WEEK 10 CATCHUP DAY

Anything you want to discuss about design patterns?

Writing interactors / anything you want to talk about pertaining to the project?

Tracing the Lab 5 code in IntelliJ to better understand CA

Interviewing (mock technical interview questions)

PLAN FOR THE REST OF THE TERM

- Thursday:
  - Class is cancelled for ARIA; see the Week 10 ARIA Scavenger Hunt (the Week 10 Review Quiz)
  - Watch the Interviewing lecture and the posted interviews.
- Week 11:
  - Regular expressions
  - GenAI and prompt engineering
  - Asynchronous software documentation module (including, how to write a README for your project)
- Week 12:
  - Review lectures
- “Week 13”
  - Project presentations on Monday Dec 2 and Tuesday Dec 3

DESIGN PATTERNS

For each design pattern, which SOLID principle(s) does it most clearly demonstrate? Justify your answer.

	S	O	L	I	D
Dependency Injection					
Factory					
Builder					
Strategy					
Observer					
Adapter					
Façade					

PROJECT TIMELINE (ON QUERCUS)

Week of Nov 11–15

- ❑ **All View and View Model code written except for actionPerformed methods.**
  - Your team should have a clear vision of what the interface for your program will be, as well as the "inputs" and "outputs" for each use case which you will be implementing.

- ❑ In-memory data access objects created although perhaps not yet implemented to facilitate testing
  - **You should have your data access interfaces defined.**
  - **Your team should also be continuing to make sure that you can make any necessary API calls, so that you can plug in the real API calls.**

- ❑ Builder written that creates the views — **run the program and make sure each view looks okay**
  - For now, you might just make separate "Main" classes for each view, depending on your application. You can connect everything later.
  - If you aren't using the starter code, equivalent code for creating all objects for the CA engine should be present and at least creating the views.

- ❑ **Code for individual use cases begun with at least one unit test per Interactor**
  - See previous examples of how to test interactors which demonstrate how to mock the presenter.

CREATING AN INTERACTOR

LEARNING OUTCOMES

- Understand how you might design and develop the code for a user story interaction

USER STORY —> CODE

Prerequisite: a fully-planned user story

Steps:

1. Figure out the data involved in the user story: create Entities
2. Plan a sequence of actions to accomplish their user story
   - Make your stories as small and specific as you can!
3. For each user interaction
   a) Create the Input Data, Output Data, Data Access Interface, Input Boundary interface, Interactor class, and Output Boundary interface
   b) Write a *very very basic* in-memory DAO that doesn't read and write files at all (or use a real API)
   c) Write a unit test for the Input Boundary
   d) Write code in the Interactor to pass the test
   e) Hook up the UI to the Interactor
   f) Write a real DAO

FIGURE OUT ENTITIES

- Let's say that your user story is to transfer money from one bank account to another.
- The user needs to choose two bank account numbers and the amount to transfer. (We'll worry about the UI in a bit, but maybe we'll use drop-down menus to choose the accounts and a "dollars" text field and a "cents" text field. Or maybe the user has to type the bank account numbers.)
- The Entities involved might be BankAccount and Money objects. Write those.

PLAN A SEQUENCE OF ACTIONS FOR TRANSFERRING

- The first action is when the user opens the app. If the app displays any persistent information, this must trigger a use case interaction. We will deal with that as a separate user story and assume it as a prerequisite for this “transfer” user story.
- The relevant actions are:
  - Choose the first bank account
  - Choose the second bank account
  - Enter the amount of money
  - Click a button (or type Return/Enter?)

FOR EACH USER ACTION: INPUT AND OUTPUT DATA

- In our “transfer” user story, there is only one user action that involves persistent data.
- The Input Data is the two bank account numbers and the amount of money to transfer; create that class.
- For the Output Data, we’ll include the new balances of the bank accounts; we might use a Map where the keys are bank account numbers (as Strings) and the values are the balances. We also need to report on the amount of money transferred. For a failing case, the Output Data will need a message describing the problem. Let’s use “Insufficient funds”. Create the Output Data class.

FOR EACH USER ACTION: INPUT AND OUTPUT BOUNDARY

- The boundaries are usually quite simple (and usually look the same)
  - `public void execute(InputData)`

  - `public void prepareSuccessView(OutputData)`
  - `public void prepareFailView(OutputData)`

FOR EACH USER ACTION: DATA ACCESS

- The “transfer” interactor will need BankAccount and Money entities.
- The amount of money is in the Input Data, so the Interactor might use a MoneyFactory.
- All the bank account information is in the persistence layer, so we might make a getBankAccount(accountNumber) method that goes into the DAI.
- Write a very, very basic DAO that implements the DAI. A simple implementation might just use a Map of bank account number to BankAccount object.
  - You can even have the basic DAO create a few BankAccount objects in the constructor so you can start testing your “transfer” interaction.
  - We’ll write a real DAO later.

