CLEAN ARCHITECTURE

CSC207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

Course Overview

Tools (Weeks 1-4)

- Java
- Version Control
- Software Tools

Design (Weeks 5-8)

- Clean Architecture
- SOLID
- Design Patterns

Professional Topics (Weeks 9-12)

- Ethics
- Internships
- GenAI

- Last week we introduced the SOLID principles and learned about Exceptions in Java.
- This week, we will introduce:
  - Clean Architecture,
  - Use Cases, and
  - Sequence Diagrams.

Questions to be answered this week...

- What is a use case?
- What is Clean Architecture (CA)?
- What is the CA dependency rule?
- What is a sequence diagram?

LEARNING OUTCOMES

- Become more familiar with terminology related to architecture and design
- Understand clean architecture and its dependency rule
- Understand how sequence diagrams can help us convey flow of execution in our program.

ARCHITECTURE

(Brief Summary of Chapter 15 from Clean Architecture textbook)

Design of the system

- Dividing it into logical pieces and specifying how those pieces communicate with each other.
- Input and output between layers.

"Goal is to facilitate the **development**, **deployment**, **operation**, and **maintenance** of the software system"

  - *"The strategy behind this facilitation is to leave as many options open as possible, for as long as possible."*

Good architecture strives to maximize programmer productivity!

POLICY AND LEVEL

Brief Summary of Chapter 19 from Clean Architecture textbook

- “A computer program is a detailed description of the **policy** by which inputs are transformed into outputs.”
- Software design seeks to separate policies and group them as appropriate. (Ideally form a directed acyclic dependency graph between components)
- A policy has an associated **level**. (e.g. “high level policy”)

POLICY AND LEVEL

- Level: “the distance from the inputs and outputs”
    - higher level policies are farther from the inputs and outputs.
    - lowest level are those managing inputs and outputs.

USE CASES FOR A PROGRAM

Imagine you were asked to write a program that allows users to

- **register a new user account** (with a username and password)
- **log in to a user account**
- **log out of a user account**
- They're planning on having a few different kinds of accounts, but we'll start with just one for now.

Bold words are *use cases*: what will the user want to do?

1. What data needs to be represented?
2. What data structure might you use while the program is running?
3. What should happen if the user quits and restarts the program?

USE CASE: USER REGISTERS NEW ACCOUNT

USE CASE: USER REGISTERS NEW ACCOUNT

- The user chooses a username
- The user chooses a password and enters it twice (to help them remember)
- If the username already exists, the system alerts the user
- If the two passwords don't match, the system alerts the user
- If the username doesn’t exist in the system and the passwords match, then the system creates the user but does not log them in

USE CASE: USER LOGS IN

USE CASE: USER LOGS IN

- The user enters a username and password
- If the username exists in the system and the passwords match, then the system shows that the user is logged in
- If there is no such username, the system alerts the user
- If the password doesn't match the one in the system, the system alerts the user

USE CASE: USER LOGS OUT

- The system logs the user out and informs the user

USE CASE: WHEN LOGGED OUT, CHOOSE USE CASE

- The user chooses between the user *registers a new account* use case and the *user logs in* use case

BURNING QUESTIONS

- What is the user interface?
    - A webpage?
    - A Java application on your computer?
    - A Python command-line program?
    - A mobile app?
- How to do data persistence?
    - A text file? json?
    - A database?
    - Google Drive/OneDrive/etc.?

- How can you design your program so that it's easy to move to a new UI?
- How can you design your program so that it's easy to save data to a different kind of storage?
- How can you design your program so that **as much code as possible** stays the same when you do these things?

DESIGN CONUNDRUMS

- How can we design the use cases so that they **do not directly depend** on the UI and persistence choices?
  - Then we can test all the use cases thoroughly without dealing with input/output!
- What are the use case APIs?
  - What is the interface to each use case?
  - What public methods do we want to provide to call the use cases from the UI?
- What persistence methods will we need in any storage?
    - Saving
    - Finding a user by username
    - Etc.

CLEAN ARCHITECTURE

BUSINESS RULES

(Brief Summary of Chapter 20 from Clean Architecture textbook)

- **Entity**: “An object within our computer system that embodies a small set of Critical Business Rules operating on Critical Business Data.”
- **Use Case:** “A description of the way that an automated system is used. It specifies the input to be provided by the user, the output to be returned to the user, and the processing steps involved in producing that output. A use case describes *application-specific* business rules as opposed to the Critical Business Rules within the Entities.”

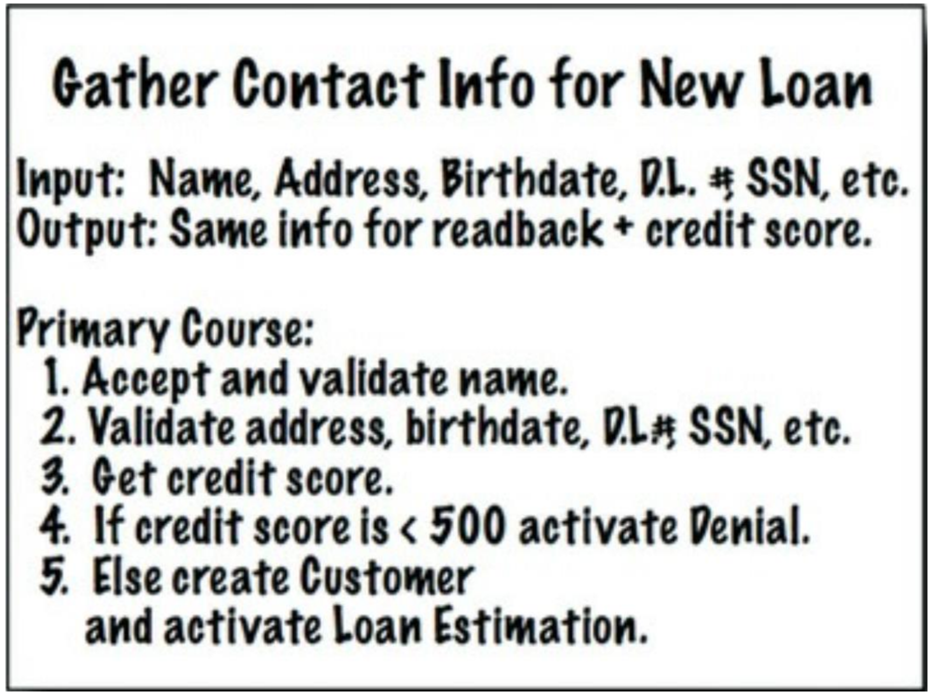
ENTITIES

- Objects that represent critical business data (variables) and critical business rules (methods).
- In Clean Architecture, **entities** are the highest-level policies (core of the program).
- Some examples
  - a Loan in a bank
  - a Player in a game
  - a set of high scores
  - an item in an inventory system
  - a part in an assembly line

USE CASE

- A description of the way that an automated system affects Entities
- Use cases manipulate Entities
- Specifies the input to be provided by the user, the output to be returned to the user, and the processing steps involved in producing that output.

Use cases know nothing about the details of the user interface or the data storage mechanism — just that they exist!

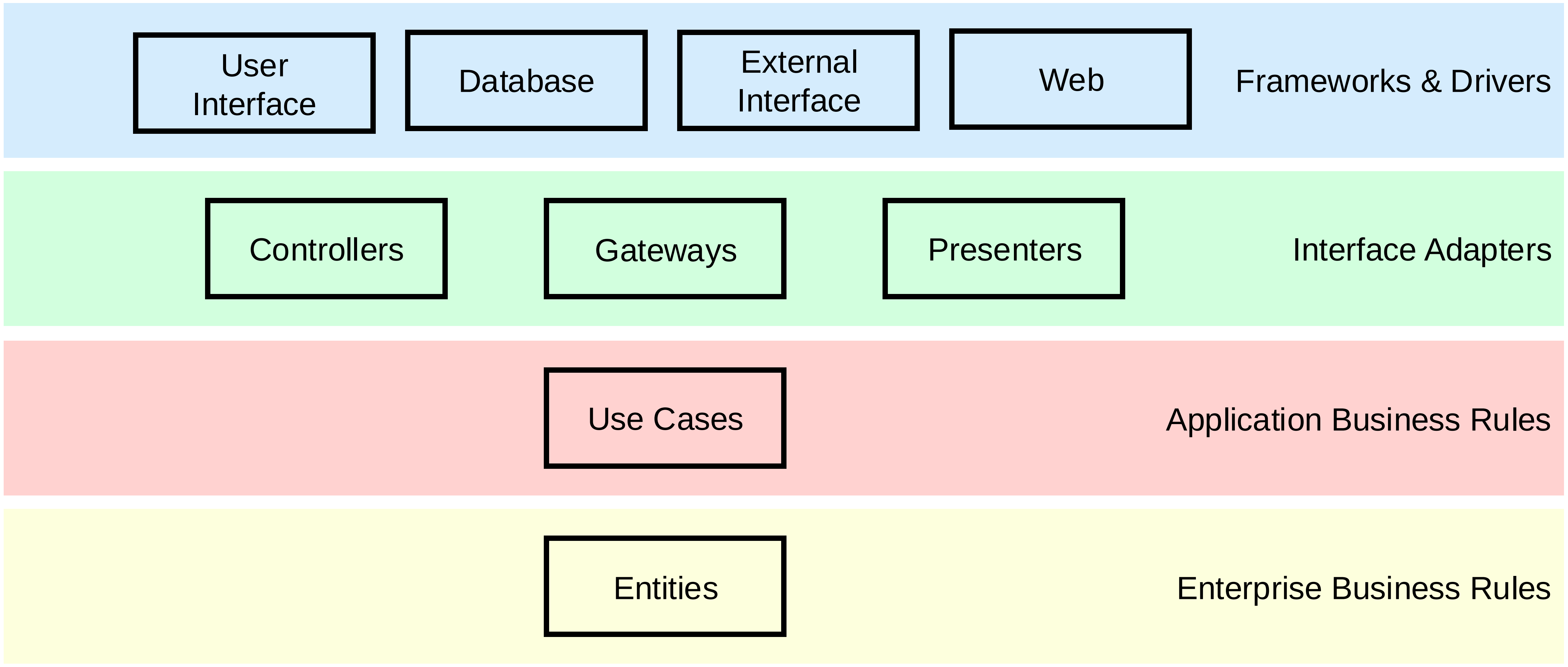


Gather Contact Info for New Loan

Input: Name, Address, Birthdate, D.L. # SSN, etc.
Output: Same info for readback + credit score.

