**Name:** Karthik Palavalli

**Title:** Insurance Company

**Project Summary:** A portal for customers to buy insurance plans. A customer can buy an insurance plan based on his own research, or could choose to get in touch with an expert in the domain, sales representative, who would then guide him with the purchase. There are broadly two types of insurance plans - Health Plan and Life Plan. The health plan deals with protecting the insured when he/she is ill. While the life plan is more concerned with providing the nominee of the insured with a monetary benefit in a scenario when the insured passes away. Both these plans have add-ons which is achieved using a decorator pattern in the current implementation.

There are in total three actors of the system. At the very top is the admin, who has super user privileges i.e., he can modify users and plan details across the whole application. Followed by a sales representative, whose goal is to help the customer with the right insurance plan. And finally the customer buys the plan.

**Features Implemented:**

| ID | Feature |
|---|---|
| ID002 | Add new users |
| ID003 | Remove users |
| ID004 | Add new plans |
| ID005 | Remove old plans |
| ID007 | Can view all plans |
| ID008 | Can view all pending appointments |
| ID010 | View current plan |
| ID011 | Buy a plan |
| ID012 | Raise appointment request |

**Features *Not* Implemented:**

| ID | Feature |
|---|---|
| ID001, ID006, ID009 | Login |
| Stretch Functionality 1 | Discount options based on customer type |
| Stretch Functionality 2 | Compare Plans |

**Class Diagram:**

The class diagram has evolved during the various stages of the project. The reasons for it being both functional and non-functional requirements. I have also accounted for the suggestions made in the Part 2 evaluation of this project. The following updates are found in the new class diagram:

1. Better structure across the insurance plans - Decorator and Composite:
    a. While the original class diagram treated every insurance plan as a single object. The new design now brings more structure in the way insurance plans are organized. They are now closer to the real word scenario of the insurance plans. I have taken inspiration from one of the leading insurance providers - HDFC Life (https://www.hdfclife.com/), in order to come up with this structure.
    b. Use of decorator - As discussed before there fundamentally exists two classes of plans, health and life. But while the insurance application form is being filled out, the customer may choose to avail additional benefits. For example, customer Jolly may choose a health plan. Health plan covers for some of the basic illnesses such as: flu, fever, cough etc. Before submitting he sees a set of options to add coverage for more illnesses (e.g. cancer, cardiac problems). On seeing these options he decides to add cancer support as well. This is a pattern of “Addition of responsibilities during runtime” hence decorator design pattern seemed to fit.
    c. Use of composite - Many insurance companies provide a feature of combining/ co-buying insurance plans at a discounted price. These plans are called group plans. This calls for the “grouping” design pattern, hence composite seemed apt.

2. Data access protection - Proxy:
    a. I have used the Proxy pattern to protect illegal access of personal data. In this case a customer cannot access plan details of different customers. He/she can view only their plan details.

## Design Patterns:

## 1- Decorator pattern

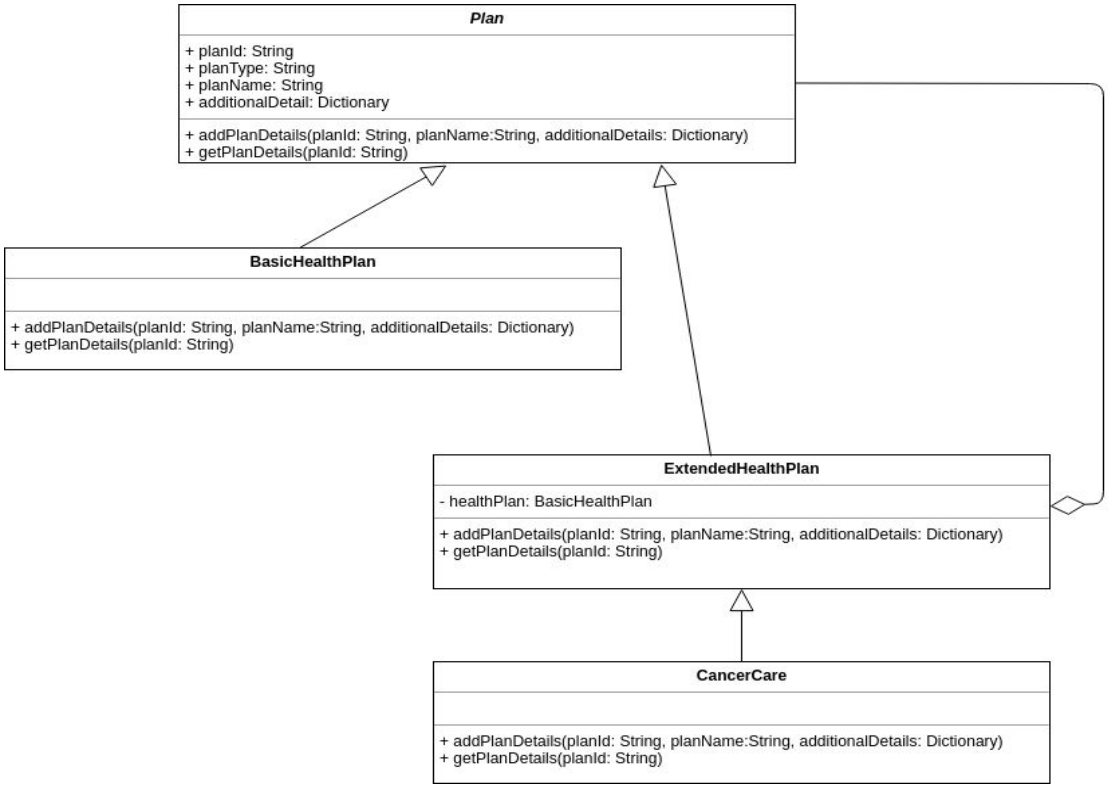



Fig 1. Decorator pattern used in insurance company

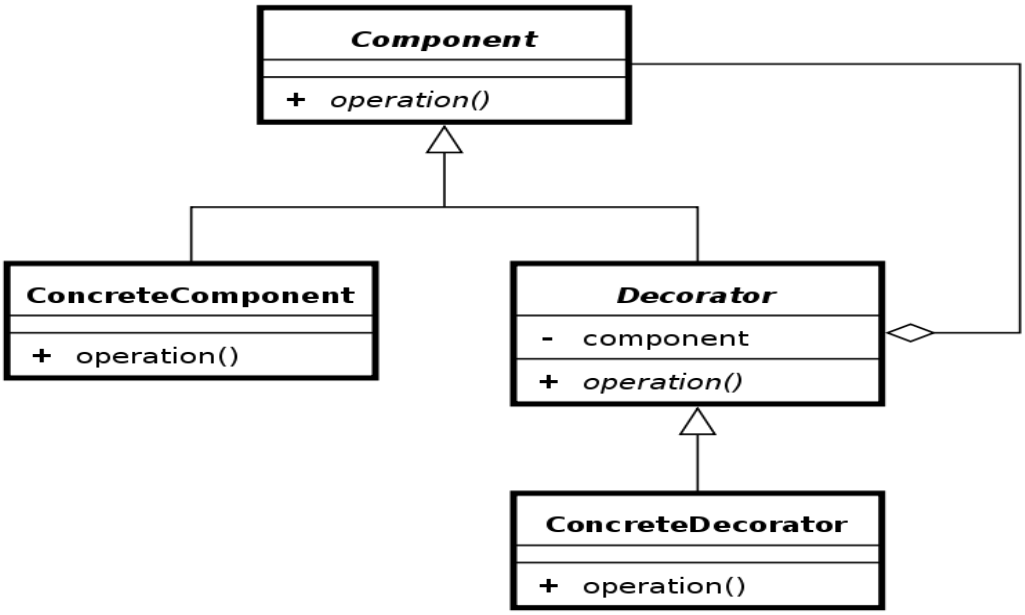



Fig 2. Decorator pattern according to Gang of Four

**Implementation details:**

**Classes involved:** Plan, BasicHealthPlan, ExtendedHealthPlan, CancerCare, CardiacCare, BasicLifePlan, ExtendedLifePlan, ULIPBenefits

Plan - Abstract class extensions of which form various types of plans.
BasicHealthPlan - Contains attributes and values of a basic health plan.
ExtendedHealthPlan - Decorator class, used to take to take the basic health plan and dynamically attach other benefits as per the user needs.
CancerCare - Concrete extended plan, storing benefits of a basic plan plus benefits related to cancer related illnesses.
CardiacCare - Concrete extended plan, storing benefits of a basic plan plus benefits related to cardiac related illnesses.

