Project – Unit 3 - Making your app (like kbb - Kelly Blue Book) scalable

Reflection Questions on Unit 2

You should review following questions to make sure you understand the outcomes from Unit 1 and document lessons learned for submission (with final unit of Car Configuration Application). You do not submit these questions for grading.

1. What role(s) does an interface play in building an API?
2. What is the best way to create a framework, for exposing a complex product, in a simple way and at the same time making your implementation extensible.
3. What is the advantage of exposing methods using different interfaces?
4. Is there any advantage of creating an abstract class, which contains interface method implementations only?
5. How can you create a software architecture, which addresses the needs of exception handling and recovery?
6. What is the advantage of exposing fix methods for exception management?
7. Why did we have to make the Automobile object static in ProxyAutomotive class?
8. What is the advantage of adding choice variable in OptionSet class? What measures had to be implemented to expose the choice property in Auto class?
9. When implementing LinkedHashMap for Auto object in proxyAuto class, what was your consideration for managing CRUD operations on this structure? Did you end up doing the implementation of CRUD operation in proxyAuto or did you consider adding another class in Model for encapuslating Auto for the collection and then introducing an instance of this new class in proxyAuto. (Think about this and if this part of your design is not self-contained, then fix it.)

Requirements (Written by someone in Marketing)

This project needs more work to make sure it's functioning correctly and to ensure that it does the right thing when there are multiple users. Here is a scenario to consider:

How will the system scale if multiple users were to update the same model, same OptionSet or perhaps the same Option value.

Plan of Attack

To ensure that your application can handle concurrency, we have to add new functionality in new classes without modifying our existing code too much.

What I am saying is that proxyAutomotive and BuildAuto (combined with interfaces) can be kept intact. Our Model package containing Automobile, OptionSet and Option classes is representing the data for one Model.

If we want to setup a scalable structure for the app, we can add a LinkedHashMap in proxyAutomotive (abstract class) with some simple API's to modify it (we can always add more methods in interfaces at any time).

We want to add a feature that will improve concurrency which can be done as follows:

1. Design a class called EditOptions that can be used to edit Options for a given model in its own thread. You should put this class in its own package called scale.

    a. Be sure to use the LinkedHashMap instance of Automobile (that is a static object) in proxyAutomotive class instance.
    b. Consider synchronizing methods in classes that are used in setup of LinkedHashMap instance of Automotive ((that is a static object) in proxyAutomotive class instance). You will have to synchronize all methods, used for creating, reading, updating and deleting parts of Automobile, OptionSet and Option classes.

2. Now code the new driver class and test it as follows:
    a. Create a driver program for this unit that allows:
        i. Instantiation of BuildAuto.
        ii. Create two threads using EditOptions, which will modify the same model as LinkedHashMap instance of Automobile (static object) in proxyAutomotive class instance.
    b. Test your implementation to ensure that two threads altering same property do not cause data corruption. (use Goofy.java semantics for this step).

3. Your deliverable for this part:
    a. Updated classes and class diagram
    b. Test program showing the successful implementation of these classes.

Design Considerations

Consideration 1

You have to synchronize methods in Model package. Since methods in OptionSet and Option class are protected, you don't really have to synchronize those methods, since each can be accessed through Automotive only. So, Automobile class methods will need to synchronized.

Consideration 2

You have to use instance of Automobile LinkedHashMap in proxyAutomotive (abstract class) in EditOptions class. What is the best way of enabling this interaction?

Following design could be possibly used:

1. Create an API to expose ProxyAutomotive class through BuildAuto and an Interface. Methods should be accessible internally, providing access to the Automobile LinkedHashMap instance. EditOptions class and BuildAuto class will be associated with each other through an API (Interface)
   (This exercise will help you learn how to design Interfaces. You will need to design this part of the exercise without help from anyone.)

2. EditOptions should use synchronized methods in the Automobile class to operate on Automobile LinkedHashMap instance in proxyAutomotive.

