ASSIGNMENT 3 – Solution Set

Question1:

B:

Part 1:

a determines p when : b = true and c = false
b determines p when: a = false and b =false
c determines p when: a = false and b = true

Part2:

a determines p when: not matter the values of b and c
b determines p when: a can be anything and c= false
c determines p when: a can be anything and b = false

C:

	A	B	C	P1	P2
1	T	T	T	T	T
2	T	T	F	T	T
3	T	F	T	T	T
4	T	F	F	T	F
5	F	T	T	T	F
6	F	T	F	F	F
7	F	F	T	T	F
8	F	F	F	7	T

D:

Part1:

For a = <2,6>
For b = <6,8>
for c = <5,6>

Part2:

For a = <1,5><1,6><1,7><1,8><2,5><2,6><2,7><2,8><3,5><3,6><3,7><3,8><4,5><4,6><4,7><4,8>

For b = <2,4><6,8><2,8><6,4>
For c = <3,4><7,8><3,8><4,7>

E:

Part1:

For a = <2,6>
For b = <6,8>
For c = <5,6>

Part2:

For a = <1,5><1,6><1,7><2,5><2,6><2,7><3,5><3,6><3,7><4,8>
For b = <2,4><6,8>
For c = <3,4><7,8>

F:

Part1:

For a = <2,6>
For b = <6,8>
For c = <5,6>

Part2:

For a = <1,5><2,6><3,7><4,8>
For b = <2,4><6,8>
For c = <3,4><7,8>

Question 2:

A:

Short-circuit evaluation does not effect active clause coverage since ACC allows to pick the particular values of the minor clauses, and therefore we can always pick values that ensure that each major clause is evaluated.

To illustrate this, consider p = a or b . If a is the major clause, then for a to determine p, clause b has to be false. This would imply that ACC would impose the test requirements {T, F} and {F, F}. For the test case {T, F}, the clause b does not need to be considered since its a minor clause and p is still being determined by major clause a.

B:

PC: not affected since each clause is not evaluated n the coverage.
CC: there are illogical test suites since short-circuited evaluation affects whether or not a particular clause is evaluated, and CC requires each clause to be evaluated to both T and F.
CoC: there are illogical test suites since CoC requires every clause to be evaluated in all possible combinations.
GACC, CACC and RACC are from the family of Active Clauses. For the same reason mentioned in part “A” there coverages are not affected.

C:

Example:

P = (a AND (NOT a)) OR b

b is never evaluated.

Question 3: (Adopted from B. T. Wilkinson)

We first find the conditions under which c_1 determines p.

d_{c_1}

$= p_{c_1=true} \oplus p_{c_1=false}$

$= (true \wedge p') \oplus (false \wedge p')$

$= p' \oplus false$

$= p'$

So, we have that c_1 determines p if and only if p' is true. In order to satisfy GACC with respect to c_1, T must then contain two test cases with p' set to true. One of these test cases must set c_1 to true and the other must set c_1 to false. Thus, we have:

$\{(c_1 : true, p' : true), (c_1 : false, p' : true)\} \subseteq T$

Now we consider whether or not *T* satisfies CACC with respect to c_1. The test case $(c_1 : true, p' : true)$ results in p being true. The test case $(c_1 : false, p' : true)$ results in p being false. Since c_1 determines p in both of these test cases and p evatuates to both true and false, *T* satisfies CACC with respect to c_1.

Question 4:

Part a)

Any software application can exhibit a set of problems from user perspective. In a case of Ehcache, the problems can be divided to the problem in using, learning, customizing, and installing the system. Some these problems are:

- Configuring the Ehcache with appropriate parameter to maximize its performance

- Difficulty in using the user's manuals and API documentations

- Installation problems due to its dependency to other packages and applications

- Problems due to the Ehcache's architecture and design. Such problems usually affect the non-functional aspects of a system. For example: serialization, and the Ehcache's design to use disk as secondary cache storage.