BasicLifePlan - Contains attributes and values of a basic life plan.
ExtendedLifePlan - Decorator class, used to take to take the basic life plan and dynamically attach other benefits as per the user needs.
ULIPBenefits - Concrete extended life plan, storing benefits of a basic plan plus benefits related interest rate.

**Reason for choosing decorator pattern:**
As described above, the customer after choosing a basic pack, is given the flexibility of adding extra benefits in the same application. This is analogous to adding toppings to a pizza during an order. Since all of this need to happen during runtime, decorator seemed to fit.

## 2 - Composite

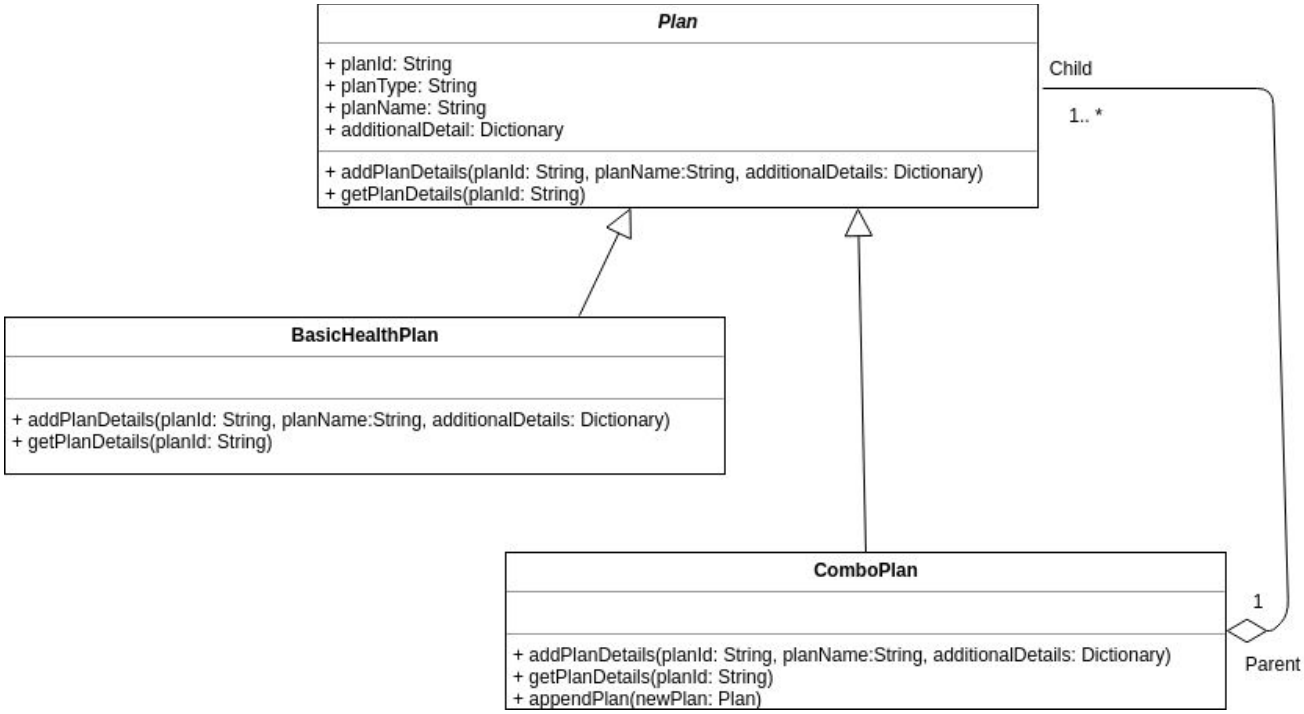



Fig 3. Composite pattern in insurance company

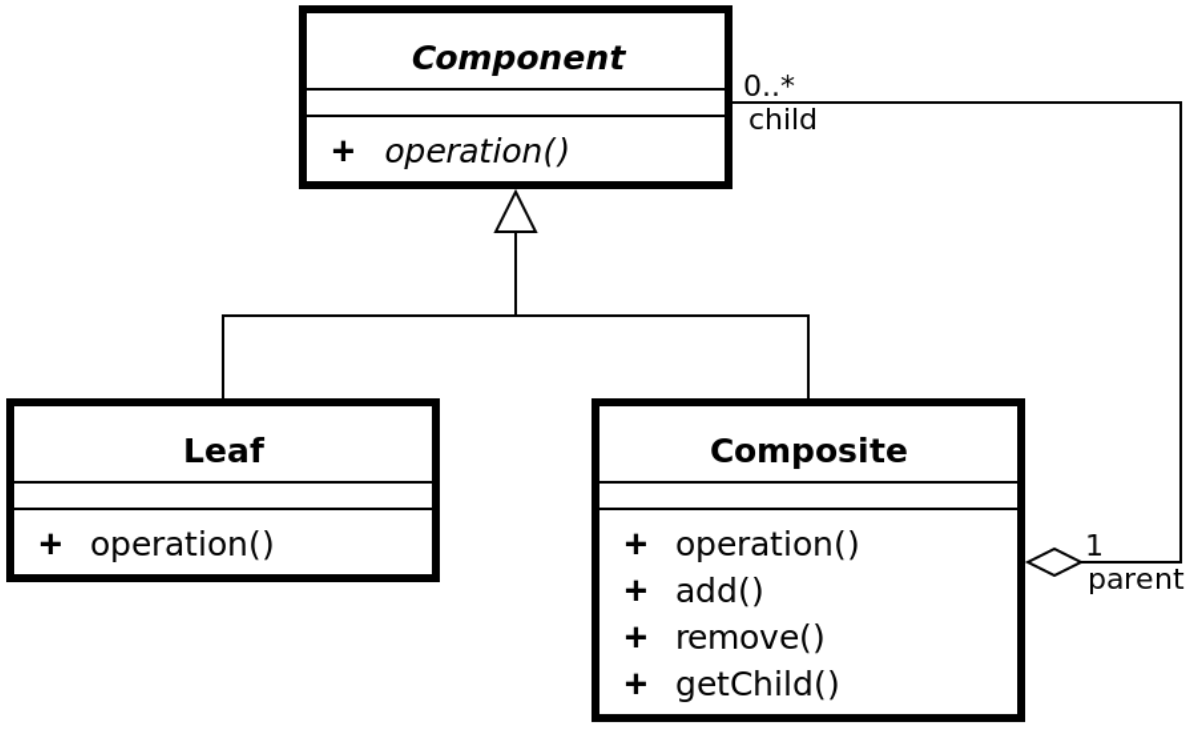



Fig 4. Composite pattern according to Gang of Four

**Implementation details:**

**Classes involved:** Plan, BasicHealthPlan, BasicLifePlan, ComboPlan

Plan - Abstract class extensions of which form various types of plans.
BasicHealthPlan - Leaf class containing the attributes and values for basic health plan.
BasicLifePlan - Leaf class containing the attributes and values for basic life plan.
ComboPlan - A composite class containing one or more types of leaves, in this case either the BasicHealthPlan or BasicLifePlan.

**Reason for choosing composite pattern:**
Many insurance providers provide the option of combining purchases across the classes of plans. This creates a win-win situation, the insurance company gets more sales and the customer benefits from additional discounts. Here we observe a pattern of ‘grouping’ items together which suggests that a object is in the form of “composition” of other objects. Hence choosing the composite pattern seemed apt.