FOR EACH USER ACTION: START THE INTERACTOR

- Create an Interactor that implements the Input Boundary
- Leave it empty for now, other than “return null” in method `execute`
- We’ll fill this in soon, but first a test!

FOR EACH USER ACTION: WRITE A TEST

- Now we have all the Interactor parts
- Write a JUnit test (like the one in the lab5 code). These all follow a pattern:
  - Create the Input Data object
  - Create a Presenter that implements the Output Boundary
    - Its job is to validate the Output Data
  - Create the Interactor, injecting the Presenter and any necessary DAOs
  - Invoke the Interactor, passing in the Input Data
- Run the test — it will fail, of course

FOR EACH USER ACTION: FINISH THE INTERACTOR

- Complete method `execute`
- For our example, this isn't a lot of code:
  - Get BankAccount objects from the DAO
  - Create a Money object
  - Subtract the money from one account and add it to the other
  - Create an OutputData object
  - Call the Presenter through the OutputBoundary
- If you get this right, your test will pass
- Now write any failing test cases, like when there isn't enough money to transfer
  - This may require updating the Interactor code

HOOK THE INTERACTOR UP TO THE UI

- Now create the View, Controller, View Model, and Presenter

THE VIEW

- You may need to create text fields or menus or whatever to get the bank account numbers and amount to transfer
- And a button to click, maybe called “Transfer”
  - Write an actionPerformed method that gathers the bank account numbers and amount to transfer and invokes the Controller
  - The information may be in the View Model or you might fetch it directly from the UI fields

THE CONTROLLER

- Write the Controller.
- The execute method has parameters for the bank account numbers and a Money object.
- Instantiate an Input Data object and invoke the Interactor.

THE PRESENTER

- Write the Presenter.
- This will be called by the Interactor.
- It updates the View Model and then instructs it to notify the View.

THE VIEW (AGAIN)

- The View needs to add itself as an observer to the View Model, if it hasn't already.
- The View needs to update based on the new View Model values.
- Write any necessary methods to do this.

CREATE THE WHOLE THING IN A FACTORY OR BUILDER

1. The View Model and DAOs can be instantiated first, because they depends on nothing
2. The Presenter only depends on the View Model, so create it next and inject the View Model
3. The Interactor needs the DAOs and the Presenter, so create it next and inject them
4. The Controller needs the Interactor, so create it next and inject them
5. The View needs the View Model and Controller, so create it next and inject them

Now it should run! Cross your fingers and try it. ☺

PROJECT TIMELINE (ON QUERCUS)

Week of Nov 11–15

- ❑ **All View and View Model code written except for actionPerformed methods.**
  - Your team should have a clear vision of what the interface for your program will be, as well as the "inputs" and "outputs" for each use case which you will be implementing.

- ❑ In-memory data access objects created although perhaps not yet implemented to facilitate testing
  - **You should have your data access interfaces defined.**
  - **Your team should also be continuing to make sure that you can make any necessary API calls, so that you can plug in the real API calls.**

- ❑ Builder written that creates the views — **run the program and make sure each view looks okay**
  - For now, you might just make separate "Main" classes for each view, depending on your application. You can connect everything later.
  - If you aren't using the starter code, equivalent code for creating all objects for the CA engine should be present and at least creating the views.

