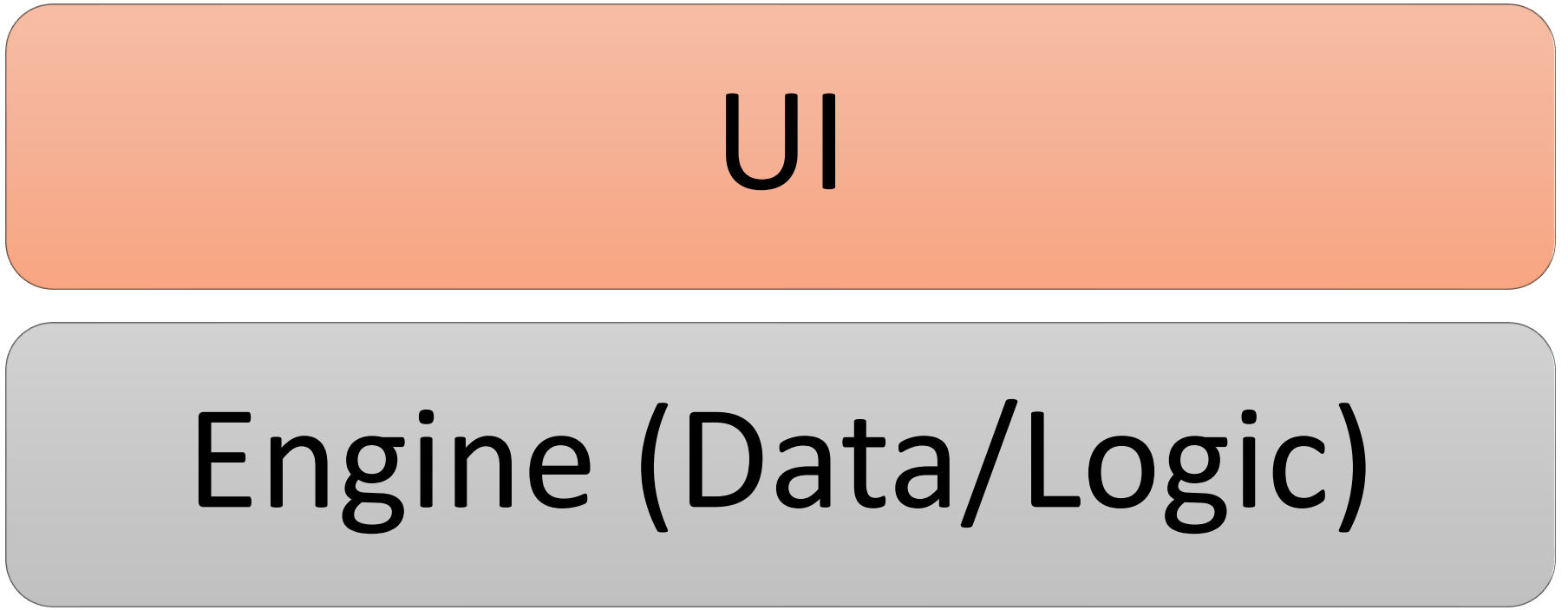
Spreadsheet Application Design Overview

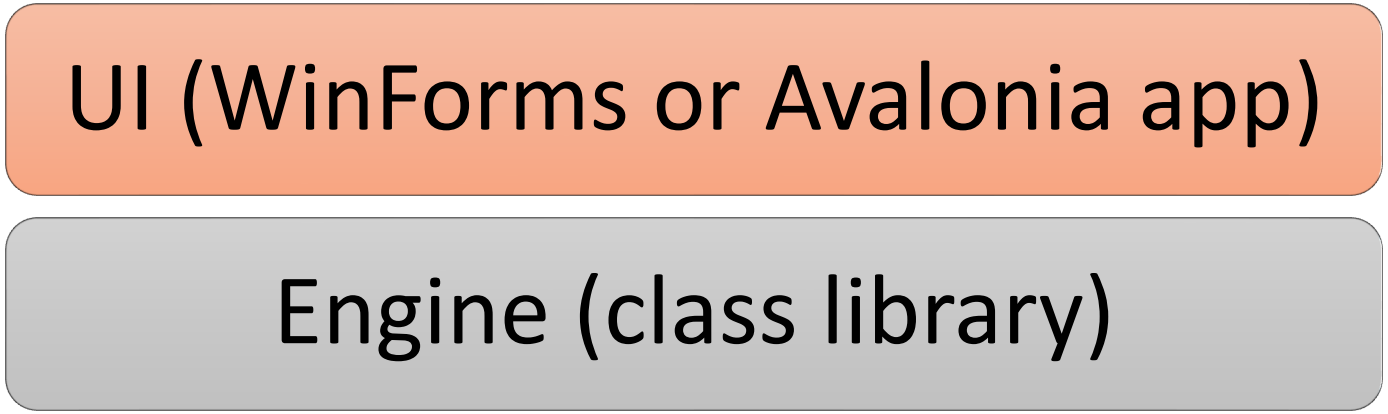
Cpt S 321

Washington State University

Recall: Decouple Logic/Data and UI

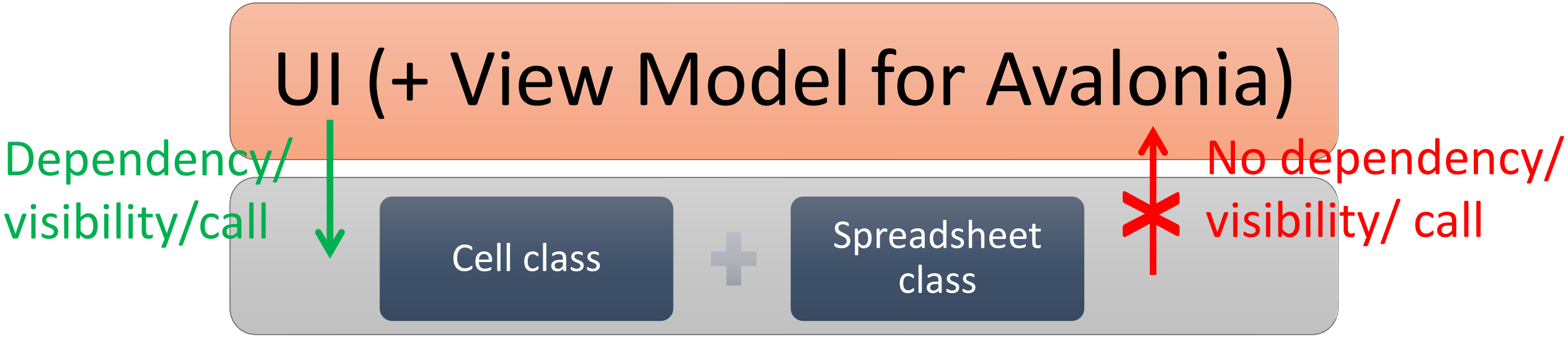


Spreadsheet Design



The spreadsheet solution consists of a WinForms or Avalonia project for the UI, a DLL class library project for the engine, and of course a Test project to test the functionality.

Spreadsheet Design



- Inside the logic engine are classes for **Cell** and **Spreadsheet**.
- Changes in the **Spreadsheet** will bubble-up to the UI through property-change notifications. UI will update appropriately.

Follow the **Model-View Separation Principle**

- States that: Model (domain) objects should not have direct knowledge of view (UI) objects.
- Thus, in the Spreadsheet application the Cell and Spreadsheet classes will not directly reference neither send messages to the UI classes such as the Form and the View Model.
- Also, the UI classes will not contain any domain logic (such as formula calculation). UI objects should only initialize UI elements, receive UI events (such as a mouse click on a button), and delegate requests for application logic on to non-UI objects (such as domain objects).

Spreadsheet Engine Design

- **Cell** class: *members*
  - Text (public string property)
    - Represents the exact text that was entered into the cell.
  - Value (public string property)
    - Represents the evaluated value of the cell. If the cell has a formula this will be the result of computing that formula. Otherwise, it has the same value as the **Text** property (hence the string type)
    - Read-only to the world outside of the **Spreadsheet** class (Why?)
  - (more properties will come later with the introduction of new features)

Spreadsheet Engine Design

- **Cell** class: *purpose*
  - Store the text and value for a single cell
    - Protect **Value** property from being set by anything other than the **Spreadsheet** class
      - You will need to figure out how to use C# language features to achieve this on the homework
  - Provide the ability to notify of property changes
    - Cell class will implement INotifyPropertyChanged interface