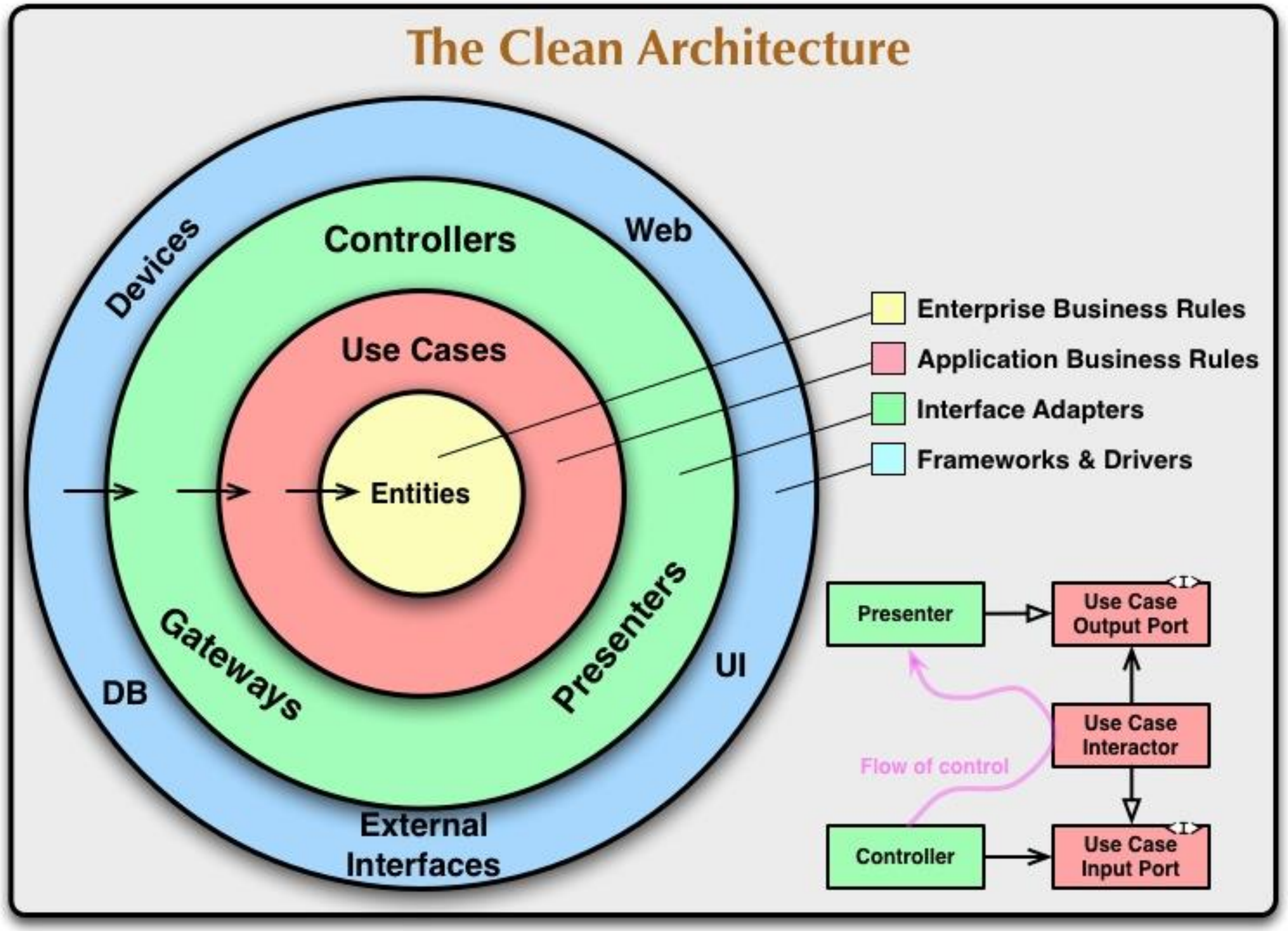
Primary Course:
1. Accept and validate name.
2. Validate address, birthdate, D.L# SSN, etc.
3. Get credit score.
4. If credit score is < 500 activate Denial.
5. Else create Customer and activate Loan Estimation.

CLEAN ARCHITECTURE



CLEAN ARCHITECTURE

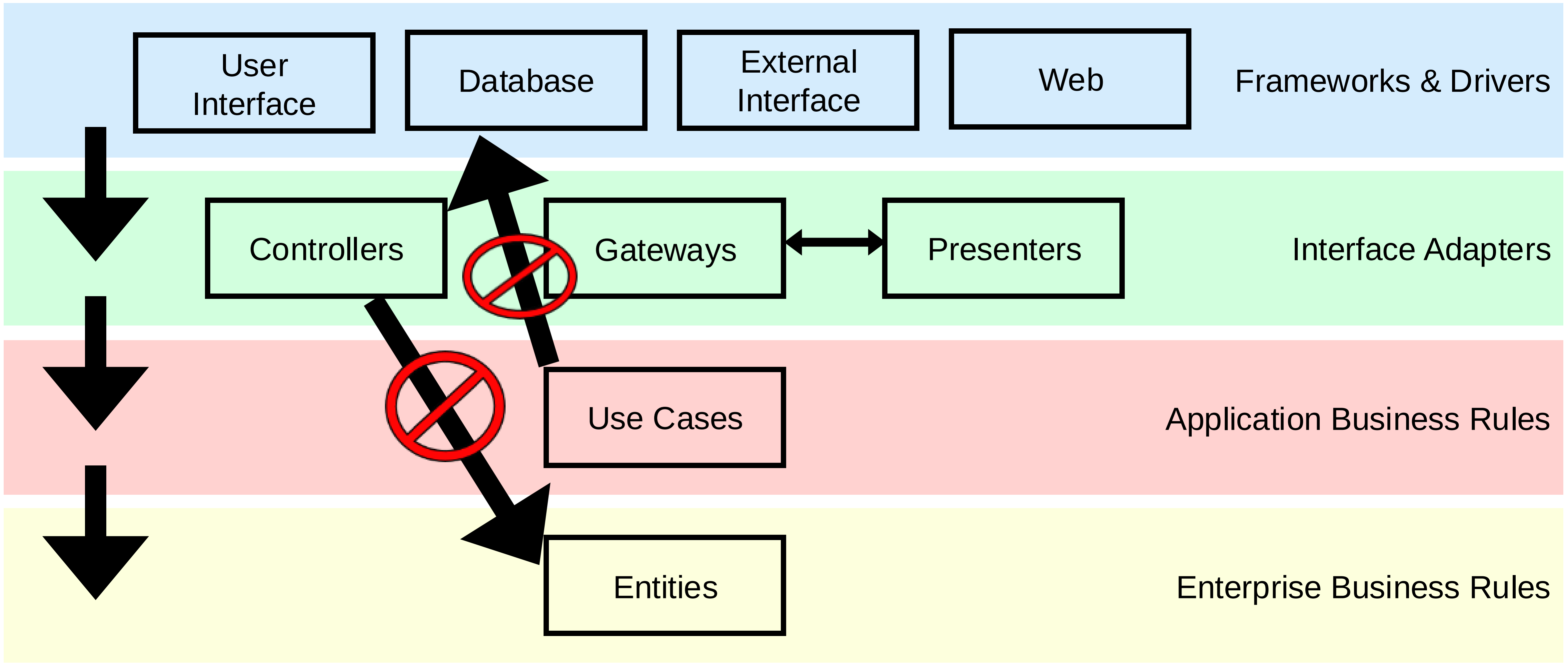
- Often visualize the layers as concentric circles.
- Reminds us that Entities are at the core
- Input and output are both in outer layers



CLEAN ARCHITECTURE – DEPENDENCY RULE

- All dependencies must point inward.
- Dependence within the same layer is allowed (but try to minimize coupling)
- On occasion you might find it unnecessary to explicitly have all four layers, but you'll almost certainly want at least three layers and sometimes even more than four.
- **The name of something (functions, classes, variables) declared in an outer layer must not be mentioned by the code in an inner layer.**
- How do we go from the inside to the outside then?
  - Dependency Inversion!
  - An inner layer can depend on an interface, which the outer layer implements.

CLEAN ARCHITECTURE – DEPENDENCY RULE



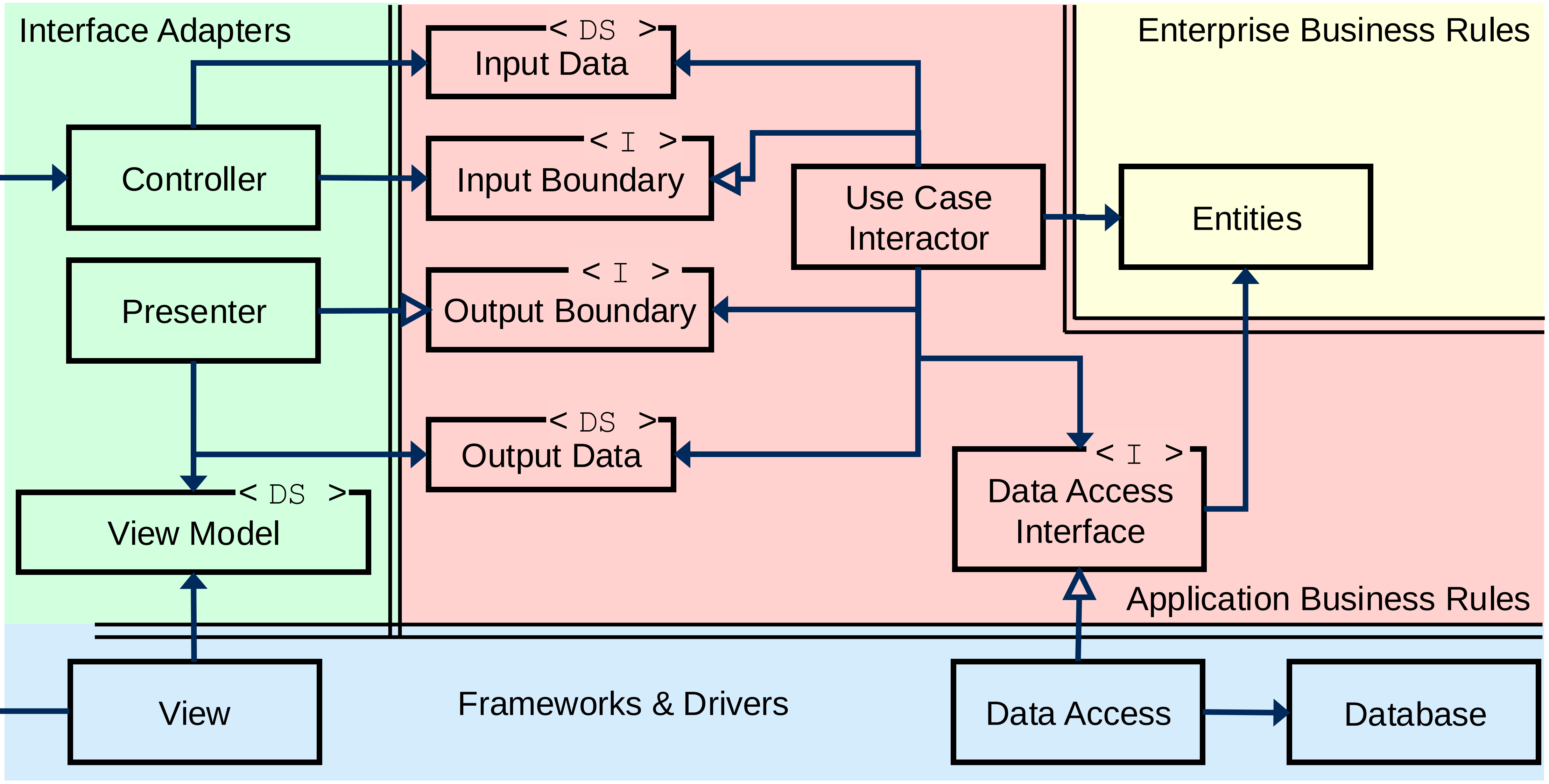
IDENTIFYING VIOLATIONS OF CLEAN ARCHITECTURE

