University of
South Australia

INFS 2044

Workshop 3b Answers

Preparation

- Read the required readings
- Watch the Week 3 Lecture
- Bring a copy of the workshop instructions (this document) to the workshop

Where We Are At

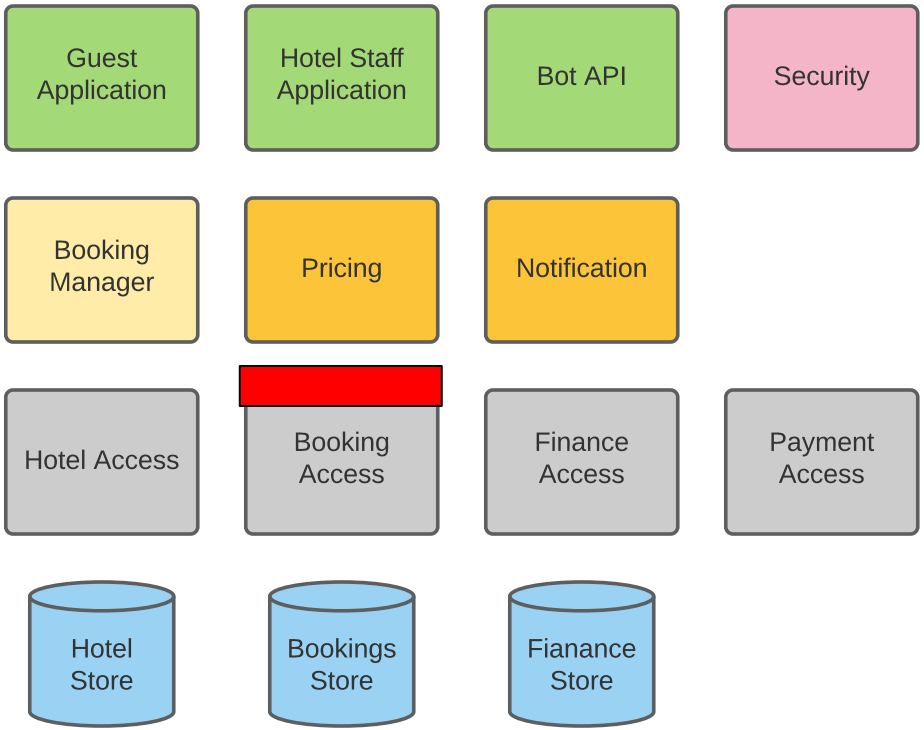
- Designed system-level operations
- Drew System Sequence Diagrams

Learning Objectives

- Design component interfaces
- Draw Sequence Diagrams to show interactions
- Correctly orient dependencies between components

Task 1. Assess Interface

- Assess the interface for the *BookingAccess* component shown on the following slides.
- Does the interface hide implementation details?
- Are the operations at an appropriate level of abstraction?


Guest Application
Hotel Staff Application
Bot API
Security
Booking Manager
Pricing
Notification
Hotel Access
Booking Access
Finance Access
Payment Access
Hotel Store
Bookings Store
Fianance Store

BookingAccess Interface #1

- bookings(): list(str) # returns a list of booking IDs
- findBookings(filter:str): list(str) # filter is an SQL query
- createBooking(): str
- setBookingRoomID(bookingID:str, roomID: str)
- setBookingCheckIn(bookingID:str, dt:Date)
- setBookingCheckOut(bookingID:str, dt:Date)
- setBookingGuest(bookingID:str, guestID: str)
- cancelBooking(bookingID:str)

BookingAccess Interface #1: Issues

- bookings()
  - Inefficient: must retrieve all bookings and filter them in caller
  - named poorly
- findBookings(str)
  - Exposes implementation details (SQL, schema)
- createBooking(), setXXX(…)
  - Poor abstraction.
  - Instead create and initialise object at once
- No operations to retrieve booking details
- No operation to check if a room is vacant during a period of time
- No operation to create Guest associated with Booking

BookingAccess Interface #2a

- getBookings(**roomID:str**, from:Date, to:Date): **list(BookingDescriptor)**
  - Operation takes a roomID, returns bookings for that room
  - BookingDescriptor is a data structure containing the booking information
- isVacant(**roomID:str**, from:Date, to:Date): **boolean**

- This interface should be fine for in-memory operations. If operations need to access a database each time and/or access via the network, this will be slow.

