Name: Punith Patil

Title: Counselling Services Management

Project Summary: Maintains counselling medical records and options for electronically accessible counselling services.

This system aims to provide an electronically maintained counselling medical records and counselling services. It will have features to view and edit medical records, video and chat functionalities, book and schedule appointments, sign new users up, prescribe medicines and home screens for different types of users.

The system should be able register new Patients. Show patients a list of available Counsellors and enable them to book/reschedule an appointment. Patients should be able to view their prescriptions, pay bills and claim insurance. The counsellors should be able to view patient records and update them. Counsellors should be able to prescribe medicines.

Features Implemented:

ID	Requirement
UR001	Add new users to the system
UR002	Schedule appointment
UR003	Reschedule appointment
UR004	Pay bills
UR005	Claim insurance
UR006	View available Counsellors
UR007	View patient records
UR008	Prescribe medicines

Features Not Implemented:

ID	Requirement
UR011	Video call capability
UR012	Chat functionality
UR009	Disable login
UR010	Login to access functionalities

UR011 and UR012 are stretch functionalities.

Class Diagram:

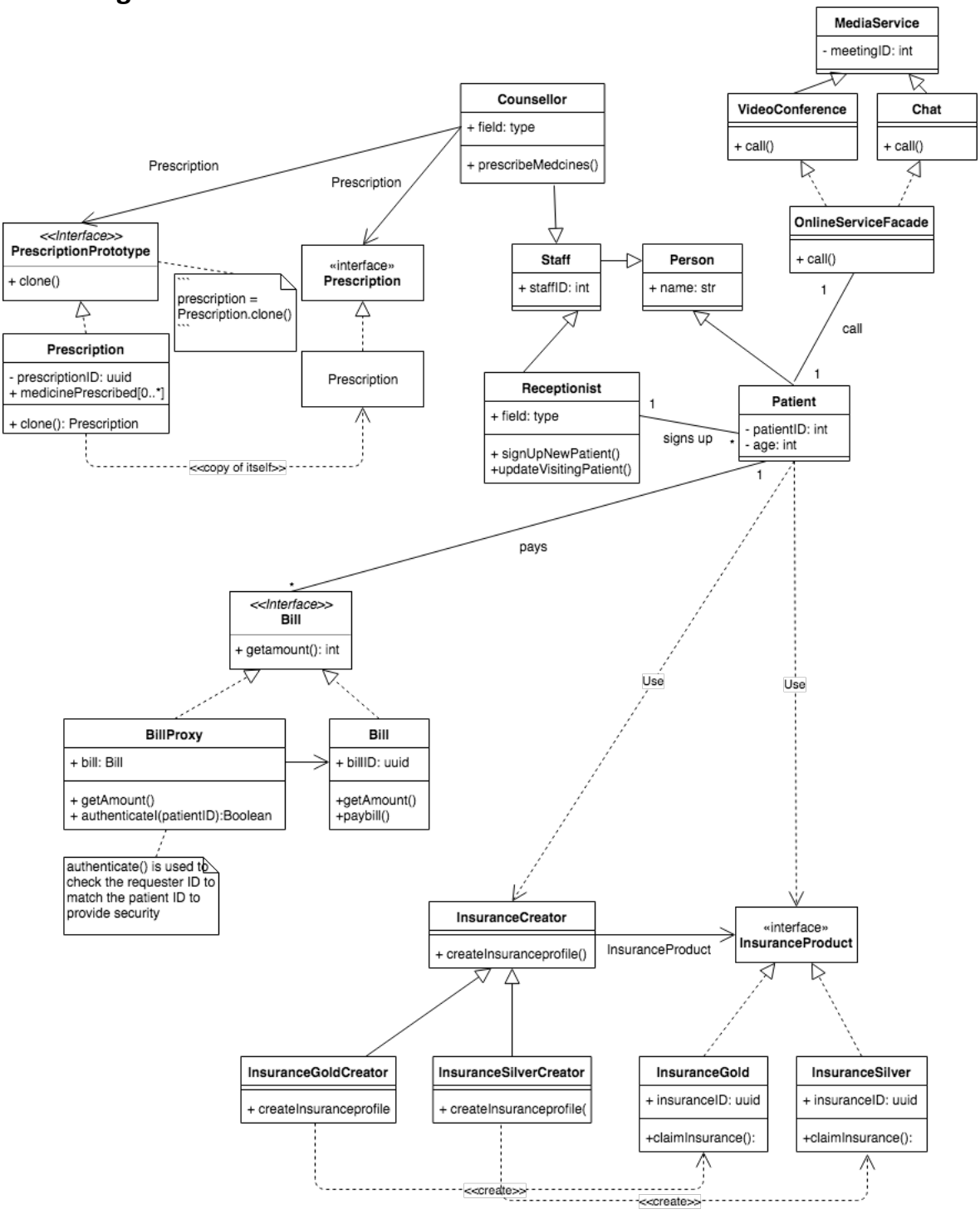

Figure 1 Final class diagram

There are several changes from the old class diagram to the new. Based on feedback from part 2, I have more classes to handle all the requirements. I have also corrected some mistakes. The new changes are caused by addition of design patterns. The application now has the following non-functional features:

1. **Security (Proxy)**
    - The feature where the patient can view/pay his bill is now protected by the **Protection Proxy Design Pattern**. When evert the patient wishes to access the bill, the patients ID is checked with the ID on the bill to ensure that the patient can only view bills they own
