**S O L I D**

Single Responsibility

Open-Close:

We worry about expectations of clients from the exposed abstraction(s). We try to be extensible to meet extra expectations of client(s) but we don’t modify the existing code because we might break some existing expectations. The Expectation(s) of client(s) is our concern here.

Liskov:

We worry about run-times. We again worry about clients’ expectations from the exposed abstraction(s). In a hierarchical structure all derived types should be able to be used as base type without breaking the expectations. We control this principle in 5 areas:

. Pre-conditions: cannot be strengthened

. Post-conditions: cannot we weakened

. Invariants: must be preserved

. Exceptions: no new one should be thrown in derived types

. Contravariance of method arguments and Covariance of return types

All pre/post conditions are mech sometimes to protect invariants. All of them are categorized in Business rules. Therea are mainly 2 ways of handling business rules:

. Structural: one-2-one or one-2-many relationships are good examples

. Behavioral: all pre/post conditions

Interface segregation

Dependency inversion

**Why to use Builder?**

Because calling constructor is complex in most of the times (building IOptions requires many fields), then we prefer to make the constructor Internal and then construction is delegated to the Builder. So that’s why we use builder in production code.

**Why to use the Manager to Update?**

According to the previous section (why to use Builder) calling the aggregate methods most of the times is a complex action, just like the constructors we prefer to make all methods Internal and then the invocations is delegated to the Manger class.

From now, in most cases we don’t use domain API(s) directly from outside of domain and instead we talk to the Managers. So, here we see that the Manager class can do both Constructing and Updating. All these efforts, help us to avoid having double standards among the code.

Point: the Managers are almost the Builders with more abilities than construction.

Point: the Managers have interfaces because we will have interaction testing for them.

public interface ITargetManager<TSelf, TTarget> : ITargetOptions  
 where TSelf : ITargetManager<TSelf, TTarget>  
 where TTarget : ITargetOptions  
{

TTarget Build();

TSelf WithTargetName(string value);  
  
 void Update(TTarget options);  
}

**Parent Aggregate**

When we talk about public API(s) of aggregate(s) we deeply refer to observable behavior(s) that are doing domain logic whether standalone or in collaboration with other aggregates’ observable behavior(s). In other words, public API(s) doesn’t necessarily mean public method(s). There are circumstance’s that an aggregate has an observable behavior which is not public to the outside of the domain. It is in service of another aggregates’ observable behavior. We call the outer aggregate as Parent Aggregate. It acts like a coordinator working with another aggregates.

*// parent aggreagate*public class FinancialPeriod : AggregateRoot  
{  
 *// public api of FinancialPeriod, it's accessible outside of domain* public Voucher SubmitVoucher()  
 {  
 *// here do what is todo  
 // then delegate the rest to another aggregate* return Voucher.CreateTemp();  
 }  
}  
  
public class Voucher  
{  
 *// public api of voucher, but it's internal and accessible to domain only* internal static Voucher CreateTemp()  
 {  
 *// here do what is todo* return default;  
 }  
}