Practical Implementation of in-memory Repository Patterns

Recently I have been given a task to demonstrate my key knowledge on OOP and Design Patterns and its practical implementation.

*The Task*

Create an in memory implementation of IRepository<T>

*Hints:The solution should be developed in a Test Driven approach, we expect the code you write to have excellent unit test coverage.Keep the solution simple, we value simple architecture and SOLID principles, but we’re not looking for over engineered design.The code should be self-explanatory, documentation should not be required to understand the code.*

I have create a solution for that but prior to that I would like to explain what is repository pattern and explain it in my own words and how I implemented by describing my thoughts.

***Repository Pattern: what is it?***

Repository pattern is a pattern for data access. This is a common design pattern inside of applications that need to perform data access.

*Why do we use the Repository Pattern?*

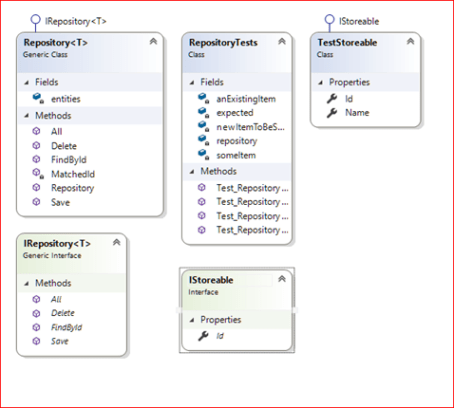
Because most of applications do need to access data from some Data source or perhaps multiple data sources, and for that we need to use data access code throughout Business Logic or Domain Model. It is true that data access code typically is very repetitive and it’s concerned with low-level infrastructure details such as opening SQL connection and managing parameters etc.

This pattern will help us to separate the data access code from business logic and increase the testability of an application without real data in the place. and we can simulate data access in unit tests and have a component that works with in-memory data instead of web service or against any database schema.This allows us to unit test to be very repeatable, very low friction and not much effort to set up and it has quick feedback cycle.

***So Let us crack on.***

A Repository is going to be a class that you build for your application and it’s going to exist somewhere between the Business Logic and the Data Source of your application. You might have multiple repositories inside of an application. You might have a repository for each type of entity that you need to store.

Here we have been given an opportunity to create a solution for in-memory Generic IRepository Implementation which allows us to Save, Delete, Retrieve all or Find data by Id. Because it is generic in nature it can be easily implemented for any other entities.  Likewise here we are talking very specific implementation for IStoreable which has a Property of IComparable. (You must have knowledge of how IComparable works and implements and further details can be found [**here**](https://support.microsoft.com/en-us/help/320727/how-to-use-the-icomparable-and-icomparer-interfaces-in-visual-c).

[](https://i1.wp.com/blog.ravilodhiya.co.uk/wp-content/uploads/2017/12/image.png)

So let us get back to our original given problem statement:

Following interfaces has been given and we have to implement the concrete class for them.

The first file is IStoreable.cs as given below:

1: using System;

2:

3: namespace RamInfosystems.Interview

4: {

5: public interface IStoreable

6: {

7: IComparable Id { get; set; }

8: }

9:

10: }

Now next file is IRepository.cs as given below:

1: using System;

2: using System.Collections.Generic;

3:

4: namespace RamInfosystems.Interview

5: {

6: // Please create an in memory implementation of IRepository<T>

7:

8: public interface IRepository<T> where T : IStoreable

9: {

10: IEnumerable<T> All();

11: void Delete(IComparable id);

12: void Save(T item);

13: T FindById(IComparable id);

14: }

15: }

And we need to create some test in Unit Test file.

My approach to solve this is by applying Test first. Clearly here the implementation of an interface is generic type and has [**constraints of type Interface**](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters). And this will harness the element which we can compare and test against it. So far nothing unusual.

So when you implement the IRepository Class you also need to implement  IStoreable as well. Which I think first you need to figure it out what need to be achieved.

1: public class TestStoreable : IStoreable

2: {

3: public string Name { get; set; }

4: public IComparable Id { get; set; }

5: }

So what we need to do here is Implement the IRepository interface and its 4 of Methods as such All, FindByID, Delete and Save. It was pretty obvious that implementation is a key for the each of operation required to be tested. I have to add a property call Name as string to attach.

I started with unit test and create my first test method named Test\_Repository\_List\_Returns\_IEnumberable\_CorrectType as follows:

1: [Test]

2: public void Test\_Repository\_List\_Returns\_IEnumberable\_CorrectType()

3: {

4: repository = new Repository<TestStoreable>();

5: this.expected = repository.All();

6: Assert.IsInstanceOf<IEnumerable<TestStoreable>>(expected);

7: }

Small explanation about the long name. I intend to do readable code as needed and what  my code does is reflect by its name for example test repository list returns when IEnumberable with correct type.

