W4156

Cementing

OOD

The class had great command of the majority of sections of our big worked example.

The one area where people got sightly tentative was the tail end of OOD where we fleshed out a conceptual class diagram by continually testing it with use-cases and a sequence diagram

Let’s work through an example ….

CU@Gym

- We were building a system to reserve machines at Columbia gym
- Our "why" was more productive gym sessions
- We found 3 personas: Gym Users (free), Alumni Gym Users (paying), Facilities Management
- We produced a roadmap of features (and made some tough tradeoffs where we decided to satisfy. We decided to satisfy the core constituency first before expanding later)
- We established a set of non-functionals including availability, capacity and scalability

	MVP	M2	M3
Reservation	View Availability Reserve Cancel		
Payment		Pay	
Analytics		View Machine Usage	
Other			e-Lockers

CU@Gym

- When we did sprint planning we found we could only accommodate two features in the first sprint

	MVP	M2	M3
Reservation	**View Availability** **Reserve** Cancel	Gym attendance stats	Notify broken machine Suspend no-shows
Payment		Pay	
Analytics		View Machine Usage	
Other			e-Lockers

- *Note: we only need to address the scope of the sprint*
  - *we don’t have time to build all the other requirements*
  - *After sprint 1 we may get feedback that changes direction anyway*

CU@Gym

- Our next challenge is to *design the structure of the code.*

- *Note: we only need to address the scope of the sprint (we don't have time to build all the other requirements within this sprint && as we demo at the end of the sprint it may change)*

- We have a set of human language requirements but need to design the structure of the code that meets the requirements and no more …..

- If we choose OO paradigm then we need to
  - Identify entities
  - Map the relationships between these entities
  - Understand for each entity what data and methods it support ….

- Thankfully we had a 'turn-handle' process

OOAD: Problem Domain to Code

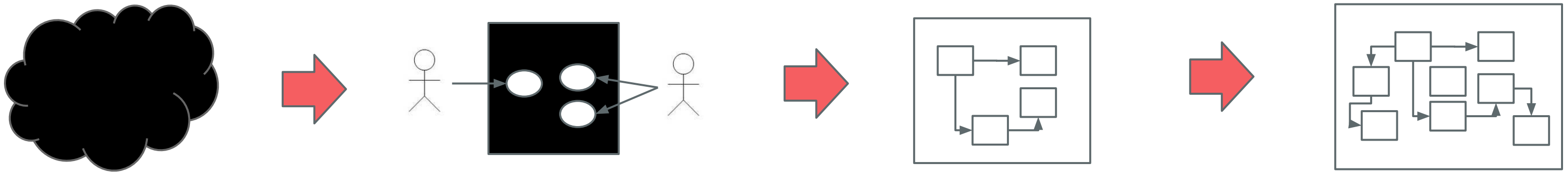
We are trying to design a system (currently “fuzzy/unclear”)

What does it need to do?

(scope and functions)

Conceptual model

Implementable Design



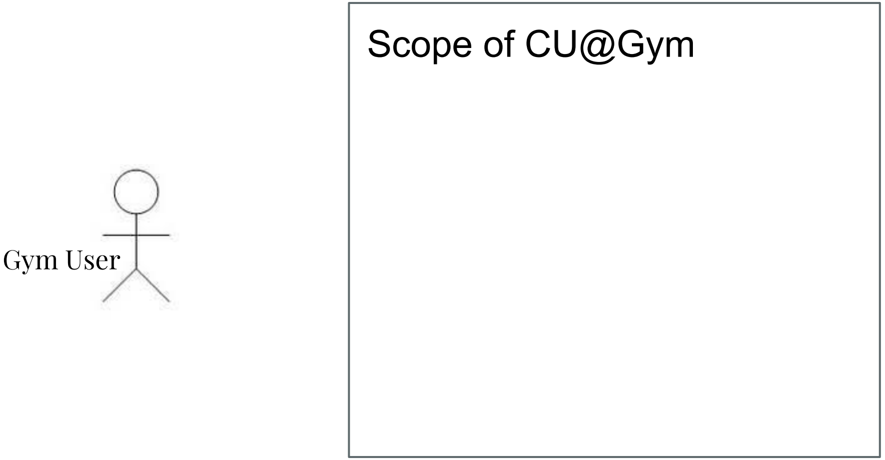
(use-cases)

(conceptual class diagram)

(implementation level class diagram)

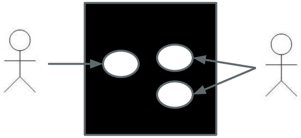
Use-Cases

(use-cases)

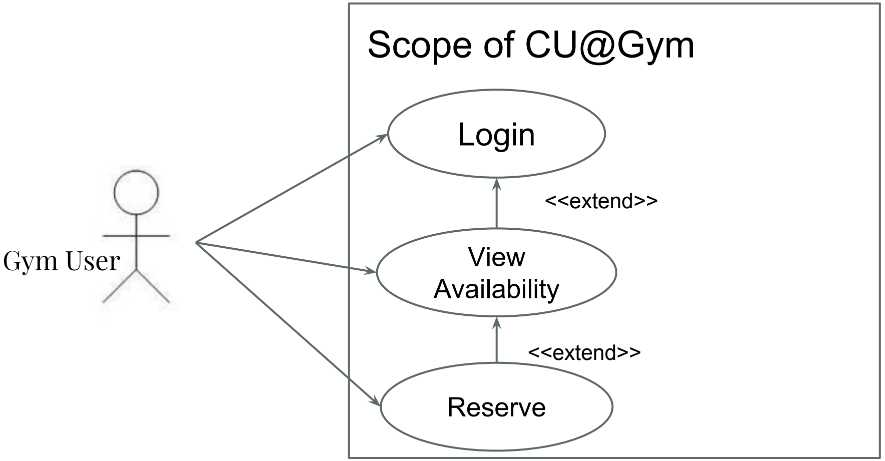


Reminder: We do not need to consider requirements and actors that are out of scope for the sprint.
We will come back and evolve the code in the second sprint to accommodate new requirements

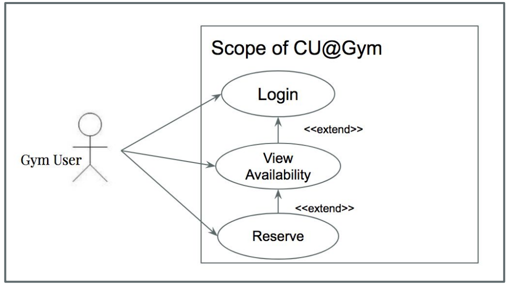
Use-Cases



(use-cases)



Use-Cases

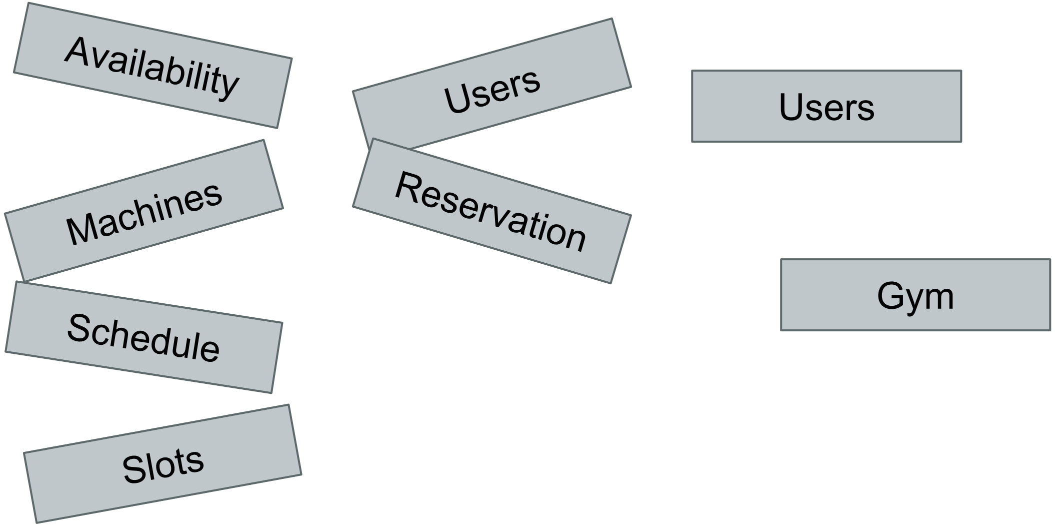


Title	View Availability
Actor	Gym User
Precondition	User exists and logged in
Basic Flow	1. Requests to view schedule 2. Provided a list of dates 3. Requests a specific date 4. Receives a set of machines organized by machine type with their availability shown in 30 minute slots
Alternate	

Title	Reserve
Actor	Gym User
Precondition	View Availability
Basic Flow	1. Select a machine and slot to reserve 2. Make reservation request 3. See successful reservation request
Alternate	3b. If machine is reserved while the user is browsing they will receive a failed reservation request

Reminder: It may feel dysfunctional we do not yet support cancels in this sprint. Yes. Not all first sprints for non-trivial products result in a fully working product. We may not be able to ship this version but we will get feedback and deal with a 'bite size chunk' of engineering complexity

Entity



Availability

Users

Users

Machines

Reservation

Gym

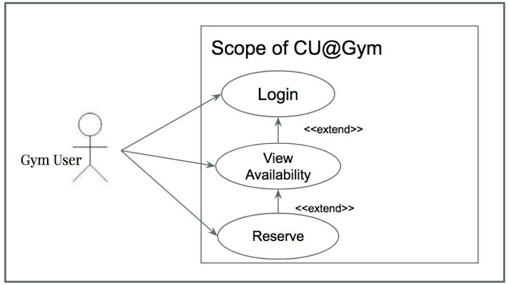
Schedule

Slots

I will move through this given I think the class had a solid grasp:

1. look for nouns from the use-cases and any other requirements materials
2. Resolve ambiguities in language and ask “what do we mean by that” - (availability, reservation, etc)

Use-Cases (Iterative)



Scope of CU@Gym

Gym User

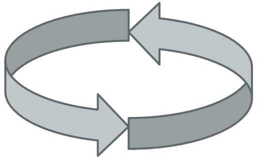
Login

<<extend>>

View Availability

<<extend>>

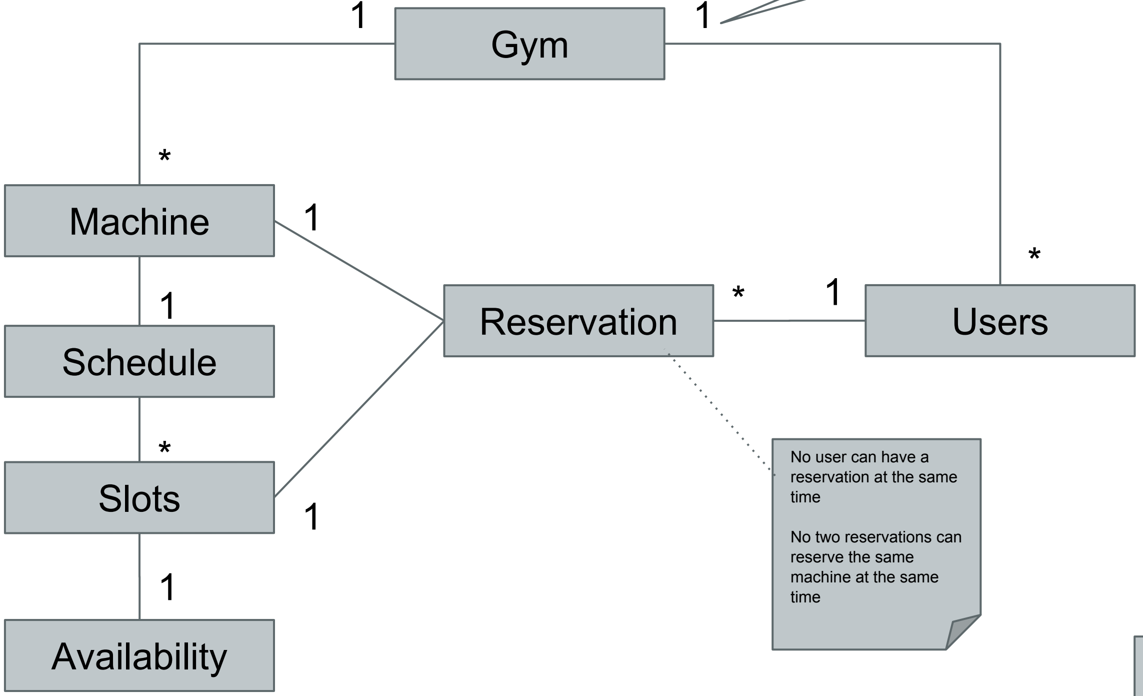
Reserve



The activity of working on the use-cases or designs often helps uncover scenarios. Therefore, even the process of use-cases and design is iterative

Title	Reserve
Actor	Gym User
Precondition	View Availability
Basic Flow	1. Select a machine and slot to reserve 2. Make reservation request 3. See successful reservation request
Alternate	3b. If machine is reserved while the user is browsing they will receive a failed reservation request 3c. If the user has reserved a machine at the same time they will receive a failed reservation request

Conceptual Class Diagram



Note: we are designing for sprint 1 scope.

No user can have a reservation at the same time

No two reservations can reserve the same machine at the same time

UML supports 'notes' where you can add additional design guidance that may not be easy to express with other notation

Post Our Lecture

After working through a few use-cases I realized we may need to refine the relationships

1. A *gym* has opening hours

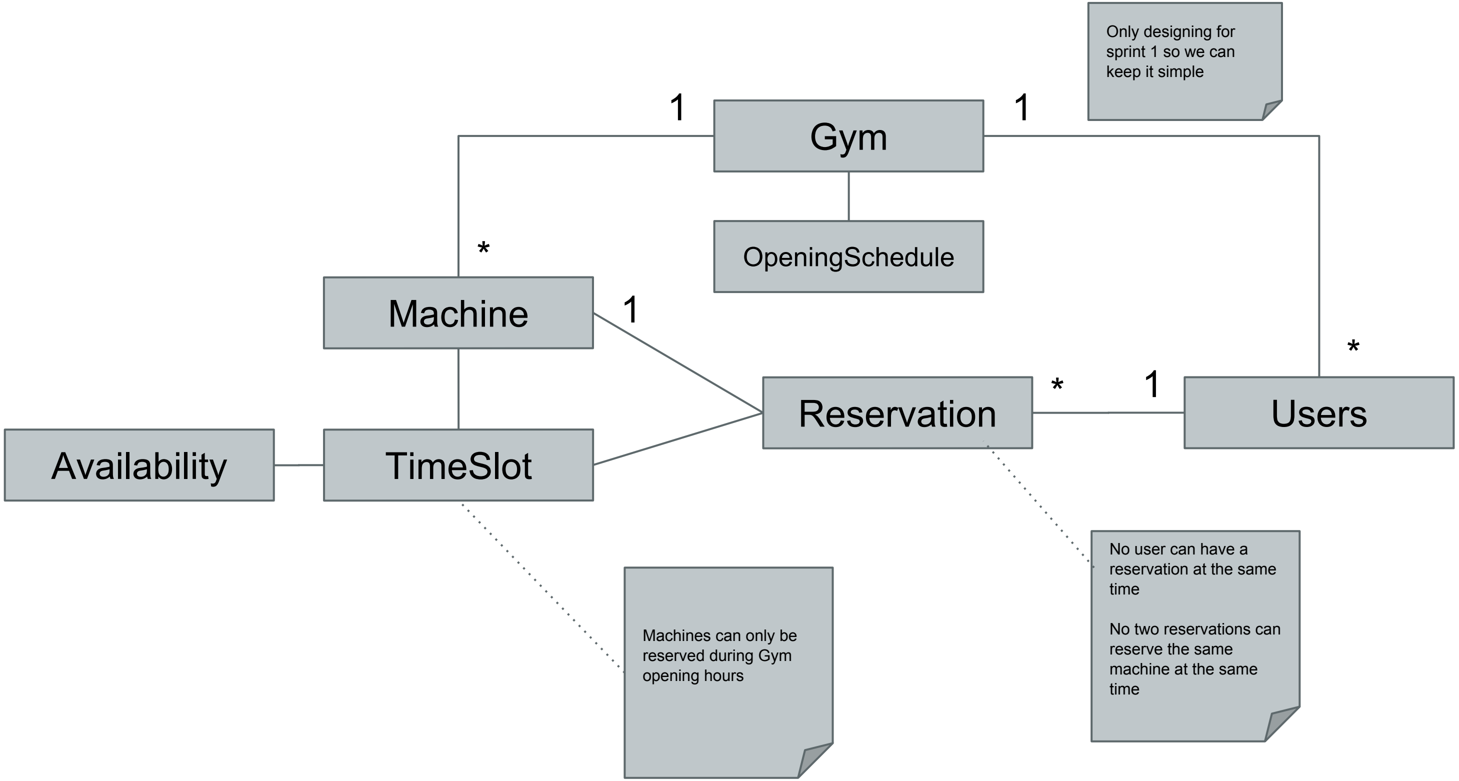
2. A *machine* can only be reserved during those opening hours

3. A cleaner / more accurate representation of the real worl is
   “Gym as having an OpeningSchedule with a machine having TimeSlots which can be reserved’

(I do also generate questions around whether we allow people to reserve whenever they want / arbitrary start time or on 15 or 30 minute increments (13:47 start time vs 13:30 or 13:45 being valid and the total duration of a reservation. The act of design generally flush out more requirements issues. My mind also asks ‘can I reserve only 60 minutes’ when gym is quiet etc. However, resist the temptation to make it more complex. Get something simple working first then we can evolve.)

Conceptual Class Diagram

UML supports 'notes' where you can add additional design guidance that may not be easy to express with other notation



Only designing for sprint 1 so we can keep it simple

Machines can only be reserved during Gym opening hours

No user can have a reservation at the same time

No two reservations can reserve the same machine at the same time

Pause

Ok - I think most people were comfortable or very comfortable to get to this stage. It may require some more practice on your own project and reading the text book for you to transition from being able to follow this to conduct independently.

We understand the functions the system needs to provide. We understand the key conceptual entities and the relationship between them.

