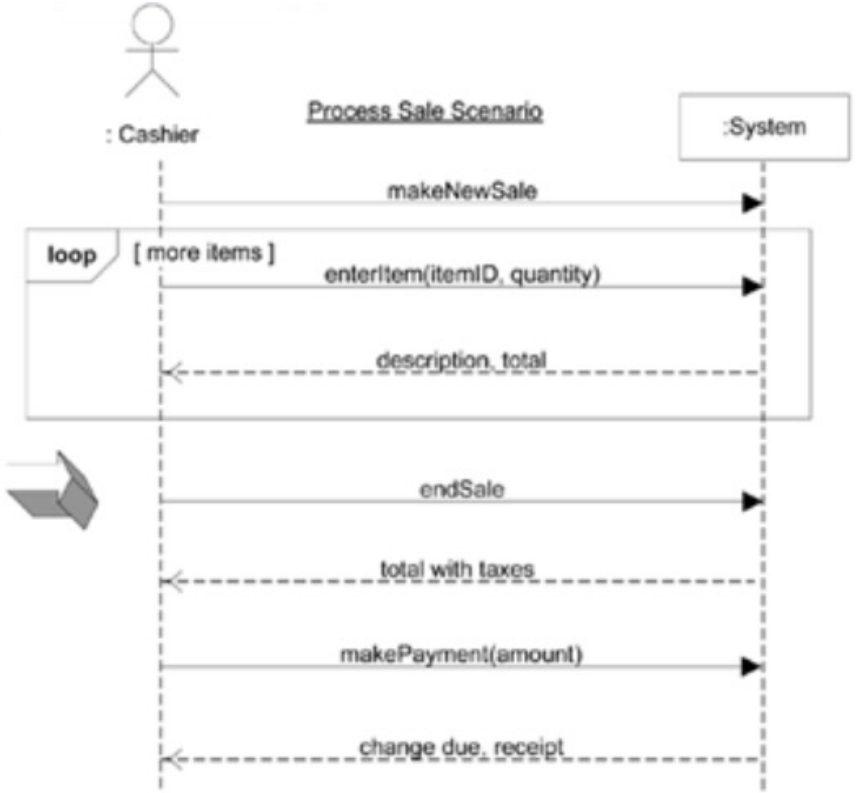
Larman, C (2005) Applying UML and Patterns

POS System Example

https://commons.wikimedia.org/wiki/File:Harbortouchterminal.jpg

Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
...



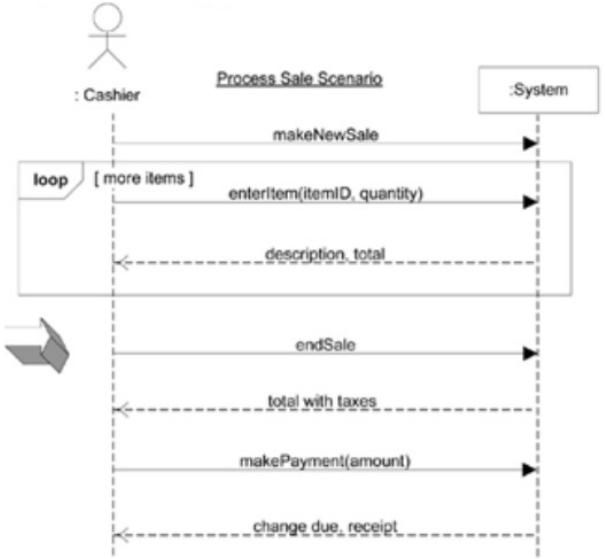
Larman, C (2005) Applying UML and Patterns

Implementation Design

- Use case realisation is the process of elaborating the detailed design for a particular use case using interactions among objects
- Define the classes, their operations, and their interactions that implement a use case scenario
- Document the classes as UML Class Diagram, and the interaction as UML Sequence- or Communication Diagram

Implementation Design Process

- For each **system input message** in the SSD:
  - Determine the internal sequence of messages that result from the **system input message**
    - » Look at the post conditions in use cases and operation contracts for inspiration
  - For each message, determine
    - » what information it needs
    - » what object (class/component) sends it
    - » what object (class/component) receives (and implements) it
  - Extend the design class/component diagram with classes/components and operations implementing the messages



Larman, C (2005) Applying UML and Patterns

Recall Operation Contract

- Operation: **enterItem(itemID: str, quantity: int)**
- Preconditions: There is a sale underway.
- Postconditions:
  - A SalesLineItem instance sli was created.
  - sli was associated with the current Sale.
  - sli.quantity became quantity.
  - sli was associated with a ProductDescription, based on itemID match.