- Look at the imports at the top of each source file in your project.
- For example, if you see that you are importing a Controller class from inside an Entity class, that is a violation of clean architecture! Likewise, if you see a Controller referencing an Entity that is also likely a violation.
- You can also achieve this visually by having IntelliJ generate a dependency graph for your project.

BENEFITS OF CLEAN ARCHITECTURE

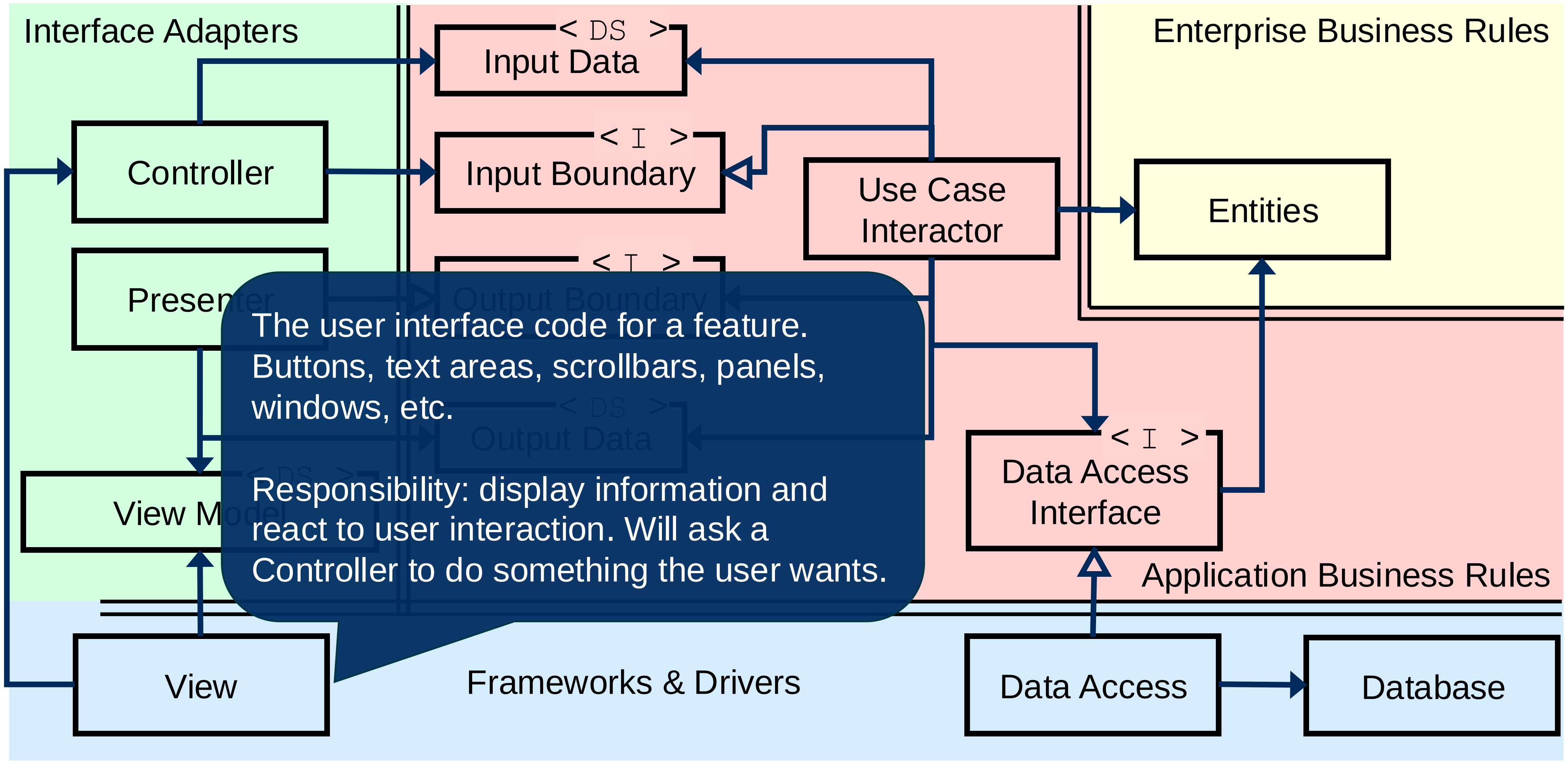
- All the “details” (frameworks, UI, Database, etc.) live in the outermost layer.
- The business rules can be easily tested without worrying about those details in the outer layers!
- Any changes in the outer layers don’t affect the business rules!

EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Computer Science
UNIVERSITY OF TORONTO

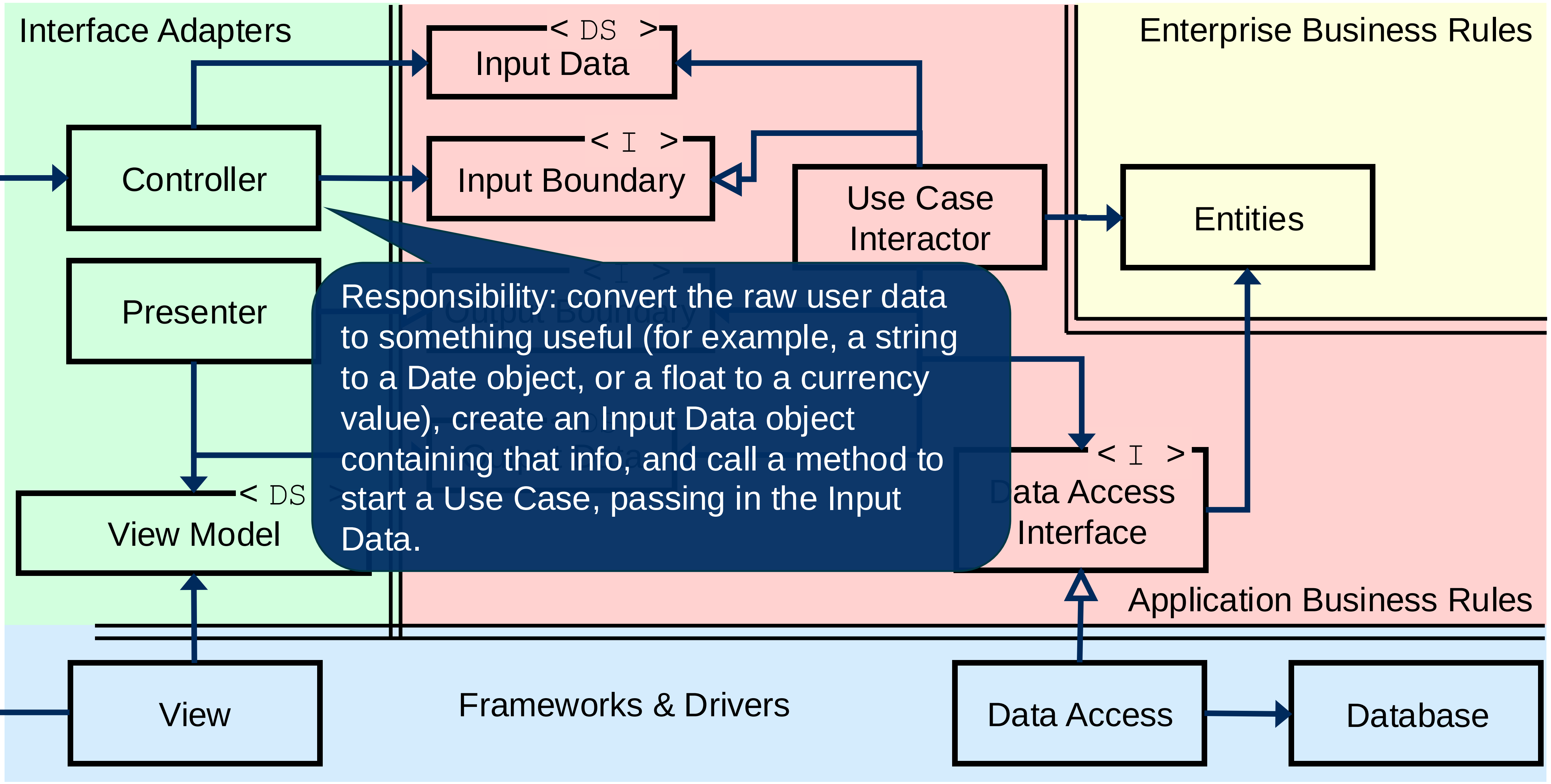
HOW THE ENGINE WORKS: THE VIEW



The user interface code for a feature. Buttons, text areas, scrollbars, panels, windows, etc.

Responsibility: display information and react to user interaction. Will ask a Controller to do something the user wants.