However, this is not yet ready to code. The ***conceptual*** design (previous page) does not address:

- What ***data*** does each entity need to contain?
- What ***methods*** does each entity need to support?
- What ***data structures*** should we use?
- Are there other ***missing entities*** as we go closer to implementation?
- What are the ***data types*** of different pieces of data?
- What about other concerns: ***persistence***, etc

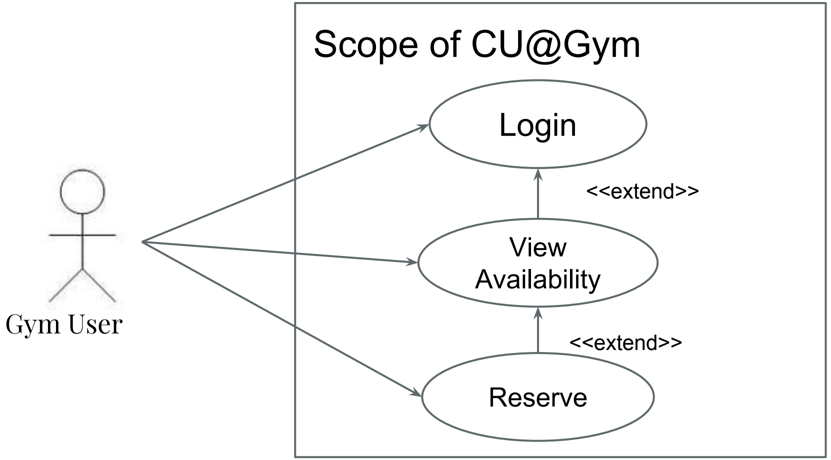
We need a ***concrete design.*** Rather than randomly guess there is a process.

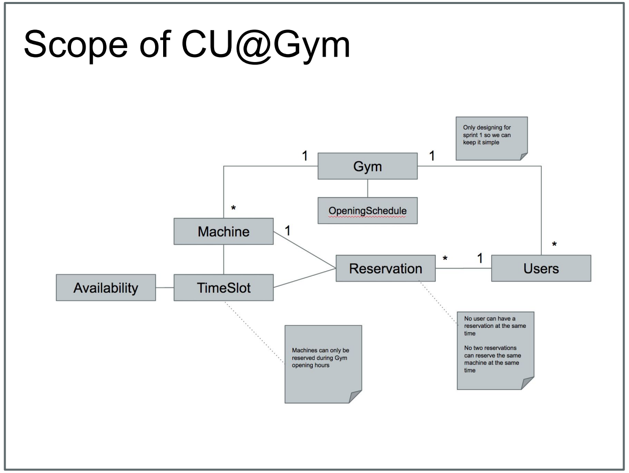
- We take ***use-cases*** and apply to the conceptual class design
- Using this process we identify data, methods and refine the design (structures, etc)
- We will end up with a more concrete class design

Refining Design

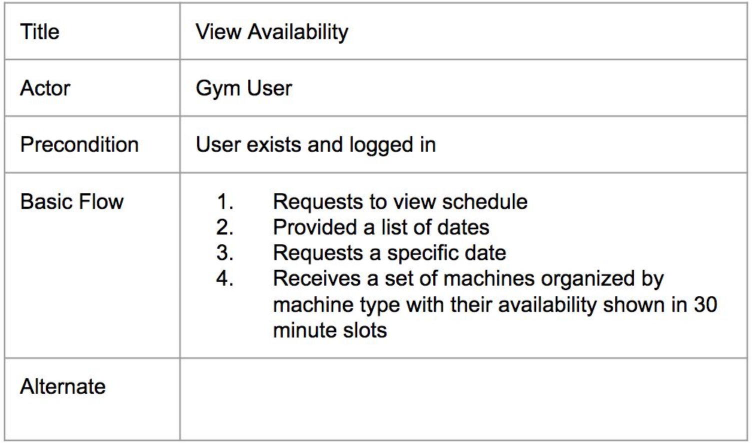
We know the use-cases the system needs to support

We know the conceptual internal structure of the system

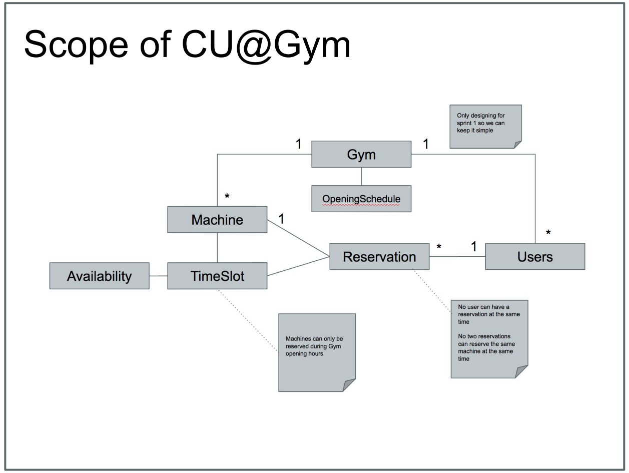



Scope of CU@Gym
Only designing for sprint 1 so we can keep it simple
1
Gym
1
OpeningSchedule
*
Machine
1
Reservation
*
1
Users
*
Availability
TimeSlot
Machines can only be reserved during Gym opening hours
No user can have a reservation at the same time
No two reservations can reserve the same machine at the same time

Let’s try to work out the missing pieces of the design by asking “How would this use-case work?”



Title	View Availability
Actor	Gym User
Precondition	User exists and logged in
Basic Flow	1. Requests to view schedule 2. Provided a list of dates 3. Requests a specific date 4. Receives a set of machines organized by machine type with their availability shown in 30 minute slots
Alternate	



Scope of CU@Gym

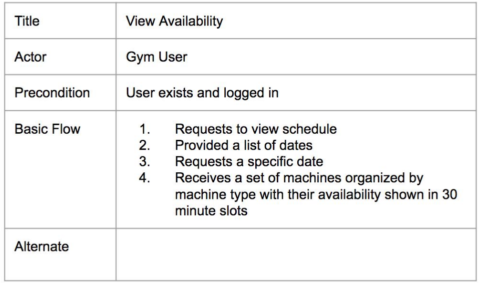
Only designing for sprint 1 so we can keep it simple