BookingAccess Interface #2b

- getBookings(roomIDs:list(str), from:Date, to:Date):
  map(str-> list(BookingDescriptor))
  - Operation takes a list of roomIDs, returns bookings for each room
  - BookingDescriptor is a data structure containing the booking information
- isVacant(list(roomIDs:list(str), from:Date, to:Date):
  map(str-> boolean)
- createBooking(roomID:str, in:Date, out:Date, guestID:str): str
- cancelBooking(bookingID:str): boolean

Task 2. Interaction Diagram

- Draw a Sequence Diagram for the interaction between two classes/components defined in the Python code fragments shown on the subsequent slides.

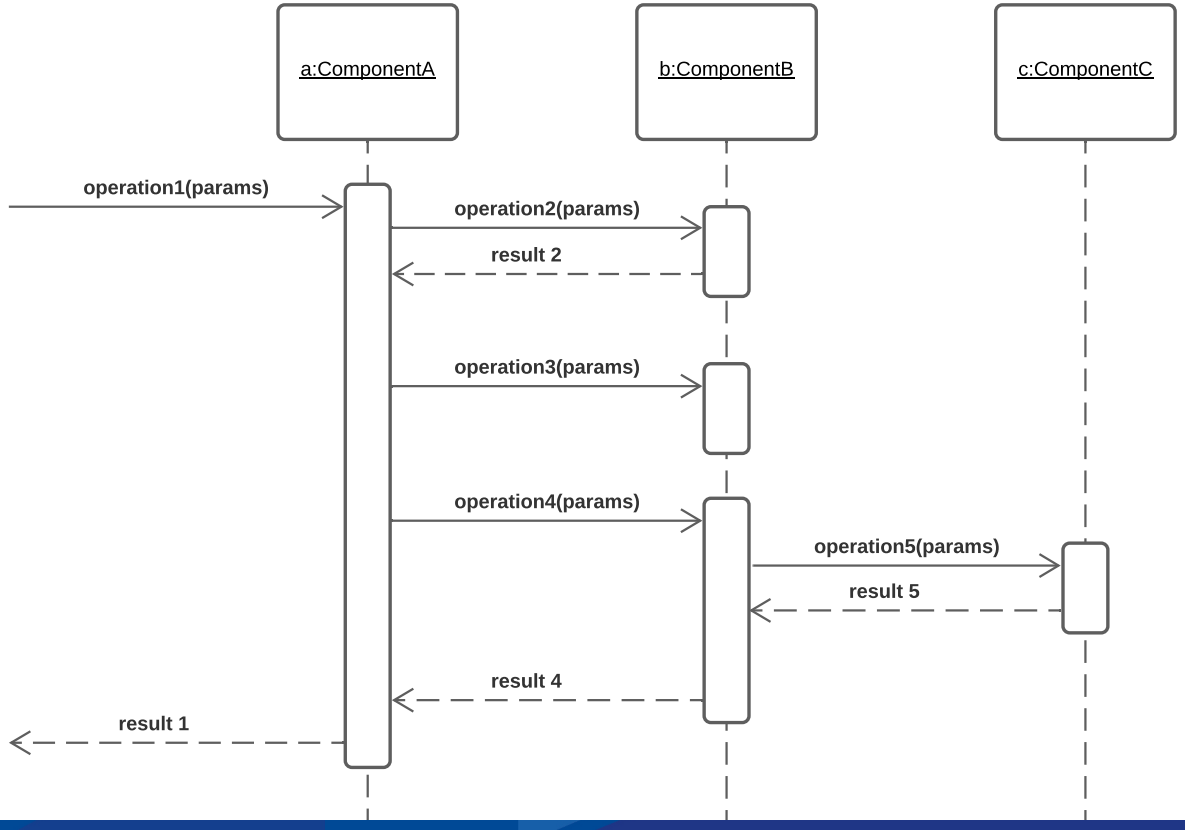
Booking Manager in Python

```
class BookingManager:
  def findRooms(...): ...
  def calculateDays(…):…

  def getBasePrice(roomID, inDate, outDate):
    roomInfo = self.hotelAccess.getRoomDetails(roomID)
    days = self.calculateDays(inDate,outDate)
    return roomInfo.dailyRate * days

  def createBooking(roomID,...):
    totalPrice = self.pricingPolicy.getTotalPrice(self, roomID, inDate, outDate)
    ...
    bookingID = self.bookingAccess.createBooking(roomID,...)
    return bookingID
```

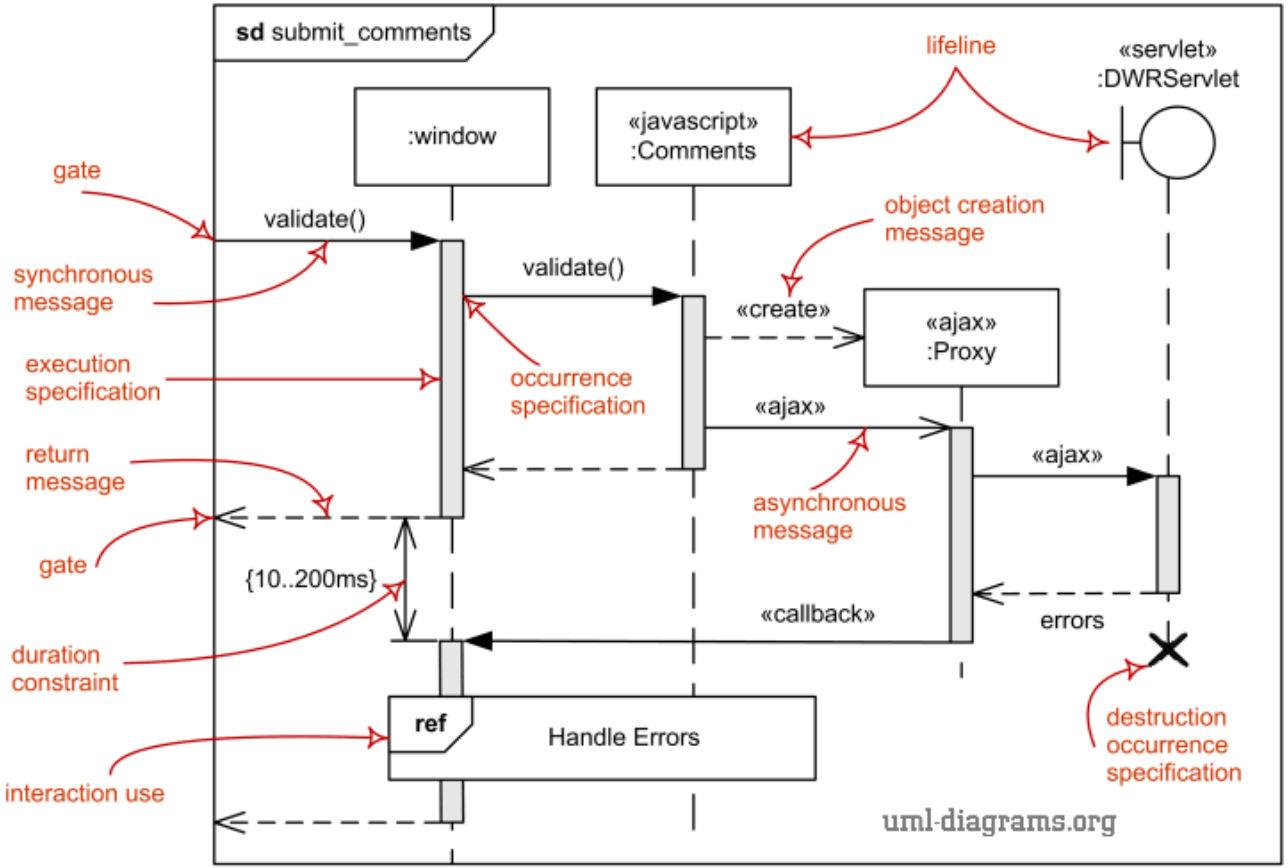
Pricing Policy in Python

```
class PercentDiscountPricingPolicy:

  def __init__(percentDiscount):
    self.percentDiscount = percentDiscount

  def getTotalPrice(manager, roomID, inDate, outDate):
    basePrice = manager.getBasePrice(roomID, inDate, outDate)
    discountedPrice = basePrice * self.percentDiscount
    return discountedPrice
```

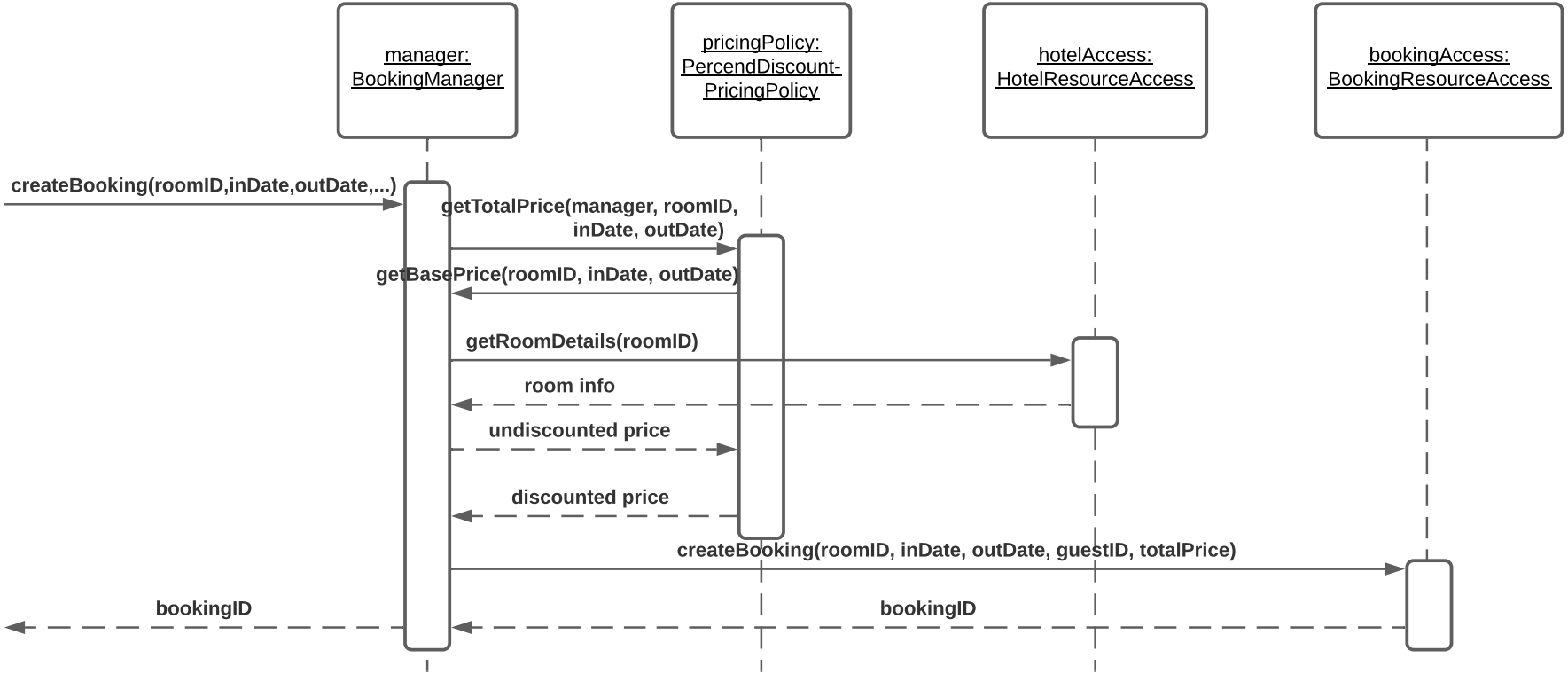
Sequence Diagram Example



UML Sequence Diagram Syntax



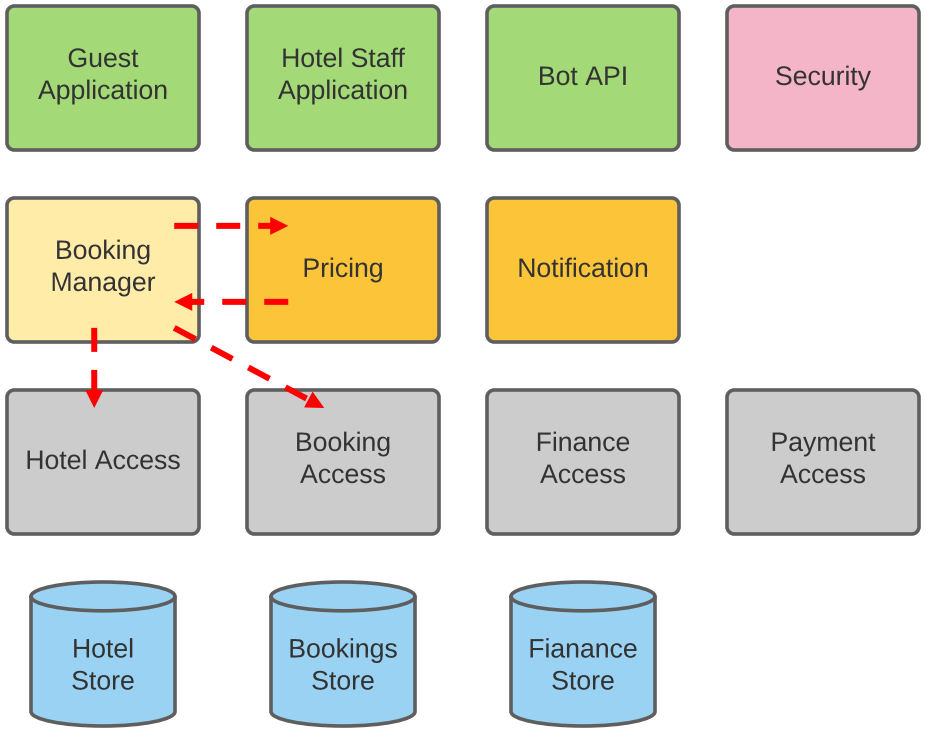
Sequence Diagram for Pricing



Task 3. Assess Boundaries

- Identify any issues that may be present in the interaction design defined in Task 2.
- Revise the design to create a better interface for the two components

Cyclic Dependency



Issues with Pricing Design

- There is a cyclic dependency
  - Pricing Policy depends on Manager, and
  - Manager depends on Pricing Policy
- Need to break the cycle between BookingManager and PricingPolicy

Revised Booking Manager