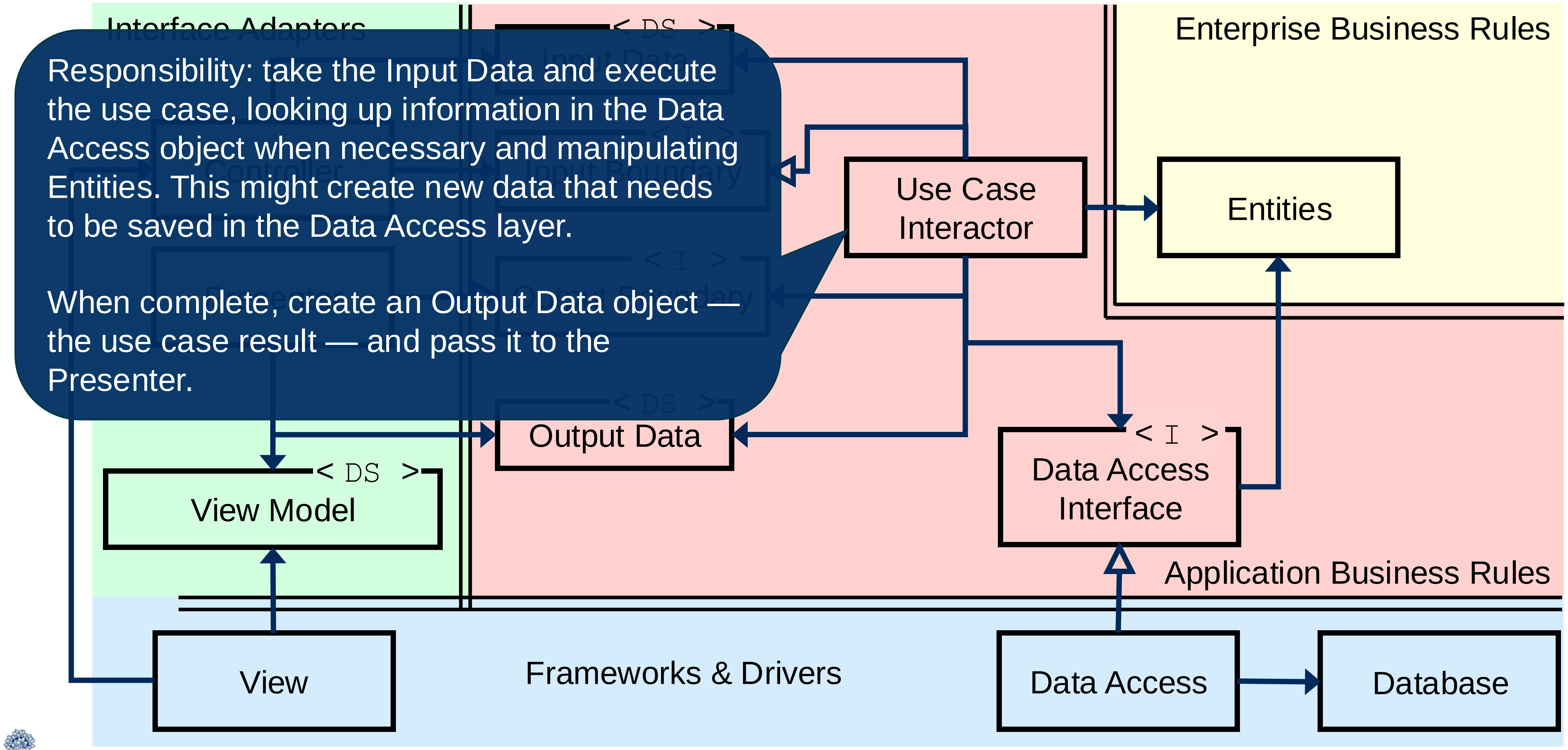
EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Responsibility: convert the raw user data to something useful (for example, a string to a Date object, or a float to a currency value), create an Input Data object containing that info, and call a method to start a Use Case, passing in the Input Data.

Computer Science
UNIVERSITY OF TORONTO

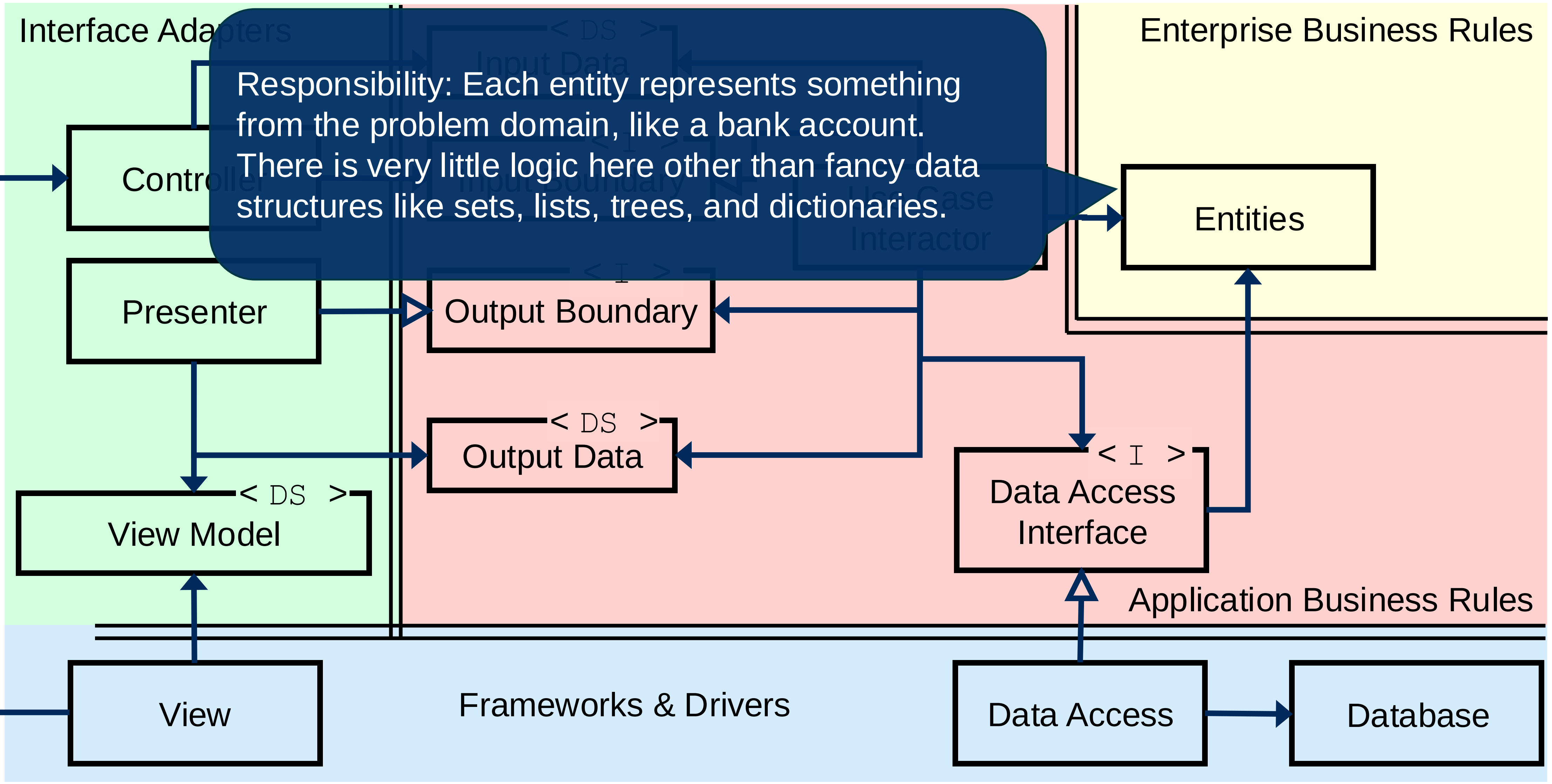
EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Responsibility: take the Input Data and execute the use case, looking up information in the Data Access object when necessary and manipulating Entities. This might create new data that needs to be saved in the Data Access layer.

When complete, create an Output Data object — the use case result — and pass it to the Presenter.

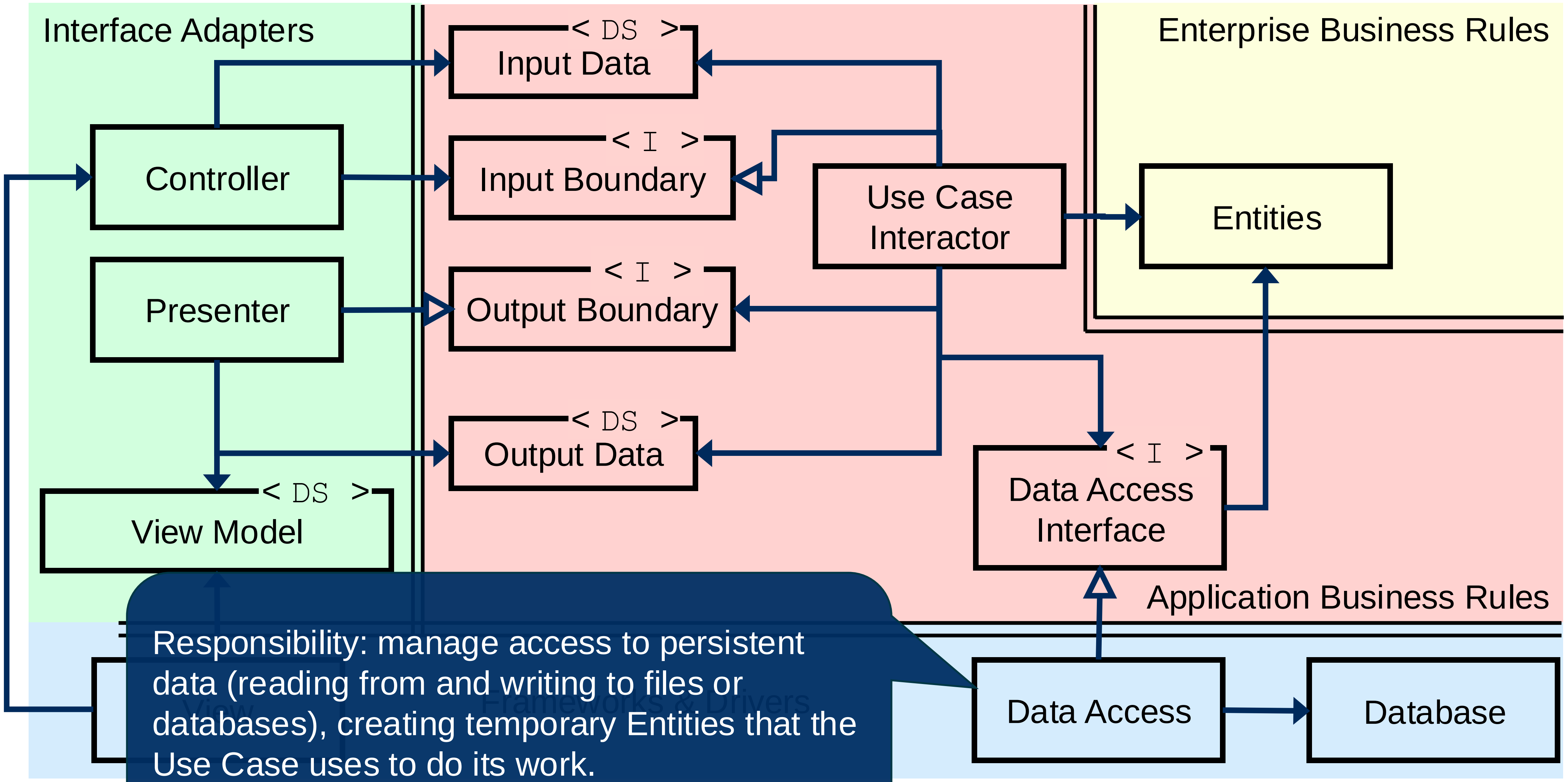
EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Responsibility: Each entity represents something from the problem domain, like a bank account. There is very little logic here other than fancy data structures like sets, lists, trees, and dictionaries.