But I don’t have any implementation of Repository or any implementation of All(). Let’s implement it as below:

1: using System;

2: using System.Collections.Generic;

3: using System.Linq;

4: using System.Text;

5:

6: namespace RamInfosystems.Interview

7: {

8: public class Repository<T> : IRepository<T> where T : IStoreable

9: {

10: private List<T> entities;

11:

12: public Repository()

13: {

14: entities = new List<T>();

15: }

16:

17: public IEnumerable<T> All()

18: {

19: return entities;

20: }

21: ///Not fully implemented.

22:

23: }

24: }

so when we call repository.All() it expected to be IEnumberable of generic type and List<T> is a same of the type.  Trick here is a concrete implementation of IStoreable gives us a opportunity to test all() method’s implementation.

Further on Delete(IComparable id) and Save(T item) implementation also based on the writing test first approach, Arrange, Act and Assert is cycle for TDD and I follow the same.

1: [Test]

2: public void Test\_Repository\_Save\_AddNewItem()

3: {

4: repository = new Repository<TestStoreable>();

5: newItemToBeSaved = new TestStoreable { Id = 1, Name = "ItemToBeSaved" };

6: repository.Save(newItemToBeSaved);

7: expected = repository.All();

8: Assert.IsTrue(((IEnumerable<TestStoreable>)expected).Contains(newItemToBeSaved));

9: }

10:

11: [Test]

12: public void Test\_Repository\_Delete\_An\_Existing\_Item()

13: {

14: repository = new Repository<TestStoreable>();

15: anExistingItem = new TestStoreable { Id = 1, Name = "AnExistingItemToBeDeleted" };

16:

17: repository.Save(anExistingItem);

18: repository.Delete(1);

19: expected = repository.All();

20:

21: Assert.IsFalse(((IEnumerable<TestStoreable>)expected).Contains(anExistingItem));

22: }

Now it is obvious for method Delete we are providing an ID (which is IComparable) and which leads to additional method to create as it requires Predicate of its type and we can fetch through the Id by lambda expression  as follows

1: private Predicate<T> MatchedId(IComparable id)

2: {

3: return match => match.Id.Equals(id);

4: }

Alright. so it help to find an Id which provided in the Delete method. Also it add an advantage to avoid a duplicate id in the Save operation shown as follows

1: public void Save(T item)

2: {

3: Delete(item.Id); // Avoid Duplicate

4: entities.Add(item);

5: }

Here it might get tricky as Lazy evaluation of Find as we are passing IComparable Id which lead us to do certain things.

1: [Test]

2: public void Test\_Repository\_Find\_By\_Id()

3: {

4: repository = new Repository<TestStoreable>();

5: someItem = new TestStoreable { Id = 2, Name = "SomeItem" };

6: anExistingItem = new TestStoreable { Id = 1, Name = "ExistingItem" };

7: newItemToBeSaved = new TestStoreable { Id = 3, Name = "NewItemToBeSaved" };

8: repository.Save(someItem);

9: repository.Save(anExistingItem);

10: repository.Save(newItemToBeSaved);

11:

12: expected = repository.FindById(2);

13:

14: Assert.AreEqual(someItem, expected);

15: }

And finally Implementation of FindById.

1: public T FindById(IComparable id)

2: {

3: return entities.Find(MatchedId(id));

4: }

This will harness all the uses of Predicates and Generic list to find and match equal ids. This technique will help to enhance in-memory sorting and searching very quickly. However I haven’t included any stress-test to generate let us say 100 Million records and measure performance against the memory load or anything as it was out of scope for me at this stage. But Later I am thinking to write a separate article for it.

I have tried to implement and develop this task using my ability to provide demonstration about how to implement the Repository Pattern for in-memory.

**Consequences**

There are a couple consequences to using the Repository Design Pattern. First, you should have an increased level of abstraction inside your application. This is good because it should mean that you’ll have less duplicated code. It also means that you should have code that’s more maintainable, more flexible, and more testable, however, you do have some additional interfaces, some additional classes, some people who are not familiar with the design pattern might need to look at this and figure out exactly what’s going on before they can catch on. When you have data access logic buried right inside of your business code, sometimes it’s very easy to look at that and understand what’s going on, but it just doesn’t scale up well, and in complex applications, you really need this increased level of abstraction just to manage the complexity. Another consequence of the Repository Design Pattern is that you are one step further away from the data. Since the Repository is shielding you from the underlying infrastructure, sometimes it can also be more difficult to get hints into that infrastructure that can help it optimize some of the operations that it needs to perform against the Data Source.

**Summary**

Here I tried to explain the Repository Design Pattern and how we can use it to keep Business Logic and Domain Model free of data access code and away from infrastructure code and actual components that talk to the database.

Using the Repository Pattern, I was able to write more maintainable code and easily unit test code. And  I showed a specific example that implemented a generic repository in C# code, the IRepository<T> interface. Using the magic of Generics, I was able to provide a single concrete implementation of this interface that covers most of our needs in an application.

If you are looking for a code or repository you can get it from code project and stack overflow as link given below: