Design guidelines in large scale programming

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

October 21, 2015

Overview

Responsibility

Responsibility - is a reason to change

- Function - do a computation
- Module - all the functions responsibilities

Single responsibility principle (SRP)

A function/module should have one, and only one, reason to change.

```
#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
```

What are the multiple responsibilities above?

Single responsibility principle (SRP)

Multiple responsibilities

- Harder to understand and use
- Unable to test
- Unable to reuse
- Difficult to maintain and evolve

Separation of concerns - UI part

The process of separating a computer program into distinct features that overlap in functionality as little as possible

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

Separation of concerns - Test

The process of separating a computer program into distinct features that overlap in functionality as little as possible

```
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 - function implementation

The process of separating a computer program into distinct features that overlap in functionality as little as possible

```
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
```

Dependency

What is **dependency**?

- Function - a function invokes another function
- Module - any function from the module invoke a function from another module

In order to increase reusability we need to manage dependency

Coupling

Coupling - a measure of how strongly one element is connected to, has knowledge of, or relies on other elements
The more the connections between one module and the rest, the harder to understand that module, the harder to re-use that module in another situation, the harder it is to isolate failures caused by faults in the module
$=>$ The lower the coupling the better
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 procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing

Cohesion

The less tightly bound the internal elements, the more disparate the parts to the module, the harder it is to understand
$=>$ The higher the cohesion the better
Cohesion is a more general concept than the SRP, modules that are follow the SRP tend to have high cohesion.

Layered Architecture

Structure the application such that:

- Minimizes coupling between modules (modules don't need to know much about one another to interact, makes future change easier)
- Maximizes the cohesion of each module (the contents of each module are strongly inter-related)

Layered Architecture

Layered Architecture - is an architectural pattern that allows you to design flexible systems using components (The components are as independent of each other as possible)

- Each layer communicates only with the layer immediately below it.
- Each layer has a well-defined interface used by the layer immediately above. (implementation details are hidden)

Layered Architecture

Common layers in an information system logical architecture

- **User interface/Presentation** (User interface related functions/modules/classes)
- **Domain/Application Logic** (provide the application functions determined by the use-cases)
- **Infrastructure** (general/utility functions/modules/classes)
- **Application coordinator**

Layered Architecture - simple example


```
#Ui
def filterScoreUI():                                    #manage the user interaction
    st = input("Start sc:")
    end = input("End sc:")
    rez = filterScoreDomain(st, end)
    for e in rez:
        print e
```

```
#domain
l = [["Ion",50],["Ana",30],["Pop",100]]
def filterScoreDomain(st, end):                         #filter the score board
    global l
    rez = filterMatrix(l, 1, st, end)
    l = rez
    return rez
```

```
#Utility function - infrastructure
def filterMatrix(matrice, col, st, end):                #filter matrix lines
    linii = []
    for linie in matrice:
        if linie[col]>st and linie[col]<end:
            linii.append(linie)
    return linii
```

Layered Architecture - organizing projects