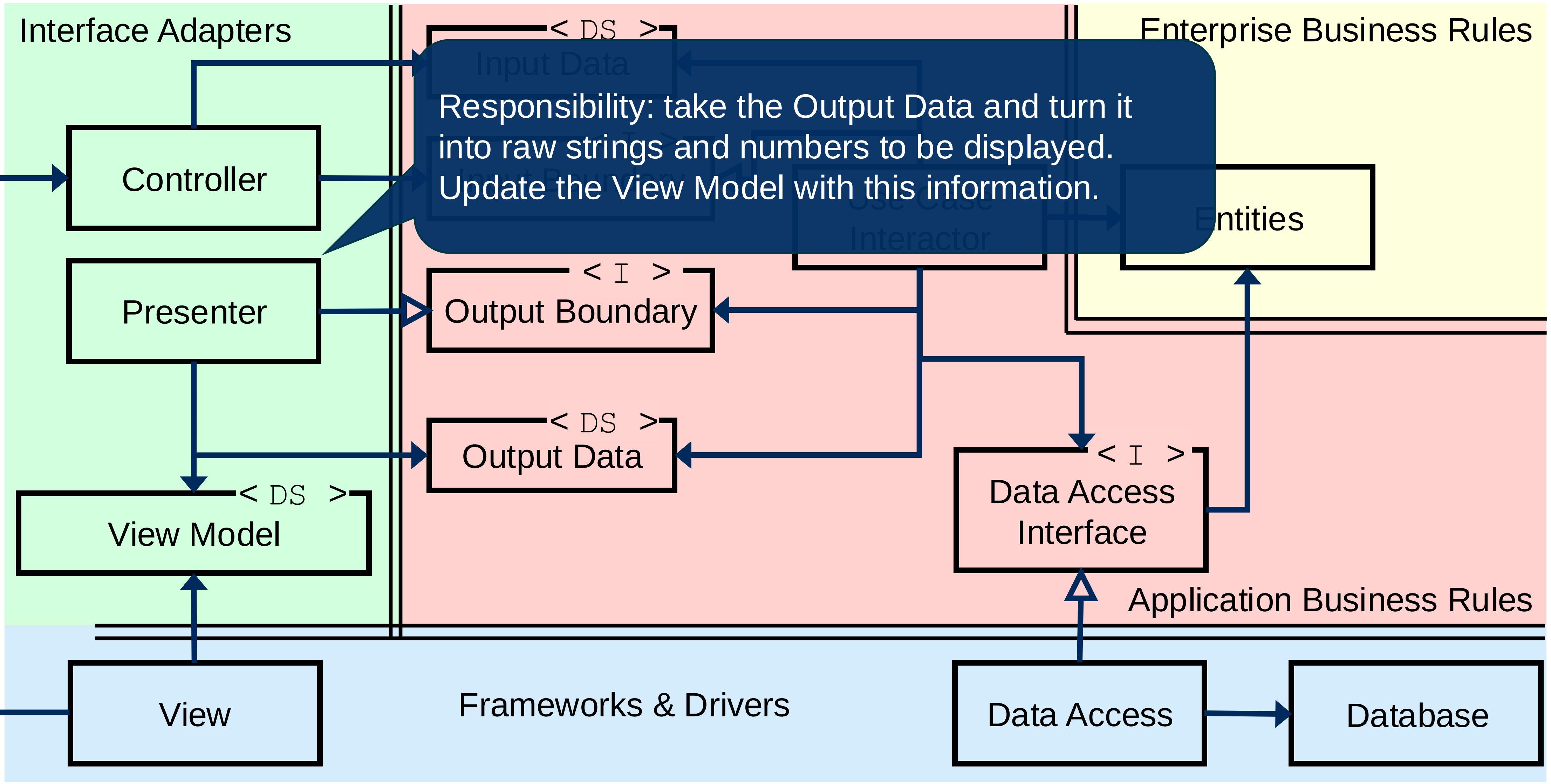
Computer Science
UNIVERSITY OF TORONTO

EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Responsibility: manage access to persistent data (reading from and writing to files or databases), creating temporary Entities that the Use Case uses to do its work.

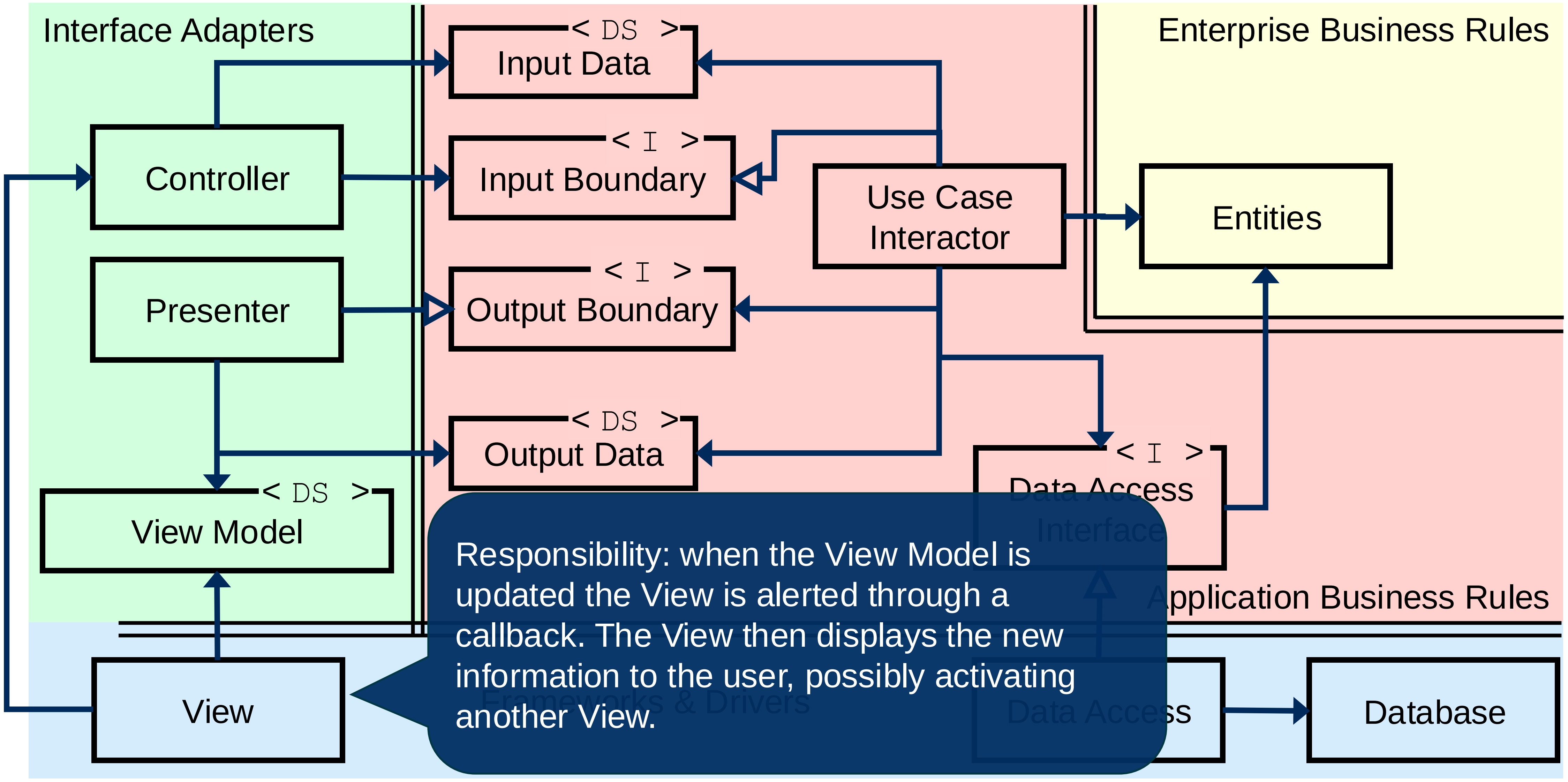
EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Responsibility: take the Output Data and turn it into raw strings and numbers to be displayed. Update the View Model with this information.