(Please analyze carefully, as to where to put synchronize methods and implement. Is it better to put in Auto class, or class that contains CRUD operations for proxyAuto or EditOptions class. You need to justify your implementation in comments added to your code).

Consideration 3

In this lab, I only expect you to change the existing value(s) of an OptionSet and Options. Please do not overwhelm yourself with coding all cases, since our goal is to learn multithreading.

Resources

Example showing how to share an object with n threads - synchronized:

```
class Program6 extends Thread {
    static String[] msg = {
"Example","of","how","messy","Java","is","without","synchronization" };

    public static void main(String[] args) {
        Goofy t1 = new Goofy("t1: ");
        Goofy t2 = new Goofy("t2: ");

        t1.start();
        t2.start();

        boolean t1IsAlive = true;
        boolean t2IsAlive = true;

        do {
            if(t1IsAlive && !t1.isAlive()) {
                t1IsAlive = false;
                System.out.println("t1 is dead.");
            }

            if(t2IsAlive && !t2.isAlive()) {
                t2IsAlive = false;
                System.out.println("t2 is dead.");
            }
        } while(t1IsAlive || t2IsAlive);
    }

    public Goofy(String id) {
        super(id);
    }

    void randomWait() {
        try {
            Thread.currentThread().sleep((long)(3000*Math.random()));
        } catch(InterruptedException e) {
            System.out.println("Interrupted!");
        }
    }

   public void run() {
        synchronized(System.out) {
            for( int i=0; i<msg.length; i++ ) {
                randomWait();
                System.out.println(getName() + msg[i]);
            }
        }
    }
}
```

Example of implementation of wait and how to notify between two threads with synchronized:

```
class goofy {
      public static void main(String [] a211)
      {
            coffee a1 = new coffee();
            cgoofy a3 = new cgoofy(a1);
            a3.start();
            try {
                  Thread.sleep(2000);
            } catch (Exception e) { }
            pgoofy a2 = new pgoofy(a1);
            a2.start();
      }
}
```

```
class pgoofy extends Thread {
      coffee a1;
      pgoofy(coffee e)
      {
            a1 = e;
      }
      public void run()
      {
            a1.put(10);
            System.out.println(a1.contents);
      }
}

class cgoofy extends Thread {
      coffee a1;
      cgoofy(coffee e)
      {
            a1 = e;
      }
      public void run()
      {
            System.out.println("consumer" + a1.get());
      }
}

class coffee {
boolean available=false;
int contents;

//indicating there nothing to get.
//waiting on each other.
public synchronized int get() {
    while (available == false) {

        try {
            // wait for Producer to put value
            wait();
        } catch (InterruptedException e) {
        }
    }
    available = false;
    // notify Producer that value has been retrieved
    notifyAll();
    return contents;
}
public synchronized void put(int value) {
    while (available == true) {
        try {
            // wait for Consumer to get value
            wait();
        } catch (InterruptedException e) {
        }
    }
    contents = value;
    available = true;
    // notify Consumer that value has been set
    notifyAll();
 }
}
```