```
class BookingManager:
  def findRooms(...): ...
  def calculateDays(…):…

  def getBasePrice(roomID, inDate, outDate):
    roomInfo = self.hotelAccess.getRoomDetails(roomID)
    days = self.calculateDays(inDate,outDate)
    return roomInfo.dailyRate * days

  def createBooking(roomID,...):
    basePrice = self.getBasePrice(roomID, inDate, outDate)
    totalPrice = self.pricingPolicy.getTotalPrice(basePrice)
    ...
    bookingID = self.bookingAccess.createBooking(roomID,...)
    return bookingID
```

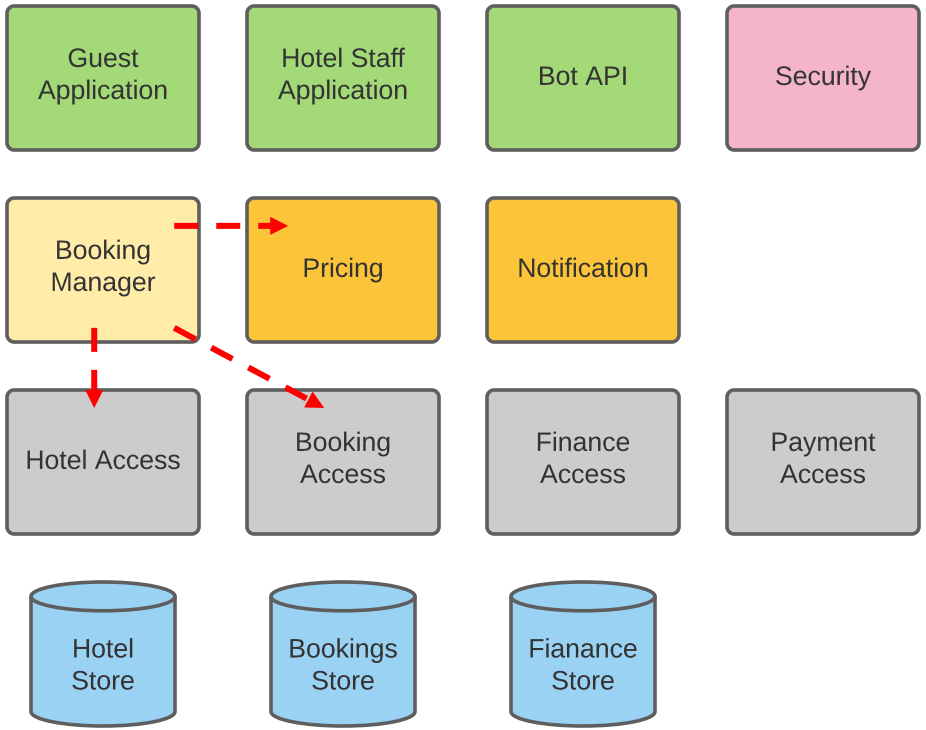
We may want to refactor createBooking() into sub-functions:

- __createGuest(…)
- __getTotalPrice(…)
- __takePayment(…)
- __createBooking(…)
- __notifyGuest(…)

Pricing Policy in Python

```
class PercentDiscountPricingPolicy:

  def __init__(percentDiscount):
    self.percentDiscount = percentDiscount

  def getTotalPrice(basePrice):
    ### no dependency on manager
    discountedPrice = basePrice * self.percentDiscount
    return discountedPrice
```

Dependencies no longer cyclic



Alternative Designs (1/2)

- Move getBasePrice(…) into Pricing Policy
  - Technically acceptable
  - May introduce code duplication since there may be multiple pricing policies
  - Mitigate by introducing an abstract base class that provides this method

Alternate Design (2/2)

- If all components are part of the same in-memory process, it may be advantageous to pass object references
  - BookingManager first creates an instance of Booking, then passes the instance to PricingPolicy
  - This provides more flexibility if other pricing policies need to be supported polymorphically; these may access different information about Booking to determine the final price

Task 4. Assess Abstractions

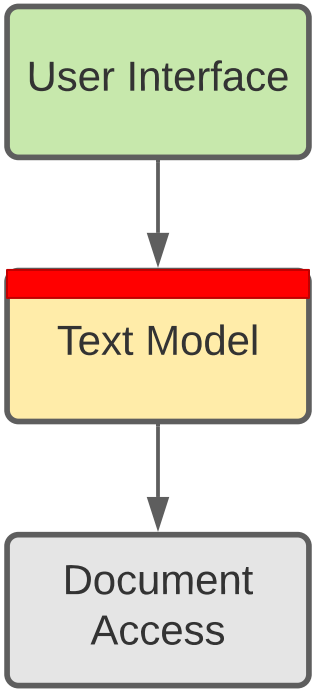
- Consider the design of a simple text editor shown on the following slides

- Discuss advantages and disadvantages of the given interface design.
- Would the component be easily reusable?
- Is the interface at the right level of abstraction?
- Is the module deep or shallow?

Text Editor Use Cases

- Load, edit, save plain text documents
- Search & replace text in the current document
- Copy and paste selected text

Text Editor Architecture



Text Editor Model Interface

- backspace(cursor:Cursor)
- delete(cursor:Cursor)
- deleteSelection(selection:Selection)

- Cursor represents the position of the cursor
- Selection represents the range of text that is selected

Issues with this Design

- Shallow abstraction
  - Replicates the user interface operations
- Large number of shallow methods
  - Each new UI operation requires a new method in the model
- Information leakage from UI into model
  - Cursor and Selection are UI concepts

Improved Text Editor Model Interface

- insert(pos:Position, newText:str)
- delete(start:Position, end:Position)
- changePosition(pos:Position,numChars:int): Position
- findNext(pos:Position,text:str): Position

- General purpose interface
- Position is independent of the UI

You Should Know

- Detect information leakage
- Assess the quality of abstraction in an interface
- Define interfaces at an appropriate level of abstraction
- Correctly orient dependencies between components
- Draw Sequence Diagrams showing interactions

Activities this Week

- Complete Quiz 3

University of
South Australia