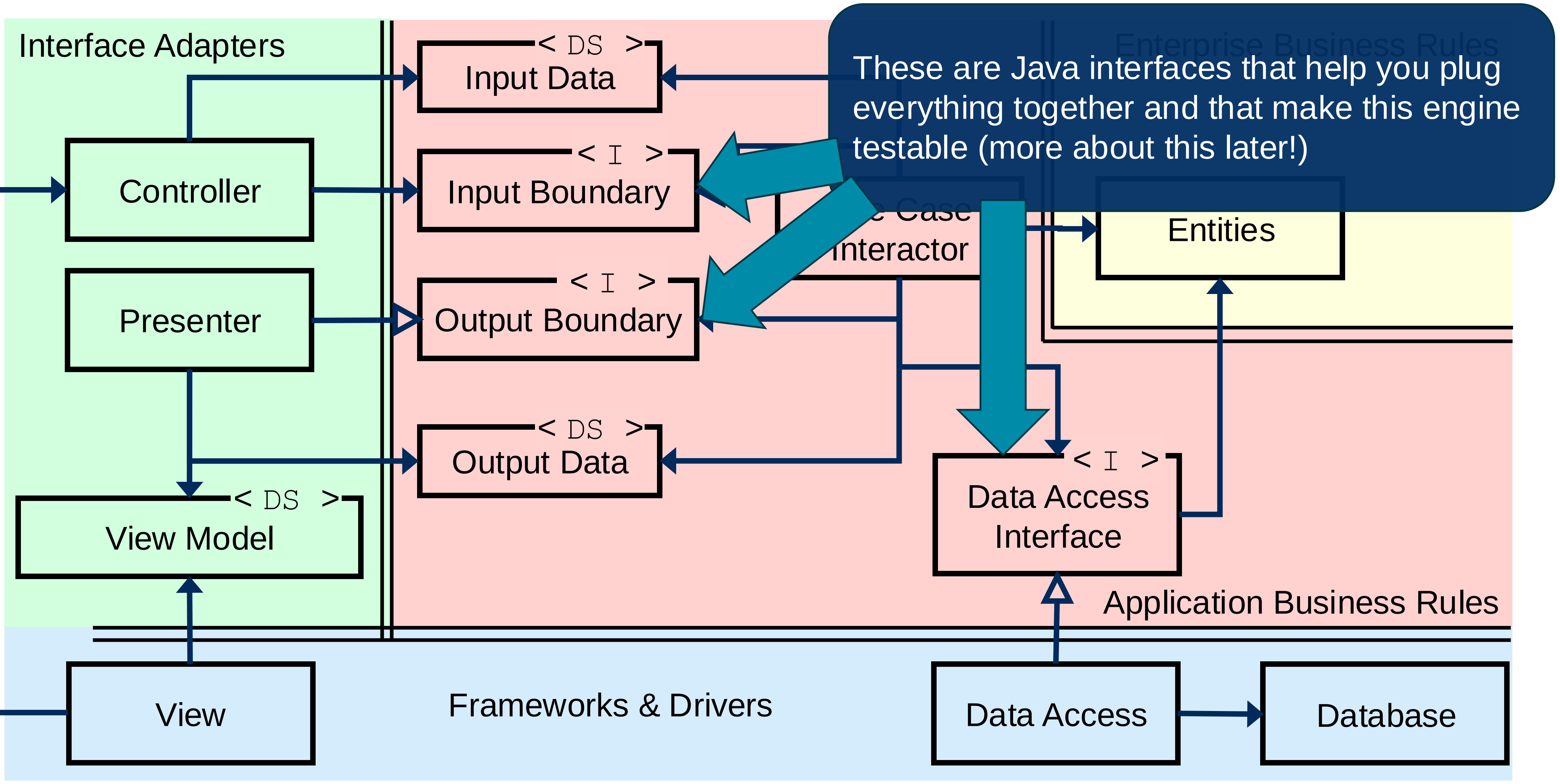
Computer Science
UNIVERSITY OF TORONTO

EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Responsibility: when the View Model is updated the View is alerted through a callback. The View then displays the new information to the user, possibly activating another View.

EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



These are Java interfaces that help you plug everything together and that make this engine testable (more about this later!)

TERMINOLOGY (OH DEAR ME!)

- **Entity**: a basic bit of data that we're storing in our program (like a user with a username and password). Often called a "model" of the real world.
- **Factory**: an object that knows how to instantiate a class or a collection of interrelated classes.
- **Use case**: something a user wants to do with the program.
- **Interactor**: the object that responds to a user interaction, usually part of a use case (implements the input boundary)
- **Input boundary**: the public interface for calling the use case
- **Output boundary**: the public interface that the Interactor will call when the use case is complete
- **Data Access Object (DAO)**: involves persistence (a file or database). Often called a Gateway or a Repository. Reads data and creates Entities
- **Controller**: the object that the UI asks to run a use case
- **Presenter**: the object that tells the UI what to do when a use case finishes
- **Model**: a model of a concept from the problem domain. A collection of data representing a concept from the problem domain. Entities are often called models.

DOES CLEAN ARCHITECTURE FOLLOW SOLID?

- Take a few minutes to think about each principle, then check the following slides for some thoughts. Continue the conversation on piazza!

CLEAN ARCHITECTURE FOLLOWS SOLID

- **Single Responsibility Principle** – Each role divides up the responsibilities for each class.
  - Only gateway classes are responsible to other hardware or software that is outside of your program.
  - Only Presenters are responsible to whomever decides which information is displayed to the user
  - Each entity is responsible for storing information about one building block of the program (example: user class in a program that allows users to login and save their session)

- **Open/Closed Principle** – Clean Architecture separates layers of code based on how close or far they are from the details. For example: what does the user see, which hardware/operating system is running the program, where are we store persistent data, etc.
  - It is easy to reuse the backend in an app for a different system (web app, Android, etc)
  - It is easy to reuse entities across different apps that follow the same Enterprise Business Rules.
  - It is possible to add new Use Cases without changing much of the orginial code.

CLEAN ARCHITECTURE FOLLOWS SOLID

- **Liskov Substitution Principle** – The various interfaces introduced should help us write code consistent with this principle, but we still need to ensure our code consistently uses the interfaces and that they are designed and documented properly.
- **Interface Segregation Principle** – Each interface has a specific role and is associated with one Use Case, so no unnecessary methods are implemented.
- **Dependency Inversion Principle** – In Clean Architecture, this is closely related to the Dependency Rule. The principle is applied to remove dependencies on the low-level details of the program!

USER LOGIN EXAMPLE

- Recall the specification for a user login system we talked about earlier.
- We had started to develop a design for it in terms of use cases.
- You can find a partial implementation at https://github.com/paulgries/UserLoginCleanArchitecture
- We'll refer to this code during lecture, but on the final coding homework you'll work with a slightly different codebase — the overall structure will look similar though.

AN EXAMPLE: REGISTER A NEW USER

As part of a program, we want to allow a new user to register an account by specifying their username and password — getting them to enter their password twice to avoid typos and help them commit it to memory.

USE CASE APIS

A class for each use case

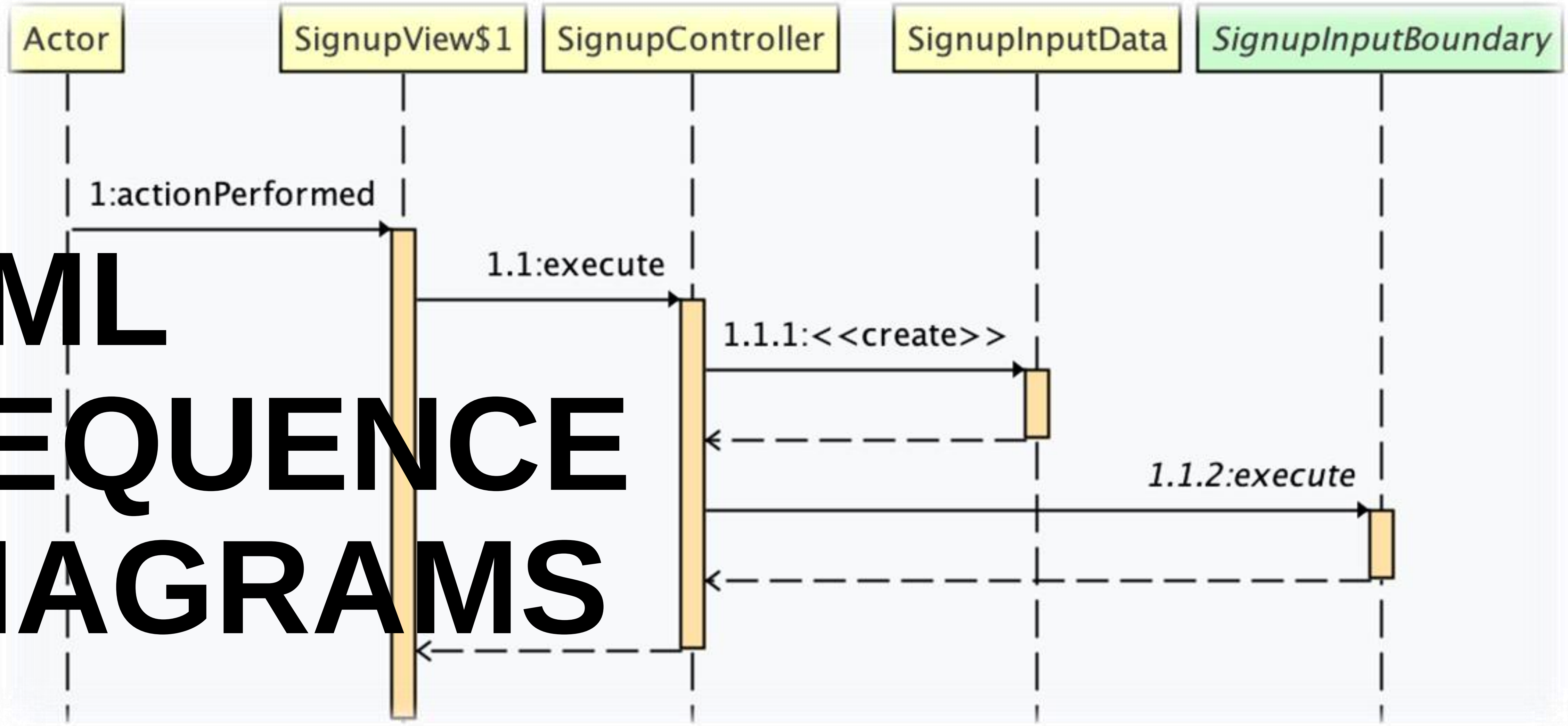
`UserRegisterInteractor`

- Public API: a method called `create` (perhaps with the username and password as parameters, or maybe they're wrapped up in a data class)
- Needs to tell the UI to prepare a success view or a failure view
- Needs to know how to make a new user
- Needs to ask the persistence layer whether a username exists
- Needs to tell the persistence layer to save a new user

REGISTER NEW USER USE CASE SKETCH

```
class UserFactory {

    public User create(String name, String password) { ... }

class UserRegisterInteractor {

  private UserFactory userFactory;    # a class to create User objects

  private UserPresenter userPresenter;  # controls the UI

  private UserRegisterGateway userRegisterGateway;  # persisting data

  public UserRegisterResultModel create(String name, String password: str) {

    # On success, this method returns an object with the

    # username and creation time, but not the password (why?) }
```

