CS455 Software Engineering

Inversion of Control

This sample is written in python 3.2 and depends on pip3

Inversion of Control (IoC) is used to increase modularity of the code and make it extensible. IoC for this sample is implemented as Dependency Injection (DI). We'll essentially decouple the dependencies between classes through a shared class called container.

“dependency_injector” is offered by python which will be used for functionalities offered by a container.

Use-Case Description

There are two essential classes Products and Operations, naturally the class Operations depends on Products and offers the possibility of different operations on a List of products (in this case all types of a particular product).

Class Products implements two basic functionality ”add” an item and “getList” returns all the items.

Class Operations implements “isProduct”, an object of this class needs an instance of Products (Hence the dependency).

Injecting Dependency

The idea is to take given classes and resolve the dependencies by hardcoding(injecting) the dependencies in the dependent class and writing a container on top of it which provides an object giving all the functionalities of the initial class.

Container

It is very trivial to observe that creation of the dependent class (Operations here) requires additional code for specification of dependencies. This leads to a hurdle which can solved by IOC containers. Container keeps track of implemented dependencies and injects them into an object on creation.

Traditional Code Fragment

```
class Product:
    __productList = []
    def __init__(self):
        pass

    def add(self,productName,price):
        self.__productList.append({'name':productName,'price':price})

    def getList(self):
        return self.__productList;

class Operations:
    def isProduct(self, product, productName):
        allproduct = product.getList();
        for it in allproduct[:]:
            if it['name'] != productName:
                allproduct.remove(it)
        return allproduct;

def main():
    flip = Product()
    flip.add("TV",1050)
    flip.add("TV",1150)
    flip.add("Fridge",10050)
    flip.add("Fridge",50000)
    flip.add("Fridge",10000)

    op = Operations();
    out = op.isProduct(flip,"Fridge")
    print(out)

if __name__ == "__main__": main()
```

IOC Code Fragment

main.py products.py procedural.py operations.

```
class Product:
    __productList = []
    def __init__(self):
        pass

    def add(self,productName,price):
        self.__productList.append({'name':productName,'price':price})

    def getList(self):
        return self.__productList;
```

main.py products.py procedural.py

```
class Operations:
    def __init__(self, product):
        self.__product = product # Injecting Dependency

    def isProduct(self, productName):
        allproduct = (self.__product).getList();
        for it in allproduct[:]:
            if it['name'] != productName:
                allproduct.remove(it)
        return allproduct;
```

main.py products.py procedural.py operations.py

```
import products as products
import operations as operations

import dependency_injector.containers as containers
import dependency_injector.providers as providers

class Products(containers.DeclarativeContainer):
    product = providers.Factory(products.Product)

class Operations(containers.DeclarativeContainer):
    operation = providers.Factory(operations.Operations, product=Products.product)

def main():
    flip = Products.product()
    flip.add("TV",1050)
    flip.add("TV",1150)
    flip.add("Fridge",10050)
    flip.add("Fridge",50000)
    flip.add("Fridge",10000)

    op = Operations.operation();
    out = op.isProduct("TV")
    print(out)

if __name__ == "__main__": main()
```

Advantages IoC

to take given classes and resolve the dependencies by hardcoding(injecting) the dependencies in the depende

1. During implementation we can focus on a particular class independent of the implementation of the depending class.
2. The framework (here Operations class) and the client (class Product) behave as if exclusive hence it makes easier if one wants to change the depending class to a different kind of calling method. (Note: as python is not type sensitive the problem is tackled fairly easily, but suppose if it were JAVA then an addition “interface” function is required to restrict the type of objects passed to dependent class)
3. As the size of the Use-Case in terms of Lines of Code and Classes, regulating using a Container and Dependency Injection is much more easy for testing purpose without and highly accepting to changes. As, we do not need to compile the entire code-base again, not even for dependent classes as it is taken care on run-time credit to Dependency Injection.