COURSE SUMMARY

CSC 207 SOFTWARE DESIGN

LEARNING OUTCOMES

- Identify what you know and what you feel you need to review before the exam!

Notes:

- No new material this week! Everything is review.
- Use the project to study for the exam: apply the course material to solidify your understanding.
- These slides aren't exhaustive but aim to highlight the main topics from the course.

COURSE EVALUATIONS

- Please do them!
  - Constructive feedback is appreciated.
- We use the feedback to improve the course.
  - You can also email the course address to provide feedback — this is best for overall thoughts on how the course is structured, as the evals are by lecture section, but 207 has coordinated sections.
- Course evals are also used for promotion and end-of-year evaluation.
  - Equivalent of the grades you get in your courses as a student!

FINAL EXAM ADVICE I

It is your chance to show us what you know. You can do this by:

- Carefully reading **all the words** in each question and looking at how many marks things are worth.
- **Clearly expressing yourself using terminology we've introduced in the course.**
- Writing something down for every question — we can't give you any credit for a blank page!
- **Be explicit in your answers — we can only give you credit for what you wrote down and clearly explained!**
- **BRING YOUR T-CARD OR YOU WON'T GET INTO THE ROOM.**

FINAL EXAM ADVICE II

Example Question

- Explain why having a Square be a subclass of a Rectangle is an example of a Liskov violation. [2 marks]

Minimal Answer

- Because a square is more constrained than a rectangle. (***We need you to elaborate though — this does not look like a 2 mark answer!***)

But…

what *is* a Liskov violation?

How is it more constrained?

Would a picture or small code example help?

We want you to convince us that you understand the concepts! (Think of the question as a prompt — but make sure to stay on topic! Be clear and concise in your answers!)

COMMUNICATION

- Communication is important when working with other people. Communication is also how we know to give you marks!
- For the exam, you should be able to communicate design ideas using:
    - Java code
    - Words
    - UML diagrams
- You should also be able to read and interpret design ideas from other people in each of the above formats. (example: write code based on a UML diagram)
- There is rarely only one correct answer when solving a design problem. But there are better answers and worse ones. You should be able to communicate the good and bad aspects of a design decision using terminology from the course.

PROFESSIONAL SKILLS

- A large part of this course aimed to give you experience that will be helpful in your professional life.
  - Working as part of a team
  - Designing a complex program over months
  - Using version control
  - Reading documentation and being self-sufficient, but also asking for help as needed.
- **Reflecting on what worked — and what didn't — will help you make better design decisions in the future!**
- If your code is easy to read, navigate, extend, and work on at the same time as other people work on it, then it is likely:
  - well organized, consistent in its design, following the SOLID principles, and well encapsulated.
  - demonstrating effective Object-Oriented Design!

FEATURES OF OBJECT-ORIENTED PROGRAMMING LANGUAGES

- *Abstraction* — the process of distilling a concept to a set of essential characteristics.
- *Encapsulation* — the process of binding together data with methods that manipulate the data and hiding the internal representation from the user.
  - The result of applying abstraction and encapsulation is (often) a class with instance variables and methods that together model a concept from the real world. (Further reading: what's the difference between Abstraction, Encapsulation, and Information Hiding?)
- *Inheritance* — the concept that when a subclass is defined in terms of another class, the features of that other class are inherited by the subclass (enabling reuse of common code).
- *Polymorphism* ("many forms") — the ability of an expression (such as a method call) to be applied to objects of different types.

UML MENTOR

- A great tool to practice designing systems using UML diagrams to satisfy a written specification.
- Developed by past CSC207 students on the UTM campus!
- https://umlmentor.utm.utoronto.ca/home
- Once you attempt a "challenge", you can post your proposed design and then discuss your design and those of other students!

SOLID DESIGN PRINCIPLES

Computer Science
UNIVERSITY OF TORONTO

FUNDAMENTAL OOD PRINCIPLES

SOLID: five basic principles of object-oriented design

(Developed by Robert C. Martin, affectionately known as “Uncle Bob”.)

Single responsibility principle (SRP)

Open/closed principle (OCP)

Liskov substitution principle (LSP)

Interface segregation principle (ISP)

Dependency inversion principle (DIP)

SOLID (PRINCIPLES OF CLASS DESIGN)

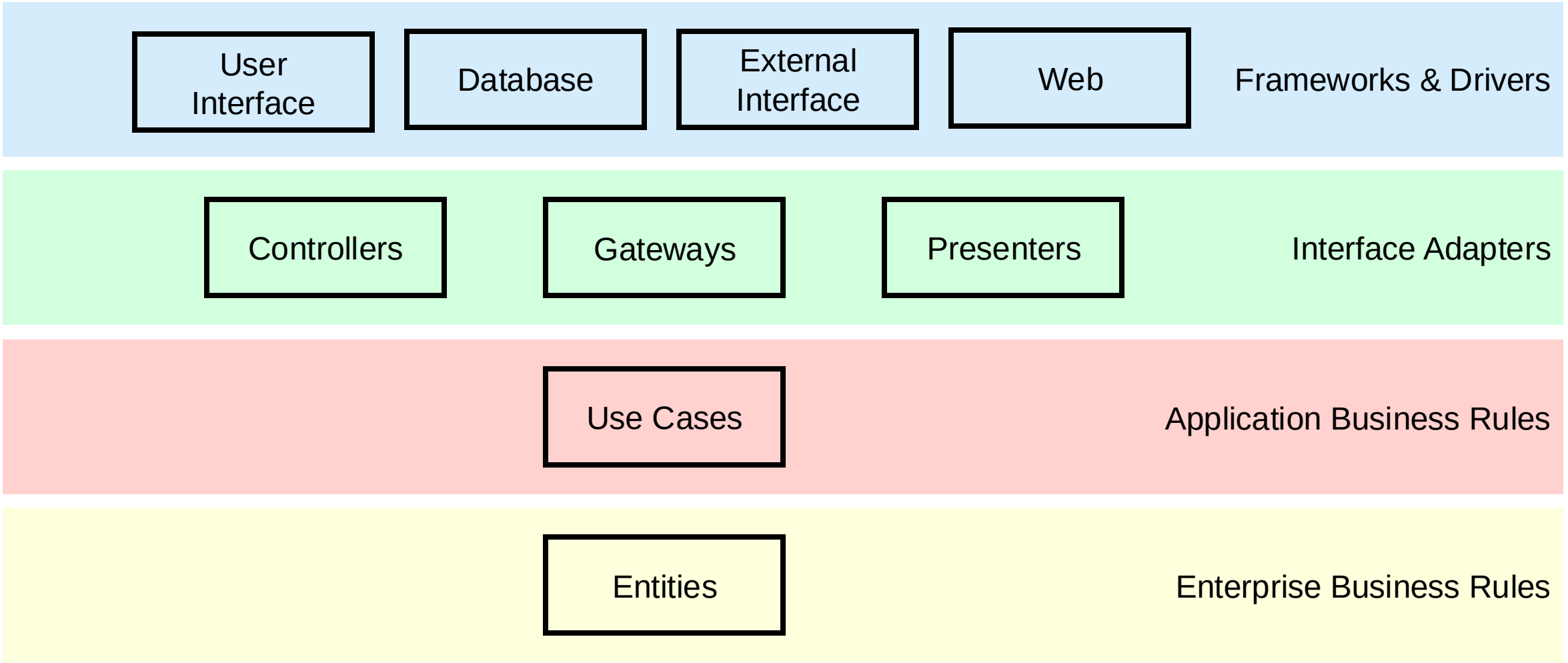
SRP	The Single Responsibility Principle'	A class should have one, and only one, reason to change.
OCP	The Open Closed Principle	You should be able to extend the behaviour of a class without modifying the class.
LSP	The Liskov Substitution Principle	Derived classes must be substitutable for their base classes.
ISP	The Interface Segregation Principle	Make fine grained interfaces that are client specific.
DIP	The Dependency Inversion Principle	Depend on abstractions, not on concretions.

