AnimalMaker – Project Overview

Contents

[1. Overview 2](#_Toc361919899)

[1.1. Assumptions 2](#_Toc361919900)

[2. Technology Overview 2](#_Toc361919901)

[2.1. Dependency Injection 2](#_Toc361919902)

[2.2. .NET Reflection 3](#_Toc361919903)

[3. Key Functional Components 3](#_Toc361919904)

[3.1. Animals 3](#_Toc361919905)

[3.2. AnimalMakerFactory 4](#_Toc361919906)

[3.3. Animal Maker(s) 4](#_Toc361919907)

[3.3.1. Maker 5](#_Toc361919908)

[3.3.2. ReflectionAnimalMaker 5](#_Toc361919909)

[3.4. Repositories 5](#_Toc361919910)

[3.4.1. DefaultRepository 6](#_Toc361919911)

[3.4.2. XmlRepository 6](#_Toc361919912)

[4. Building & Testing 6](#_Toc361919913)

[4.1. Building the solution 6](#_Toc361919914)

[4.2. Run unit tests 7](#_Toc361919915)

# Overview

‘AnimalMaker’ is a simple software solution that is intended to be used as a plug-in into Acme Zoo’s application suit to serve the purpose of creating Animal entities. Development was based on TDD approach. This solution contains two main projects as follows;

1. Zoo.AnimalMaker.Core : Plugin project serving as the animal maker in order to create animal entities based on a customizable mechanism (explained in section 2.)
2. Zoo.AnimalMaker.Test : Test project with a set of unit tests that can be executed using Rake CI task tool.

# Assumptions

Assumed that the customer requires different animals to be defined as different software entities (classes) so that whenever a new animal type is introduced, a new class for that animal type has to be added to the system.

However, the system is modified with its design such that the properties associated to any animal type can be fed dynamically using dependency injection & .NET reflection.

# Technology Overview

With the latest software design trends, possible new patterns & principles were added into the solution to provide the maximum flexibility, extensibility and maintainability. Two main features can be highlighted as follows.

# Dependency Injection

Under the ‘SOLID’ design principles, Dependency Inversion has become one of the key design principles to follow in latest software solutions. This will reduce or ditch the dependencies occur on high level modules on lower level modules and will provide a way of dynamically injecting the dependencies whenever required. For this solution, I have used ‘Ninject’ as the dependency injector to maintain simplicity, but can use any other DI container (Unity, Autofac etc) to gain further customizability such as registering dependencies on a config file. Further technical details on how DI is being used in this solution is described in section 3 ( Key functional Components )

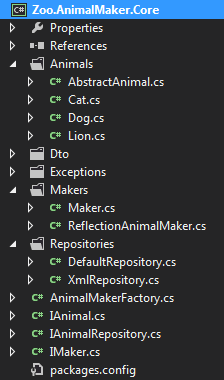
# .NET Reflection

Based on the main assumption we made, animal entities are not generic where each animal type will have its own class such as Dog, Cat, Lion etc. This will restrict the animal making mechanism to create animals dynamically so that the animal maker will have a hard coded list of animals to serve the ‘make’ request to create a particular animal type.

But with this solution, a more dynamic way is introduced with .NET reflection, where it will search for all the animal types available using .NET reflection and will instantiate the required type using the ‘Activator’. This will introduce more flexibility of introducing new animals, where it’s simplified to just adding a new class into ‘Animals’ folder and you don’t have to modify the maker. How this is being used in the solution is described in detail in section 3.

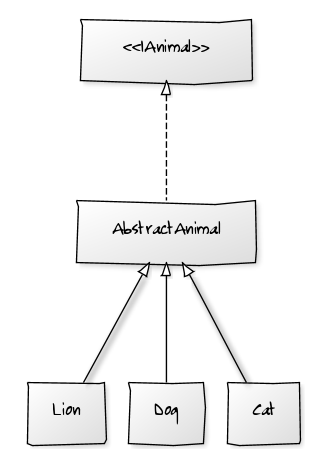
# Key Functional Components

Core project will have the following hierarchy where similar entities have been grouped to have an easy to use namespace hierarchy.



