Software Archirtecture

THIS DOCUMENT COVERS

Introduction

Architecture

Architecture consists broadly of two parts. The first part takes a set of requirements and creates from them a high level conceptual architecture. The high level architecture is then decomposed into smaller subsystems and components. The architecture defines the interfaces, responsibilities and interactions of these components. The architect will specify logical layers and services and the interfaces between them. Non functional requirements influence the choice of technology.

The second part involves making the hard decisions early a project which are hard to change later on. For example deciding to use KDB and switching later on to Sybase is not an easy change.

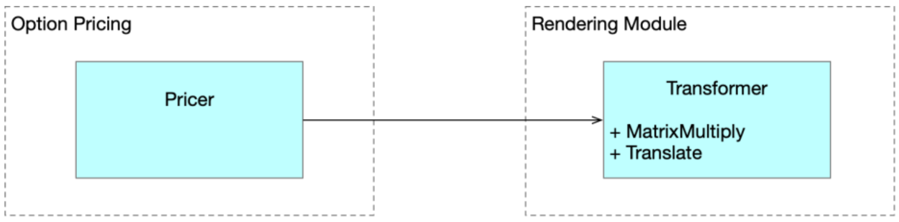
Architectural Patterns

- Layering
- Paritioning
- Service Orientated architecture
- Hexagonal architecture
- Micro services

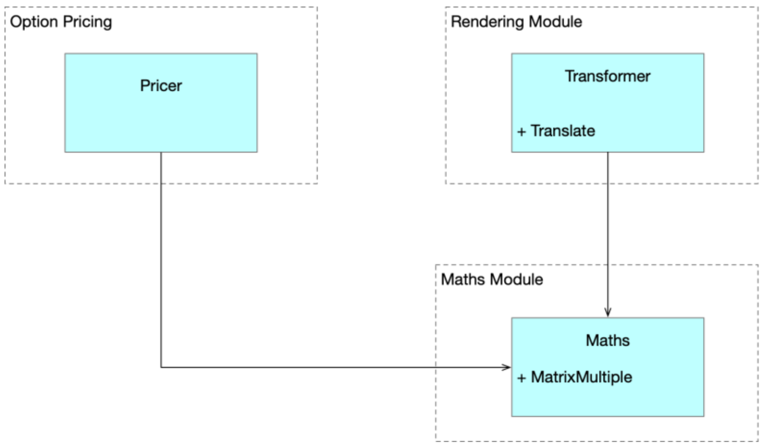
S.O.L.I.D.

Single Responsibility Principle

A single class should have only one responsibility. If a class has multiple responsibilities then changes to one responsibility can break the logic of the other responsibilities thus increasing brittleness. In cases where a module exposes a class that provides multiple responsibilities, clients of that module may be forced to recompile and redeploy when a responsibility on which they have no logical dependency changes.



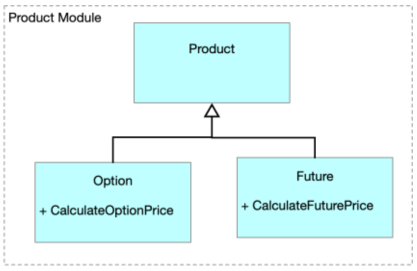
We can fix this as follows



Mixing business rules and persistence is a classic example of breaking the Single Responsibilty Principle. Patterns such as Façade, Proxy and DAO patterns can be used in such situations

Open Closed Principle

Types should be open for extention and closed for modification. Consider the following situation which breaks the open closed principle. Consider the following simple type hierarchy and a piece of client code which breaks the Open/Closed Principle

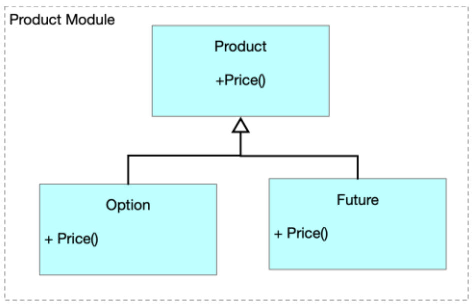


```
double PriceProducts(IEnumerable<Product> products)
{
        double total = 0.0;

        foreach (var product in products)
        {
                        if (product is Option)
                        total += ((Option)product).PriceOption();
                if (product is VarSwap)
                        total += ((VarSwap)product).PriceVarswap();
        }

        return total;
}
```

The use of abstractions and polymorphism enable us to modify our hierachy and PriceProducts method to obey the open closed principle.



```
double PriceProducts(IEnumerable<Product> products)
{
        double total = 0.0;

        foreach (var product in products)
                total += product.Price();

        return total;
}
```