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

CLEAN ARCHITECTURE

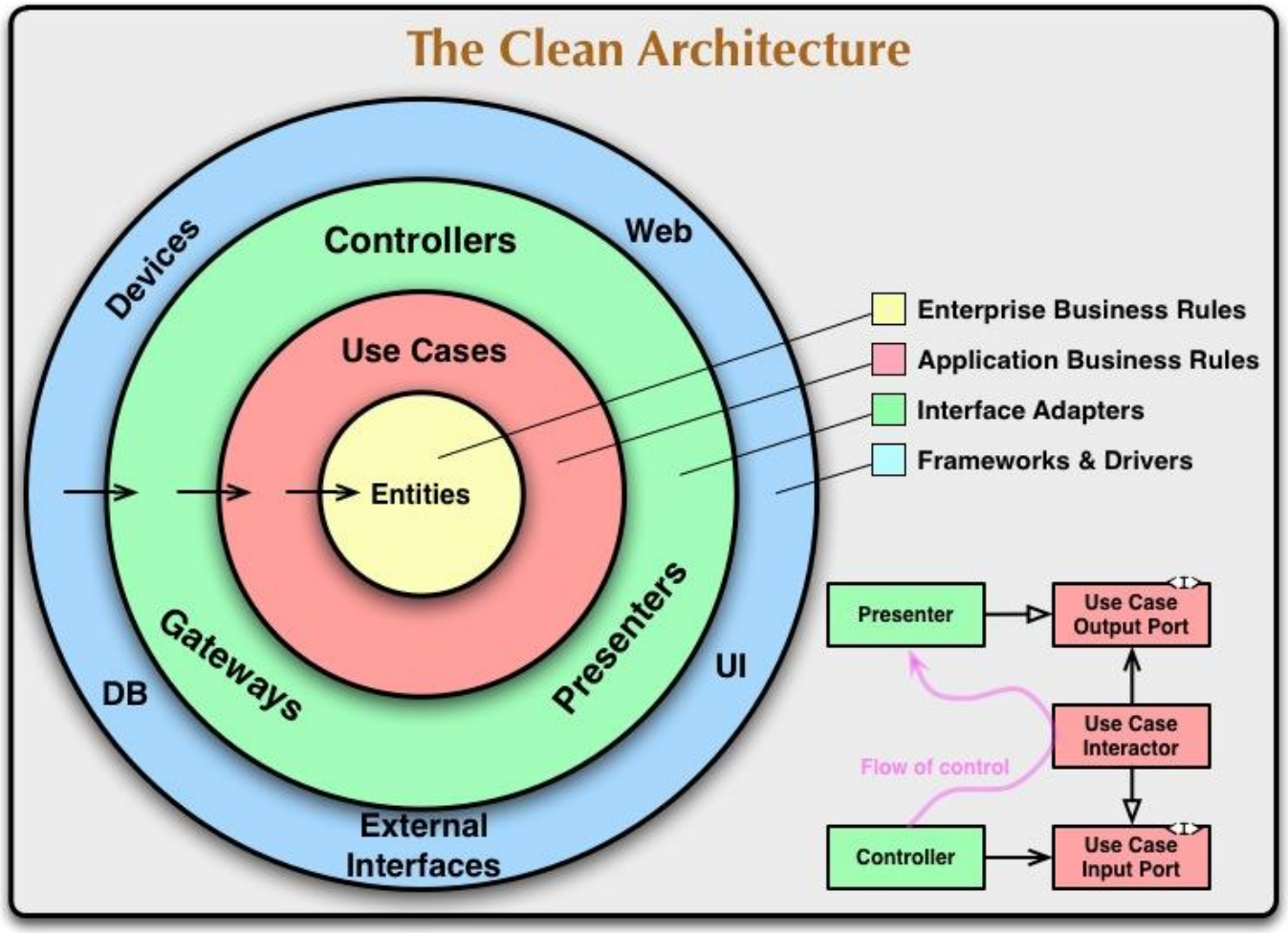
CSC207 SOFTWARE DESIGN

CLEAN ARCHITECTURE

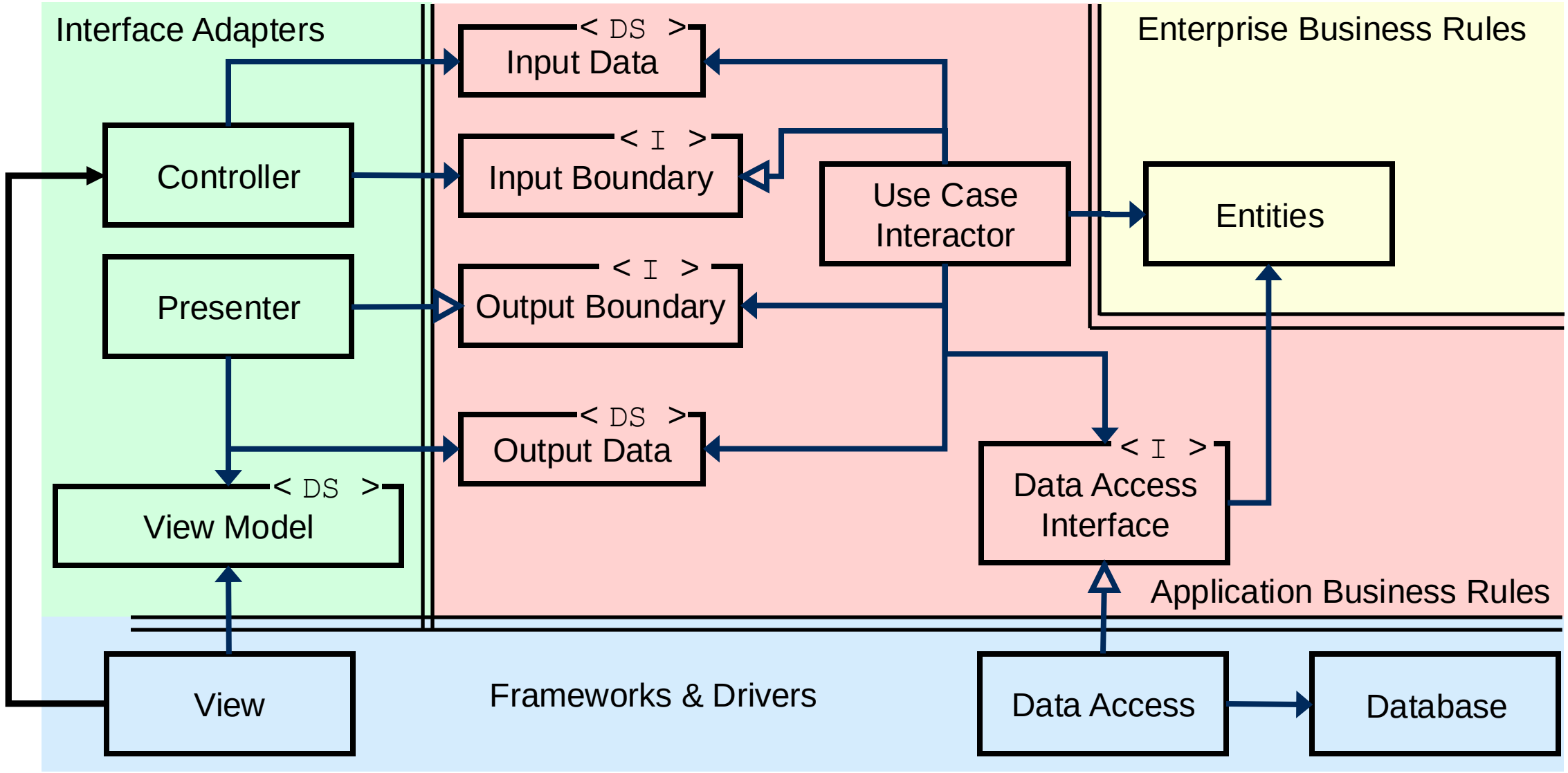


CLEAN ARCHITECTURE

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



OUR CA ENGINE



CLEAN ARCHITECTURE – DEPENDENCY RULE

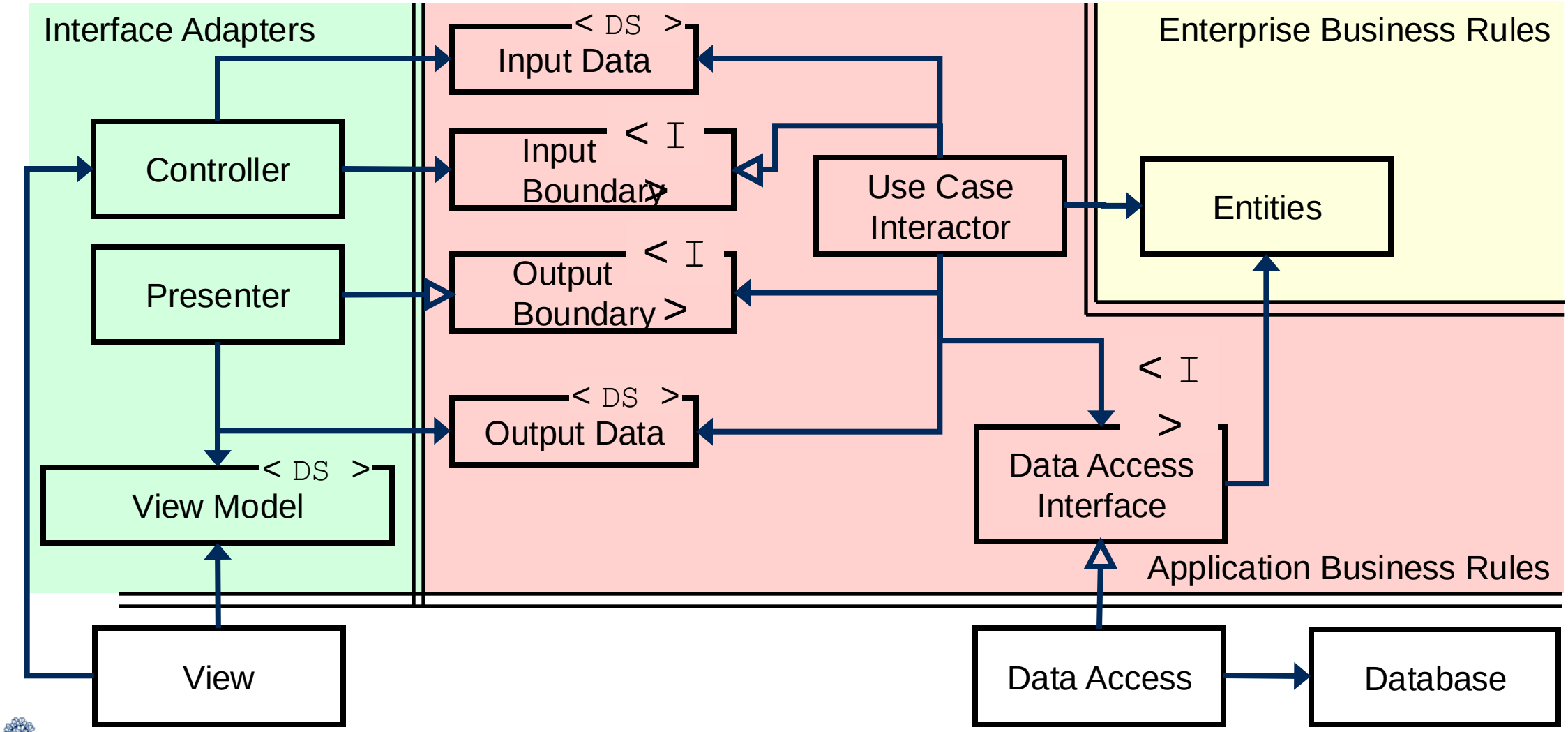
- Dependence on adjacent layer — from outer to inner
- Dependence within the same layer is allowed (but try to minimize coupling)
- On occasion you might find it unnecessary to explicitly have all four layers, but you'll almost certainly want at least three layers and sometimes even more than four.
- **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. The outer layer is injected into the inner layer.