# Animals

All Animal entities will be implementing the IAnimal interface, but to avoid repetitively defining the common properties, an abstract class is introduced as ‘AbstractAnimal’. This abstract class will have all the common properties and the mechanism to populate property values for each animal instance. Therefore all animal objects will be an extension of this abstract class and they all will inherit the common properties. A repository will be injected into all animal objects in their constructors and all constructors will call the base constructor (in AbstractAnimal) to get their properties populated with the injected repository (IAnimalRepository implementation).



# AnimalMakerFactory

This is the Dependency injector where the ‘AnimalMaker’ consumers can decide which AnimalMaker (section 3.3) or the AnimalRepository (section 3.4) to be used dynamically.

By calling the public ‘GetInstanceOf<T>’ method, any consumer can get an instance of the type T.

e.g

\_animalMaker = AnimalMakerFactory.GetInstanceOf<IAnimalMaker>();

Bindings can be defined in the Ninject register module’s Load method. If there’s a new type of dependency to be defined for DI, it can simply be added as another entry as follows.

//IMaker binding [ options: Maker|ReflectionAnimalMaker ]

Bind<IMaker>().To<ReflectionAnimalMaker>();

//IAnimalRepository binding [ options : DefaultRepository|XmlRepository ]

Bind<IAnimalRepository>().To<XmlRepository>();

# Animal Maker(s)

There are two types of Animal makers have been introduced to the system and are inside the ‘Makers’ folder. All AnimalMakers supposed to implement the IAnimalMaker interface so that the AnimalMaker to be used can be decided on runtime using the AnimalMakerFactory dependency injector.

One of the animal makers can be injected using the AimalMakerFactory into the animal making process and this gives a flexibility of introducing new animal making mechanisms in future.

# Maker

This is the legacy animal maker having a hard coded switch statement to decide which animal to be instantiated . This doesn’t have the flexibility of introducing new animals without changing the maker.

# ReflectionAnimalMaker

This AnimalMaker uses .NET reflection to fetch the available animal types in the system by searching for all the entities in the assembly that implements the IAnimal interface. Then it will instantiate an instance of the expected type based on the animal type name parameter provided.

This gives the flexibility of introducing new animal types without changing the AnimalMaker. This AnimalMaker is capable of fetching any new animal types added into the assembly without changing the AnimalMaker class.

This can be made further extensible by isolating the animal types into a separate assembly so that only that assembly is required to compile on new animal type addition.

# Repositories

After the animal is created using the AnimalMaker, its properties are populated using a repository. This gives the flexibility of storing the properties in a convenient method such as in a database or in a XML file etc. Similar to Animal Maker, repository will be injected at runtime using Ninject DI container to feed the property values.

Currently two types of repositories are available but can add further repositories to feed property values into Animal objects. (e.g if properties supposed to be read from a database, it can be added as a new repository and require to bind in the AnimalMakerFactory module.

Repositories are injected into an Animal object at its constructor and is passed to AbstractAnimal constructor by calling the base constructor.

e.g

in Animal Class

public class Dog : AbstractAnimal

{

public Dog(IAnimalRepository animalRepository): base(animalRepository)

{

}

}

In AbstractAnimal Class

private readonly IAnimalRepository \_animalRepository;

protected AbstractAnimal(IAnimalRepository animalRepository)

{

\_animalRepository = animalRepository;

\_animalRepository.FeedProperties(this);

}

# DefaultRepository

Legacy repository with all the property values are hard coded. Need to update at each time a new Animal type is added. Can be used for testing but not in real time system.

# XmlRepository

This will fetch the property values from a XML file where the data file path can be defined in the consumer’s config file.

This gives a high level of extensibility & maintainability where as we can change the data file at any moment, add new animal types/properties without recompiling the plugin and change property values at any moment. (e.g if the DailyFeedCost value goes up, for all the animals, it’s just a matter of updating the data xml file without doing any change in the code.

# Building & Testing

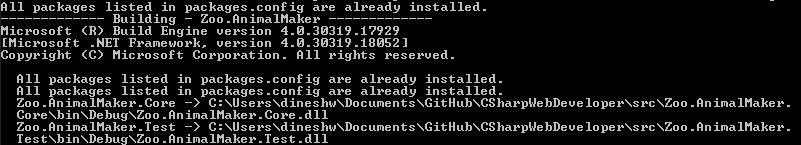
System can be build & test using the following Rake tasks.

# Building the solution

Command:

**rake build:build[debug]**

Result (in PowerShell )



# Run unit tests

Command:

**rake test:all**

Result (in PowerShell )

