

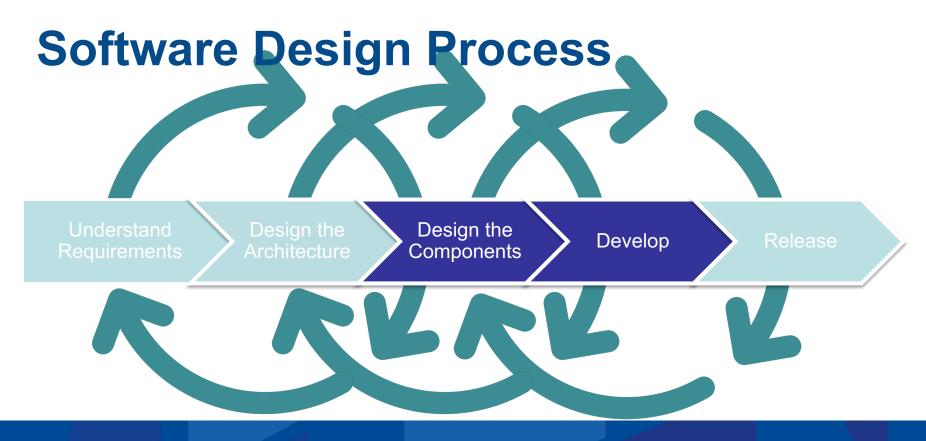
INFS 2044

Workshop 4b

Preparation

- Read the required readings
- Watch the Week 4 Lecture
- Bring a copy of the workshop instructions (this document) to the workshop







Where We Are At

- Designed system- and component- interfaces
- Drew Sequence, Communication, and Class Diagrams
- Assessed feasibility of an interaction design



Learning Objectives

- Develop use case realisations through interaction design and design principles
- Apply the Dependency Inversion principle to manage dependencies

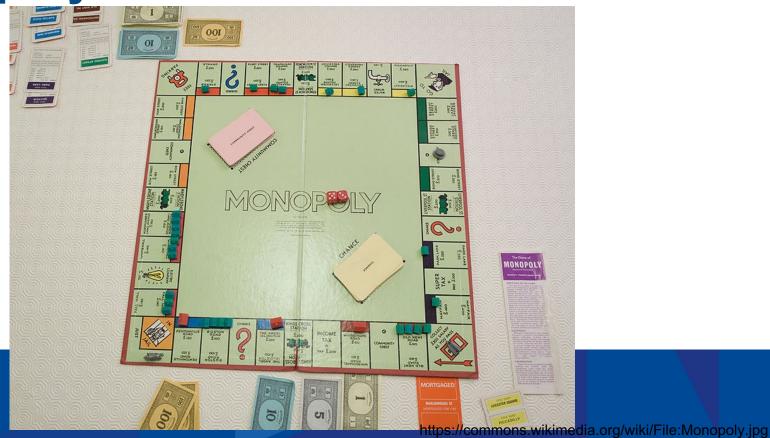


Task 1. Iterative Impl. Design

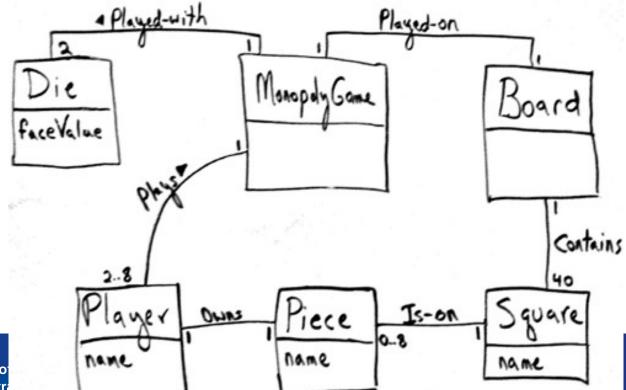
- Design an implementation for a simple Monopoly Game
- Your tutor will lead this activity.



