CodeCommander

File Name: **CodeCommander.docx**	**Pages: 10**
Created By: Ariel Ben Horesh	**Version: 1**
Updated By:	**Date: 7-Jun-11**

Revision History

Date	Version	Modified By	Modifications
7-Jun-11	1.0	Ariel Ben Horesh	First draft

Reference Documents

#	Document name	Document id.
1		
2		
3		

Glossary

Terms / Abbreviations	Definition
Rx	Reactive Extensions
CQRS	Command and Query Responsibility Segregation
BCL	Base Class Library

Table of Content

1 Overview

CodeCommander is a commands handling framework built on top of the Reactive Framework and ReactiveUI framework.

1.1 Purpose

1. Unified implementation of command handling behaviors between different domains.
2. Simple API for handling command execution, runtime behaviors and inputs.
3. Efficient command execution and filtering.
4. Supporting typical scenarios for commands: one-way, async, lifetime, limited, etc.

1.2 Description

Commands execution is a repetitive pattern that is commonly used within SOA.

Wikipedia defines Bertrand Meyer’s Command and Query Separation Principle (CQRS):

It states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, asking a question should not change the answer. More formally, methods should return a value only if they are referentially transparent and hence possess no side effects.

Wikipedia

In a nut shell every service exposes 2 sets of "abilities" each has it's own considerations.
In some domain contexts command execution may include various challenges that needs to be addressed such as command filtering, commands life time managements etc. CodeCommander introduce solid API for executing, tracking and controlling commands.

2 Code Route Top Level Design

2.1 General Class Diagram

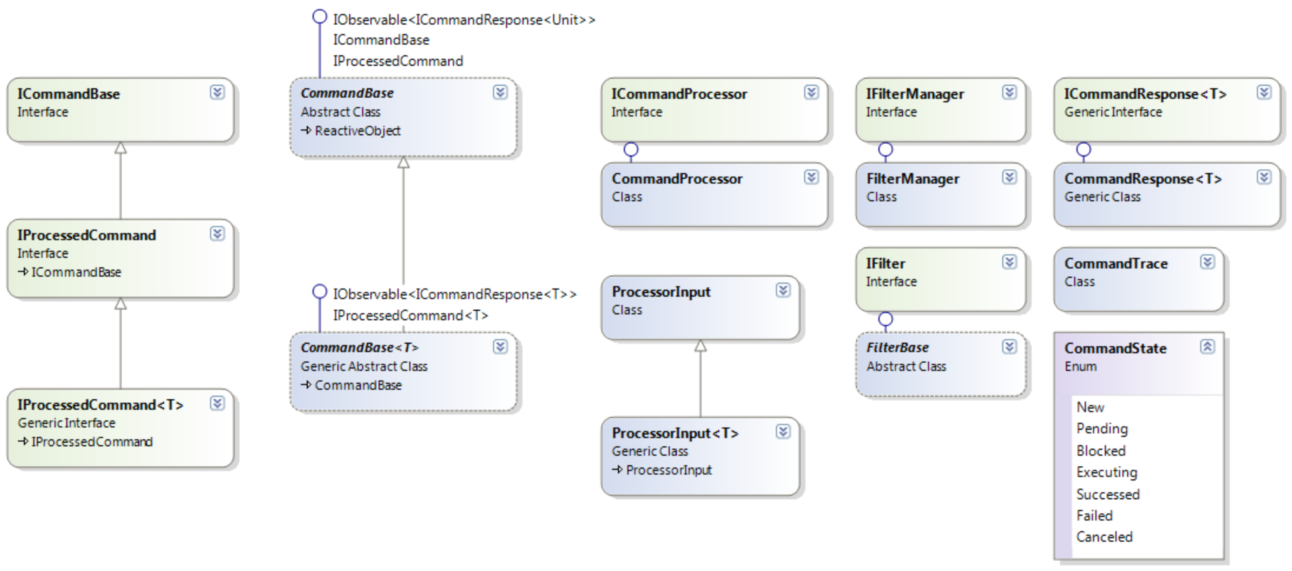

Figure 1 - Class Diagram

2.2 Main class description

2.2.1 Concepts

1. Rx interfaces: IObserver and IObservable

   Rx is a library for composing asynchronous and event-based programs using observable collections. For this task the Rx team had incorporated the essential Rx interfaces into the BCL.

```
public interface IObservable<out T>
  {
    IDisposable Subscribe(IObserver<T> observer);
  }

  public interface IObserver<in T>
  {
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
  }
```

   Both interfaces serve a complementary role. The IObservable<T> interface acts as a data source that can be observed, meaning it can send data to everyone who's interested to hear about it. Those interested parties are represented by the IObserver<T> interface.

