N-Way Set Associative Cache

# Overview

This document was created to give an overview of the **N Way Set Associative Cache** for other developers and technical teams. Here we cover insight to the design decisions and usage specific features to allow new developers to come up to speed on the amazing software created by ***The Trade Desk*** developers.

# What is an N-Way Set Associative Cache?

It’s important to understand this particular implementation of an **N Way Set Associative Cache** as this data structure is very common in hardware level software where you are implementing at the Bits and Bytes level. Most modern day memory management system in fact use these principles at the very lowest level by manipulating bits within a single byte.

However, at the application level, our implementation is built at a much higher level using existing .NET Framework constructs. Unfortunately, there is no out of the box solution for a **N Way Set Associative Cache**, but great software is used as building blocks and the *.NET Framework* is packed with useful classes to extend. In our case we will use a generic ***Dictionary*** that is used to contain the actual cache data. A ***LinkedList*** will be used to maintain a linked list to be used by the eviction policy. This could have been implemented using other constructs but this seems to be a simple and easy to understand code base that can be modified as required with little or no difficulty.

# Requirements

### In Memory

Our cache implementation is entirely in memory currently. If this requirement changes we can easily create a distributed version my extending the ***IAssociativeCache*** interface with a custom implementation with little or no code changes to application level code.

### Type Safe Cache

The Cache is 100% type safe that is determined at instantiation for ultimate flexibility for the developer to store any type of data.

### Extendable and Distributable by Clients

The Cache is interfaced based as to create the opportunity for third party developers to write their own implementations. To extend the interface a developer would include a reference to the Cache DLL and implement a couple methods and be ready to go with no coding changes required at the application level code.

### User Data Use Case Implementation

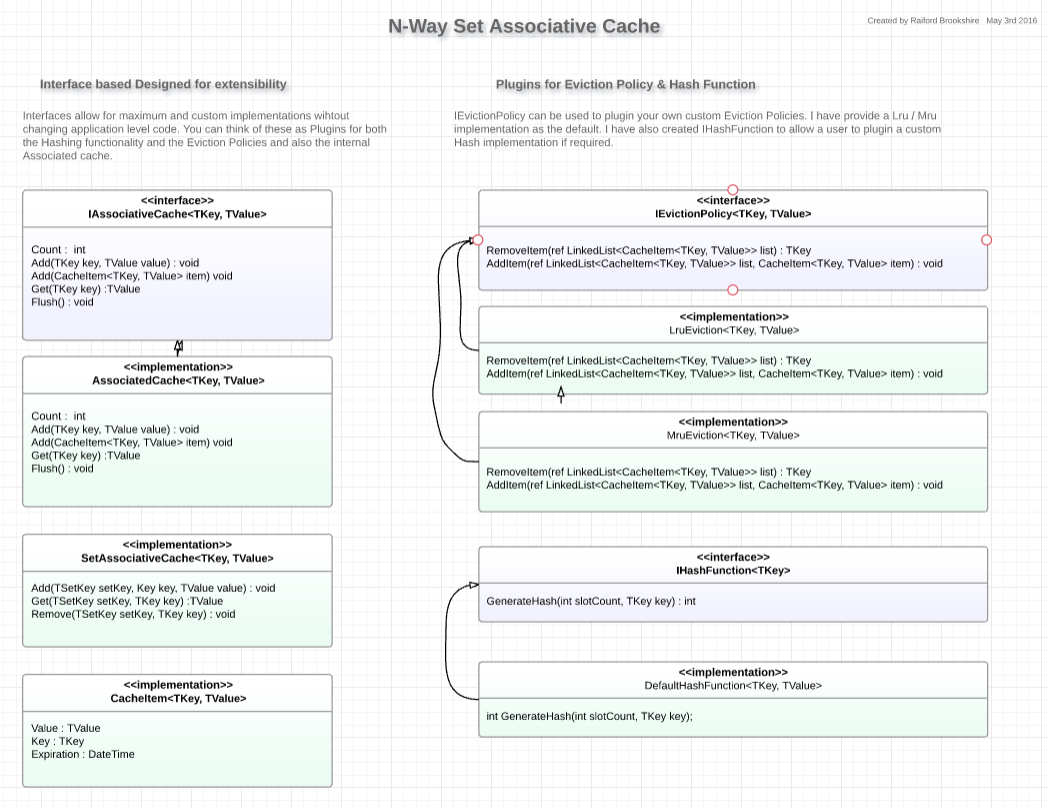
Provided is a Unit Test that with 2 main tests that test the following:

* **AssociativeCache**
* **SetAssociativeCache**

The Unit tests are easy to understand and test the majority of the expected functionality using Asserts along the way to test all assumptions and expectations. These unit tests are made to be included in a build so any ‘refactoring’ issues will be identified immediately and will need to be fixed before a deployment. This is a very good thing once the code base builds over the life of the software

# Interface & Class Design

The Caching classes consist of several interfaces. The design decision was made to make public the interfaces for both the hashing function and the replacement algorithm or Eviction Policy. This creates a very flexible interface based design to “Plug-In” various implementations as required.



# Visual Studio Solution

I have created the following projects:

##### TradeDesk.Caching

Contains all the Caching specific classes. This is the only class to distribute to client developers

##### TradeDesk.Caching.Tests

Contains a single test class with 2 tests that test both the Associative Cache and Set Associative Cache. This class is not distributed to client developers

##### TradeDesk.Models

Contains simple object model that can be used for testing purposes. This class would be required to run the unit tests.

# Additional Ideas or Implementations

Just some ideas that can be implemented as required

* **Expiration Time** – I added the required properties in the ***CacheItem***, but a useful implementation could easily be provided to allow items to ‘expire’ from the cache.
* **Dependency Injection** can be used in a large scale application to inject the implementations of choice for the cache.
* **Thread safety –** Currently the classes use the “Synchronized” .NET Attribute to force the method into a single thread. This should work, but for a little extra, this can implement a more advance locking strategy such as critical sections or other of the many .NET blocking constructs.
* **Production Ready** – I would want to fully test the cache in a complete application using real data to ensure the functionality is as expected. 99% of the time a well written Unit Test will be sufficient, but extended testing is a must before going live.