    - When **Text** or **Value** changes, invoke notification event

Spreadsheet Engine Design

- **Spreadsheet** class: *members*
  - Cell GetCell(int columnIndex, int rowIndex)
    - Method that returns the cell at the specified column and row indices
    - Indices are zero-based
  - Cell GetCell(string cellName)
    - Method that returns the cell by name (e.g. “A1”)
    - Can be implemented by parsing the string and calling the other overload

Spreadsheet Engine Design

- **Spreadsheet** class: *purpose*
  - Store the 2D array of cells in the spreadsheet
  - Allow observers to subscribe to change events and provide notifications of any change in the spreadsheet
    - Subscribe to change events in all internally managed cells
    - When they change, forward the event to observers of the spreadsheet CellPropertyChanged event

Important: Follow Good Design Principles/Patterns!

- Follow the **Model-View Separation Principle / MVVM**

- Promote encapsulation (i.e., set the proper access modifiers to your entities)

- You are expected to use all principles/patterns
  1. That apply for the current HW and
  2. That we have seen in class

General Responsibility Assignment Software Patterns/Principles: GRASP

- Guidelines for assigning responsibility to classes and objects in object-oriented design
- Guided by one main question "*Whose responsibility is it?*"
- Nine principles and *we are already familiar with some of them*:
  - **Information Expert:** Assign the right responsibilities to the right entities (i.e., to the right classes, methods, etc.).
    Problem: What is a basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfill it.

GRASP (cont.)

- **Low Coupling**
  - Problem: How to reduce the impact of change?
  - Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
- **High Cohesion**
  - Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - Solution: Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.
- **Polymorphism**
  - Problem: How handle alternatives based on type?
  - Solution: When related alternatives or behaviors vary by type, assign responsibility for the behavior using polymorphic operations to the types for which the behavior varies (instead of explicit branching based on type).