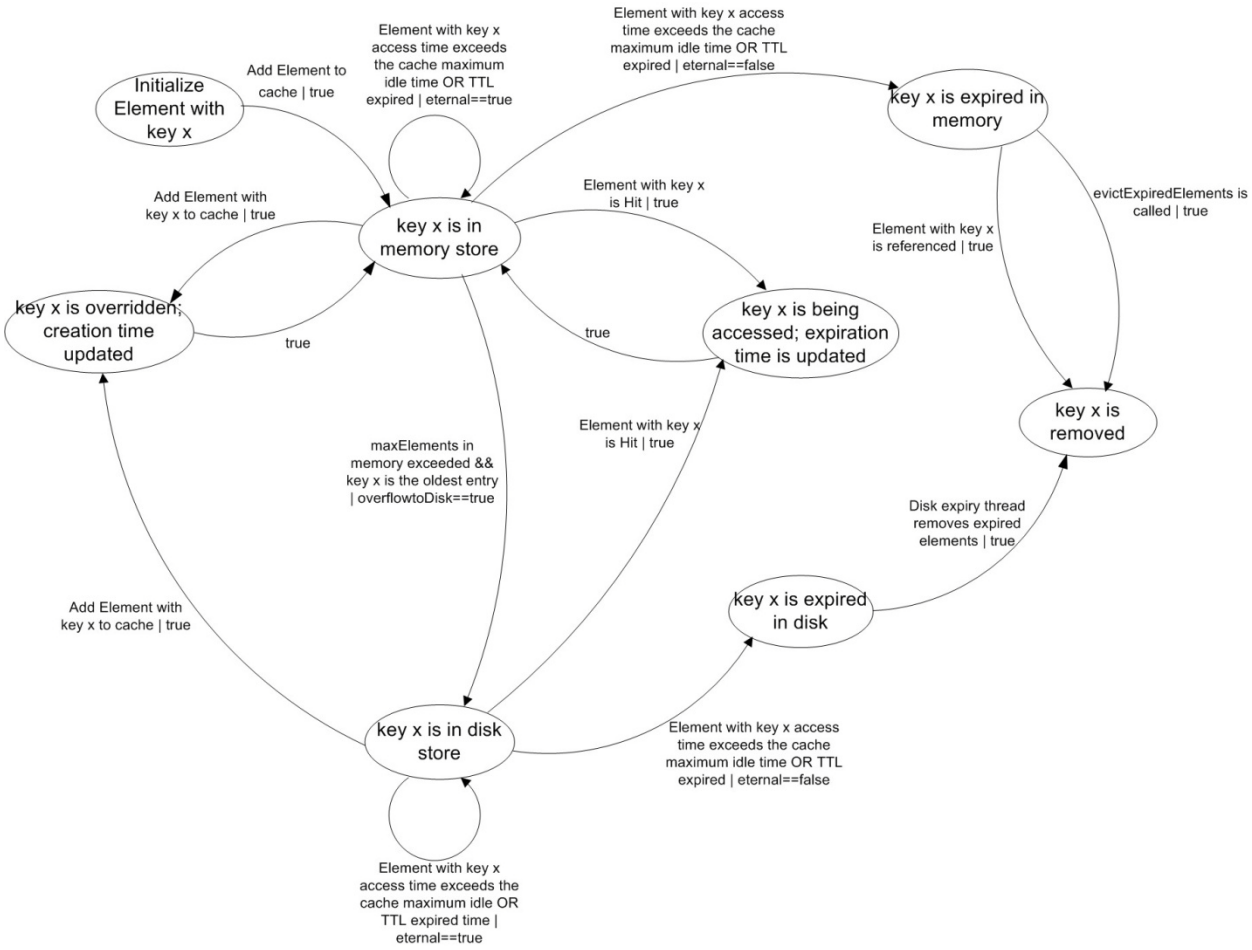
Anyone can think of other problems based on his/her experience. However, we can only accept those bad experiences as problems if they do not contradict with the original purpose of using encache, which is providing a cache service in this case. As an example, using system's free memory to store cached data is not an acceptable problem!

Part b)

FSMs can vary based on their abstraction level. Here, I selected a sample FSM which has a reasonable level of detail.

Some students discarded secondary memory (disk) in their FSM model. I accepted these FSM models, as far as these types of assumptions were explicitly mentioned, and the systems were set up with appropriate configuration variables to disable the disk chache.

Sample FSM: (Adopted from Noor Mohiuddin)



Part c) (Adopted from Noor Mohiuddin)

The test cases are adopted for the FSM model of part C. The ehcache should be configured with appropriate values to facilitate the test process. The reason for choosing such low values was to make testing easier. The JUnit tests provide statement (node) coverage for all the code that is executed according to the FSM. Tests were provided for each state as well as transitions from the states.

```
<ehcache>
  <diskStore path="java.io.tmpdir" />
  <defaultCache maxElementsInMemory="3" eternal="false" timeToIdleSeconds="5"
    timeToLiveSeconds="10" overflowToDisk="true" maxElementsOnDisk="4"
    diskPersistent="false" diskExpiryThreadIntervalSeconds="2"
    memoryStoreEvictionPolicy="LRU" />
</ehcache>
```

The test cases are attached to the end of this document.

Bonus)

Based on the submitted solutions, there were several interesting suggestions for improving Ehcache test suite. Few of these suggestions are:

- Concurrency testing in the suite

- Scalability testing (including load-testing)

- Add profiles for performance testing. This can help identifying bottlenecks in system performance.

- Integrate some testing frameworks to improve a systematic and automatic testing process.