BENEFITS OF CLEAN ARCHITECTURE

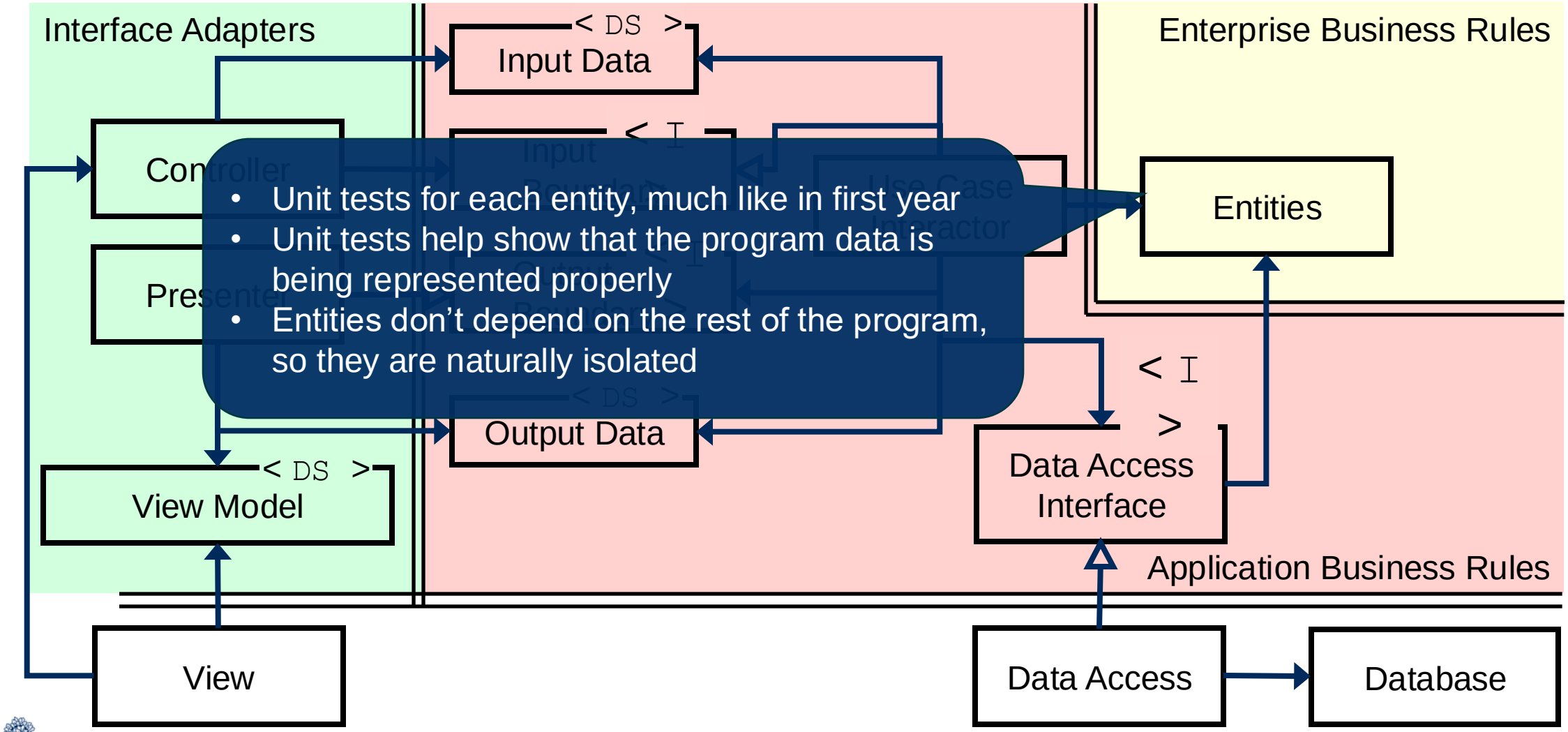
- All the “details” (frameworks, UI, Database, etc.) live in the outermost layer.
- The business rules can be easily tested without worrying about those details in the outer layers!
- Any changes in the outer layers don’t affect the business rules!

TESTING

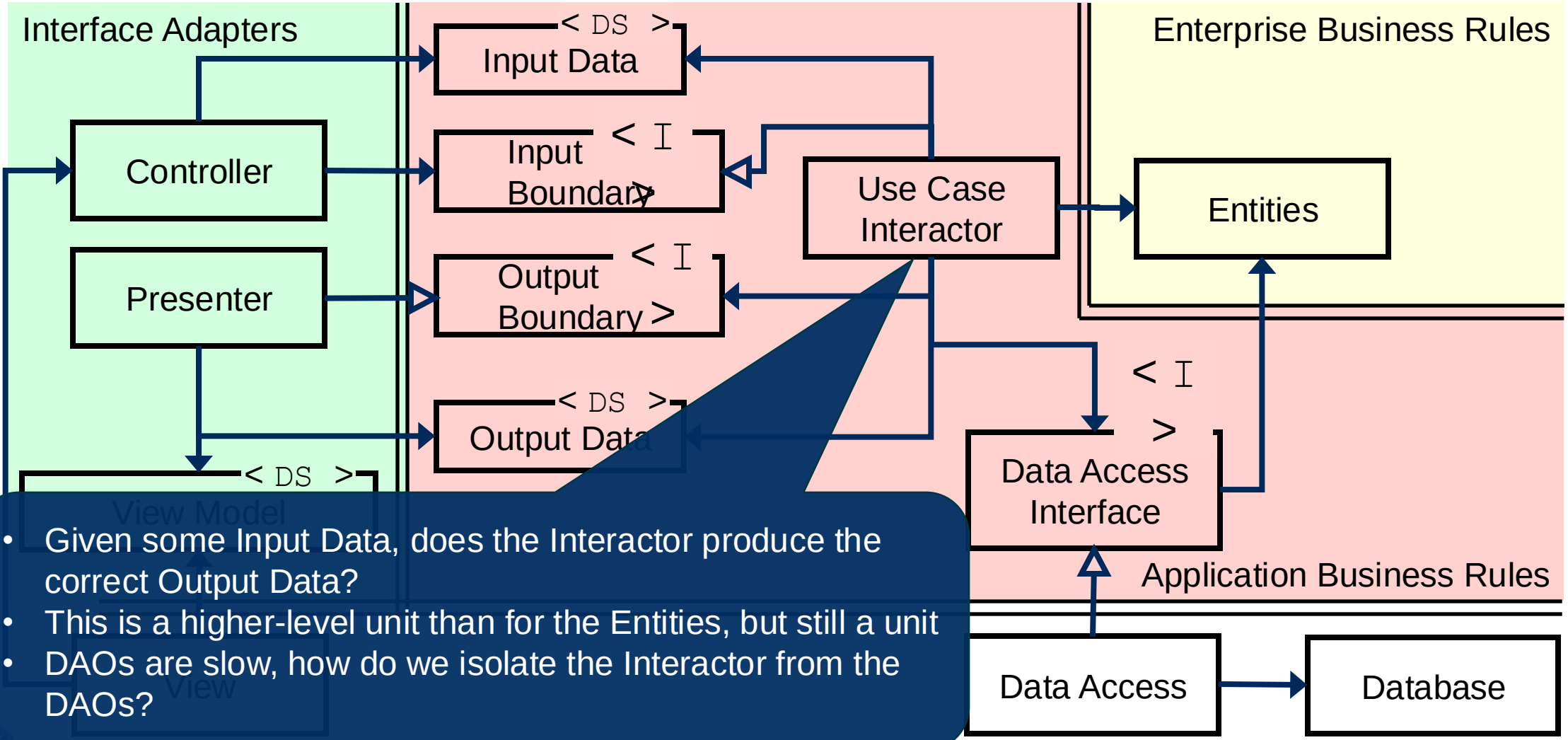
HOW CAN WE TEST THIS?



