Lecture Notes for Week 7
========================

You can refer to the material from lecture, the Week 7 Slides and Code on Quercus, the following website, or any online source that you find to be helpful.
https://sourcemaking.com/design_patterns

+ Object Oriented Design and Clean Architecture

1. Why is it useful to think of design as a decision-making process, as opposed to a feature? Give at least three reasons.

If you try to get a "perfect" design, you may take too long before actually trying to implement it.
A "perfect" design may not work in practice, so you can't blindly follow it when coding.
Design patterns are based of a problem in code that already exists, thus giving more reason to keep designing as you go.

2. Why might you want to separate some of your Use Case classes from each other? Why might you want to separate your Controller code from your Presenter code?

Reusability, as you cannot reuse code from a class if its calling methods from another class that you did not also copy.

You may want to display different things based on user options, but you want the functionality of the content to stay constant. (Ex. dark mode)

3. How can you decide on what to name your packages?

Package names are very flexible, but they should be named in a way that tells the developer exactly the type of classes are in it (clean arch. layers, program functionality, etc.)

4. Describe two things you can do to a class A to lessen a dependency on class B. (Hint: one is a design pattern, the other is polymorphism)

Dependency injection
Using an interface

An example of using polymorphism to lessen a dependency:

```
for(Drawable d : shapes)
  d.draw();
```

The above code is independent of the Rectangle and Circle classes, even those both classes implement Drawable.

+ Design Patterns

5. What are some of the goals that design patterns help you work towards? (Example: better encapsulation)

6. For each of the design patterns that we cover this week, describe:
(i) What problem does this solve?
(ii) How do we implement the pattern? In other words, how does the anti-pattern (the "before" code) differ from the pattern (the "after" code)?
(iii) What is the benefit of implementing the design pattern?

Here are the design patterns we are covering in Weeks 6, 7, and 8:
Dependency Injection, Iterator, Observer, Strategy, Factory Method, Abstract Factory, Adapter, Builder, Façade,

Dependency Injection
i) When assigning values to instances variables, we don't want to create the values in the class
ii) Create the content outside of the class and have the constructor or some method take it as a parameter

iii) Good for extensability, as subclasses can inherit the methods to use aswell

Factory
i) We want to interact with a set of object types that are related
ii) Use a class to create instances of an object by deciding a constructor call in the seperate class.
iii) Open/Closed and extendable, as we can add other objects to it and maybe change the types of objects

Abstract Factory
i) Like a factory, but is completely independent of the objects it creates and has another layer of classes
ii) Create multiple factory classes for every set of related objects
iii) Good for abstraction, as the objects being created are not related to the abstract factory at all

Façade
i) A class has too many responsibilities
ii) delegate responsibilities to other classes and use them inside the façade
iii) Good for encapsulation, single responsibility

Builder
i) There is a complex interaction of objects that need to be instantiated.
ii) Use a class to create the objects
iii) Good for single responsibility and abstraction

7. For each of the previous design patterns, does your project design have the conditions for implementing the design pattern? Why or why not?

Dependency Injection: We may have a class to store account credentials, and we need to save those to a file. So the class can take in a gateway to call saving to a file
Factory: There are different types of users, so different information may be needed for the constructor parameters. A factory class could be used to create user accounts
Abstract Factory: The UI may be different for different types of user, so a abstract factory could create different UI generators for different user types
Façade: A class that has to store user info may have to interact with many classes because users can do many things.
Builder: If as now, there doesn't seem to be a complex task to complete, so there isn't a use for this pattern