Example of how to implement wait and notify between two threads with synchronized (a practical implementation):

```
package examples.threads.bank;
/** A class to demonstrate wait and notify methods
  */
public class BankAccount {
   private int balance = 0;
   private boolean isOpen = true;
   /** The method withdraws an amount from the
     * account. If funds are insufficient, it will
     * wait until the funds are available or the
     * account is closed.
     * @param amount The amount to be withdrawn from
     *    the account
     * @return true if the withdrawal is successful,
     *    false otherwise
     * @exception InterruptedException If another
     *    thread calls the <b>interrupt</b> method
     */
   public synchronized boolean withdraw( int amount )
                throws InterruptedException {
      while ( amount > balance && isOpen() ) {
         System.out.println( "Waiting for "
                            + "some money ..." );
            wait();
      }
      boolean result = false;
      if ( isOpen() ) {
         balance -= amount;
         result = true;
      }
      return result;
   }
   /** The method to deposit an amount into the
     * account provided that the account is open.
     * When the deposit is  successful, it will notify
     * all waiting operations that there is now more
     * money in the account
     * @param amount The amount to be deposited into
     *    the account
     * @return true if the deposit is successful,
     *    false otherwise
     */
   public synchronized boolean deposit( int amount ) {
      if ( isOpen() ) {
         balance += amount;
         notifyAll();
         return true;
      } else {
         return false;
      }
   }
   /** Check to see if the account is open
     * @return true if it is open, otherwise false
     */
   public synchronized boolean isOpen() {
      return isOpen;
   }
   /** Close the bank account */
   public synchronized void close() {
      isOpen = false;
      notifyAll();
   }
}

package examples.threads.bank;

/**
  * A class to demonstrate wait and notify methods
  */
public class Banking {
   /** The test method for the class
     * @param args[0] Time in seconds for which
     *    this banking process should run
     */
   public static void main( String[] args ) {
      BankAccount ba = new BankAccount();

      // create the spender thread
      Spender spenderThread = new Spender( ba );

      // create the saver thread which is a two-step
      // process because Saver implements Runnable
      Saver aSaver = new Saver( ba );
      Thread saverThread = new Thread( aSaver );

      spenderThread.start();
      saverThread.start();

      int time;
      if ( args.length == 0 ) {
         time = 10000;
      } else {
         time = Integer.parseInt( args[0] ) * 1000;
      }
      try {
         Thread.currentThread().sleep( time );
      } catch ( InterruptedException iex ) {
         /* ignore it */
      }
      // close the bank account
      ba.close();
   }
}

package examples.threads.bank;
/**
  * A class to demonstrate wait and notify methods
  */
public class Saver implements Runnable {
   private BankAccount account;
   /** Class constructor method
     * @param ba The bank account where this saver
     *    puts the money
     */
   public Saver( BankAccount ba ) {
      account = ba;
   }
   /** The method the saver uses to put away money */
   public void run() {
      while( account.isOpen() ) {
         try {
            if ( account.deposit( 100 ) ) {
               System.out.println(
               "$100 successfully deposited." );
            }
            Thread.currentThread().sleep( 1000 );
         } catch ( InterruptedException iex ) {
            // display the exception, but continue
            System.err.println( iex );
         }
      }
   }
}
package examples.threads.bank;

import examples.threads.bank.*;

/**
  * A class to demonstrate wait and notify methods
  */
public class Spender extends Thread {

   private BankAccount account;

   /** Class constructor method
     * @param ba The bank account from which
     *    this spender takes the money
     */
   public Spender( BankAccount ba ) {
      account = ba;
   }

   /** The method the spender uses
     * to take out money
     */
   public void run() {
      while( account.isOpen() ) {
         try {
            if ( account.withdraw( 500 ) ) {
               System.out.println(
               "$500 successfully withdrawn." );
            }
            sleep( 1000 );
         } catch ( InterruptedException iex ) {
            // display any interruptions but continue
            System.err.println( iex );
         }
      }
   }
}
```

Grading your submission

1. Program Specification/Correctness (12 points)

    a. Class diagram is provided.
    b. No errors, program always works correctly and meets the specification(s).
    c. The code could be reused - as a whole or each routine.
    d. Multithreading is implemented in EditOptions class or in a separate class.
    e. Object locking is implemented in methods of Automobile class.
    f. Interface is used to enable interaction between EditOption and BuildAuto class.
    g. Demonstrates (code and test) Object Locking usage (removing synchronization causes data corruption).

2. Readability(2 point)
    a. No errors, code is clean, understandable and well-organized.
    b. Code has been packaged and authored based on Java coding standards.

3. Documentation (2 point)
    a. The documentation is well written and clearly explains the functionality implemented in the code.

4. Code Efficiency (4 points)
    a. No errors; code uses the best approach in every case. The code is extremely efficient, readable and understandable.