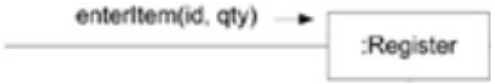
Responsibilities

- Doing
  - Doing something itself (creating objects, doing calculations, etc.)
  - Initiating an action in other objects
  - Controlling activities in other objects
- Knowing
  - Private encapsulated data
  - Related objects
  - Things it can derive or calculate

Use the Domain Model

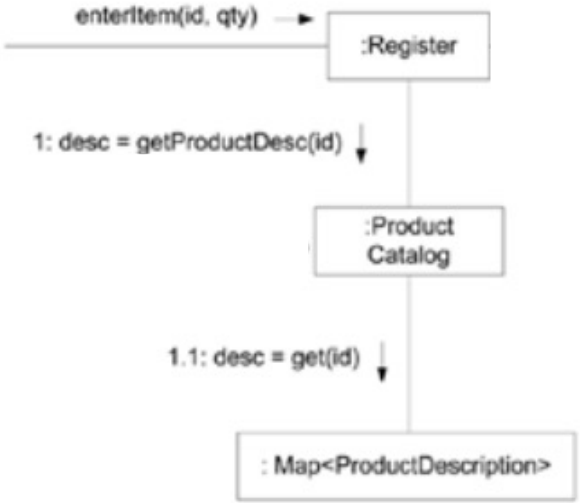
- The domain model captures the concepts and data elements that are relevant to an application domain.
  - Independent of the implementation
  - Stable
- Aligning the implementation with the domain model lowers the “gap” between domain concepts and implementation concepts
  - Makes the implementation more stable
  - Easier to understand

Incremental Behaviour Design



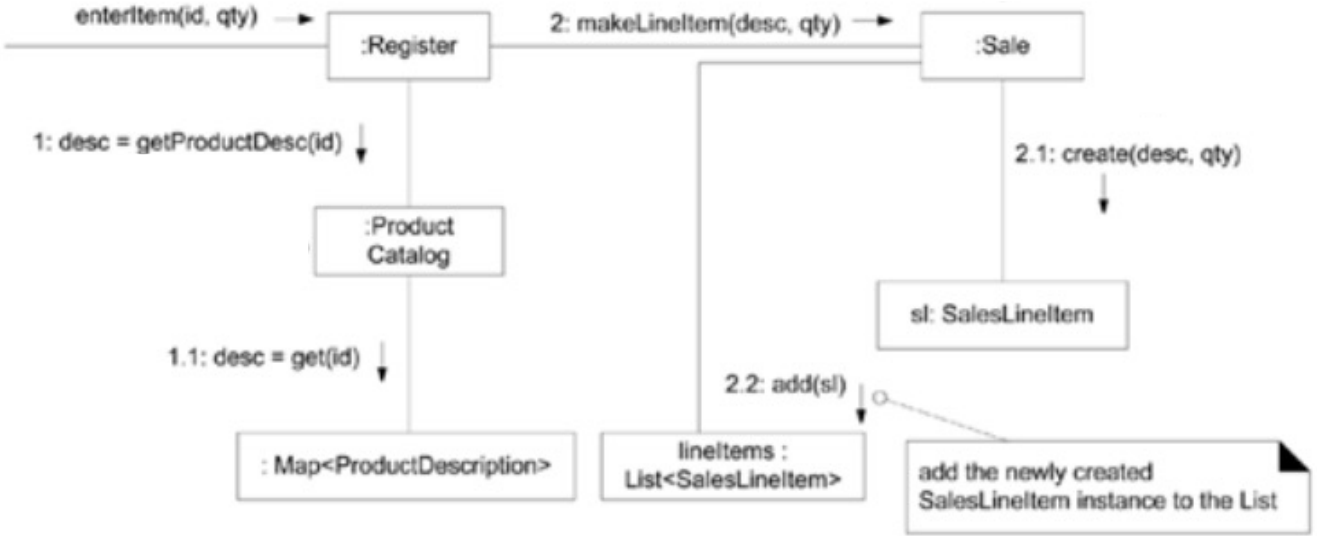
Larman, C (2005) Applying UML and Patterns

