Week 5

Last Week

1. Answer any questions from last week that your lecture section didn't cover related to refactoring, exceptions, and packages.
   I did not have any questions last week...
2. Which packaging strategy does the J-Shell case study code appear to be using?
   By component.

Clean Architecture Recap

3. What is a 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.

Design Patterns

4. What is a design pattern?
   - 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.
5. For each of the design patterns covered during lecture, explain:
   i. What problem it solves,
   ii. How the pattern works (including the roles of each class), and
   iii. How the pattern solves the problem from (i).

- ITERATOR
  i. Problem: Want a way to iterate over the elements of the container. Want to have multiple, independent iterators over the elements of the container. Do not want to expose the underlying representation: should not reveal how the elements are stored.
  ii. How the pattern works: Create a pair of two classes: an IteratorClass which implements the IteratorInterface, and an IterableClass which implements the IterableInterface. As defined by the IteratorInterface, the IteratorClass should provide a method to tell whether there is a next element

and another method to return the next element. As defined by the IterableInterface, the IterableClass should provide a method to return an iterator of the IterableClass.

iii. How the pattern solves the problem from (i): No underlying container is defined and exposed by this approach. The elements can be iterated by first checking whether there is a next element then can be fetched by the next() method. Each iterator is independent and points to different "next" elements.

- OBSERVER
  i. Problem: Need to maintain consistency between related objects. Two aspects, one dependent on the other. An object should be able to notify other objects without making assumptions about who these objects are.
  ii. How the pattern works: Create an Observable class which maintains a list of Observers. Whenever there is a change, update every observer and clear the "changed" status.
  iii. How the pattern solves the problem from (i): Consistency is maintained in the objects when the related Observer are notified and updates the objects.
- STRATEGY
  i. Problem: Multiple classes that differ only in their behaviour (for example, use different versions of an algorithm), but the various algorithms should not be implemented within the class want the implementation of the class to be independent of a particular implementation of an algorithm. The algorithms could be used by other classes, in a different context. Want to decouple — separate — the implementation of the class from the implementation of the algorithms.
  ii. How the pattern works: Within a context class, store an instance of a class that implements a specific strategy.
  iii. How the pattern solves the problem from (i): Under different contexts, we can use different strategies as defined by the Strategy instance in the context class.
- DEPENDENCY INJECTION
  i. Problem: We are writing a class, and we need to assign values to the instance variables, but we don’t want to hard-code the types of the values. Instead, we want to allow subclasses as well.
  ii. How the pattern works: If a class ever needs to maintain a list of objects in another types, create those instances outside any of the class methods then store those instances directly using one of the class methods.
  iii. How the pattern solves the problem from (i): The pattern already describes how it solves the problem...
- SIMPLE FACTORY
  i. Problem: One class wants to interact with many possible related objects. We want to obscure the creation process for these related objects. At a later date, we might want to change the types of the objects we are creating.
  ii. How the pattern works: Define an interface that enforces a method to be implemented by the related classes. In the Factory class, we can get such objects and call the method on different types of the related classes.
  iii. How the pattern solves the problem from (i): The pattern already describes how it solves the problem...
- FAÇADE
  i. Problem: A single class is responsible to multiple “actors”. We want to encapsulate the code that interacts with individual actors. We want a simplified interface to a more complex subsystem.

ii. How the pattern works: Create individual classes that each interact with only one actor. Create a Façade class that has (roughly) the same responsibilities as the original class. Delegate each responsibility to the individual classes. This means a Façade object contains references to each individual class.
iii. How the pattern solves the problem from (i): The pattern already describes how it solves the problem...

- BUILDER
  i. Problem: Need to create a complex structure of objects in a step-by-step fashion
  ii. How the pattern works: Create a Builder object that creates the complex structure.
  iii. How the pattern solves the problem from (i): The pattern already describes how it solves the problem...