2. **Efficiency (Prototype)**
    - The prescription class will instantiate a prescription pad for each counsellor with common details like the counsellor's name, designation and other details. Whenever the counsellor writes a new prescription, the **Prototype Design Pattern** implementation will enable us to just copy the object and then initialize it with details of the patient. Counsellors write many prescriptions, and this can save resources and time by not instantiating new instances every time a counsellor writes a prescription. It is designed to emulate a prescription pad custom made for the counsellor and sheets (clone to generate sheets) for new patients.
3. **Scalability (Façade)**
    - The Online Services offered by this application relies on external software. For example, to make a video call one might use Hangouts by Google. But this might change from time to time as we have no control over what Productivity Tools the college purchases. Therefore, a **Façade Design Pattern** will give us a consistent implementation for our code. If there is a move away from Hangouts, we can write code for the new platform without changing any of our existing implementation. This design pattern was not implemented as getting the Hangouts API to work was a lot of work. The implementation interface will exists, we can add to it when the online service (collaboration services) is chosen and the API for it is decided and figured out.
4. **Long term flexibility (Factory)**
    - This is used when we are deciding what insurance a patient will have. There are two options, Gold and Silver as of now. As the new patient is registered their respective insurance instance is created at runtime. This also has the advantage of easily being extendable. To add a new insurance type, one had to implement the Insurance interface without any additional code change and without breaking the existing implementation.

Design Patterns:

1: Proxy Design Pattern

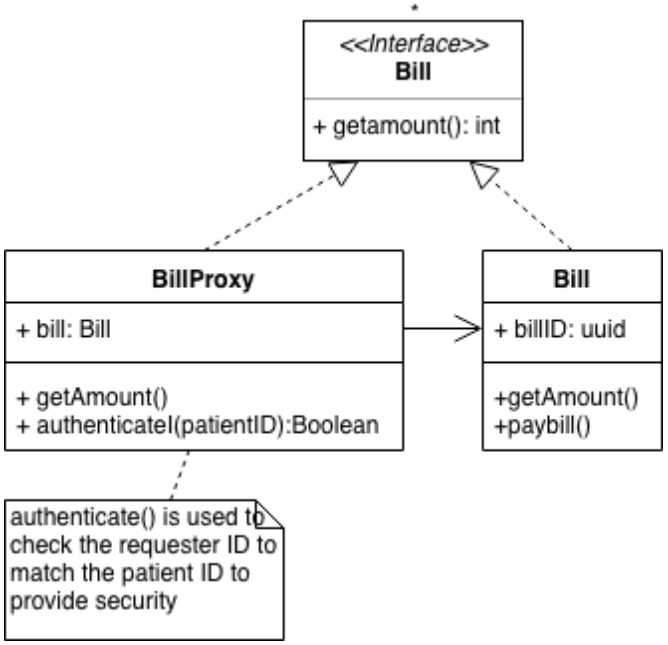

Figure 2 Proxy Design Pattern

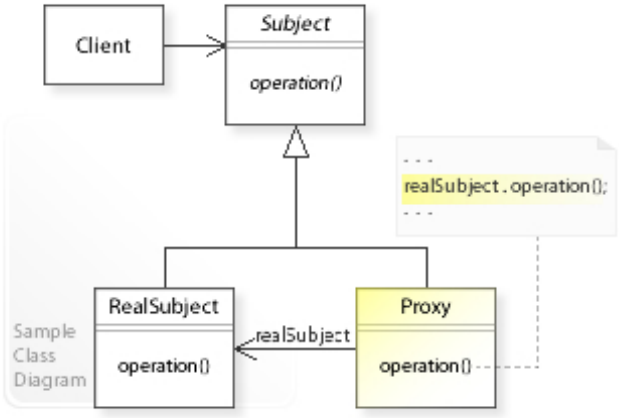

Figure 3 Proxy Design Pattern Wikipedia

Implementation details:

Classes involved: `AbstractBill, Bill, BillProxy`

`AbstractBill` – Contains abstract methods that the patient will request secured details.
`Bill` – Contains methods and field that represent the Bill Object

`BillProxy` – Contains the authenticate() method that will validate incoming request to access the Bill object. If the patient ID in the incoming request matches the ID inside the requested Bill object, the access will be granted. These classes can be seen inside the `bill.py` file in the `csm` module.

Reason for choosing Proxy:

The access to the Bill object must be protected. Patients should be able to view only their bills and not bills of other patients. The must also not be able to make payment to bills other than what is in their name. This is a privacy concern and protection proxy allows us to implement all of the above requirements.

2. Prototype Design Pattern

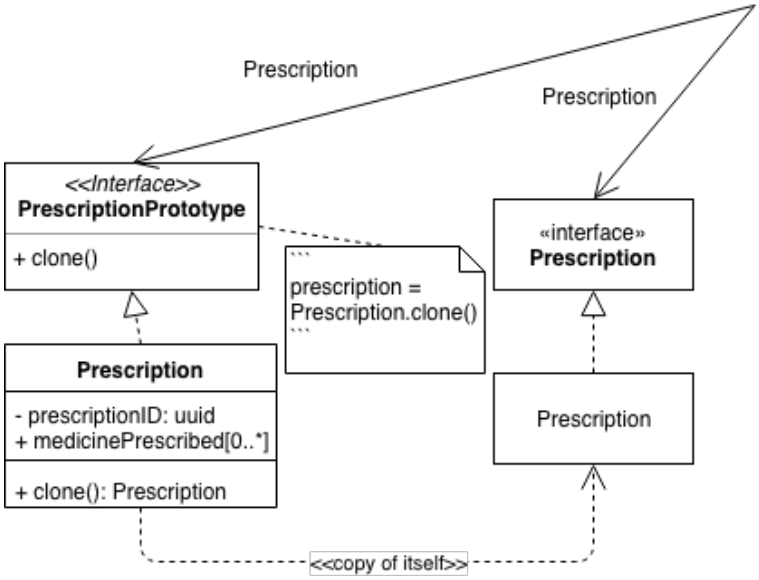

Figure 4 Prototype Design Pattern

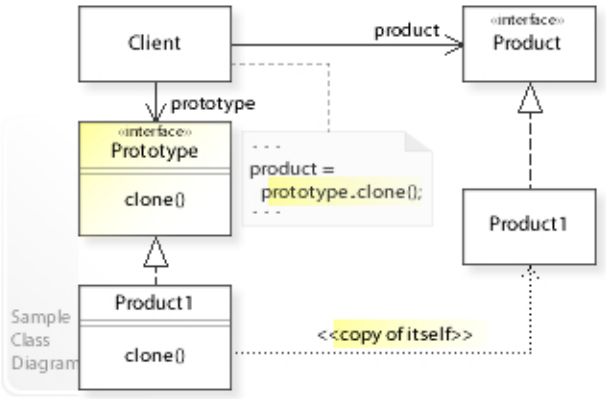

Figure 5 Prototype Design Pattern Wikipedia

Implementation Details:

