Tweety Test Application

How to start the application

1. The solution is based on Visual Studio 2013 (if you use the Ultimate edition you can also open the Tweety.Model project containing the layer diagram)
2. Build the solution to allow the download of the NuGet packages
3. Set the Tweety.ConsoleUI as Start-up project
4. Run the solution

Architecture

The solution consists of the following components:

Contracts	It contains all the Interfaces (for dependency injection) and the Entity classes
ConsoleUI	It contains the console implementation of the user interface. As it represents the composition root of the application, it uses the Unity dependency injection framework to resolve the resolution root (using Convention over Configuration)
Controllers	It contains the logic to handle the user text commands, call the appropriate service classes to get the job done, format the result in a user readable way.
Logic	It contains the core service classes and the memory-based implementation of the repository

The following layer diagram shows the dependencies between the components of the solution.

There is weak logical dependency of ConsoleUI on Controllers and of Controllers on Logic but it is all abstracted by the DI Framework.

Note: You can find this Layer Diagram in the Tweety.Model project in the VS solution and you can use it to automatically validate the code against the defined architecture.

Tests

There are two test projects in the solution:

Tweety.UnitTests	Each test class tests a target class in isolation creating test doubles when the target class requires any dependency. The mocking framework used is Microsoft Fakes. The TDD approach has been based on this test project.
Tweety.SystemTests	The test classes contained by this project tests the System as a whole without any isolation. It has the responsibility for verifying that the components work well together and that the dependency injection configuration works properly.

Trade-off decisions

I developed this test app by keeping in mind the following objectives:

- Simple readable maintainable code
- Efficient code
- Extensibility
- TDD
- SOLID principles

However in some cases I had to take some trade-off decisions because those objectives were conflicting between them.

Trade-off decision 1 – Number of classes

16 classes (plus test classes) might seem too many for this simple application. However, using less classes would have broken the Single Responsibility principle and actually reduced the maintainability.

Trade-off decision 2 – Number of assemblies

4 assemblies (plus 2 test assemblies) might seem too many for this simple application. However the separation of the classes in different assemblies is essential to extensibility, to apply the Open/Closed principle and to avoid the Entourage anti-pattern.

Trade-off decision 3 – Number of tests

47 tests might seem too many for this simple application since their code need to be maintained in addition to the core code base. However, properly applying the TDD approach brought me to write many tests and they are in fact an insurance that allows refactoring and extensibility with a very low risk of regression.

Trade-off decision 4 – Interfaces with single implementation

Some of the interfaces have only one implementation. This makes the code a bit more complex and verbose but it also makes the solution more flexible and allows Dependency Injection.

Trade-off decision 5 – Single assembly for Interfaces

All the interfaces and entity classes of the solution are contained by a single assembly. Splitting them in multiple assemblies would have reduced the coupling between components because each component would depend only on the assemblies that contains the interfaces that they implement or depend on.

However that would have increased the complexity of the solution and reduce code readability.

Trade-off decision 6 – Command handling based on command classes and Regex

The handling of the user entered commands is based on a CommandHandler class and a set of command classes that implement ICommand. Each command exposes a regular expression signature that the handler uses to match the command and to parse the arguments.

This approach increases the complexity of the code but also increases the maintainability and allows for full extensibility. A new command can be added without even recompiling any of the assembly because it would just need to be registered in the configuration file.

Trade-off decision 7 – Repository class contained by Tweety.Logic assembly

Since the memory-based repository consists of just one class, creating a specific assembly for it seemed over engineering to me. For this reason I placed the UserRepository class inside the Tweety.Logic assembly.

Should the solution be using a different implementation of the repository in the future, the application would still depend on an assembly (Tweety.Logic.dll) that contains the memory-based implementation of the repository even if it is not used anymore.

However the memory-based implementation does not depends on any external libraries so keeping it inside the Tweety.Logic assembly is not too bad as it does not bring the Entourage anti-pattern.

Trade-off decision 8 – User entity relying on itself as a storage for navigation properties

The User entity defines the navigation properties “Timeline” and “Following” declared as IList<>.

The memory based implementation of the user repository initializes them with a concrete instance of List so the User objects are used, in a way, as a storage for the “Timeline” and “Following” collections.

This is ok for memory-based repository but a db based repository would need to retrieve “Timeline” and “Following” together with the User object resulting in inefficient use of network, storage and computational resources.

This has been, in a way, mitigated by declaring “Timeline” and “Following” as “virtual” so that ORMs such as Entity Framework would be able to apply some lazy loading logic to it. Anyway, extending the solution to add a non-memory based repository might result in facing some challenges.

A more extensible solution would have included a data context class containing the collections for Users and Stories and some navigation logic attaching to the properties instead of the concrete lists. However this would have resulted in over engineering and the requirements were specifically referred to a memory-based repository.