## 3 - Proxy

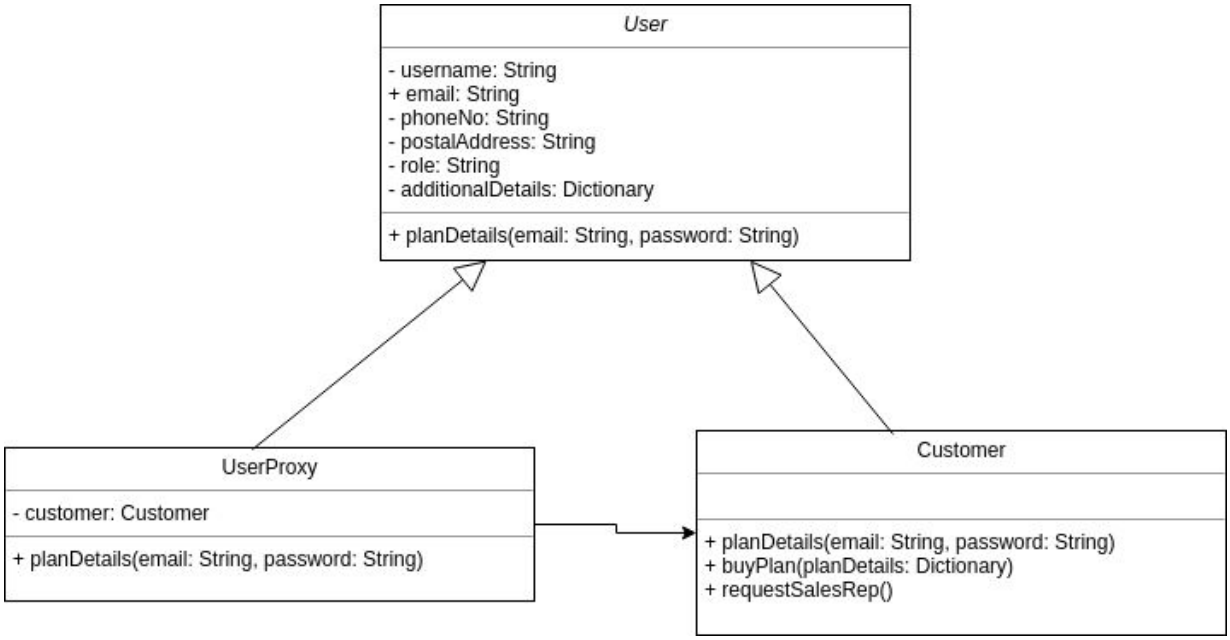



Fig 5. Proxy pattern in insurance company

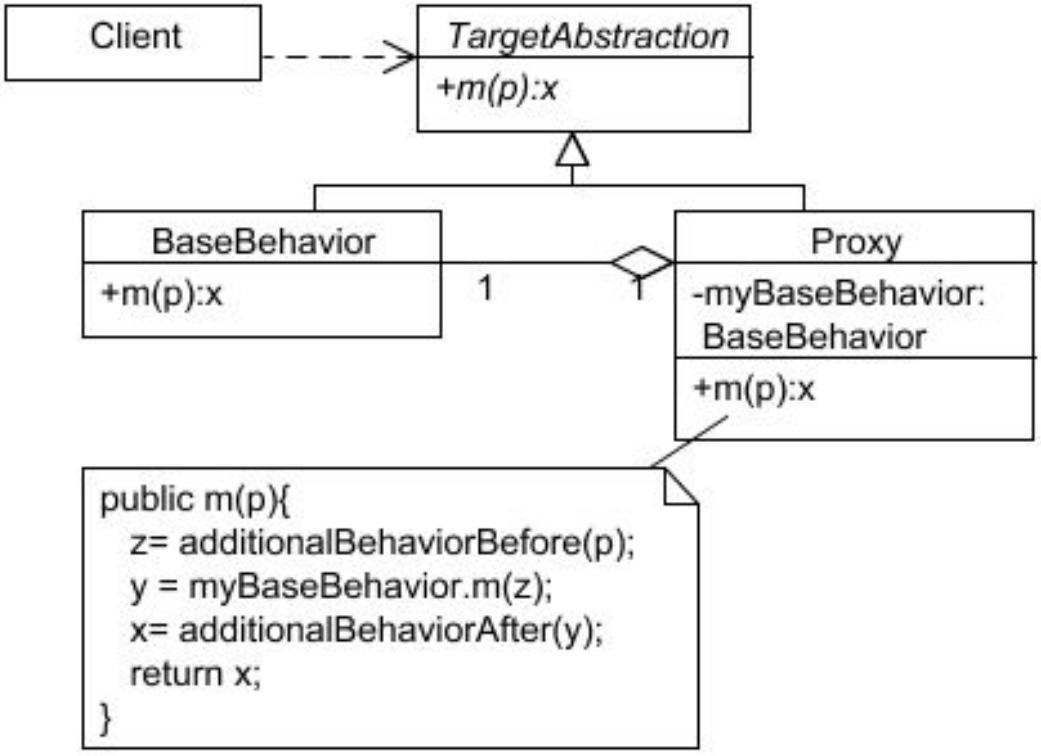



Fig 6. Proxy design pattern

**Implementation details:**

**Classes involved:** User, UserProxy, Customer

User - Contains abstract attributes and methods related to user
Customer - Concrete class used to represent the customer of the insurance company. Contains the attributes and values related to the customer
UserProxy - User proxy class, used to check if the details requested by the customer belongs to him/her. If not raises an “Unauthorized access” error. Else return the active plan details of the customer

**Reason for choosing proxy pattern:**
The plan details of a customer is confidential. While credential check such as password is good, it’s also a good practise to provide extra layer of isolation using a pattern. In this case we need protection of access across different customer objects, hence a protection proxy seemed suited for the job.

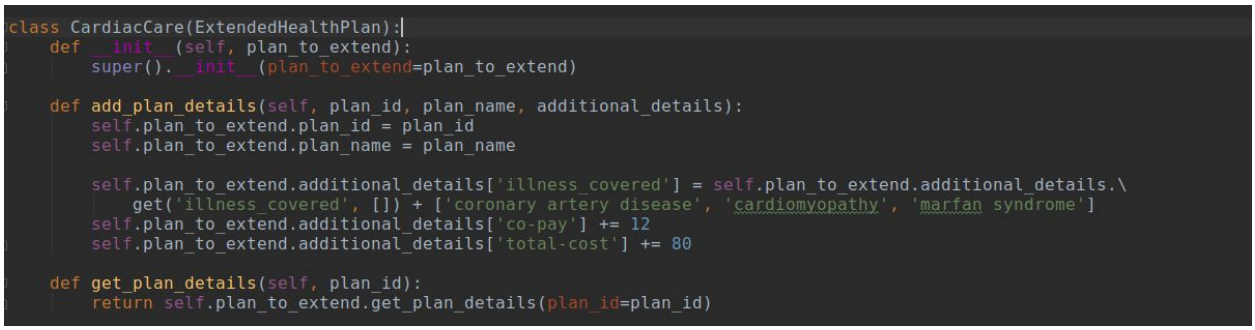
**Key Learnings**

- In the process of building this application, not only did I get to learn a great deal of design patterns but actually got to use them in-order to make my code more neat. Well of course there was the effort of redesigning the classes, but I guess in the long run it's for the best. For example in case of the Plan classes and its extensions, the newer class diagram makes it more realistic, and I had fun modelling the real life insurance structure to classes.
- Another key learning for me was the ability to model databases using code (DAO pattern). For this project in particular I used sqlalchemy as my ORM. Here is a snapshot of the schema written using that beautiful too -

```
import uuid

import datetime
from sqlalchemy import Column, Integer, String, DateTime, Boolean, JSON
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.engine.url import URL
from sqlalchemy import create_engine, orm


postgres_db = {'drivername': 'postgres',
               'username': 'postgres',
               'password': 'postgres',
               'host': 'localhost',
               'port': 5432,
               'database': 'insurancecompany'}

db_string = URL(**postgres_db)
db = create_engine(db_string)

Session = orm.sessionmaker(bind=db, autoflush=False)
db_session = Session()

Base = declarative_base()


class UserDB(Base):
    __tablename__ = 'users'

    iid = Column(Integer, primary_key=True)
    name = Column(String)
    email = Column(String)
    password = Column(String)
    role = Column(String)
    additional_metadata = Column(JSON)

    def __init__(self, name, email, password, role, additional_metadata):
        self.name = name
        self.email = email
        self.password = password
        self.role = role
        self.additional_metadata = additional_metadata
```

