Design Principles for Modular Programs

Lect. PhD. Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

Overview

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

```
#Function with multiple responsibilities
#implement user interaction (read/print)
#implement a computation (filter)
def filterScore():
    st = input("Start score:")
    end = input("End score:")
    for c in l:
        if c[1]>st and c[1]<end:
            print c
```

Single responsibility principle - functions

Why could the **filterScore()** function change?

- The program's input format or channel changes
  - e.g. command/menu based UI as in Lab2-4
  - How about GUI/web/mobile/voice-based UI?
- The filter has to be updated

NB!

The **filterScore()** 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!)**

```
#Function with multiple responsibilities
#implement user interaction (read/print)
#implement a computation (filter)
def filterScore():
    st = input("Start score:")
    end = input("End score:")
    for c in l:
        if c[1]>st and c[1]<end:
            print c
```

Separation of concerns - the UI

The refactored function below only addresses the UI, functionalities are delegated to the **filterScore()** function

```
def filterScoreUI():
    st = input("Start sc:")
    end = input("End sc:")
    rez = filtrareScore(l,st, end)
    for e in rez:
        print e
```

Separation of concerns - the test

The **filterScore()** function can be tested using a testing function such as the one below

```
def testScore():
  l = [["Ana", 100]]
  assert filterScore(l,10,30)==[]
  assert filterScore(l,1,30)==l
  l = [["Ana", 100],["Ion", 40],["P", 60]]
 assert filterScore(l,3,50)==[["Ion", 40]]
```

Separation of concerns - the operation

The **filterScore()** function only has one responsibility!

```
def filterScore(l,st, end):
    """
    filter participants
    l - list of participants
    st, end - integers -scores
    return list of participants
        filtered by st end score
    """
    rez = []
    for p in l:
        if p[1]>st and p[1]<end:
            rez.append(p)
    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 one module and others, the harder to understand that module, the harder to re-use it in another situation, the harder to test it and isolate failures
- **Low coupling** - facilitates the development of programs that can handle change because they minimize the interdependency 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 a software, requiring little interaction with code from other parts of a 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

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

Layered Architecture

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

Layered Architecture

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

- Each layer communicates only with the one immediately below
- Each layer has a well-defined interface used by the layer immediately above (hide implementation details)

Layered Architecture

Common layers in an information system architecture

- **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

A first example - Lab2-4

Layered architecture in Lab2-4

- **User interface** - Functions related to the user interaction. Contains input and data validation, print operations. This is the only module where input/print operations are present.
- **Functions** - Contains the functions required to implement program features
- **Application coordinator** - Initialize the UI and start the application.