ENTITIES: UNIT TESTING

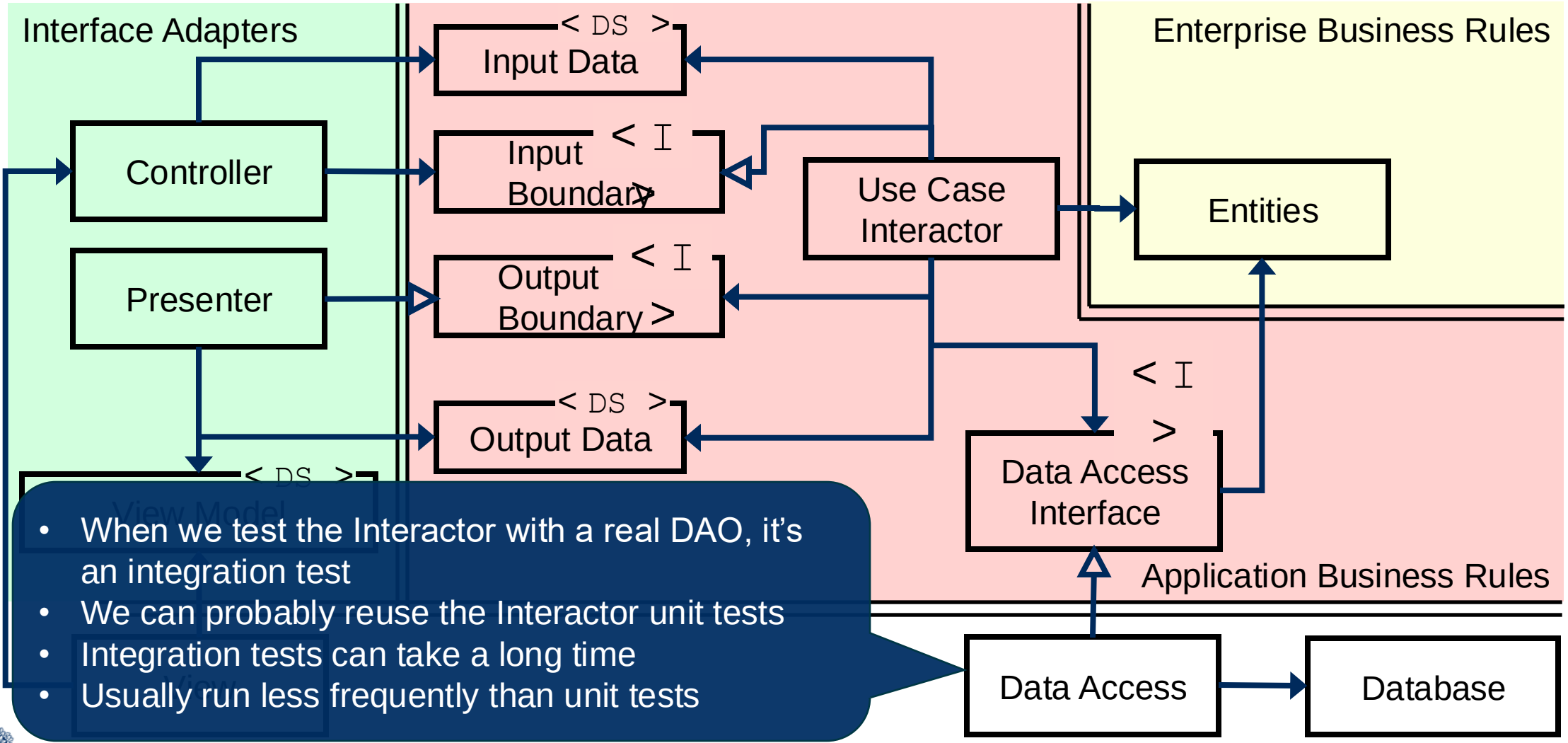


INTERACTORS: UNIT TESTING

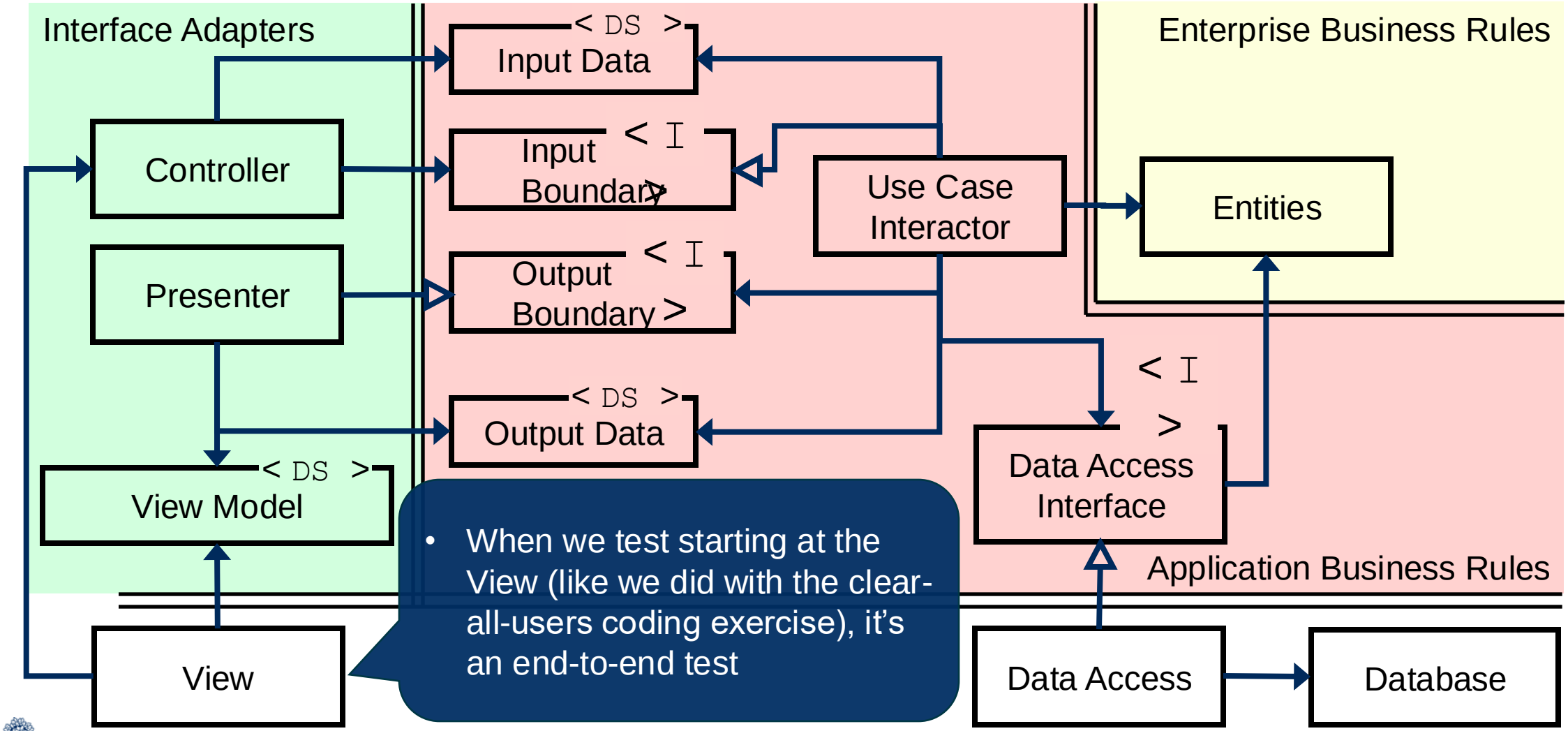


- Given some Input Data, does the Interactor produce the correct Output Data?
- This is a higher-level unit than for the Entities, but still a unit
- DAOs are slow, how do we isolate the Interactor from the DAOs?

INTERACTORS AND DAOS: INTERACTION TESTING



HOW CAN YOU ISOLATE EACH LAYER?



USE CASE INTERACTOR TESTING

Interactors receive Input Data from a Controller and produce Output Data for a Presenter

Your test class serves as the Controller

Your test class needs to create a Presenter that checks the Output Data and any Entity and persistence mutation

USE CASE INTERACTOR: THE SUCCESS TEST

```
void successTest() {
    SignupUserDataAccessInterface userRepository = new InMemoryUserDataAccessObject();

    SignupOutputBoundary successPresenter = // Make a presenter here that asserts things (see next slide)

    SignupInputData inputData = new SignupInputData("Paul", "password", "password");


    SignupInputBoundary interactor = new SignupInteractor(
            userRepository, successPresenter, new CommonUserFactory());
    interactor.execute(inputData); // This will eventually send Output Data to the successPresenter
}
```