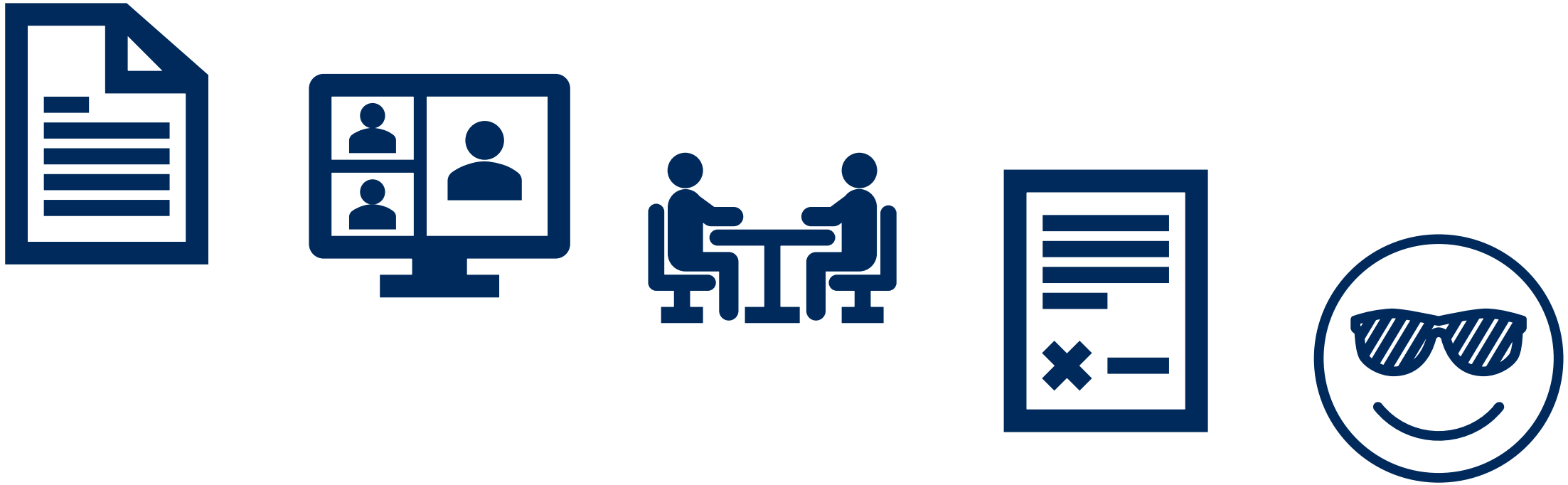
- ❑ **Code for individual use cases begun with at least one unit test per Interactor**
  - See previous examples of how to test interactors which demonstrate how to mock the presenter.

TRACING THE LAB 5 CA CODE

- Time permitting, we'll open IntelliJ and trace the program execution of Main and some of the interactors.
  - Understand how the CA engine gets built
  - Understand how a Use Case gets executed
  - Understand how the View gets updated

TECHNICAL INTERVIEW QUESTIONS

CSC 207 SOFTWARE DESIGN



RESOURCES

- Technical interview questions
  - ChatGPT 4 about CSC207: https://chat.openai.com/share/c53a1999-2475-4646-bd8e-3f3eee17a95a
    - Note: we didn't do Iterator this year, so you can ignore that (don't forget, though, someone could ask you about it in a real interview!)
  - Top 18 Clean Architecture Interview Questions: https://www.fullstack.cafe/interview-questions/clean-architecture
    - Check out the other topics!

SOME SAMPLE QUESTIONS

- The following slides contain some sample interview-style questions.
- We encourage you to do some mock interviews with your peers using these questions.
- This can also be great practice for the final exam!
- The question with the blue background in each category is the recommended question if you choose to try just one of them.

5 CATEGORIES OF QUESTIONS

- SOLID
- Clean Architecture
- Design Patterns
- Git and GitHub
- Teamwork

TEAMWORK: COMMUNICATION AND PROBLEM-SOLVING

- Effective communication is crucial in software development teams. Provide an example of a challenging situation you faced in a team, such as a disagreement over design decisions or dealing with a tight deadline?
- How did you communicate and collaborate with your team to resolve this issue?

GIT: BRANCHING AND MERGING

- Explain the process and best practices for branching and merging in Git.
- How would you handle a situation where you need to merge a feature branch into the main branch, but you encounter merge conflicts?
- Describe a strategy for managing branches in a collaborative project on GitHub.

DESIGN PATTERNS: OBSERVER

- Explain the Observer Pattern and its use cases in software design.
- How does it facilitate communication between objects?
- Please describe an example where you have implemented the Observer Pattern in a project or how you would use it to solve a specific problem, such as creating a notification system in an application.

CA: ADAPTING EXTERNAL FRAMEWORKS TO CLEAN ARCHITECTURE

- Clean Architecture emphasizes that external libraries and frameworks should not dictate the system's architecture. Describe a situation where you had to integrate an external library or framework into a system designed with Clean Architecture principles.
- How did you ensure that the integration did not violate the dependency rule and kept the business logic independent of external influences?

SOLID: DEPENDENCY INVERSION PRINCIPLE

- Describe the Dependency Inversion Principle and its role in creating maintainable and scalable software.
- Can you give an example of how Dependency Injection, a technique associated with DIP, can be used to improve a software design?
- What are the benefits and potential challenges of implementing DIP?

DESIGN PATTERNS (MOCK INTERVIEW QUESTION)

Pick a design pattern and a SOLID principle — ask the interviewee to discuss how they are related.

	S	O	L	I	D
Dependency Injection					
Factory					
Builder					
Strategy					
Observer					
Adapter					
Façade					