Incremental Behaviour Design



Larman, C (2005) Applying UML and Patterns

Incremental Behaviour Design

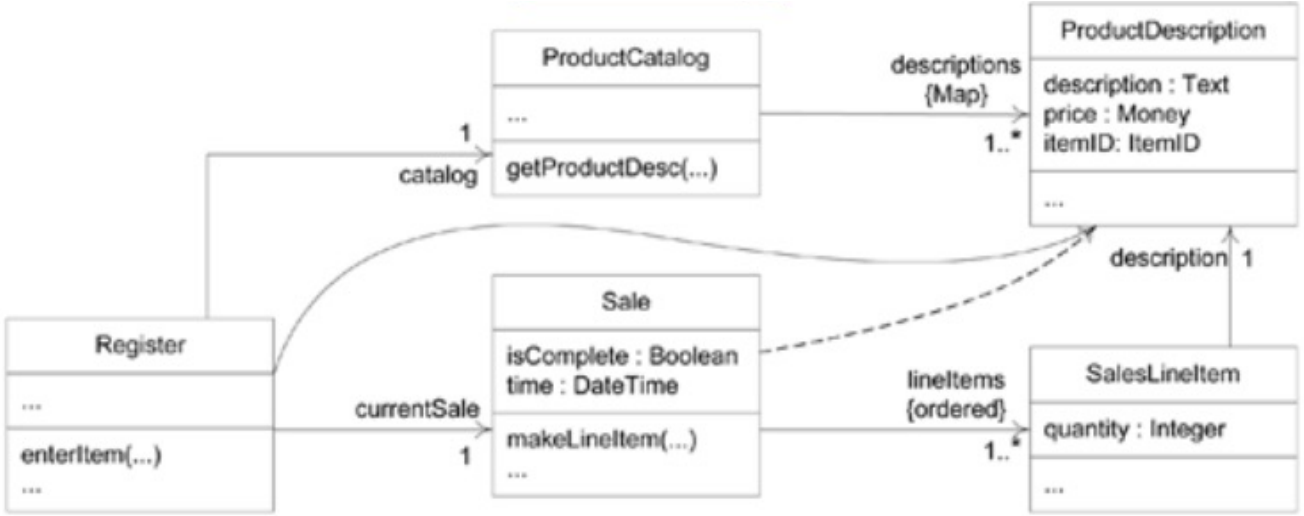


Larman, C (2005) Applying UML and Patterns

Design for Visibility

- Objects can only send messages to objects they can reach
- Object A can send a message to B if and only if
  - A holds a reference to B in a variable
  - A obtains a reference to B from another object it can reach
  - A obtains a reference to B through an operation parameter

Implementation Class Diagram



Larman, C (2005) Applying UML and Patterns

Design Principles (SOLID)

- High Cohesion
- Low Coupling

- **S**ingle Responsibility
- **O**pen-Closed Principle
- **L**iskov's Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Cohesion

- A measure of the focus or unity of purpose within a single class (high or low cohesiveness)
  - high cohesiveness if all responsibilities of a component are consistent and make sense for purpose of the component
- One class has low cohesiveness if its responsibilities are broad or makeshift
- It is best to have classes that are **highly cohesive**
- If deciding between two alternative designs, choose the one where overall cohesiveness is high

Coupling

- A measure of how closely related components are linked
  - Tightly coupled if there are lots of associations between classes/components
  - Tightly coupled if lots of operations are invoked in another class/component
- It is best to have components that are **loosely coupled**
- If deciding between two alternative designs, choose the one where overall coupling is less

Forms of Coupling

- X has an attribute of type Y
- X calls an operation of Y
- X has a method that references Y
  - parameters, local variable, returned object
- X is a subclass of Y
- X implements interface Y

Single Responsibility (SRP)

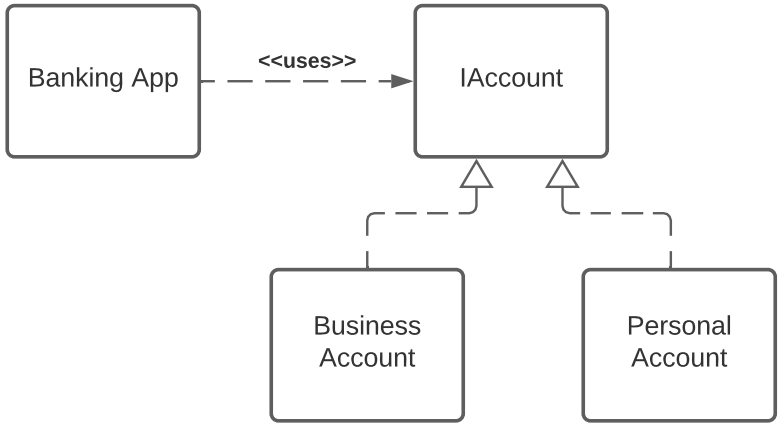
- *“A module should have one, and only one, reason to change.”*
- *“A module should be responsible to one, and only one, actor.”*
- Cohesion is the force that binds together the code responsible to a single actor.
- The SRP does NOT demand that each module does only one thing.

