Search Criteria Guard

The goal of the application is to allow specifications for search (criteria's) to be specified by the client in order to perform a custom search. **Error! Reference source not found.** demonstrates the requirement. I intestinally keep it simple for the sake of the exercise.

Figure 1

Where do I start?
This is probably the most difficult question – where do I start? From the beginning, of course. I will try to make it sort of TDD way, and keeping the Agile concepts in head to respect some of the OO principles I have learned lately – no overdesigning.

So what's the plan? The plan is to have a plan! (Valiant cartoon, recommended). What do we have?

- Criteria's
- Search results provided by some service based on criteria's customer has provided

But what if criteria's are wrong? We should be able to handle it

- Error message if criteria's are wrong

Now the question – should the service for search result handle the validity of criteria's? Nope, it only should consume it as-is, trusting it to be valid. Therefore Criteria's have to an object with its own behavior and "business rules" around it that can be tested and become a dependency for the search service. Let's hit the design through the tests.

My first tests are all about the SearchService. The concern of this component is to invoke the model to bring some data based on SearchCriteria. I can almost smell 2 different dependencies:

1. Model
2. SearchCriteria

I will not expand on the Model to the complexity of the entire exercise and my limited knowledge at this point about Domain Driven Design, but will come to this sooner or later, as this is definitely the way to write applications. SearchCriteria I will cover to make the example to work.

SearchServiceTest as showed in Listing 1

- Sanity check – can we get the object at all? → `Should_be_able_to_instanciate_service()` – state based test

- Given a certain SearchCriteria as a dependency, will the system under test (SUT) leverage the dependency, i.e. will be SearchCriteria used when SearchService is required to return result → `Should_be_able_to_return_search_results_with_a_given_search_criteria()` – interaction based test using mocked dependency

Later, when SearchCriteria is tested and implemented, we add more tests to SearchService

- Is NullSearchResult object (Null Object pattern) returned on an invalid SearchCriteria as a result of min date being bigger than the max date→ `Should_return_empty_search_result_due_to_bad_dates`
- Is NullSearchResult object (Null Object pattern) returned on an invalid SearchCriteria as a result of min status being bigger than the max status→ `Should_return_empty_search_result_due_to_bad_dates`

```
  [TestFixture]
  public class SearchServiceTest
  {
    private MockRepository mock;

    [SetUp]
    public void Setup()
    {
      mock = new MockRepository();
    }

    [TearDown]
    public void TearDown()
    {
    }

    [Test]
    public void Should_be_able_to_instanciate_service()
    {
      ISearchService sut = CreateSUT();
      Assert.IsNotNull(sut, "failed to instantiate service");
    }

    [Test]
    public void
Should_be_able_to_return_search_results_with_a_given_search_criteria()
    {
      ISearchCriteria mockSearchCriteria = mock.CreateMock<ISearchCriteria>();
      ISearchService sut = CreateSUT(mockSearchCriteria);

      using (mock.Record())
      {
        Expect.Call(mockSearchCriteria.IsValid()).IgnoreArguments().Return(true);
      }

      using (mock.Playback())
      {
        ISearchResult result = sut.GetResults();
        Assert.IsNotNull(result);
```

```
      }
    }

    [Test]
    public void Should_return_empty_search_result_due_to_bad_dates()
    {
      ISearchService sut = CreateSUT(new
SearchCriteria(DateTime.Today.AddYears(1), DateTime.Today, 0, 0));
      Assert.AreEqual(SearchResult.NullSearchResult, sut.GetResults());
    }

    [Test]
    public void Should_return_empty_search_result_due_to_bad_statuses()
    {
      ISearchService sut = CreateSUT(new SearchCriteria(DateTime.Today,
DateTime.Today, 3, 2));
      Assert.AreEqual(SearchResult.NullSearchResult, sut.GetResults());
    }

    private ISearchService CreateSUT(ISearchCriteria searchCriteria)
    {
      return new SearchService(searchCriteria);
    }


    public ISearchService CreateSUT()
    {
      return new SearchService(new SearchCriteria(new DateTime(), new DateTime(),
0, 0));
    }
  }
```