- Portability Testing (e.g. test the system on various hardware and Operating systems)

-Non-functional (quality) testing.

```
// Assignment 3
// Question 4, Part C
// Additional test cases for Ehcache
// By: Noor Mohiuddin

package net.sf.ehcache;

import net.sf.ehcache.*;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import java.io.Serializable;
import org.junit.*;

public class EhcacheTest implements Serializable {

    private static final long serialVersionUID = 1L;
    CacheManager c_manager;
    Cache cache;

    //Initialise the cache manager and cache
    @Before
    public void createCache(){
        c manager = CacheManager.create();
        c manager.addCache("Cache");
        cache = c_manager.getCache("Cache");
    }

    ////////////////////////////////////////////////Initialisation
    Test//////////////////////////////////////////////////
    //Initialise element with key x = 1 and add to cache
    @Test
    public void testElement(){
        Element e1 = new Element("key1","value1");
        cache.put(e1);
        assertEquals(cache.get("key1"),e1);
    }
    ////////////////////////////////////////////////Memory
    Tests//////////////////////////////////////////////////
    //Test if the element is transferred from memory to disk when memory full
    and key 1 is oldest
    @Test
    public void testEvictToDisk(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        cache.put(e1);                                    //timeToIdle is 5
        seconds
        delay(1000);
        cache.put(e2);
        cache.put(e3);
        cache.put(e4);
        assertTrue(cache.getDiskStore().containsKey("key1"));
    }

    //Test if the element is transferred from memory to disk when memory full
    and key 1 is not oldest
    @Test
    public void testNotEvictToDisk(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        cache.put(e2);                                    //timeToIdle is 5
        seconds
```

```
        delay(1000);
        cache.put(e1);
        cache.put(e3);
        cache.put(e4);
        assertFalse(cache.getDiskStore().containsKey("key1"));
    }

    //Ensure the expiration time is renewed when the element with key 1 is hit
    @Test
    public void testMemoryHit(){
        Element e1 = new Element("key1","value1");
        cache.put(e1);
        delay(1000);
        //Time left to idle reduced
        e1 = cache.get("key1");
        //Hit
        long timeLeftToIdle = (e1.getExpirationTime()-
        e1.getLastAccessTime()); //Calculate new time left to idle
        assertTrue(timeLeftToIdle==5000);
        //Time to idle is 5s
    }

    //Ensure the creation time is updated (hence TTL is renewed) when element
    with key 1 overridden
    @Test
    public void testMemoryOverride(){
        Element e1 = new Element("key1","value1");
        cache.put(e1);
        long oldCreationTime = cache.get("key1").getCreationTime();
        delay(1000);
        cache.put(new Element("key1","valuex"));
        long newCreationTime = cache.get("key1").getCreationTime();
        assertFalse(newCreationTime==oldCreationTime);
    }

    //Ensure the status of the element with key 1 is expired when timeToIdle
    expires and eternal is false
    @Test
    public void testExpired(){
        Element e1 = new Element("key1","value1");
        cache.put(e1);
        delay(6000);
        //Time to idle is 5s
        assertFalse(cache.getKeysWithExpiryCheck().contains("key1"));
    }

    //Ensure the status of the element with key 1 is not expired when
    timeToIdle expires and eternal is false
    @Test
    public void textExpiredEternal(){
        Element e1 = new Element("key1","value1");
        e1.setEternal(true);
        //Eternal is set
        cache.put(e1);
        delay(6000);
        //Time to idle is 5s
        assertTrue(cache.getKeysWithExpiryCheck().contains("key1"));
    }

    //Ensure the status of the element is expired when TTL expires regardless
    of hits
    @Test
    public void testExpiredTTL(){
        Element e1 = new Element("key1","value1");
        cache.put(e1);
        for(int i=0;i<=10;i++){
            e1 = cache.get("key1");
```

```
            //Keep hitting key 1 to renew idle time
            delay(1000);
        }
        assertFalse(cache.getKeysWithExpiryCheck().contains("key1"));
    }

    //Ensure the status of the element is not expired when TTL expires and
    eternal is true regardless of hits
    @Test
    public void testExpiredTTLEternal(){
        Element e1 = new Element("key1","value1");
        e1.setEternal(true);
        cache.put(e1);
        for(int i=0;i<=10;i++){
            e1 = cache.get("key1");
            //Keep hitting key 1 to renew idle time
            delay(1000);
        }
        assertTrue(cache.getKeysWithExpiryCheck().contains("key1"));
    }

    //Ensure an expired element with key 1 is removed completely from memory
    when accessed
    @Test
    public void testRemoveWhenAccessedMemory(){
        Element e1 = new Element("key1","value1");
        cache.put(e1);
        delay(6000);
        //Time to idle is 5s
        e1 = cache.get("key1");
        assertFalse(cache.getKeys().contains("key1"));
    }

    //Ensure an expired element with key 1 is removed completely from memory
    when evictExpiredElements() is called
    public void testRemoveWhenEvictCallMemory(){