USE CASE INTERACTOR: THE PRESENTER

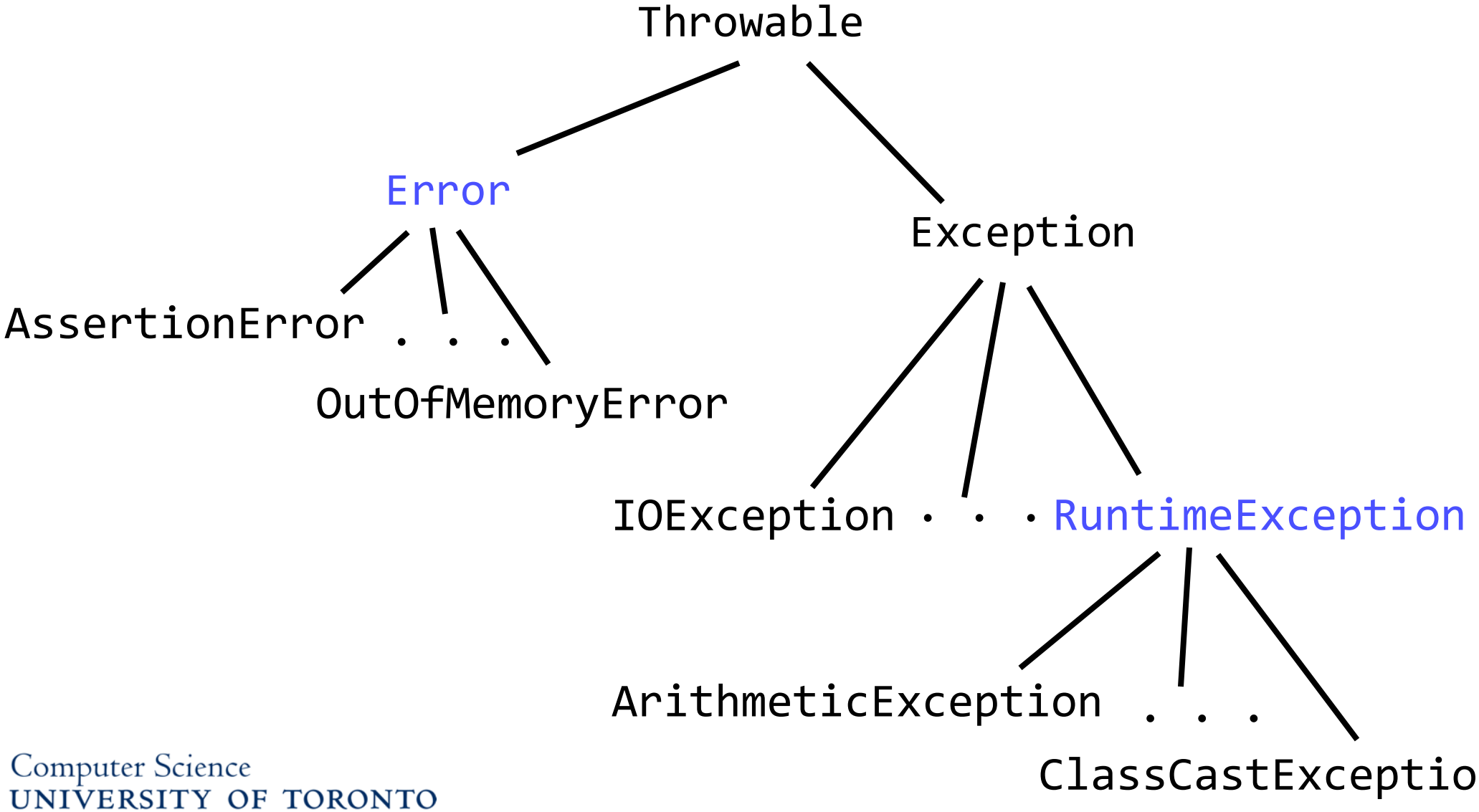
This goes in the successTest method:

```
// This instantiates an anonymous SignupOutputBoundary implementing class
SignupOutputBoundary successPresenter = new SignupOutputBoundary() {
    @Override
    public void prepareSuccessView(SignupOutputData user) {
        // 2 things to check: the output data is correct, and the user has been created in the DAO.
        assertEquals("Paul", user.getUsername());
        assertNotNull(user.getCreationTime()); // any creation time is fine.
        assertTrue(userRepository.existsByName("Paul"));
    }

    @Override
    public void prepareFailView(String error) {
        fail("Use case failure is unexpected.");
    }
};
```

EXCEPTIONS

THE THROWABLE HIERARCHY



CHECKED VS UNCHECKED EXCEPTIONS

- Checked Exceptions:
    - A method will include "throws ExceptionClassName" in its declaration
    - The compiler will check to make sure that the Exception is caught by the method that calls it, or the method that calls that method, or...
- Unchecked Exception:
    - The program will throw these without checking to see if it is eventually caught
    - Will crash the program and print the stack trace to the terminal
    - You can then see where the Exception was thrown and fix the code so that does not happen again (helpful when debugging!)

DON’T USE THROWABLE OR ERROR

- `Throwable` itself, and `Error` and its descendants are probably not suitable for subclassing in an ordinary program.

- `Throwable` isn't specific enough.
- `Error` and its descendants describe serious, unrecoverable conditions, and are typically at the system level.
  - e.g. `OutOfMemoryError` (a child of `VirtualMachineError`, which is a child of `Error`)

- Throw a specific exception, so that the client code can catch that specific exception.

PACKAGING

HOW COULD YOU PACKAGE THIS CODE?

- **`OrdersController`**: A web controller, something that handles requests from the web
- **`OrdersService`**: An interface that defines the "business logic" related to orders.
- **`OrdersServiceImpl`**: The implementation of the orders service.
- **`OrdersRepository`**: An interface that defines how we get access to persistent order information.
- **`JdbcOrdersRepository`**: An implementation of the repository interface that saves information in a database.