Computer Science
UNIVERSITY OF TORONTO

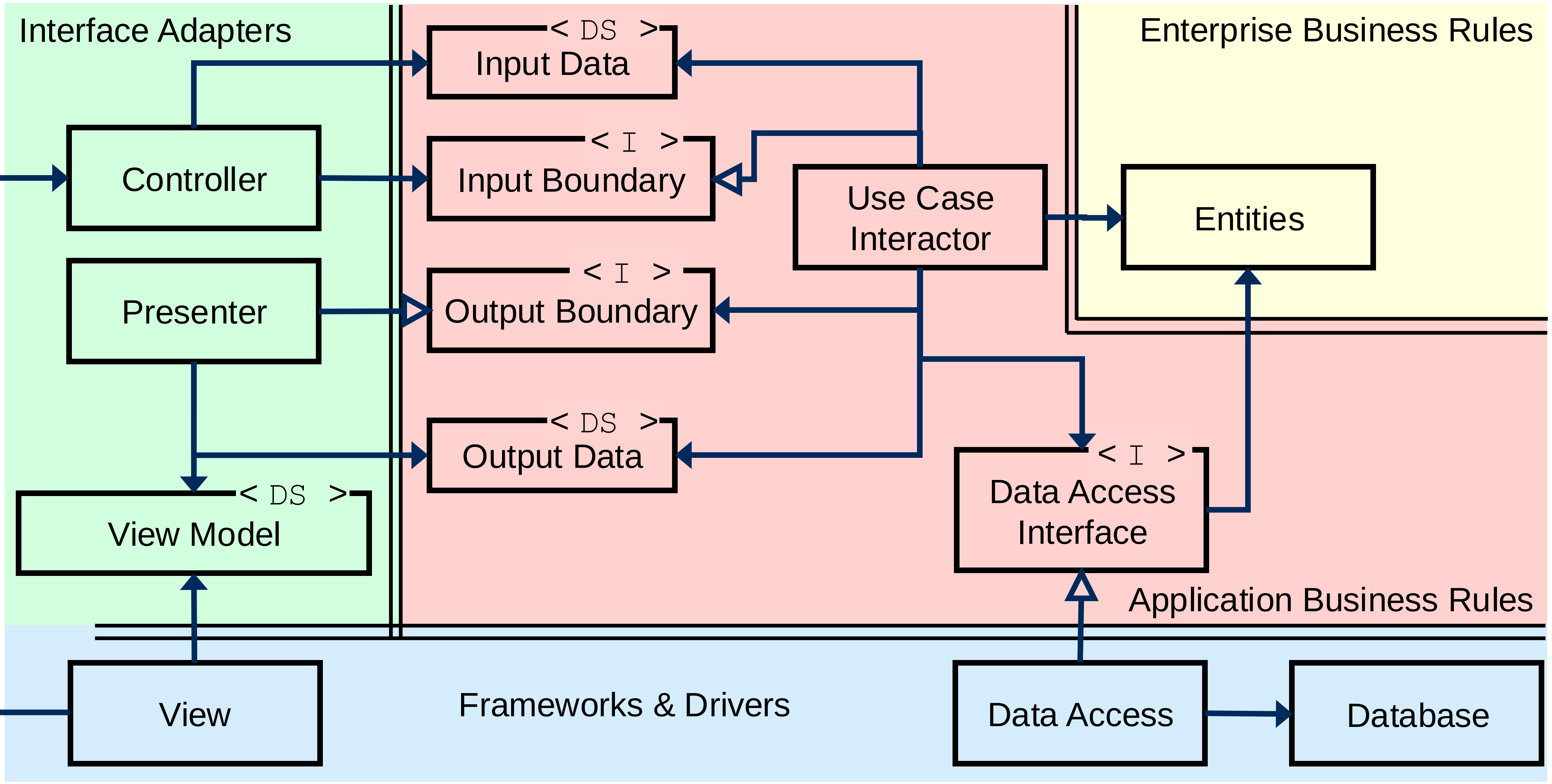


UML SEQUENCE DIAGRAMS

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE

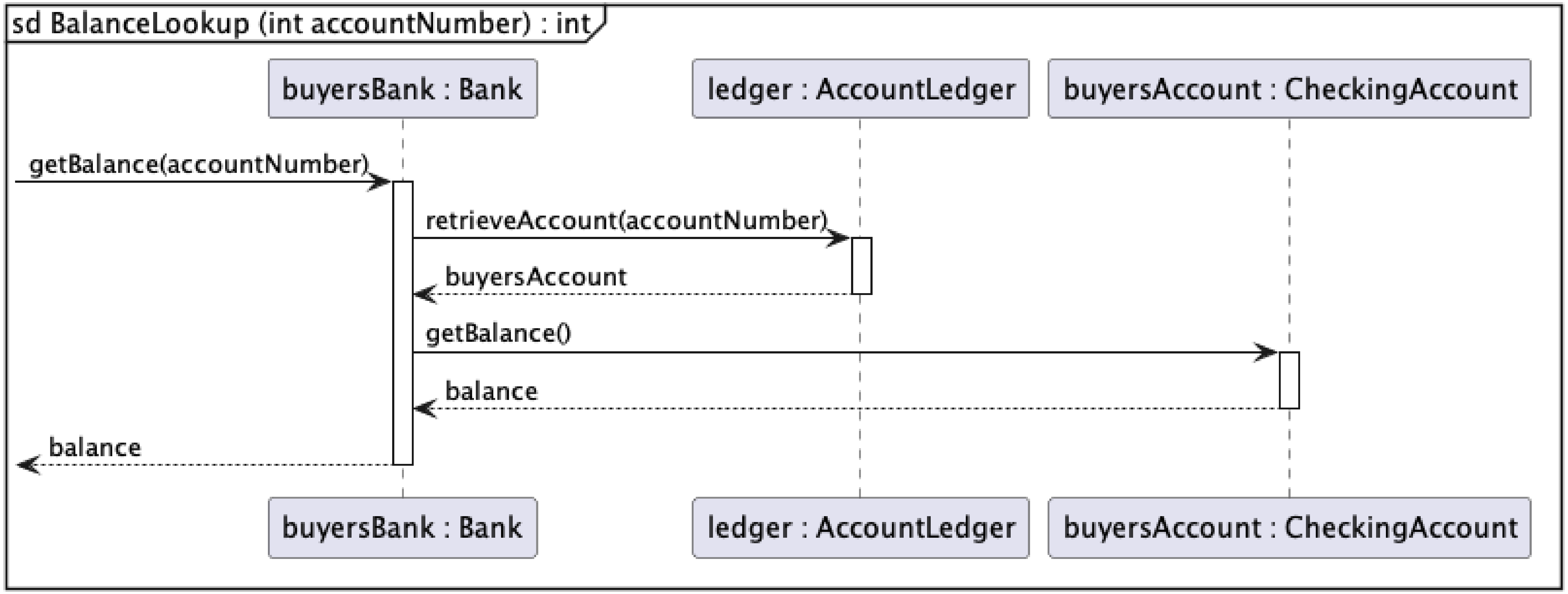


Computer Science
UNIVERSITY OF TORONTO

HOW DATA FLOWS IN THE CLEAN ARCHITECTURE

- In the Clean Architecture, data flows from the user to an action method then through the Controller to the Use Case Interactor. The Interactor will create, read, update, and delete any appropriate Entities for that interaction, using DAOs to create, read, update, and delete the corresponding data in a database or a set of files.
- When the Interactor is done, the resulting data needs to flow from the Interactor through the Presenter back to the View.
- Each View may contain several user actions, each of which has 1 action method. Each action method has 1 Controller. Each Controller has 1 Interactor. Each Interactor has 1 to several Data Access Objects and 1 Presenter. The Presenter has 1 to several View Models. The various Views are observing the appropriate View Models so that each knows when to update itself.
- Let’s draw all that!

A FIRST SEQUENCE DIAGRAM


sd BalanceLookup (int accountNumber) : int
buyersBank : Bank
ledger : AccountLedger
buyersAccount : CheckingAccount
getBalance(accountNumber)
retrieveAccount(accountNumber)
buyersAccount
getBalance()
balance
balance
buyersBank : Bank
ledger : AccountLedger
buyersAccount : CheckingAccount

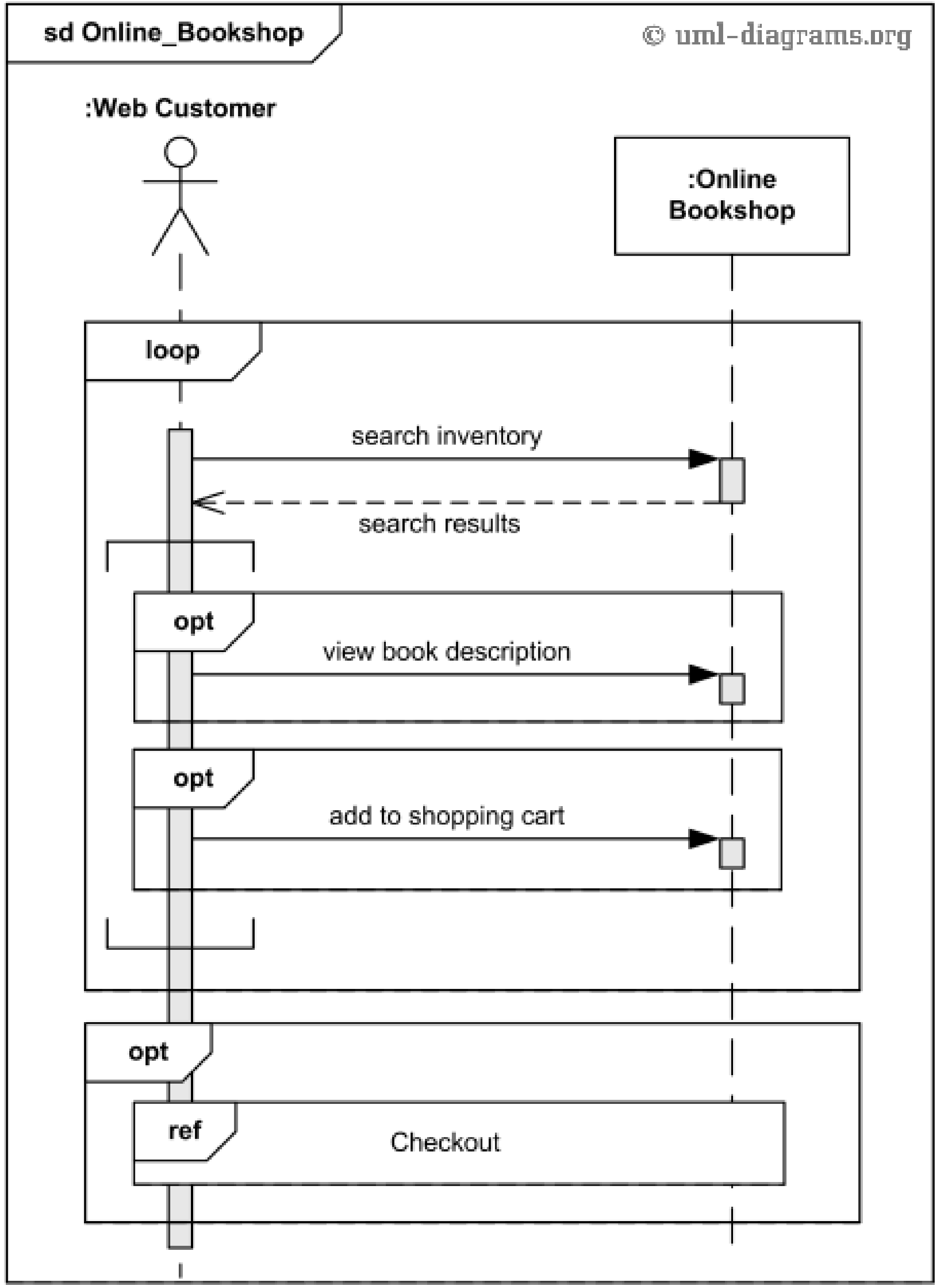
https://developer.ibm.com/articles/the-sequence-diagram/

