Design Principles. The Layered Architecture pattern

Assoc. Prof. Arthur Molnar

Babeș-Bolyai University

2024

Overview

1. Design principles for modular programs
   - Single Responsibility Principle
   - Separation of Concerns
   - Dependency
   - Layered Architecture

Organizing source code

What does it mean to organize source code?

- Determine what code goes where ... d'uh!
- We split code into functions, classes and modules
- The purpose of this section is to discuss a few principles that help us do it correctly

Modules

Discussion

What do we mean by **organizing the code correctly?**

Organizing source code

We use a few key design principles that help determine how to organize source code

- Single responsibility principle
- Separation of concerns
- Dependency
- Coupling and cohesion

Single responsibility principle

- Each function should be responsible for one thing
- Each class should represent one entity
- Each module should address one aspect of the application

Single responsibility principle - functions

Let's take the function below as example

- Implements user interaction
- Implements computation
- Prints

```
1     def filter_score(scoreList : list) -> None:
2       st = input("Start score:")
3       end = input("End score:")
4       for score in scoreList:
5         if score.get_value() > st and score.get_value() < end:
6           print(score)
```

Single responsibility principle - functions

Why could the **filter_score** function change?

- The program's input format or channel changes
  - e.g. menu/command based UI as in Assignment 5/6
  - How about GUI/web/mobile/voice-based UI?
- The filter has to be updated

NB!

The **filter_score** function has 2 responsibilities

Single responsibility principle - modules

How did we characterize a module?

[modules] ... each of which accomplishes one aspect within the program and contains everything necessary to accomplish this.

Single responsibility principle - modules

Discussion

Is there any similarity between how we design a function and a module?

Single responsibility principle

Multiple responsibilities are...

- Harder to understand and use
- Difficult/impossible to test
- Difficult/impossible to reuse
- Difficult to maintain and evolve

Separation of concerns

- Separate the program into distinct sections
- Each section addresses a particular concern
- **Concern** - information that affects the code of a computer program (e.g. computer hardware that runs the program, requirements, function and module names)
- Correctly implemented, leads to a program that is easy to test and from which parts can be reused

Separation of concerns - example

Let's take the function below as example **(again!)**

```
1     def filter_score(scoreList : list) -> None:
2       st = input("Start score:")
3       end = input("End score:")
4       for score in scoreList:
5         if score.get_value() > st and score.get_value() < end:
6           print(score)
```

Separation of concerns - the UI

The refactored function below only addresses the UI, functionalities are delegated to the **filter_score** function

```
1     def filter_score_ui(scoreList : list) -> None:
2       st = input("Start score:")
3       end = input("End score:")
4       result = filter_score(scoreList,st,end)
5       for score in result:
6         print(score)
```

Separation of concerns - the test

The **filter_score** function can be tested using a testing function such as the one below

```
1     def test_filter_score():
2       lst = [person("Ana",100)]
3       assert filter_score(l,10,30)==[]
4       assert filter_score(l,1,30)==lst
5       lst = [person("Anna",100), person("Ion",40), person("P",60)]
6       assert filter_score(lst,3,50)==[person("Ion",40)]
```

Separation of concerns - the operation

The **filter_score** function only has one responsibility!

```
1     def filter_score(lst : list, st : int, end : int) -> int:
2       '''
3       Filter participants
4       lst - list of participants
5       st, end - integer scores
6       return list of participants filtered by score
7       '''
8       rez = []
9       for p in lst:
10        if p.get_score() > st and p.get_score() < end:
11          rez.append(p)
12      return rez
```

Separation of concerns

NB!

These design principles are in many cases interwoven!

Dependency

What is a **dependency**?

- Function level - a function invokes another function
- Class level - a class method invokes a method of another class
- Module level - any function from one module invokes a function from another module

Example

Say we have functions **a**,**b**,**c** and **d**. **a** calls **b**, **b** calls **c** and **c** calls **d**. What might happen if we change function **d** ?

Coupling

- **Coupling** - a measure of how strongly one element is connected to, has knowledge of, or relies on other elements
- More connections between a module and others make that module harder to understand, harder to re-use, harder to test and isolate any failures
- **Low coupling** - facilitates the development of programs that can handle change because they minimize the dependency between functions/modules

Cohesion

- **Cohesion** - a measure of how strongly related and focused the responsibilities of an element are.
- A module may have:
  - **High Cohesion**: it is designed around a set of related functions
  - **Low Cohesion**: it is designed around a set of unrelated functions
- A cohesive module performs a single task within an application, requiring little interaction with code from other parts of the program.

Cohesion

- Modules with less tightly bound internal elements are more difficult to understand
- Higher cohesion is better

NB!

Cohesion is a more general concept than the single responsibility principle, but modules that follow the SRP tend to have high cohesion.

Cohesion

NB!

Simply put, a cohesive module should do just one thing - **now where have I heard that before... ?**

How to apply these design principles

- **Separate concerns** - divide the program into distinct sections, so that each addresses a separate concern
- Make sure the modules are **cohesive** and **loosely coupled**
- Make sure that each module, class have **one responsibility**, or that there is only one reason for change

Layered Architecture

Layered Architecture

We employ the layered architecture pattern keeping in mind the previously discussed detailed design principles

Structure the application to

- **Minimize module coupling** - modules don't know much about one another => makes future change easier
- **Maximize module cohesion** - each module consists of strongly inter-related code => avoid creating dependencies between different parts of the application
- **Dependencies flow from the higher to the lower layer**, and never in reverse => helps avoid spaghetti code

Layered Architecture

Layered Architecture - an architectural pattern that allows you to design flexible systems using components

- Each layer communicates only with the layer immediately below
- Each layer has a well-defined interface used by the layer immediately above (hide implementation details)
- We provide the object required by each layer as a parameter of the constructor, as this allows us to change the object with a different implementation later (e.g., file- vs SQL-repository).

Layered Architecture

Common layers in the architecture of an information system

- **User interface, Presentation** - user interface related functions, classes, modules
- **Domain, Application Logic** - provide application functions determined by the use-cases
- **Infrastructure** - general, utility functions or modules
- **Application coordinator** - start and stop application, instantiate components

Layered Architecture

Demo

Take a look at the example code in lecture.examples.ex36_rational_calc

Exceptions and layered architecture

Following the rules below allows your programs to raise exceptions where they happen (and where the problem can be described) and be caught where we can do something about it (tell the user, usually)

How to integrate exceptions into layered architecture programs

- UI module(s) should not do a lot of processing
- Non-UI modules must not have any UI input/output
- Exceptions are created (and raised) where the problem takes place (e.g., RepositoryError in a file-based repository)
- Exceptions can traverse program layers (e.g., Service-level classes) and must be caught at the UI level