Listing 1

SearchCriteriaTest as showed in Listing 2

- Sanity check → `Should_be_able_to_instanciate_search_criteria()`
- Test validity based on criteria parameters (the rules I came up with are as long as minimum is less or equal to the maximum, criteria is considered to be valid) → `Should_be_able_to_return_search_criteria_validity(int minYear, int minMonth, int minDay, int maxYear, int maxMonth, int maxDay, int minStatus, int maxStatus, bool result)` df- in this test case I am utilizing the MbUnit's ability to run same test with different input values

```
    [Test]
    public void Should_be_able_to_instanciate_search_criteria()
    {
      ISearchCriteria sut = CreateSUT();
      Assert.IsNotNull(sut, "failed to instantiate service");
    }

    [RowTest]
    [Row(2008, 1, 1, 2008, 1, 15, 0, 0, true)]
    [Row(2008, 1, 15, 2008, 1, 1, 0, 0, false)]
```

```
    [Row(2008, 1, 1, 2008, 1, 15, 1, 3, true)]
    [Row(2008, 1, 1, 2008, 1, 15, 3, 1, false)]
    public void Should_be_able_to_return_search_criteria_validity(int minYear, int
minMonth, int minDay, int maxYear, int maxMonth, int maxDay, int minStatus, int
maxStatus, bool result)
    {
      DateTime minDate = new DateTime(minYear, minMonth, minDay);
      DateTime maxDate = new DateTime(maxYear, maxMonth, maxDay);
      ISearchCriteria sut = CreateSUT(minDate, maxDate, minStatus, maxStatus);
      Assert.AreEqual(result, sut.IsValid());
    }

    private ISearchCriteria CreateSUT(DateTime minDate, DateTime maxDate, int
minStatus, int maxStatus)
    {
      return new SearchCriteria(minDate, maxDate, minStatus, maxStatus);
    }

    public ISearchCriteria CreateSUT()
    {
      return CreateSUT(new DateTime(), new DateTime(), 0, 0);
    }
  }
```

Listing 2

So what do I have so far:

1. Design for SearchService and SearchCriteria based on practical usage
2. Design by Contract of the listed above
3. Principle of Dipendency Injection (SearchCriteria is a dependency for SearchService) and Inversion of Control
4. Dependency on abstraction and not concrete type for search criteria
5. Encapsulation of business rules around searching criteria in an object
6. Distinguished separation of concerns – SearchService knows nothing about SearchCriteria details and nuances, except what it should know – is criteria valid or not.

The implementation of the listed above classes and their contracts in Listing 3, Listing 4, Listing 5, and Listing 6 are entirely based on the tests I conducted. This ensures not only that the code is tested, but also documents well what should be the expected behavior, i.e. an alternative documentation for the design.

```
  public interface ISearchService
  {
    ISearchResult GetResults();
  }

  public interface ISearchCriteria
  {
    bool IsValid();
  }

  public interface ISearchResult
```

```
  {
    // details are omitted to simplify example
  }
```

Listing 3

```
  public class SearchCriteria : ISearchCriteria
  {
    private readonly DateTime maxDate;
    private readonly int minStatus;
    private readonly int maxStatus;
    private readonly DateTime minDate;

    public SearchCriteria(DateTime minDate, DateTime maxDate, int minStatus, int
maxStatus)
    {
      this.minDate = minDate;
      this.maxDate = maxDate;
      this.minStatus = minStatus;
      this.maxStatus = maxStatus;
    }

    public bool IsValid()
    {
      return DatePartOfCriteriaIsValid() && StatusPartOfCriteriaIsValid();
    }

    private bool StatusPartOfCriteriaIsValid()
    {
      return minStatus <= maxStatus;
    }

    private bool DatePartOfCriteriaIsValid()
    {
      return minDate <= maxDate;
    }
  }
```

Listing 4

```
  public class SearchResult : ISearchResult
  {
    public static readonly ISearchResult NullSearchResult = new
NullSearchResultObject();

    private class NullSearchResultObject : ISearchResult
    {
    }
  }
```

Listing 5

```
  public class SearchService : ISearchService
  {
    private readonly ISearchCriteria searchCriteria;

    public SearchService(ISearchCriteria searchCriteria)
    {
```

```
      this.searchCriteria = searchCriteria;
    }

    public ISearchResult GetResults()
    {
      if (searchCriteria.IsValid())
      {
        return new SearchResult();
      }
      return SearchResult.NullSearchResult;
    }
  }
```

Listing 6

Tools used

- MbUnit
- Rhino.Mocks
- NAnt
- Visual Studio 2008
- Console2

NAnt and automated build

I wanted to have my build script files to be partitioned and structured in a manner where I can invest minimum of the effort to kick off a new solution or a project in a solution, and being able to configure automated build (and tests) fast. The scripts are not the best, but this is just the first attempt to make it happen. The only targets I was actually running were “build” and “test”.

Usefulness of this post

Someone said to me that what you just learned today might be looked by someone else tomorrow. In absolutely no way I am trying to teach people what have just barely learned. The motivation is to get others inspired and trigger feedbacks in order to evaluate what I am proposing.

Thanks

- JP Boodhoo for introducing myself to the world of patterns, TDD, and DDD
- Glen for opposing ideas I had and pushing to prove that they are not just words
- Mr. Mo for ideas and encouragement to publish the post
- Adam for fueling me up on the NAnt cryptic style and partitioning of it in general