Monopoly

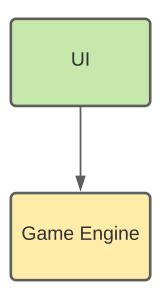


Domain Model





Component Architecture



System Operations

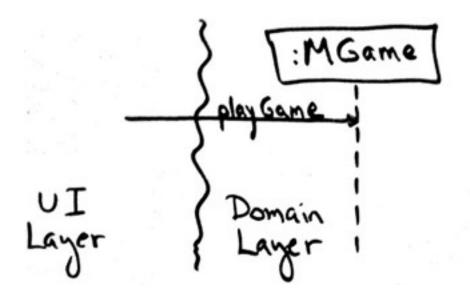
- Initialize()
- playGame()

Heuristic #1: Choosing a Manager

- Design Heuristic
 - System operations should be received by a dedicated object that orchestrates the implementation of these operations
 - Represents the overall "system," "root object," a specialized device, or a major subsystem.
 - Called *-Manager, *-Controller, *-Handler
- Related patterns
 - Controller
 - Façade



Introducing the Manager





Partial Class Diagram

MGame

- + initialize()
- + playGame()

The Game Loop

```
for N rounds:
    for each player p:
        p takes a turn
```

Who is responsible for controlling the game loop?

- The Manager?
- Another object?

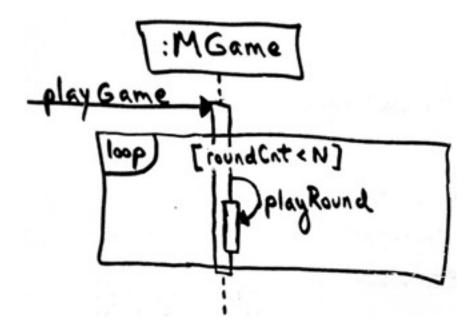


Heuristic #2: Information Need

Information Needed	Who Has the Information?
the current round count	No object has it yet, but the domain model suggests assigning this responsibility to the MonopolyGame object is justifiable.
all the players (so that each can be used in taking a turn)	Taking inspiration from the domain model, MonopolyGame is a good candidate.



Implementing the Game Loop





Partial Class Diagram

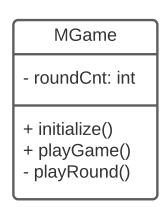
MGame

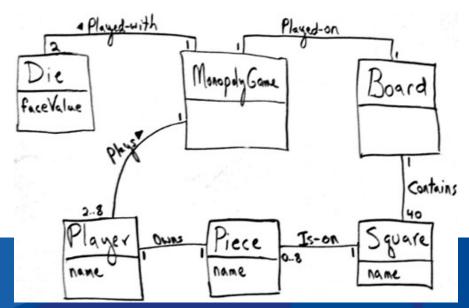
- roundCnt: int

- + initialize()
- + playGame()
- playRound()

Taking a Turn

 Involves rolling the dice and moving a piece to the square indicated by the total dice face values





Who has the information?

Information Needed	Who Has the Information?
current location of the player (to know the starting point of a move)	Domain model suggests that a <i>Piece</i> knows its <i>Square</i> and a <i>Player</i> knows its <i>Piece</i> . Therefore, a <i>Player</i> software object could know its location.
the two <i>Die</i> objects (to roll them and calculate their total)	Taking inspiration from the domain model, <i>MonopolyGame</i> is a candidate since we think of the dice as being part of the game.
all the squares—the square organization (to be able to move to the correct new square)	the <i>Board</i> is a good candidate.

Heuristics to Apply #3

- Who has the majority of information? (supports low coupling)
 - Does not help here
- When there are alternative design choices, consider the coupling and cohesion impact of each.
 - Giving MonopolyGame more work impacts its cohesion.
 Player and Board are not doing anything yet, but no clear winner among them.

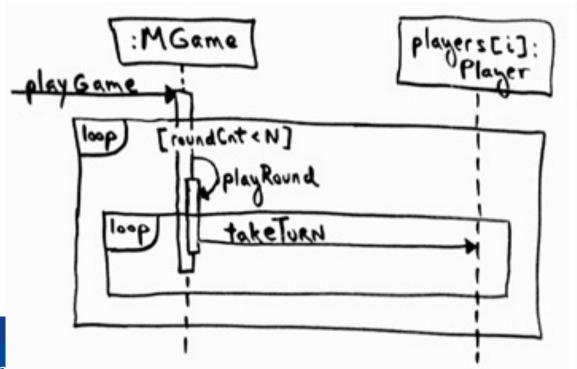


Heuristics to Apply #4

- When there is no clear winner, consider future evolutions of the software (volatility)
 - For the full game, taking a turn involves spending cash to buy property
 - The Player object knows a player's cash total
 - Player is a good candidate (but not because the human player takes the action)