Machines can only be reserved during Gym opening hours

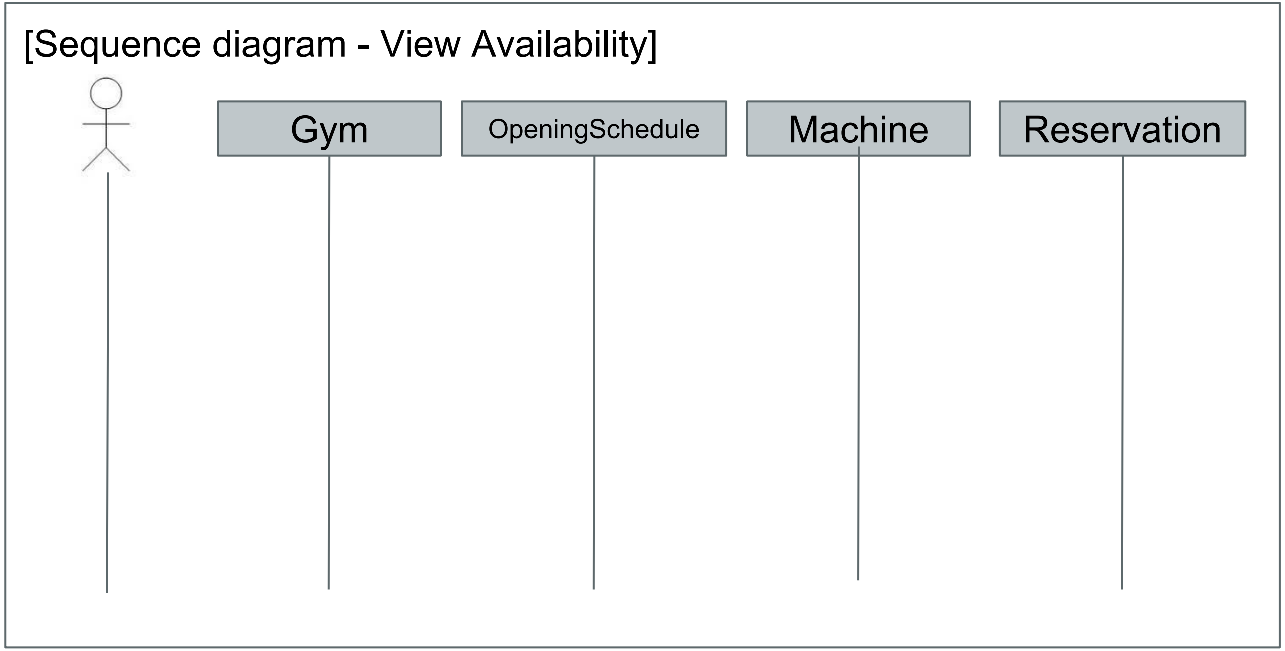
No user can have a reservation at the same time

No two reservations can reserve the same machine at the same time

To do this we need to switch to a *behavioral view* and understand how to achieve the use-case

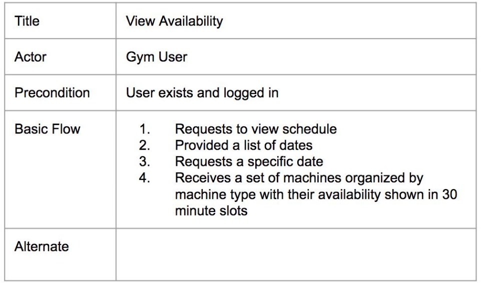


Title	View Availability
Actor	Gym User
Precondition	User exists and logged in
Basic Flow	1. Requests to view schedule 2. Provided a list of dates 3. Requests a specific date 4. Receives a set of machines organized by machine type with their availability shown in 30 minute slots
Alternate	

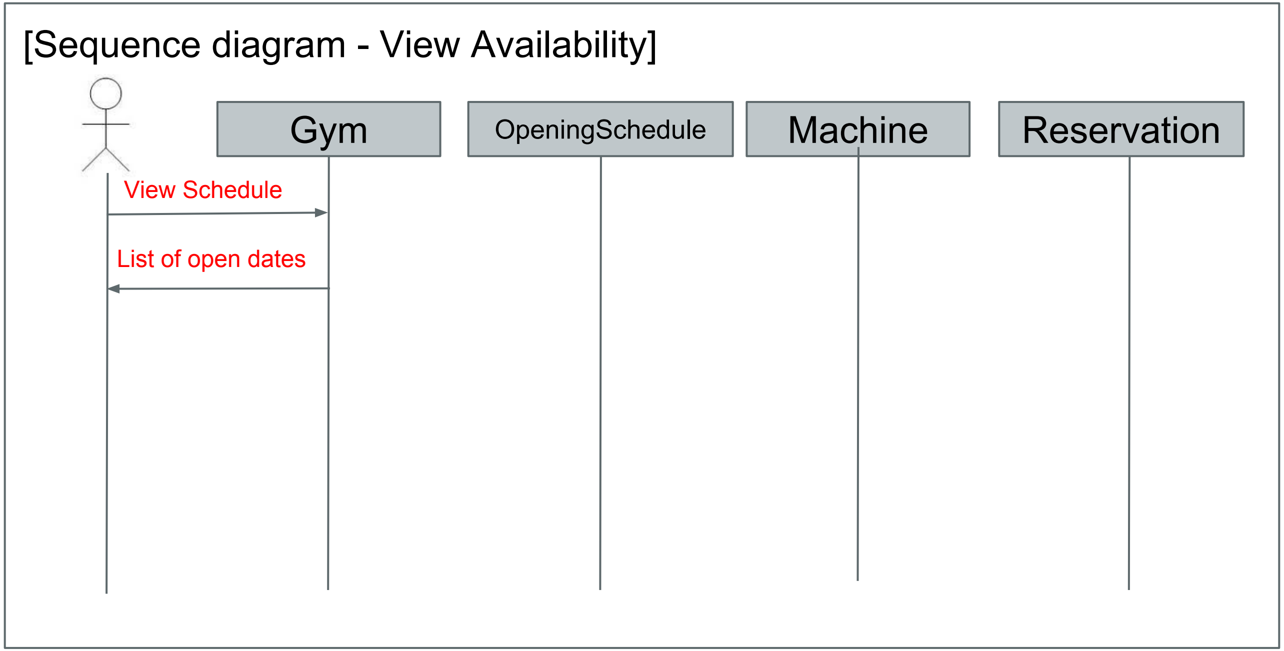


Sequence Diagram: View Open Days

View Availability



Title	View Availability
Actor	Gym User
Precondition	User exists and logged in
Basic Flow	1. Requests to view schedule 2. Provided a list of dates 3. Requests a specific date 4. Receives a set of machines organized by machine type with their availability shown in 30 minute slots
Alternate	



[Sequence diagram - View Availability]

Gym

OpeningSchedule

Machine

Reservation

View Schedule

List of open dates

Again - the act of working through the use-cases raise more questions. How many days in advance can you book? If we had done a prototype many of these issues would have been flushed out.