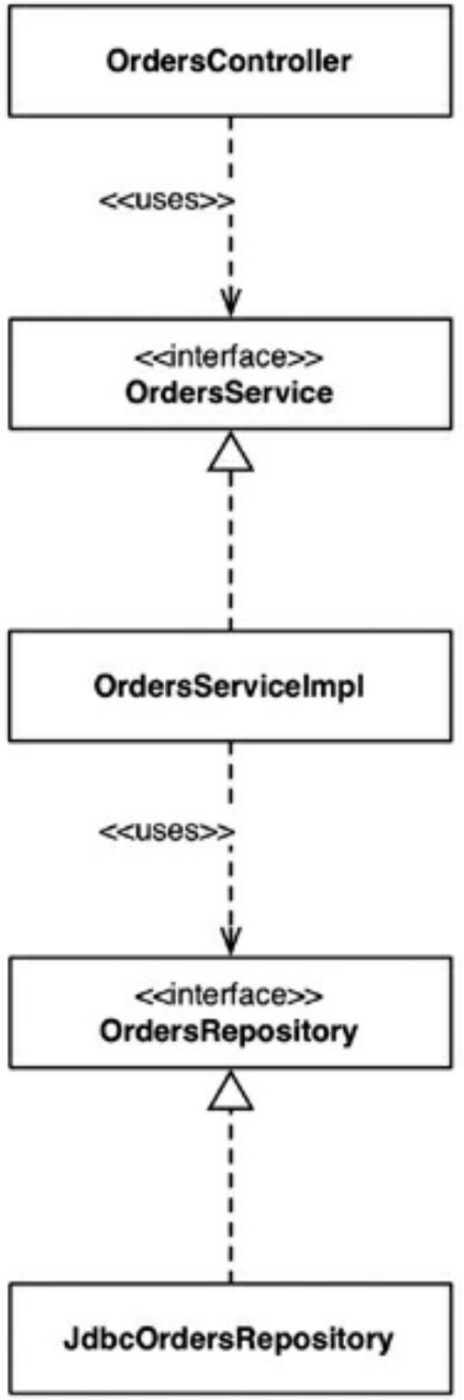
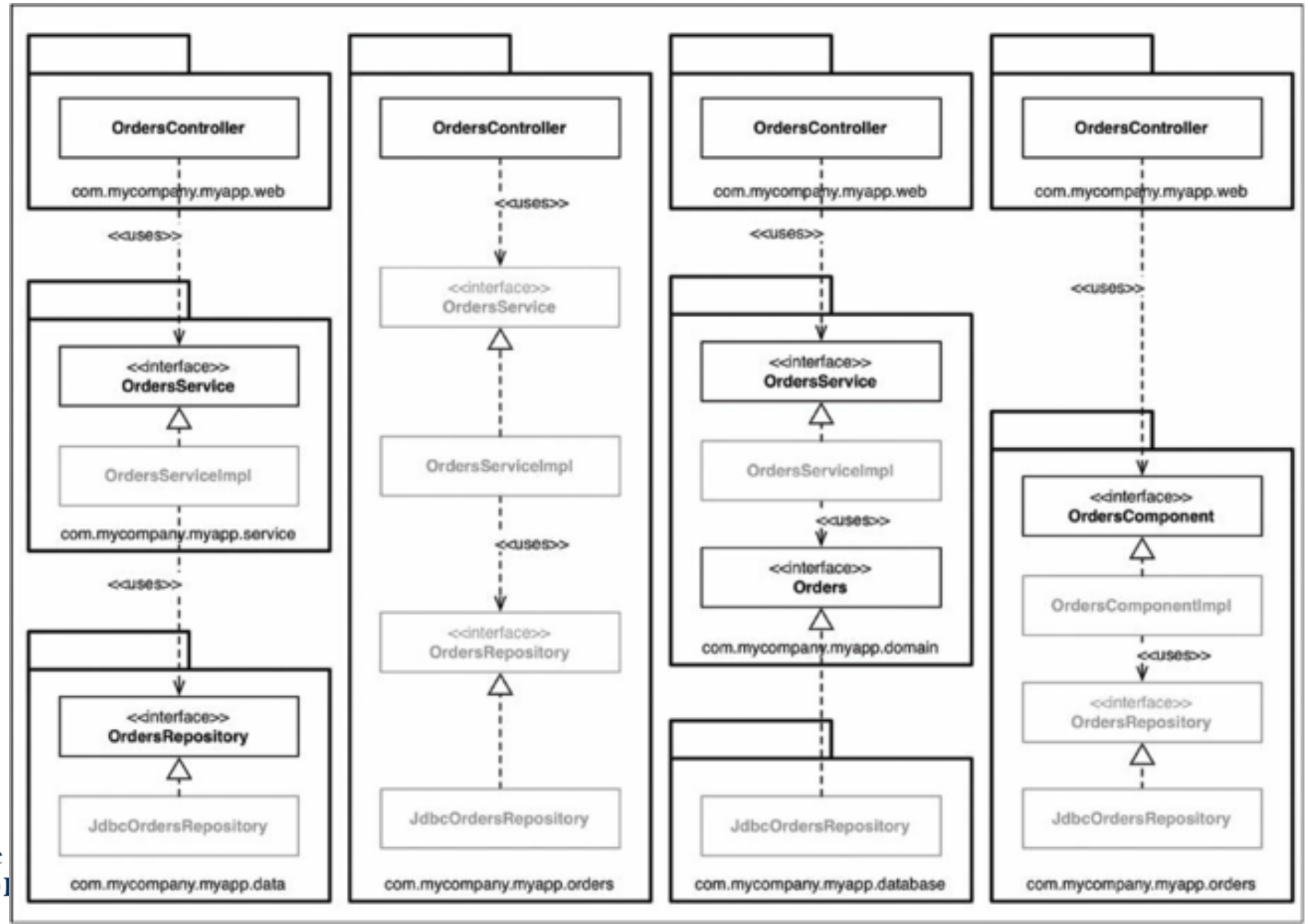
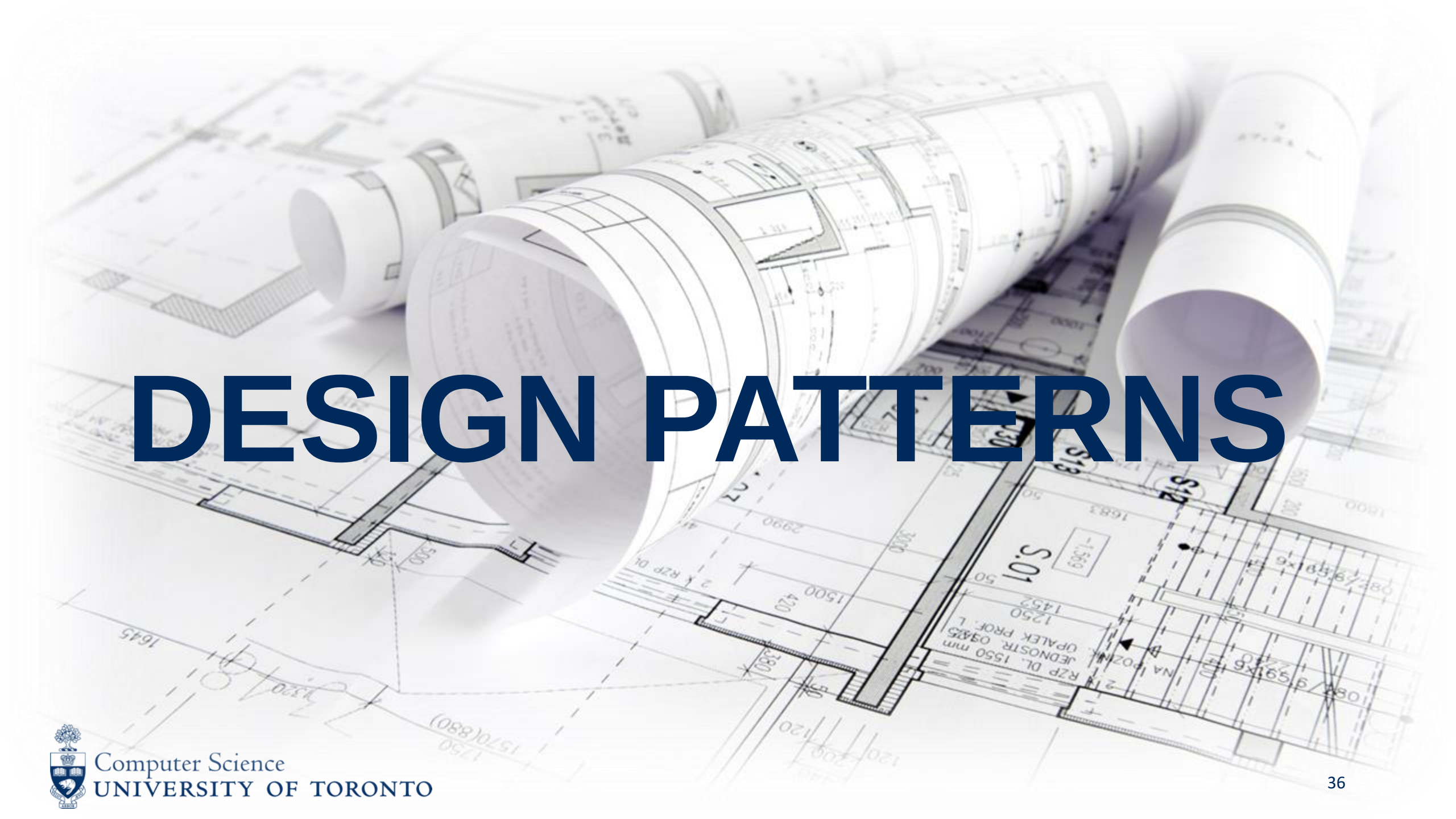


Figure 34.8
Chapter 34
Clean Architecture





DESIGN PATTERNS

DESIGN PATTERNS (REVIEW)

- A **design pattern** is a general description of the solution to a well-established problem.
- Patterns describe the shape of the code rather than the details.
- They’re a means of communicating design ideas.
- They are not specific to any one programming language.

- You’ll learn about lots of patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).

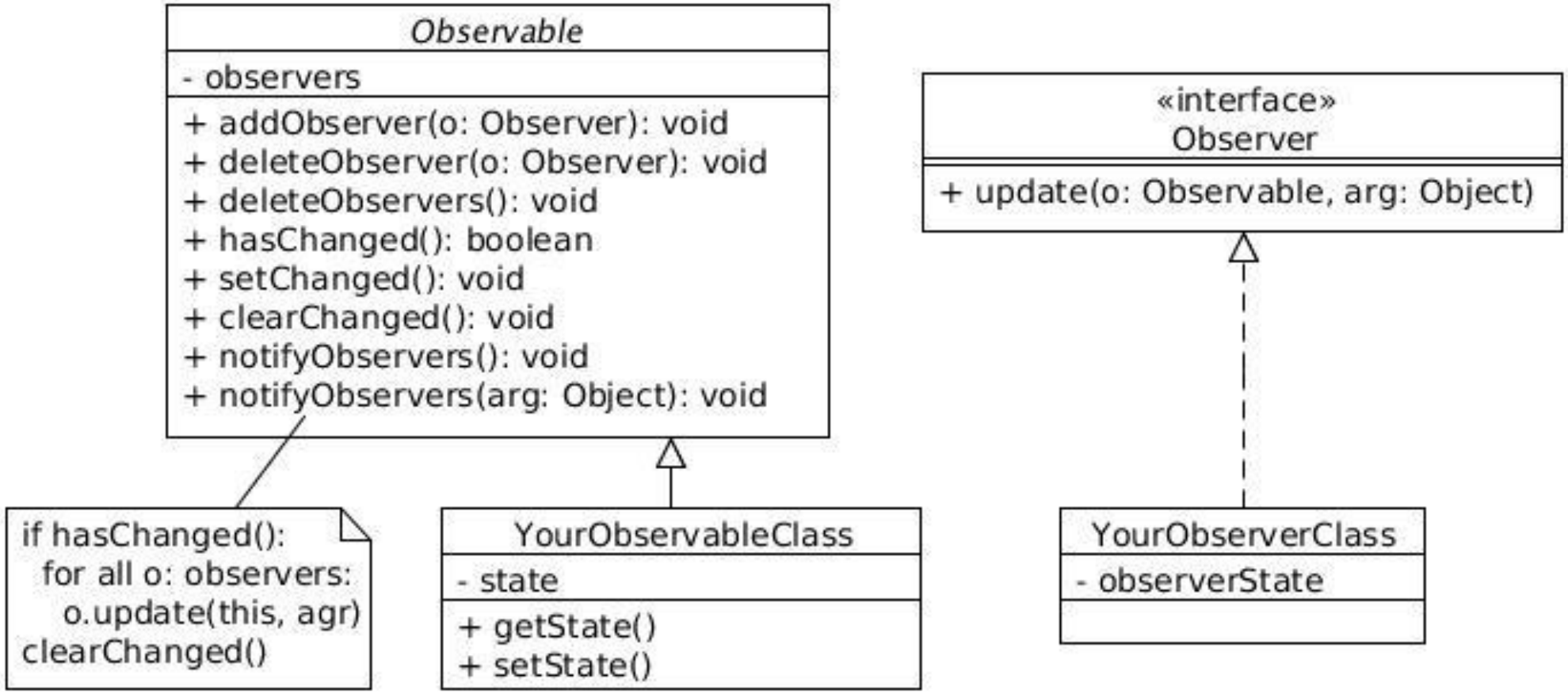
REMINDER: LOOSE COUPLING, HIGH COHESION

These are two goals of object-oriented design.