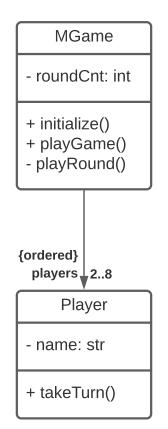


Taking a Turn Interaction





Partial Class Diagram





Taking a Turn Actions

- 1. calculating a random number total between 2 and 12 (the range of two dice)
- 2. calculating the new square location
- 3. moving the player's piece from its old location to its new square location



Taking a Turn Implementation (1/2)

- 1. calculating a random number total between 2 and 12 (the range of two dice)
 - Domain Model suggests creating a Die object
- 2. calculating the new square location
 - The Board knows all the squares and hence can calculate the new square location given the old location and an offset
- 3. moving the player's piece from its old location to its new square location



Taking a Turn Implementation (2/2)

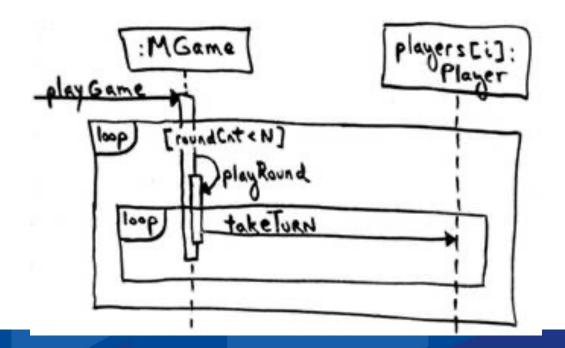
- 3. moving the player's piece from its old location to its new square location
 - Player to know its Piece, and a Piece its Square location (or even for a Player to directly know its Square location).
 - Piece will set its new location, but it may receive that new location from its owner, the Player.
 - The Player coordinates all this as it is responsible for taking a turn



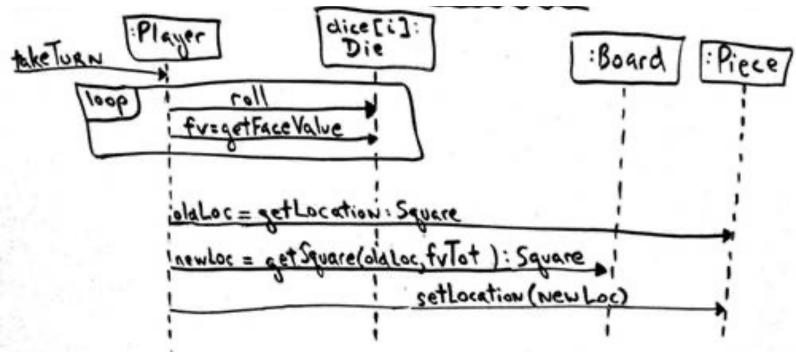
Taking a Turn Implementation (3/2)

- 3. moving the player's piece from its old location to its new square location
 - Player must collaborate with Die, Board, and Piece.
 - Player must have visibility (a reference) to these objects
 - Initialise Player with references to these objects at start-up

Final Design (1/3)

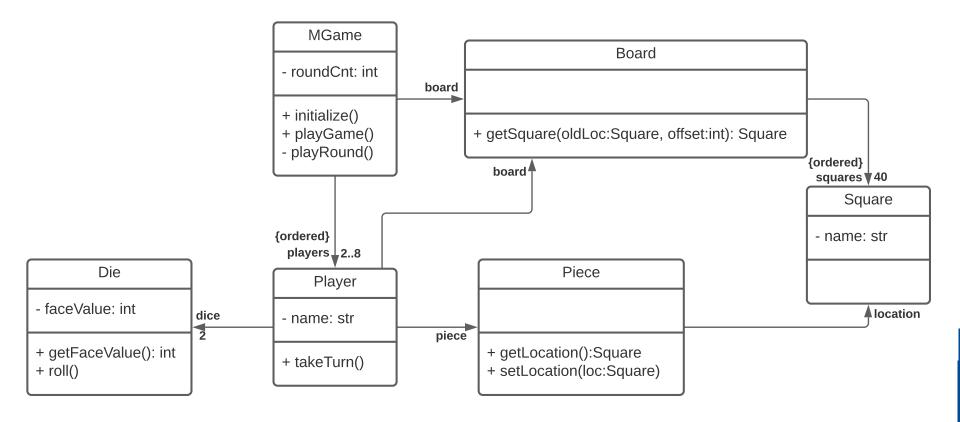


Final Design (2/3)





Final Design (3/3)