Classes Involved: `PrescriptionPrototype`, `Prescription`, `PrescriptionFactory`

`PrescriptionPrototype` – Abstract class that defines the clone() method for the Product class to inherit

`Prescription` – The product class that implements the prototype interface PrescriptionPrototype. It also overwrites the clone method to return an object of itself

`PrescriptioProtorype` – This class serves as the factory to create instances by copying and existing instance. This class implements prescription_pads, a dictionary that maps a counsellor ID to is corresponding Prescription Object. When a new counsellor is created, an object is created and stored in the prescription_pads dictionary. When the counsellor writes a prescription the get_prescription_sheet() method will return a copy of the instance in the prescription_pads dictionary

Reason for choosing Prototype:

The Prescription object will contain some common details across different objects. For example, each counsellor will have his name on the prescription. Only the patient name and the medicines prescribed will change. By have a Prescription instance for each counsellor with the details already filled in, we can return copies for this instance for new prescription and only change the patient name and prescription. This will allow us to create Prescription objects faster. If there was no Prototype, each Prescription object would have redundant tributes. Prototype ensures than common attributes across different Prescriptions can be generalized.

3. Façade Design Pattern

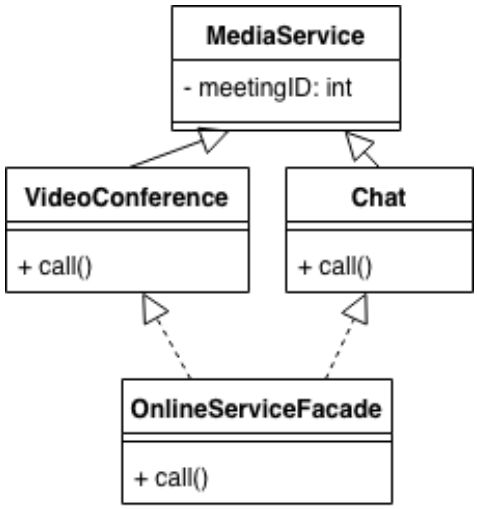

Figure 6 Facade Design Pattern

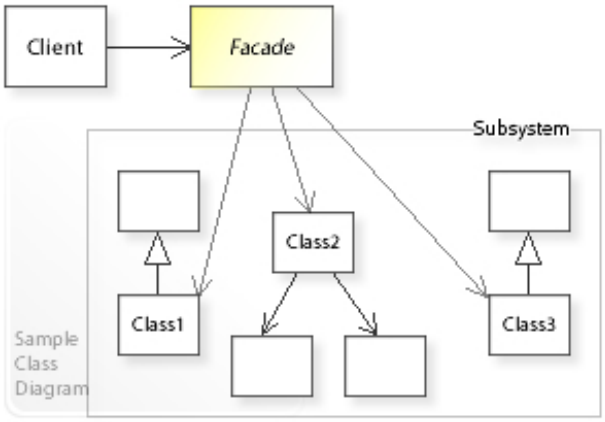

Figure 7 Facade Design Pattern Wikipedia

Implementation Details:

Classes Involved: `OnlineServicesFacade`

`OnlineServicesFacade` – This serves as the class that will instantiate both VideoConference and Chat class. It will implement a method called call that will be used to make calls via Video or chat. The client will only call the call method to start the conversation. This abstraction ensure that if the way Video conferencing or chatting is done, can be changed and the client code will not have to change as the façade will always provide the call() method which will intern call methods on the Video Conference and Chat classes. One can even extend this to add a voice call feature, and adding in the option to call via voice in the façade.

