**TDD**

We do it not because we want to test the software. We do it because it’s the way of development. It’s the way how a complex software can be built. Every complex problem can be modeled by small and simple parts and steps. That’s what TDD means to us.

Never start development by complicated/complex objects. We start development from the simplest object. Which object exactly? Usually, the object that has no dependency to other objects for example, value objects.

Start with happy path scenarios:

Happy path scenarios usually refer to 3 categories of scenarios including:

. constructing the SUT with only required references

. constructing the SUT with both required and optional references

. constructing with valid limitation of boundaries

Then we follow the development journey with focusing on exceptional scenarios including:

. construction exceptions

. boundary limitation

. business invariants and rules

**Test Builder**

We usually create a test builder for every SUT. A test builder has all properties(references) of the SUT (this archives by inheritance). One the differences between a test builder and a production builder is that every instance of a test builder initiates itself by all valid required references. This is very useful technique help us to have high maintainable tests.

How is relation of the test builder and production object?

Usually, all production objects are inherited from an abstract/interface, so we simply inherit the test builder from the SUT abstract/interface.

interface ISut

{

String Id;

}

Class Sut : ISut

{

Sut(string id)

{

Id = id;

}

String Id;

}

Class SutTestBuilder : ISut

{

SutTestBuilder (string id)

{

Id = “some id”;

}

String Id;

SutTestBuilder WithId(string value)

{

Id = value;

}

Sut Build()

{

Return this;

}

}

**The Test Manager**

This approach has very similarities with the Test Builder. A test manager not also builds an SUT but also can modify it too. In most circumstances, the test manager is used in combination with the domain manager and this archives through delegation.

Conventions:

. a test manager doesn’t inherit from any abstraction

. a test manager is generic when the target is polymorphic

. a generic test manager needs TSelf argument in order to return it in With() methods

. a generic test manager needs TTarget argument in order to return it in Build() method

. a generic test manager needs TManager argument in order to SutBuilder to outside world

* TSelf is type of the test manager itself
* TTarget is type of domain interface
* TManager is type of domain manager

public abstract class TestTargetManager<TSelf, TI, TIBuilder>  
 where TSelf : TestTargetManager<TSelf, TI, TIBuilder>  
 where TIBuilder : ITargetManager<TIBuilder, TI>  
 where TI : ITarget

{

public TManager SutBuilder;  
 protected abstract TManager CreateManager();  
  
 public TTarget Build()  
 {  
 return SutBuilder.Build();  
 }

}

**The Polymorphic Test**

In hierarchical models it is desirable to be alerted about the Liskov principle. This concern can be answered in tests by following the polymorphism in the tests too. The general approach says that for base type we have tests and also for derived type we have tests too but we provide it by inheritance. In means the derived test class inherit from base type test class.

Conventions:

. base test class is generic

. generic base test class is abstract

. generic base test class needs TTestManager argument to use it as SutBuilder

public abstract class TargetTests<TM, TI, TIM>  
 where TM : TestTargetManager<TM, TI, TIM>  
 where TIM : ITargetManager<TIM, TI>  
 where TI : ITarget

{

protected TargetTests()  
 {  
 Manager = Activator.CreateInstance<TTestManager>();  
 }  
  
 protected TestTargetManager<TTestManager, TTarget, TManager> Manager;  
 protected TTarget SUT;

}