In order to receive notifications from an observable collection, one uses the Subscribe method to hand it an IObserver<T> object. In return for this observer, the Subscribe method returns an IDisposable object that acts as a handle for the subscription. Calling Dispose on this object will detach the observer from the source such that notifications are no longer delivered. Similarities with the += and -= operators used for .NET events are not accidental, but the Subscribe method provides more flexibility. In particular, an unsubscribe action can result in quite some bookkeeping, all of which is handled by the Rx framework.

Observers support three notifications, reflected by the interface’s methods. OnNext can be called zero or more times, when the observable data source has data available. For example, an observable data source used for mouse move events could send out a Point object every time the mouse has moved. The other two methods are used to signal successful or exceptional termination.

2. Command Processing

   Command Processor is the API entry point for the whole framework. It works with commands which inherit from CommandBase, specif implementation of the commands are obviously Out of Scope for the framework and beside that it takes no other assumptions on the need to be executing commands. When a command is published an IObservable is returned and this is not by coincidence, the interface described above meets are need perfectly. OnNext is mapped for every fulfillment a command might receive. Note: that some commands may have successful fulfillments more than once during their lifetime. OnCompleted is called when the commands has finished itself successfully and it's leaving the framework, expect no further notifications for this command. The last one OnError is the less surprising, for whatever reason, a commands meets his end and didn't carry the task he was supposed to execute you will get the OnError notification.

   Command Processor is responsible for handling the execution of commands. For every new command it will try to execute it by first passing it through the IFilterManager (see below). The Command Processor also handles cancelation of command or groups of commands.

   One of the main command execution scenarios CodeCommander supports is the query commands, or just command that need some kind "synchronous" behavior such as the connect command, until it gets a confirmation that a connection has been made the command couldn't be sure it had fulfilled itself.
   For this reason the Command Processor also takes an IObservable as dependency in its constructor, this observable is used pass in input values, whenever OnNext is called the Command Processing will iterate through the executing commands and offer them a resolution with the input values.

3. Commands
   Commands are the 2nd piece of the puzzle, and arguably the most important, they contain the actual execution logic. Each command implements the following life cycle state machine.

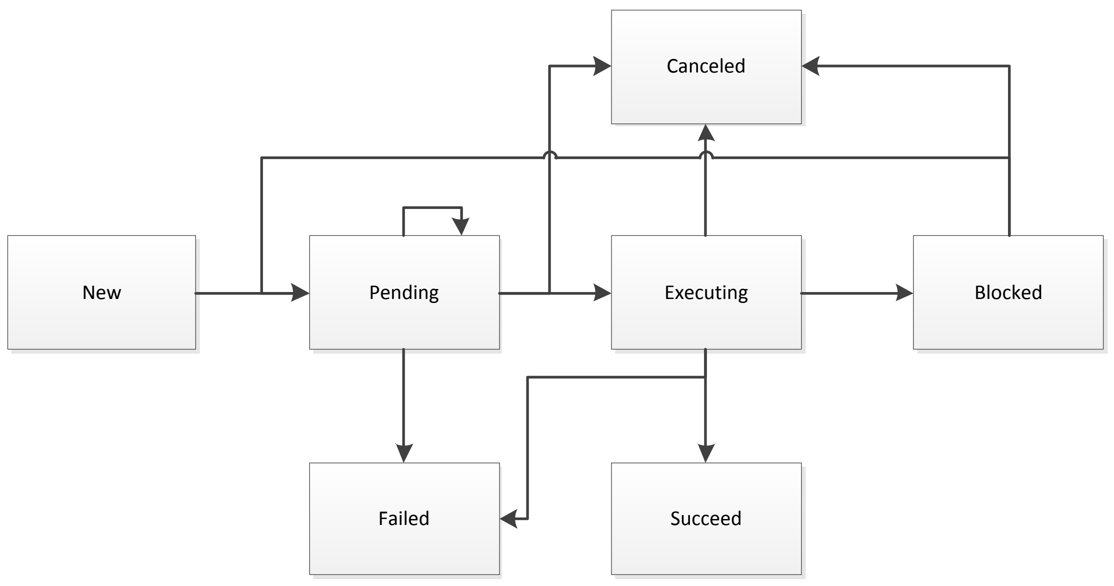

Figure 2 – Command state machine

Command states:

a. New: the basic initial state a command is born with.
b. Pending: On the moment a command is published into the Command Processor it earn the Pending state. Unless indicated differently a Command which has not passed the filter phase will remain at the Pending state, Command may fail if it has exceeded preset amount of time to remain pending or if it indicated it must fail if filtered out. Otherwise a command will begin executing.
c. Executing: When a command is executed the processor than calls the CanExecute method, CanExecute is handled by the user, if from any reason CanExecute returns false, than the command was rejected for execution by the client, it now moves the Blocked state.
d. Blocked: When a command has failed the CanExecute phase it has moved to the blocked state. It should remain in this state until the client calls the API to unblock a certain command, than it should pass again the pending state to start the execution phase all over.
e. Succeed: When the command finishes its purpose successfully it will transition itself to this state and will be regarded as completed by the framework.
f. Failed: When the command finishes its purpose unsuccessfully it will transition itself to this state and will be regarded as completed by the framework.
g. Canceled: The user may decide anytime during execution phase to cancel the command, the command will move to the cancel state and will be regarded as completed by the framework. Note: Cancelation is obviously logical, command might already been executed and it will just move to the Cancel state without waiting for a return input.

Commands are also configurable to support different execution behavior more on that below on the Configuration aspect.

4. Filters
Filters are simple constructs that respect ordering. When a command is pending it should pass through a chain of ordered filters, each consisting of a condition. Suppose Filters has

changed by the use in the middle of execution, it should be aborted and reentered into the chain.

5. Inputs
   As we have discussed above the Command Processor react to inputs introduced by the client. Input may be contain any arbitrary data to allow commands to intelligently resolve themselves by those inputs.
6. Configuration
   Commands support various behaviors through configuration:
   a. Should fail if filtered: indicate whether a command transitions itself to a failed state in case it didn't pass the filters chain.
   b. Pending Timeout: indicate how much a command can maintain its Pending state. If specified once exceeding the command will transition to the failed state.
   c. Executing Timeout: indicate how much a command can maintain its Executing state. If specified once exceeding the command will transition to the failed state.
   d. Should execute forever: Command indicating this should never complete. It will remain in execution until canceled or end of the process.

2.2.2 Interfaces

1. ICommandBase
   The most basic fundamental piece every command need to provide, the ability to specify what it needs to do upon executing. The interface also let the command implement a CanExecute to determine the command execute in both the command and the framework agreement.

2. IProcessedCommand
   A relatively complicated interfaced to represent what a command needs to implement in order to be used within the framework. Usually every command would inherit from CommandBase, this should reduce to a minimum complexity.

3. IProcessedCommand<T>
   This interface let us specify a return value, this is especially good for commands which should produce results, such as "GetValue" commands or even commands that listen for changes in lower layers. T is the type of return value.

4. ICommandProcessor
   This interface specifies the API for the command processing class, the part responsible for handling command execution and queueing. It also used to cancel commands in process.
5. IFilterManager
6. This interface specifies the API for the filter management class, the part responsible for handling command filtering, filtering is a concept that is enabling commands to be published out of order and still respect certain states where the commands are suited for execution. For example while one command is executing it might be requirement and no other command should execute simultaneously with it. Filters are a way to apply this restriction.

7. IFilter
   Every class implement IFilter is a candidate to restrict execution of commands within the framework.
8. ICommandResponse: When command fulfill itself it return a command response so it allows easy tracking of the fulfilled command and its return value.

2.2.3 Classes

1. CommandBase and CommandBase<T>
   This is the base class serving for all commands handled by the CommandProcessor, it consist of handling the state machine life cycle of a command, and calling various hooks, which the client may use to call different behaviors.
   CommandBase<T> indicate there is a return value for the command.

2. CommandPorcessor: the class responsible for handling command publishing, handling the commands queue for pending commands and pushing a command through the Filter chain.

3. ProcessorInput and ProcessurInput<T>
   Those are the inputs the CommandProcessor receive so it can iterate through the commands in hand to offering them a chance to resolve themselves.

4. FilterManager
   Implements the Filters management, Adding and removing of filters and checking whether a command can pass the filter chain.

5. FilterBase
   Simple base class, containing common implementation of a filter for derived filters.

6. CommandResponse<T>
   When a command fulfill itself it returns an instance of this class so it carries information for the client, such as the command information and return values.

7. CommandTrace
   Commands track their own state changes. Every time a command transition it track it down for usage by client.

2.2.4 Enumerations

1. CommandState
   Indicates the valid states a command can reside in.

2.3 Using the framework

- In order to use the framework, a client need to provide implementations for Commands and Filters. It is possible to derive from CommandBase or CommandBase<T>. Filters usually derive from FilterBase.
- The client need to provide an IFilterManager, it is recommended to use the built in FilterManager the framework provides but any implementation is suitable.
  The client may add and remove filters as it see fits throught out the life time of the command processing.
- Setting up an Observer to provide inputs for the CommandProcessor.
- The client need to create an ICommandProcessor. It is recommended to use the build in CommandProcessor the framework provides. The default implementation require inputs observer and a FilterManager.
- You are all set to go. Create an instance of a command and publish it to the CommandProcessor.
- Register for notifications with the Observerable returned by the publish, and keep track of your command.

2.4 Summary

CodeCommander is an efficient, lightweight command executing framework, it is reactive and handles different kind of scenarios through commands behavior configuration and filters.