Open-Closed Principle (OCP)

- "*A software artifact should be open for extension but closed for modification.*"
- The behaviour of a software artifact ought to be extendible, without having to modify that artifact.
- Goal is to make the system easy to extend without incurring a high impact of change.

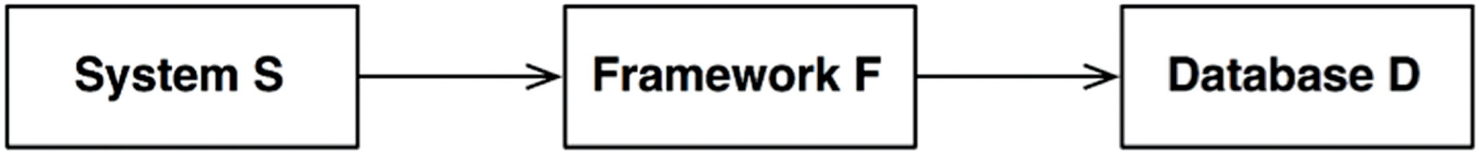
Liskov's Substitution Principle (LSP)

- Objects of a supertype shall be replaceable with objects of its subtype without breaking the program.



Interface Segregation Principle (ISP)

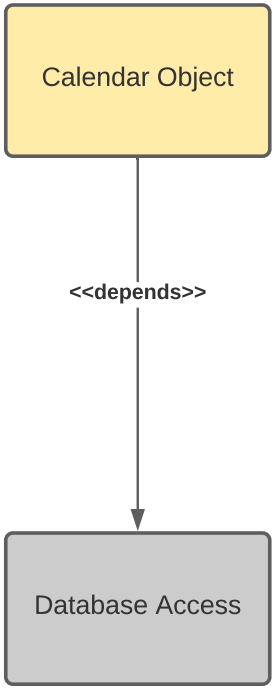
- “Clients should not be forced to depend upon interfaces that they do not use.”
  - Unrelated operations should not be in the same interface.

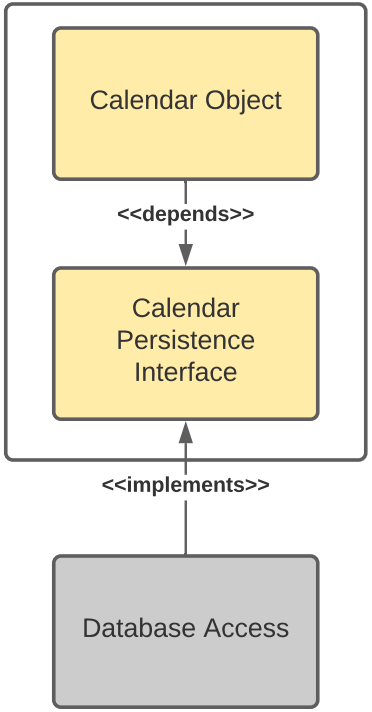


Dependency Inversion Principle (DIP)

- *“Code dependencies refer only to stable abstractions, not to (volatile) concrete elements”*
  - **Never mention the name of anything concrete and volatile.**
  - **Don’t refer to volatile concrete classes.**
    - » Refer to abstract interfaces instead.
  - **Don’t derive from volatile concrete classes.**
  - **Don’t override concrete functions.**