        Element e1 = new Element("key1","value1");
        cache.put(e1);
        delay(6000);
        //Time to idle is 5s
        cache.evictExpiredElements();
        assertFalse(cache.getKeys().contains("key1"));
    }


    ////////////////////////////////////////////Disk
    Tests////////////////////////////////////////////////////
    //Ensure element with key x is returned to memory and expiration time
    reset when accessed
    @Test
    public void testDiskHit(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        cache.put(e1);
        delay(1000);
        //Time left to idle reduced
        cache.put(e2);
        cache.put(e3);
        cache.put(e4);
        e1 = cache.get("key1");
        //Hit
        long timeLeftToIdle = (e1.getExpirationTime()-
        e1.getLastAccessTime()); //Calculate new time left to idle
        assertTrue(cache.isElementInMemory("key1") && timeLeftToIdle==5000);
        //Time to idle is 5s and element should be in memory
```

```
    }

    //Ensure element with key x is contained in memory when the element is
    overridden as well as TTL renewed
    public void testDiskOverride(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        cache.put(e1);
        long oldCreationTime = cache.get("key1").getCreationTime();
        delay(1000);
        cache.put(e2);
        cache.put(e3);
        cache.put(e4);
        cache.put(new Element("key1","valuex"));
        long newCreationTime = cache.get("key1").getCreationTime();
        assertFalse(newCreationTime==oldCreationTime);
    }

    //Ensure that element with key 1 is expired from the disk when timeToIdle
    Expires and eternal is false
    @Test
    public void testExpiredFromDisk(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");

        cache.put(e1);
        delay(2000);                                    //timeToIdle is 5
        seconds
        cache.put(e2);                                  //overflow the memory
        cache.put(e3);
        cache.put(e4);
        delay(4000);                                    //Total time is now 6s
        assertFalse(cache.getKeysWithExpiryCheck().contains("key1"));
    }
    //Ensure that element with key 1 is not expired from the disk when
    timeToIdle Expires and eternal is true
    @Test
    public void testExpiredFromDiskEternal(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        e1.setEternal(true);
        cache.put(e1);
        delay(2000);                                    //timeToIdle is 5
        seconds
        cache.put(e2);                                  //overflow the memory
        cache.put(e3);
        cache.put(e4);
        delay(4000);
        assertTrue(cache.getKeysWithExpiryCheck().contains("key1"));
    }

    //Ensure that element with key 1 is expired from the disk when TTL Expires
    and eternal is false
    @Test
    public void testExpiredDiskTTL(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        cache.put(e1);
        for(int i=0;i<=8;i++){
```

```
            e1 = cache.get("key1");                                    //Keep
            hitting key 1 to renew idle time
            delay(1000);
        }
        cache.put(e2);
        //overflow the memory and disk
        cache.put(e3);
        cache.put(e4);
        delay(2000);                                                   //this
        will ensure TTL expires before idle time
        assertFalse(cache.getKeysWithExpiryCheck().contains("key1"));
    }

    //Ensure that element with key 1 is not expired from the disk when TTL
    Expires and eternal is true
    @Test
    public void testExpiredDiskTTLEternal(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        e1.setEternal(true);
        cache.put(e1);
        for(int i=0;i<=8;i++){
            e1 = cache.get("key1");                                    //Keep
            hitting key 1 to renew idle time
            delay(1000);
        }
        cache.put(e2);
        //overflow the memory and disk
        cache.put(e3);
        cache.put(e4);
        delay(2000);                                                   //this
        will ensure TTL expires before idle time
        assertTrue(cache.getKeysWithExpiryCheck().contains("key1"));
    }
    //Ensure that the disk expiry thread will remove expired elements
    @Test
    public void testRemovedByThread(){
        Element e1 = new Element("key1","value1");
        Element e2 = new Element("key2","value2");
        Element e3 = new Element("key3","value3");
        Element e4 = new Element("key4","value4");
        cache.put(e1);
        for(int i=0;i<=8;i++){
            e1 = cache.get("key1");                                    //Keep
            hitting key 1 to renew idle time
            delay(1000);
        }
        cache.put(e2);
        //overflow the memory and disk
        cache.put(e3);
        cache.put(e4);
        delay(2000);                                                   //this
        will ensure TTL expires before idle time
        delay(2000);
        //diskExpiryThreadInterval = 2s
        assertFalse(cache.getKeys().contains("key1"));
    }


    public void delay(int i){                                          //This
    method makes the thread sleep for i seconds
        try{
            Thread.sleep(i);
        }catch (Exception e){
            System.out.println(e);
```

```
        }
    }
}
```