Reason to use Façade:

The way VideoConferencing or chatting is done might change. For example, CU Boulder uses Google suit of products. Hangouts can be used for calling, and the classes can be defined for Hangouts specifically. In the future they may move to WebEx as a conferencing tool. Without the façade the move to WebEx would cause changes in the patient and counsellor classes. Hence Façade is needed to keep the functionalities of the Online Services independent from the other classes that use the Online Services.

4. Factory Design pattern

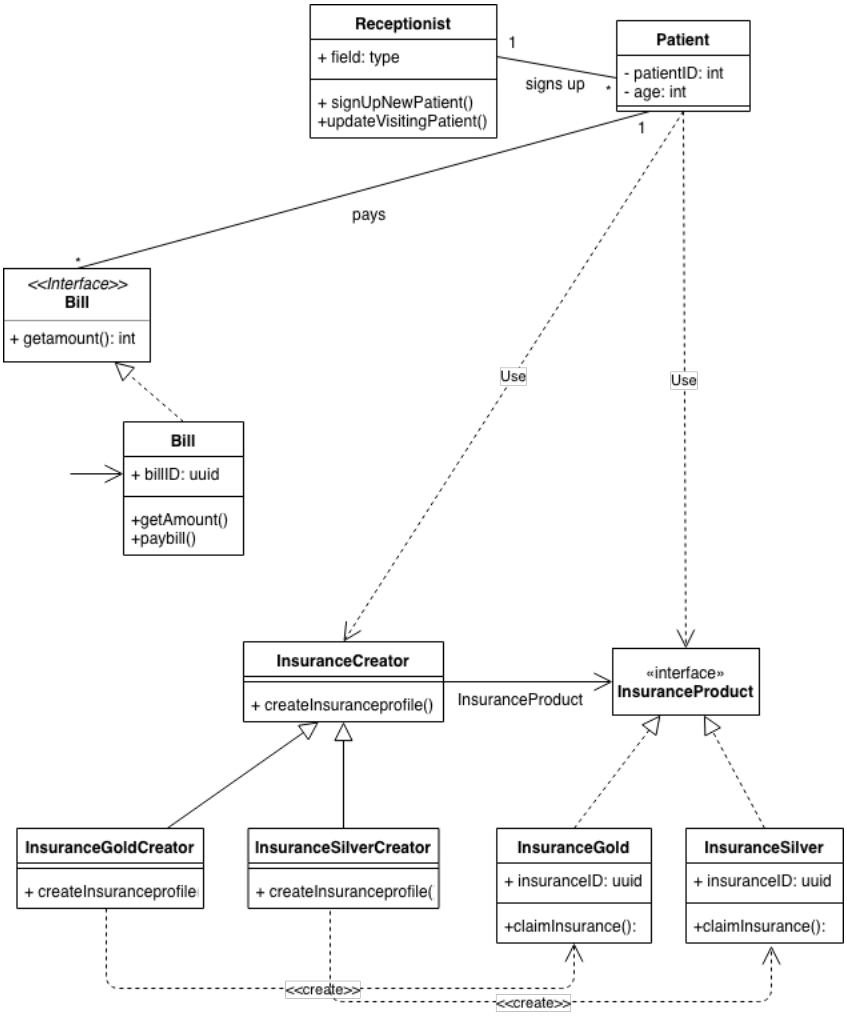

Figure 8 Factory Design Pattern

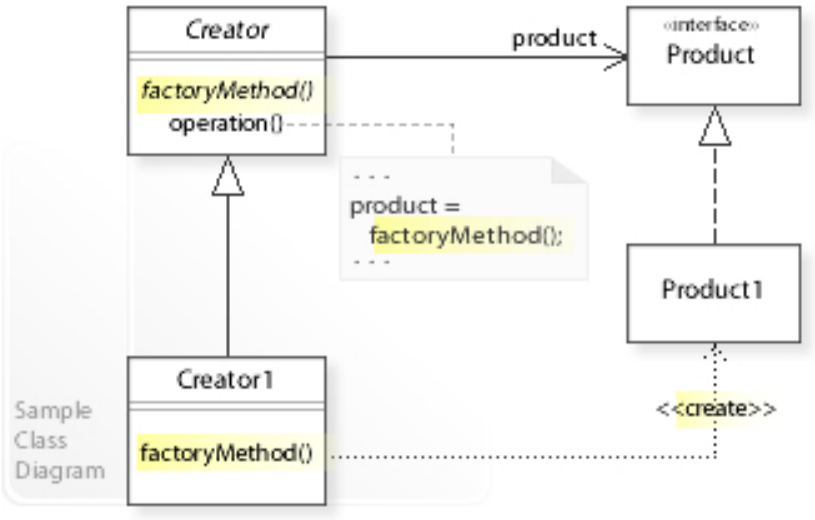

Figure 9 Factory Design Pattern Wikipedia

Implementation Details:

Classes implemented: `InsuraceCreator`, `InsuranceGoldCreator`, `InsuranceSilverCreator`, `InsuranceProduct`, `InsuranceGold`, `InsuranceSilver`

`InsuraceCreator` – Abstract class that defined the create method create_insurance_profile() method.
`InsuranceGoldCreator` – Concrete creator class that will return an instance of InsuranceGold class when the create_insurace_profile() method is called.
`InsuranceSilverCreator`- Concrete creator class that will return an instance of InsuranceSilver class when the create_insurance_profile() method is called
`InsuranceProduct` – Abstract class that defines the claim_insurace() method and any other insurance product must implement this class. This ensure that all insurance products have a common way to claim insurance
`InsuranceGold` – Concrete Product class that contains methods and attributes to claim insurance, check limit etc of the Gold product insurance.
`InsuranceSilver`- Concrete Product class that contains methods and attributes to claim insurance, check limit etc of the Silver product insurance.

Reason to use Factory:

There are multiple insurance products offered to the patients. One way to proceed is to allow creation of objects can be spread throughout the system. But when a new Insurance Product is introduced incorporating their changes is the entire system is cumbersome and time consuming. This login will have to be added in the Patient and Bill classes. Instead by ensuring a common factory, object creation must to be done through this factory. Then when adding a new insurance

product, only changes in the factory has to be made. This is a much cleaner and scalable way to implement creation of objects that are chosen at run time depending on what insurance the patient has at the time of creation of the patient object.