Sequence Diagram: View Open

1. We have identified that the Gym entity has a member which is the openingSchedule (we already know this from conceptual model)

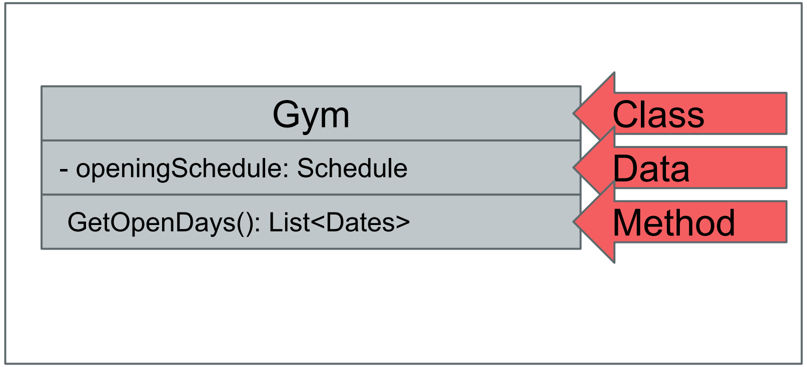
2. The relationship is 1:1 so it is a member variable

3. We have a couple of options.

A) Gym could provide ‘getter’ access to OpeningSchedule
gym.getOpeningSchedule().GetOpenDays()

OR

B) *Gym* could provide the method GetOpenDays() which inside the implementation will access the openingSchedule member variable



Gym

- openingSchedule: Schedule

GetOpenDays(): List<Dates>

Class

Data

Method

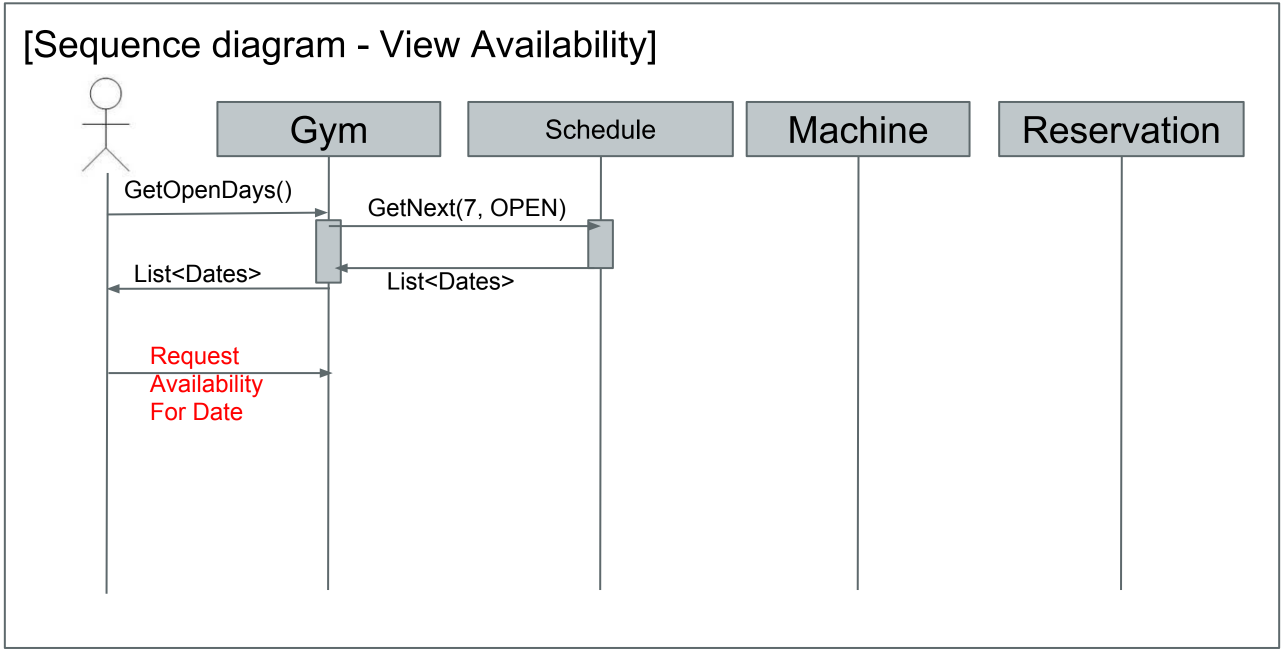
I have decided to go with the latter where Gym will be a facade and provide a set of convenience methods.

Sequence Diagram: View Open Days

- This analysis method is simply a repeated application of this first step.
- “Walk through a use-case and use the needs of the use-case to infer methods, data and expose incomplete design”
- We will continue to apply
  - the rest of the use-case
  - other use-cases
  - until design is sufficiently complete that we can code the implementation

Sequence Diagram: View Availability

Title	View Availability
Actor	Gym User
Precondition	User exists and logged in
Basic Flow	1. Requests to view schedule 2. Provided a list of dates 3. Requests a specific date 4. Receives a set of machines organized by machine type with their availability shown in 30 minute slots
Alternate	



[Sequence diagram - View Availability]

Gym

Schedule

Machine

Reservation

GetOpenDays()

GetNext(7, OPEN)

List<Dates>

List<Dates>

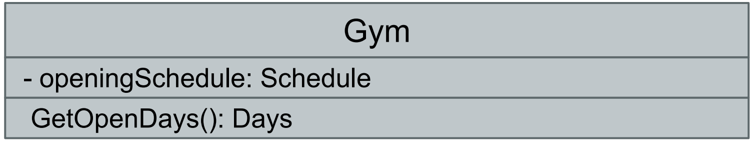
Request Availability For Date

Sequence Diagram: View Availability

1. I did think about modelling request availability as a command object (a specific class called ‘RequestAvailability’ with a single member date). However, it seems slight overkill having a class with a single member variable so Gym will support GetAvailability(Date)

2. I have all sorts of design options on how to implement Schedule (and I could make it very generic). However, at this stage let’s keep it simple.

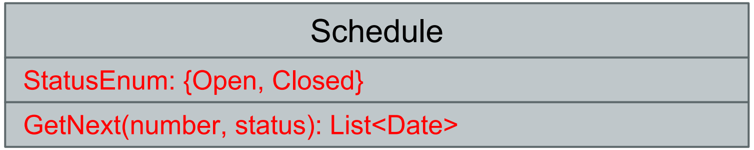
3. It is also worth noting I have a lot of design freedom as to how we implement the ***internal algorithms and data structures*** for classes. A schedule cloud be an array of dates or a sparse set of Dates and Availability etc. One of the advantages of OO is we abstract and encapsulate this complexity. For the moment I defer designing the internal data structures of Schedule



Gym

- openingSchedule: Schedule

GetOpenDays(): Days



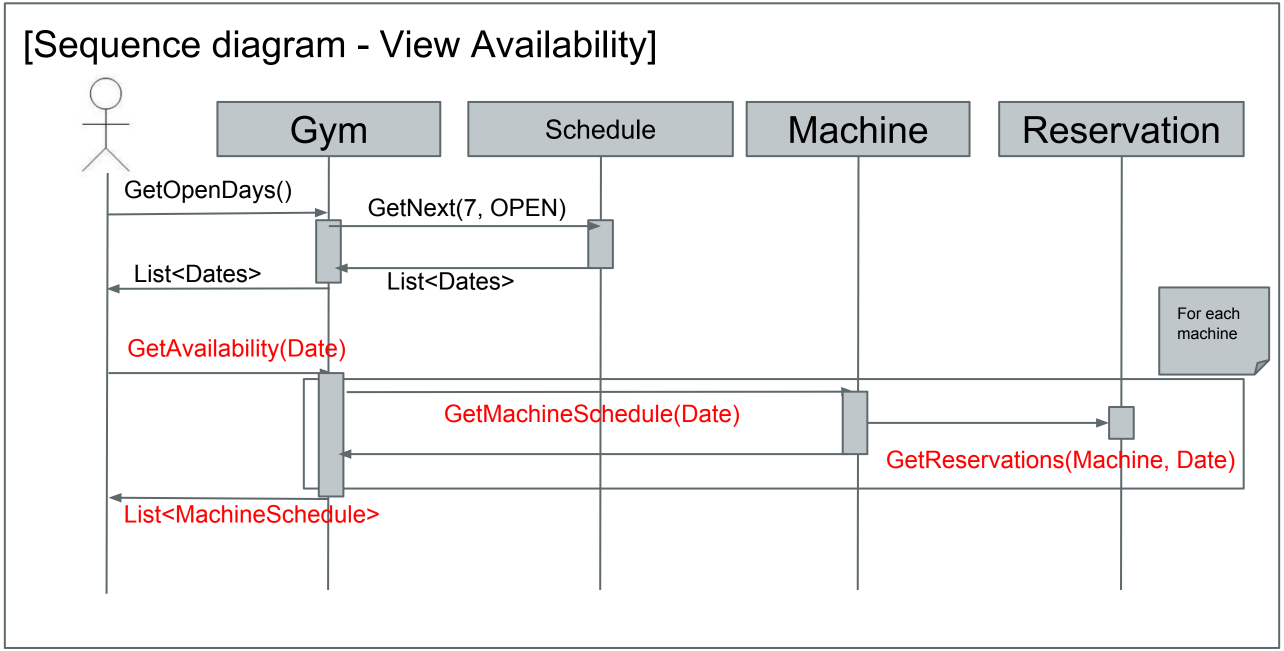
Schedule

StatusEnum: {Open, Closed}

GetNext(number, status): List<Date>

Sequence Diagram: View Availability

Title	View Availability
Actor	Gym User
Precondition	User exists and logged in
Basic Flow	1. Requests to view schedule 2. Provided a list of dates 3. Requests a specific date 4. Receives a set of machines organized by machine type with their availability shown in 30 minute slots
Alternate	



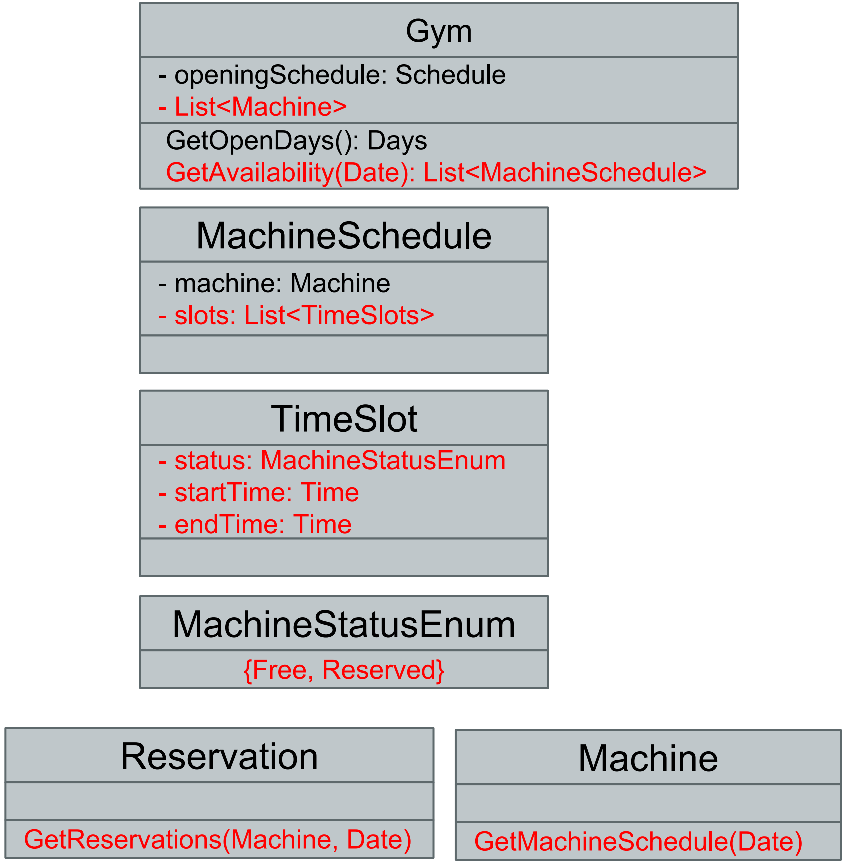
Sequence Diagram: View Availability

1. I add that Gym has a List of Machines. The conceptual class diagram already tells me there is a 1:m relationship. There are many data structures for 1:m. Let's make it a simple List for the moment

2. The return type for GetAvailability is interesting. I could have provided the full set of machines but the caller does not want to implement and work this out itself. Therefore, I choose to introduce a MachineSchedule class

From Gym I could return a list of machines and reservations. Why create a specific return type of MachineSchedule?

- I want the return type to be simple (what the consumer actually wants is the filtered set of machines with schedule for each). They don't want to have to iterate through reservations etc
- I also want to provide an *immutable* view. In this scenario the caller asked for availability. If I provided them access to the domain objects they could (depending on language and package visibility rules) make changes themselves
- I use an enum for the status of a machine



Gym

- openingSchedule: Schedule
- List<Machine>

GetOpenDays(): Days
GetAvailability(Date): List<MachineSchedule>

MachineSchedule

- machine: Machine
- slots: List<TimeSlots>

TimeSlot

- status: MachineStatusEnum
- startTime: Time
- endTime: Time

MachineStatusEnum

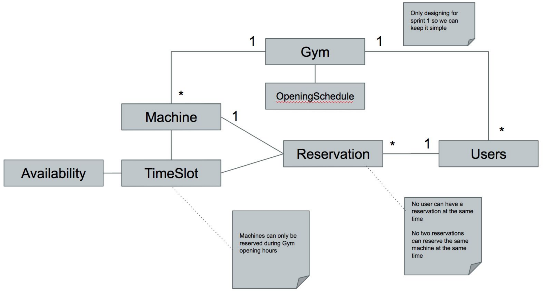
{Free, Reserved}

Reservation

GetReservations(Machine, Date)

Machine

GetMachineSchedule(Date)



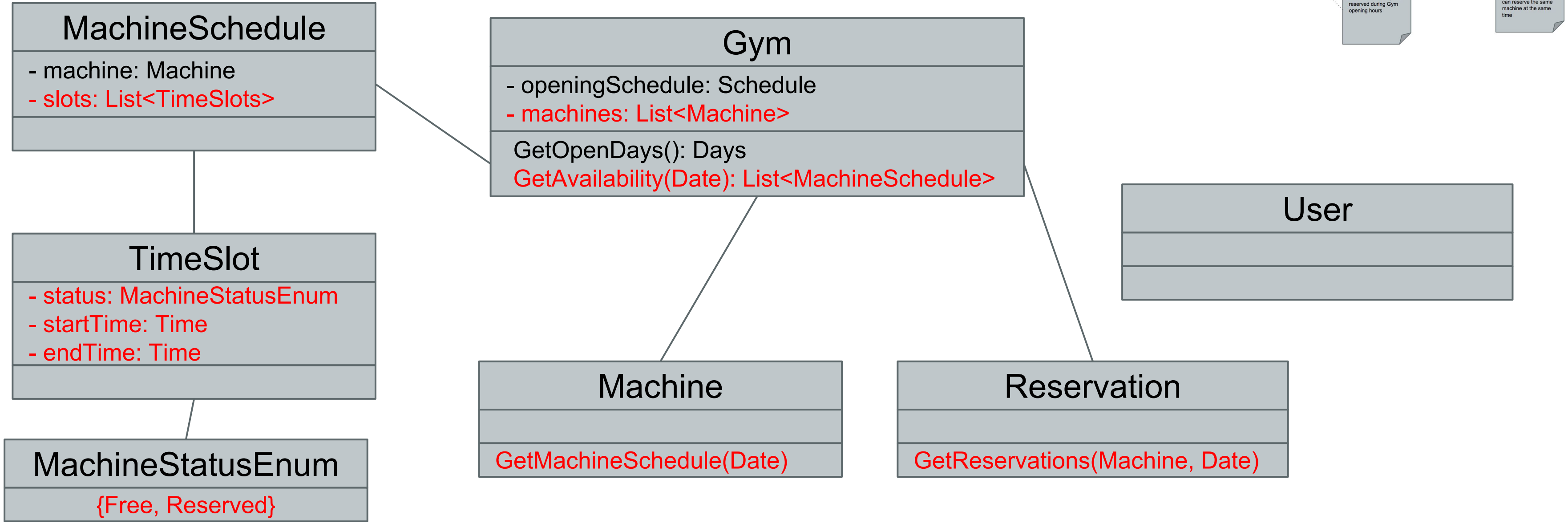
Only designing for sprint 1 so we can keep it simple

Machines can only be reserved during Gym opening hours

No user can have a reservation at the same time

No two reservations can reserve the same machine at the same time

Interim Class Diagram



MachineSchedule

- machine: Machine
- slots: List<TimeSlots>

Gym

- openingSchedule: Schedule
- machines: List<Machine>

GetOpenDays(): Days

GetAvailability(Date): List<MachineSchedule>

User

TimeSlot

- status: MachineStatusEnum
- startTime: Time
- endTime: Time

Machine

GetMachineSchedule(Date)

Reservation

GetReservations(Machine, Date)

MachineStatusEnum

{Free, Reserved}

Repeat

Recap. We have

1. Produced use-cases
2. Identified nouns and relationships to produce a conceptual class diagram

<At this stage we have a conceptual class diagram>
<We want to flesh it out by working out the methods and data each object needs to support>

3. We take our use-cases and ‘play’ the use-case through the class diagram to understand how to make the objects interact and to ‘flush out’ the methods and data each objects need to hold

4. It is an iterative process and we often uncover new findings about the rules of the problem domain. This may cause us to temporarily go back and redraft the conceptual class structure

5. However as we get it more right we find the design ‘solidifies’

6. So far this is for a ***single*** use-case. We would complete for other use-cases which would result in an implementable class diagram

7. **Exercise to the reader:** Apply the other use-cases to the class diagram to yield an implementable design.