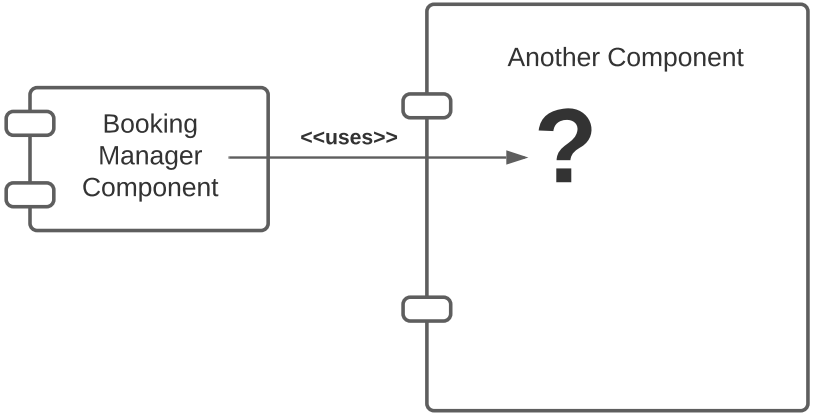
DIP: The Database is a Detail



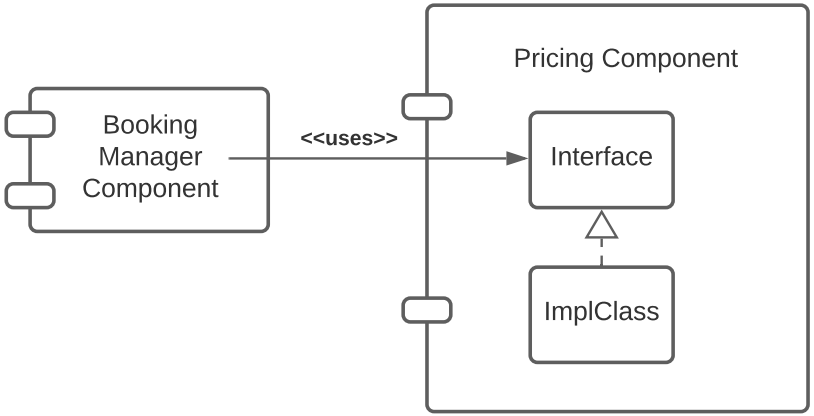


Implementing Boundaries

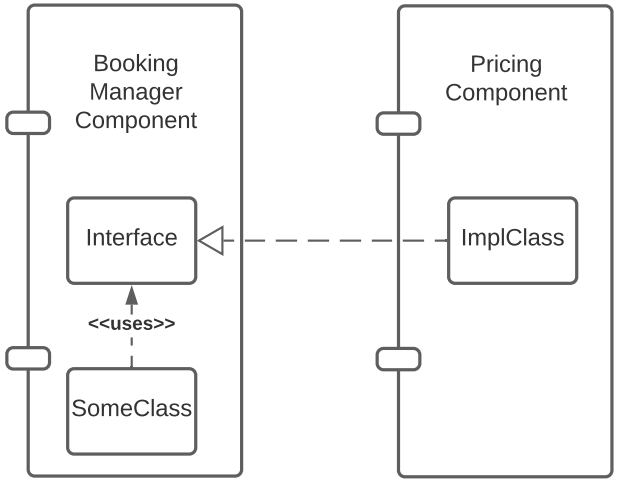
- How to implement crossing a boundary?



Simple Interface Boundary

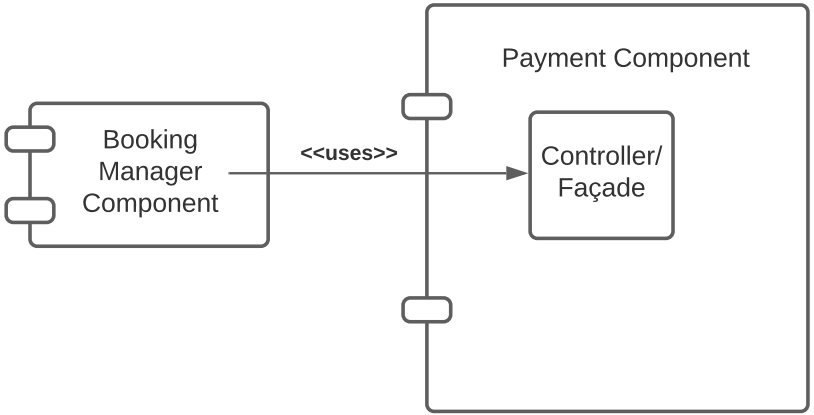


(a) Direct Dependency



(b) Dependency Inversion

Boundary Controller



Summary

- Use case realization means we take a use case scenario and produce a design that satisfies its requirements.
- We use design principles to assign responsibilities to implementation elements
- Carefully craft boundaries to manage dependencies
- Document implementation design using class- and interaction diagrams

Activities this Week

- Read the required readings
- Participate in Workshop 4
- Complete Quiz 4
- Start on working on Assignment 1

University of
South Australia