Key Learnings and Takeaways

The process of analysis if done at the start with great care can alleviate the painful changes later in the project development life cycle. For example, before implementing the factory design pattern I had to make code changes in the Patient class and the Bill class. The factory design pattern used ensures than when a new insurance product is released, one can easily add to the system by add the new type of insurance and incorporating it in the factory. Now all the client needs to do is just request for the new insurance product and will be served a new an instance of it. All the changes reside in the Factory class and no changes need to be made in the Patient or the Bill class.

The forethought of security is important. Designing the billing system with no security check is considered a violation of privacy of the patient. The Protection Proxy implementation is necessary anywhere sensitive data is handled. In client server architecture where clients request data, they must be validated first and only then responded with the data.

The part of the online services also was a complex thing. I have not implemented that feature to the fullest degree as to set up a full-fledged video/voice/chat call. As I tried to make the decision on what protocol to use, what service to use it became apparent that when there are several external services available and there are not in your control, it is always wise to implement a façade and then take care of the intricacies of each services as and when they are decided to be implemented and incorporated into the system.

The refactoring exercised in class helped me refactor my code. For example, in the insurance.py files, I had two classes InsuranceGold and InsuranceSilver. Depending on what a patient had subscribed to, I would use an if-else to determine what class to instantiate. These if-else statements were found in the patient class as the receptionist class. After applying the factory design pattern, I have completely eliminated the if-else and centeralised the creation of instances through the Factory. Therefore, I can add new Insurance Products without change I client code, it is easily extendable. The `insurance.py` file now has abstract creator class, concrete creator classes, abstract product class and concrete product classes (InsuranceGold and InsuranceSilver). I have also refactored out constants, by defining them in a file called constants.py. I have used guard clauses.