PLANTUML FOR THE PREVIOUS EXAMPLE

```
@startuml
'https://plantuml.com/sequence-diagram

mainframe sd BalanceLookup (int accountNumber) : int

[-> "buyersBank : Bank" : getBalance(accountNumber)
activate "buyersBank : Bank"

"buyersBank : Bank" -> "ledger : AccountLedger" : retrieveAccount(accountNumber)
activate "ledger : AccountLedger"
return buyersAccount

"buyersBank : Bank" -> "buyersAccount : CheckingAccount" : getBalance()
activate "buyersAccount : CheckingAccount"
return balance

return balance

@enduml
```

Computer Science
UNIVERSITY OF TORONTO

ANOTHER SEQUENCE DIAGRAM



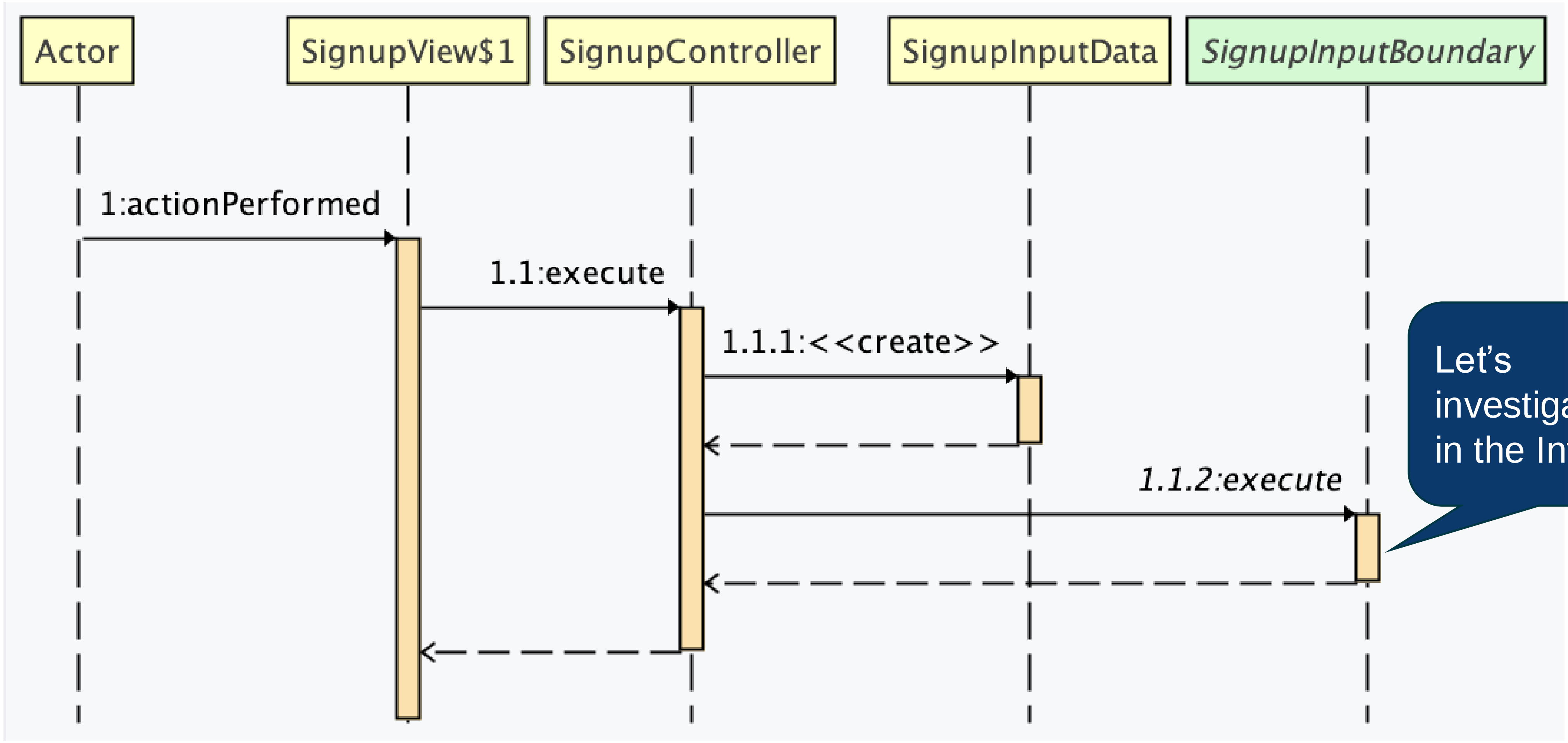
https://www.uml-diagrams.org/online-shopping-uml-sequence-diagram-example.html

Computer Science
UNIVERSITY OF TORONTO

WHAT SEQUENCE DIAGRAMS SHOW

- A UML Sequence Diagram tells the story of how and when methods are called during code execution.
- You can read the code to figure out the sequence of calls and draw one of these, using as much or as little detail about parameters and return values as you like. You can indicate variable names, or not.
- The SequenceDiagram plugin in IntelliJ can do these kinds of things too!
- You can also make a Sequence Diagram as part of your design process *before* writing any code.

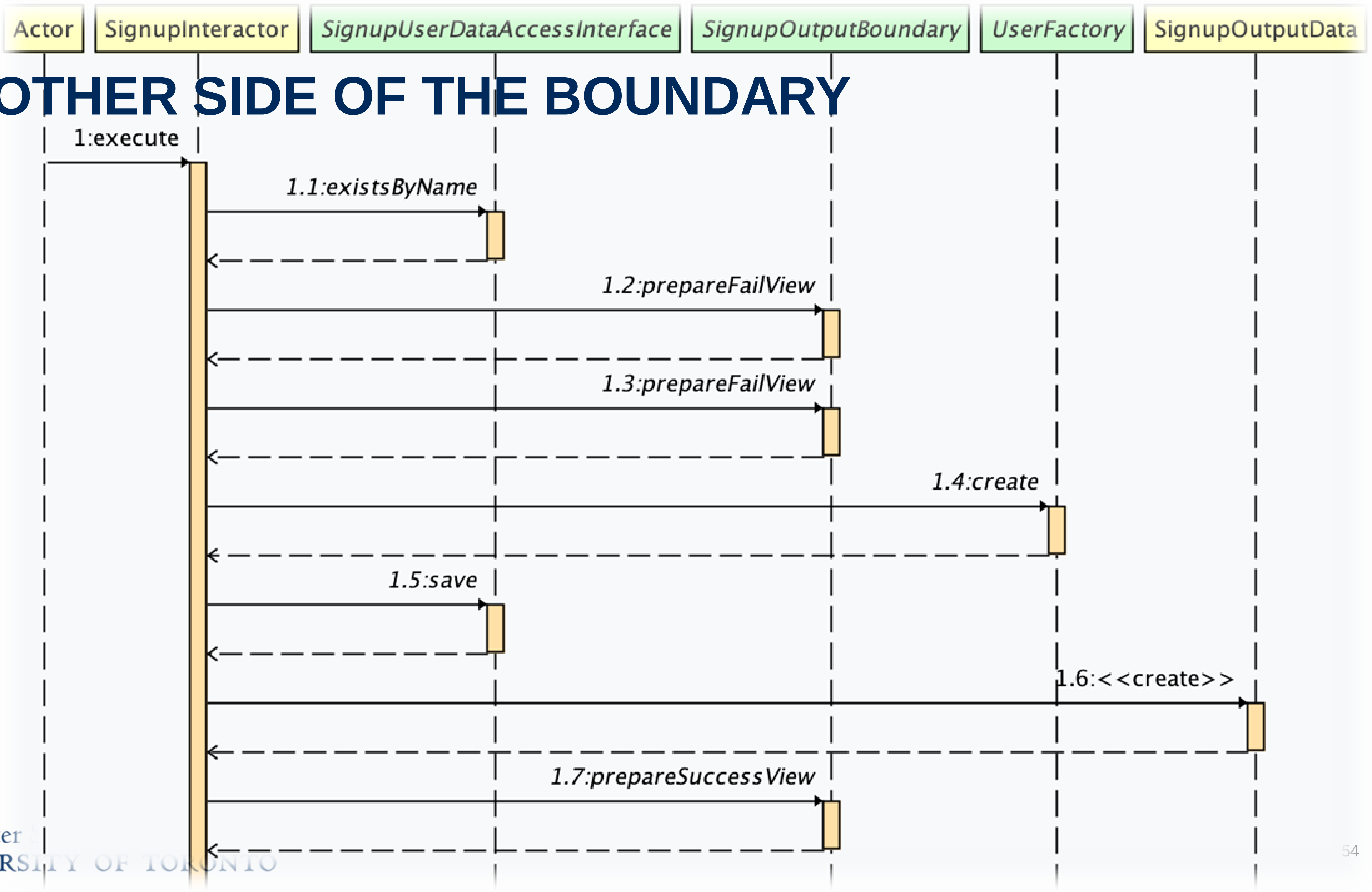
EXAMPLE FROM THE LOGIN CODEBASE



Let's investigate this in the Interactor

Computer Science
UNIVERSITY OF TORONTO

THE OTHER SIDE OF THE BOUNDARY



Computer
UNIVERSITY OF TORONTO

THE PRESENTER’S STORY

- Let’s go to IntelliJ to investigate.