OO Design Process Summary

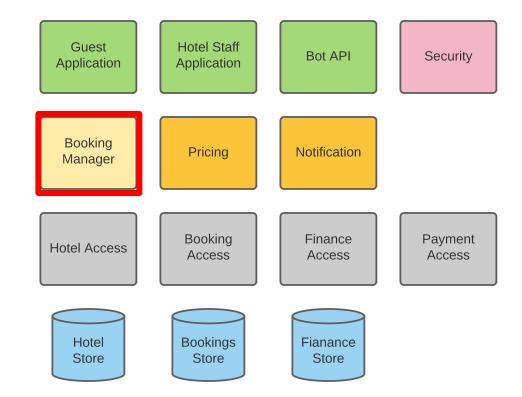
- Introduce Classes inspired by Domain Model
 - Lowers "Representational Gap"
- Allocate responsibilities according to Design Principles and heuristics as guides
- Incrementally build the interaction sequences and the class diagram



Task 2. FindRooms() Implementation

- Design an implementation for system operation findRooms(...) in the BookingManager component in the Booking System architecture.
- Use the component decomposition on the next slide instead a domain model for populating the design.



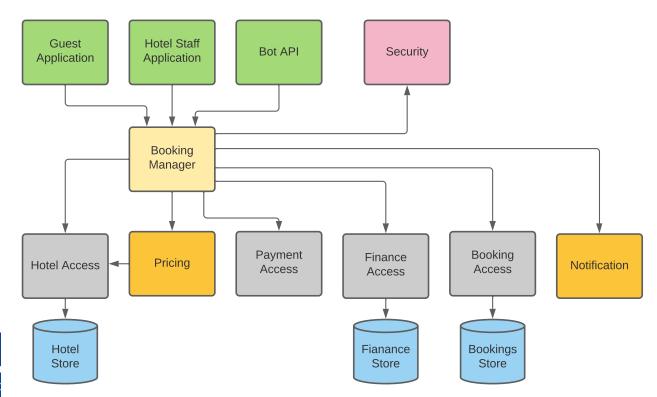


UC01 Make Booking Main Scenario

- 1. User enters date range.
- 2. System presents available rooms, their descriptions, and daily rates.
- 3. User selects room.
- 4. System presents total price.
- 5. User enters contact details and payment details.
- 6. System verifies that room is available for the period, creates booking for the room and associates booking with guest, verifies payment, records payment confirmation, and issues booking confirmation.



Comm. Diagram for Make Booking





Booking Manager Operations

- findRooms(in:Date,out:Date): list(RoomDescriptor)
- getTotalPrice(roomID:str, in:Date, out:Date): int
- createBooking(roomID:str, in:Date, out:Date, pd:PaymentDetails, cd:ContactDetails): str



findRooms(...) Contract

- findRooms(checkin:Date,checkout:Date): list(RoomDescriptor)
 - RoomDescriptor is a data structure that includes
 - » room ID
 - » room type
 - » description
 - » daily rate in AUD
- Preconditions: none
- Postconditions: none



BookingAccess Interface

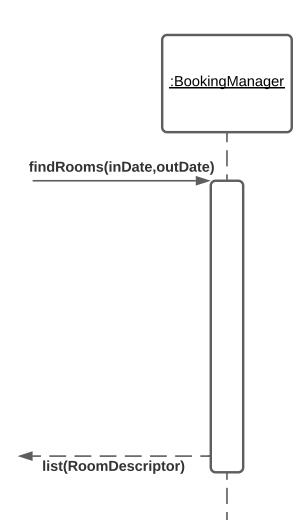
- getBookings(roomIDs:list(str), from:Date, to:Date):
 map(str-> list(BookingDescriptor))
 - Operation takes a list of roomIDs, returns bookings for each room
 - BookingDescriptor is a data structure containing the booking information
- isVacant(list(roomIDs:list(str), from:Date, to:Date): map(str->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



Design Implementation Sequence

- Design the implementation sequence for findRooms(...) in the BookingManager
- Draw an interaction diagram showing the interaction sequence among operations in the components
- Stop at the boundaries of HotelAccess and BookingAccess
 - Don't show how these operations are implemented







Task 3. Dependency Inversion

- Consider the implementation of the Booking Manager on the following slide.
- Examine how the Booking Manager and the Resource Access components interact
- What implications are there in terms of coupling, cohesion, and information leakage?



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.



You Should Know

- Apply heuristics and design principles to incrementally design an object-oriented implementation of component interfaces
- Apply the Dependency Inversion Principle to decouple concrete implementations



Activities this Week

- Complete Quiz 4
- Start on working on Assignment 1





University of South Australia