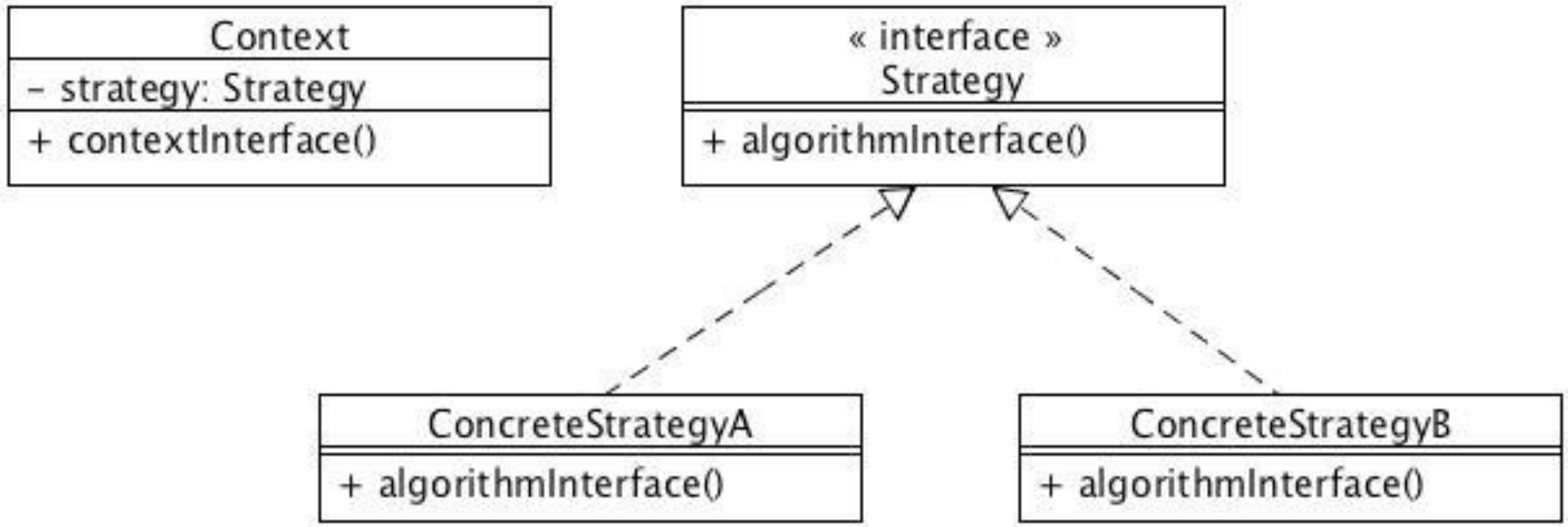
- **Coupling**: the interdependencies between objects. The fewer couplings the better, because that way we can test and modify each piece independently.
- **Cohesion**: how strongly related the parts are inside a class. High cohesion means that a class does one job, and does it well. If a class has low cohesion, then an object has parts that don't relate to each other.

Design patterns are often applied to decrease coupling and increase cohesion.