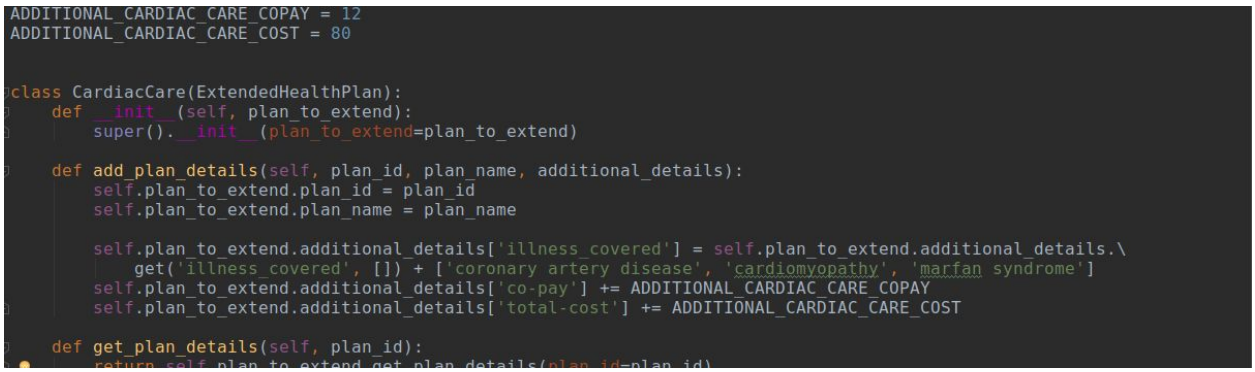
- The process of converting the anti-patterns to good code was somehow satisfying. It was kinda nice to see my code more maintainable. Well at least for me it seemed more admirable :). Here are quick snapshots of before and after (removing magic numbers) -

*Before*



```
class CardiacCare(ExtendedHealthPlan):
    def __init__(self, plan_to_extend):
        super().__init__(plan_to_extend=plan_to_extend)

    def add_plan_details(self, plan_id, plan_name, additional_details):
        self.plan_to_extend.plan_id = plan_id
        self.plan_to_extend.plan_name = plan_name

        self.plan_to_extend.additional_details['illness_covered'] = self.plan_to_extend.additional_details.\
            get('illness_covered', []) + ['coronary artery disease', 'cardiomyopathy', 'marfan syndrome']
        self.plan_to_extend.additional_details['co-pay'] += 12
        self.plan_to_extend.additional_details['total-cost'] += 80

    def get_plan_details(self, plan_id):
        return self.plan_to_extend.get_plan_details(plan_id=plan_id)
```

*After*



```
ADDITIONAL_CARDIAC_CARE_COPAY = 12
ADDITIONAL_CARDIAC_CARE_COST = 80


class CardiacCare(ExtendedHealthPlan):
    def __init__(self, plan_to_extend):
        super().__init__(plan_to_extend=plan_to_extend)

    def add_plan_details(self, plan_id, plan_name, additional_details):
        self.plan_to_extend.plan_id = plan_id
        self.plan_to_extend.plan_name = plan_name

        self.plan_to_extend.additional_details['illness_covered'] = self.plan_to_extend.additional_details.\
            get('illness_covered', []) + ['coronary artery disease', 'cardiomyopathy', 'marfan syndrome']
        self.plan_to_extend.additional_details['co-pay'] += ADDITIONAL_CARDIAC_CARE_COPAY
        self.plan_to_extend.additional_details['total-cost'] += ADDITIONAL_CARDIAC_CARE_COST

    def get_plan_details(self, plan_id):
        return self.plan_to_extend.get_plan_details(plan_id=plan_id)
```

- Finally, I learnt a lot by trying to split the code into the Models, Views and Controllers. It definitely taught me to better organize and solve the code components.