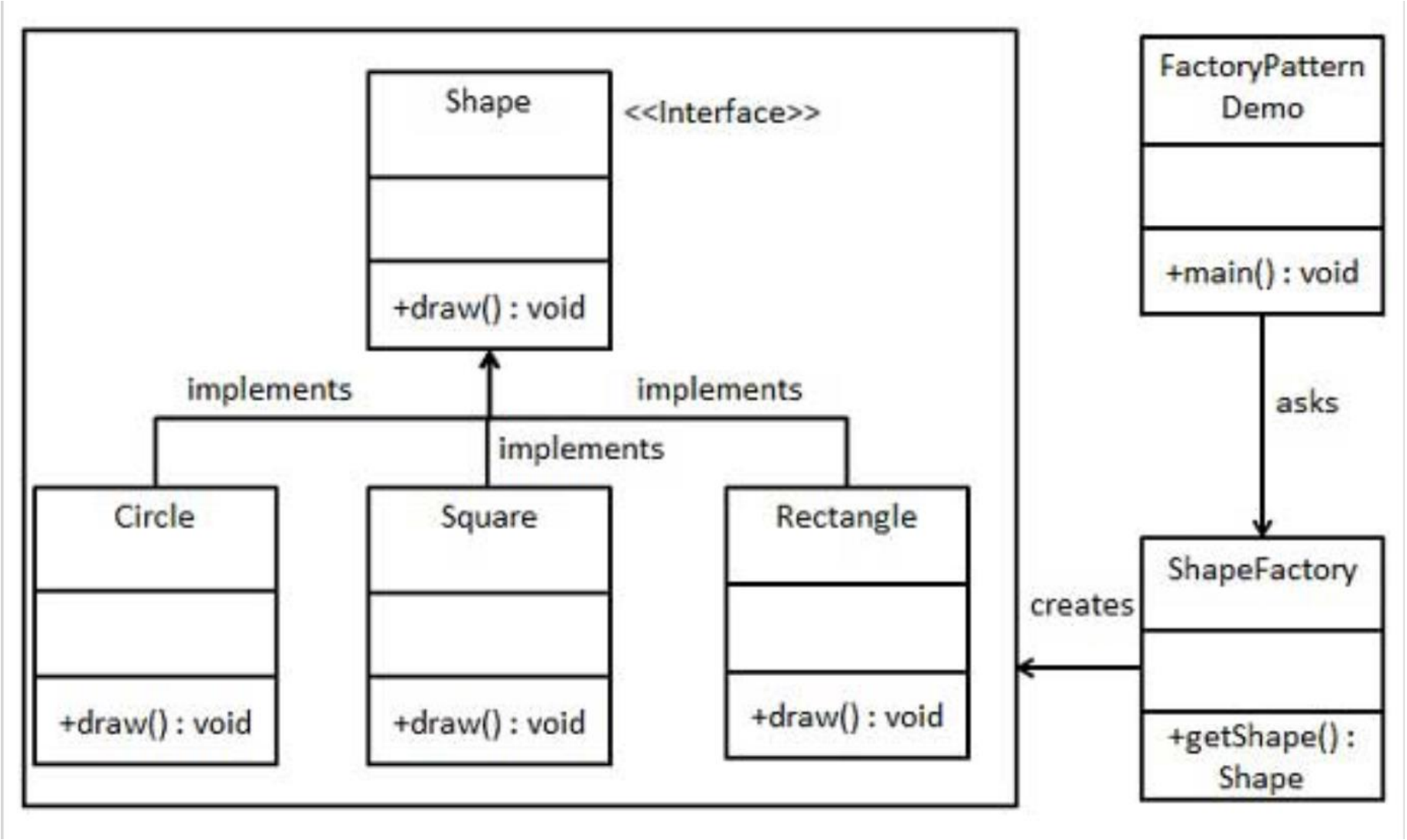
OBSERVER: JAVA IMPLEMENTATION



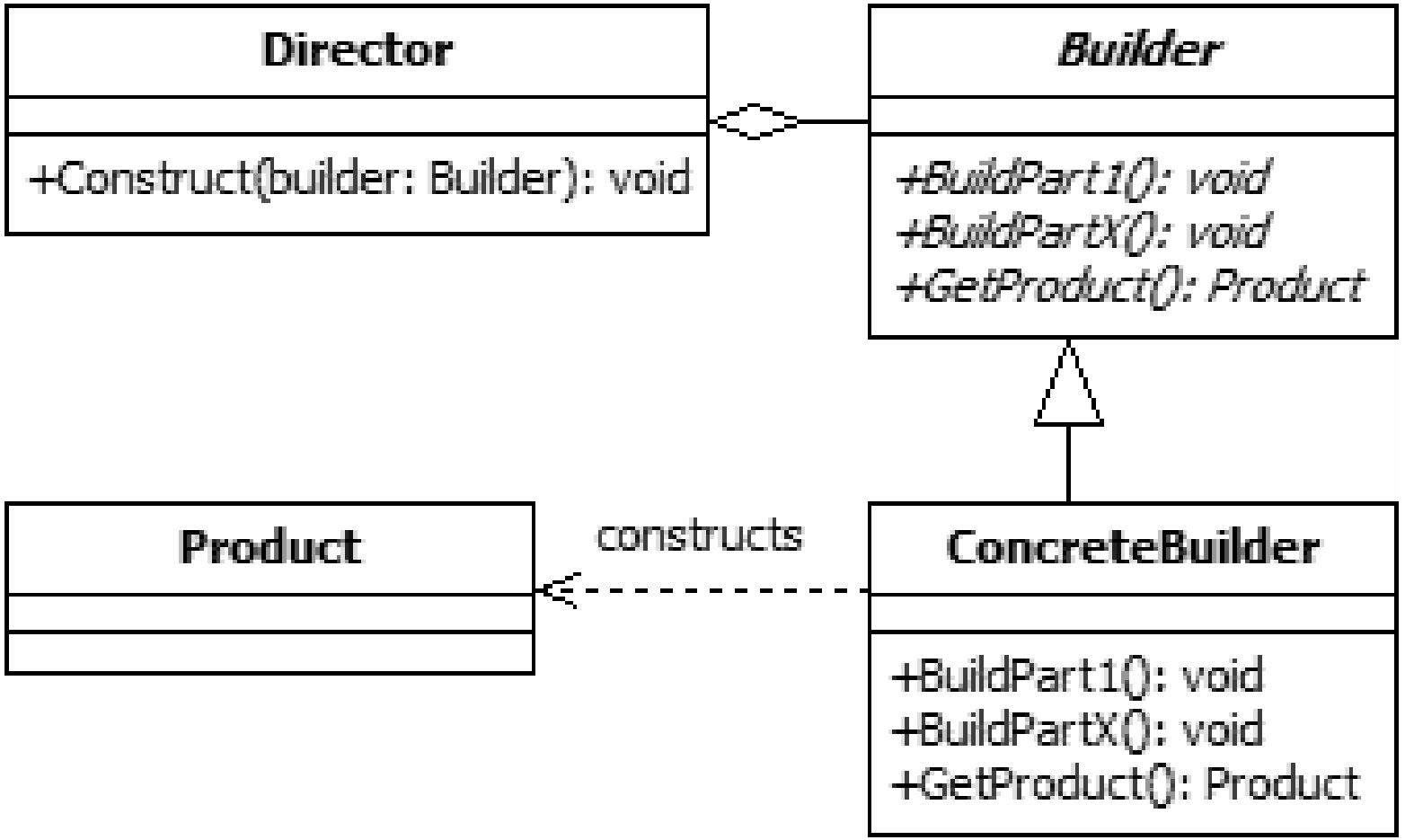
STRATEGY: STANDARD SOLUTION



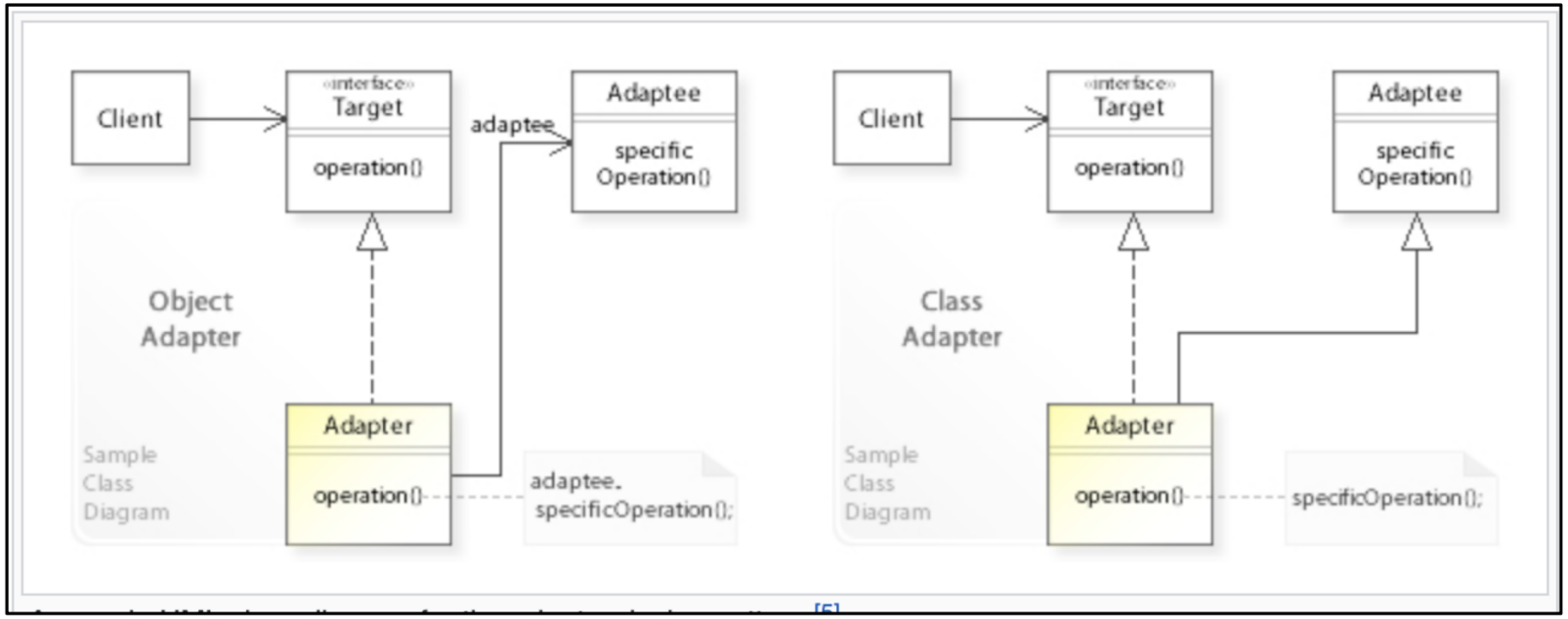
FACTORY : AN EXAMPLE

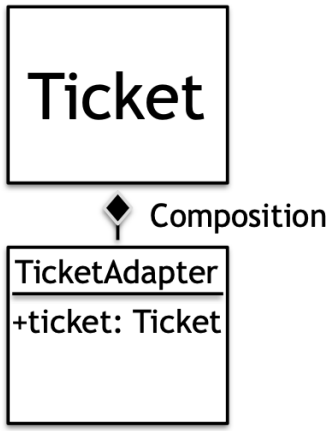


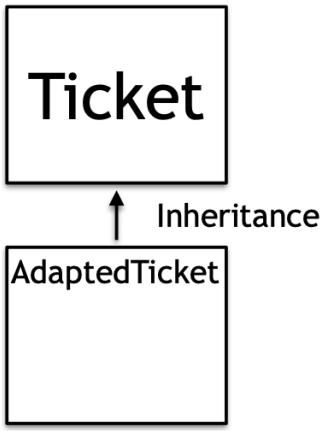
BUILDER DESIGN PATTERN



ADAPTER DESIGN PATTERN







OTHER TOPICS

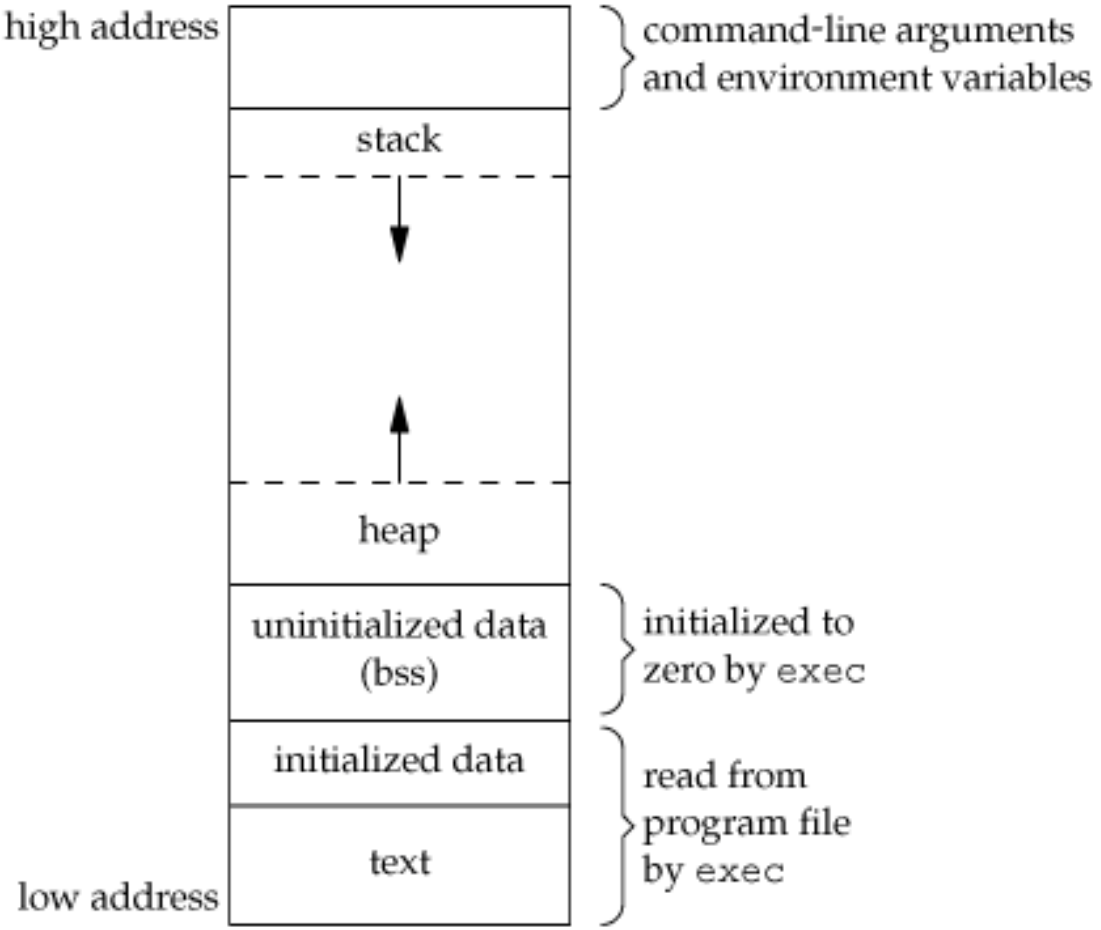
- Embedded Ethics
  - models of disability
  - accessibility
  - Principles of universal design

- Regex
  - the basics
  - regex crossword

- Software Documentation
  - ACCEU and writing README files

NEXT COURSE: CSC209

- C and shell programming
- More about build systems / how compilation works
- More git from the command line!
- Low-level programming
- Memory management (Java and python both took care of this for you!)
- Even more about low-level in CSC369: Operating System!



New view of computer memory

https://medium.com/@vikasv210/memory-layout-in-c-fe4dffdaeed6

OTHER COURSES

Graph of how the various CS courses connect: https://courseography.cdf.toronto.edu/graph

Collection of CS course syllabi: https://web.cs.toronto.edu/undergraduate/courses/outlines

- **Software Engineering**
- **Compilers**
- **Programming languages**
- **Networks**
- **Operating Systems**
- Algorithms
- Theory of Computation
- Computer Vision
- Machine Learning
- **Numerical Computation**
- Natural Language Processing
- Artificial Intelligence
- **Human Computer Interaction**
- **Web Programming**
- **Computers and Society**

THANK YOU FOR AN AWESOME SEMESTER!