Exercises - Design Patterns

1: AbstractFactory

In this exercise, we shall define an abstract factory for creating car parts.
This abstract factory would have 2 concrete sub-classes: one for mini-vans and one for family cars.

You are provided with the following product classes:

- *abstract class Part*
- *abstract class Hood extends Part*
- *class MiniVanHood extends Hood*
- *class FamilyCarHood extends Hood*
- Similarly : *Wheel, MiniVarWheel, FamilyCarWheel*

Implement the following classes, relying on the UML Diagram for AbstractFactory:

- *abstract class AbstractCarFactory*
- *class MinivanFactory extends AbstractCarFactory*
- *class FamilyCarFactory extends AbstractCarFactory*

NOTE: your factories need to assign sequential numbers to parts as they are created (use the same counter for all part types).
WHY is this better than letting the Part constructor assign sequential numbers ?

How to decide which kind of factory to use ?
Implement a factory-of-factories:

```
public class FactoryManager {
        private int carType;   // family or mini-van
        ...
        public FactoryManager(){
                // TODO: read current car-type from properties file
        }
        public AbstractCarFactory getFactory() {
                // TODO
        }
}
```

Create a test method that:

- Obtains a car factory
- Retrieves various car parts from it (call the parts toString() method)

Note: feel free to re-arrange the package structure.

Level 2:

How would you go about adding Pooling capabilities to your factory ?
Points to consider:

- Returning objects to pool
  (Advanced: reference counting)
- Configuring your pool size
- The data structure
- **Locking**
  Note: how may times would you need to lock/unlock when required to create 10 cars ? How would you solve this (given that some users may still required single parts) ?
- **Is your pooling code general & reusable**, or would you need to duplicate if you are also asked to pool Employee & Supplier objects ?

2: Builder

Use the Builder pattern for car assembly.
Please rely on the Builder UML Diagram.

Provided classes:

abstract class Part
Hood, MiniVarHood, FamilyCarHood (as in ex.1)
Wheel, MiniVanWheel, FamilyCarWheel (as in ex.1)

And cars:

- **Abstract car and its concrete sub-classes. Note we chose a different data-structure for each kind of car, since builder is usually most useful for encapsulating different complex classes with different internal structures**

```
abstract class Car {
}

class MiniVan extends Car {
        private List allParts ;  // list of all parts (list size=5)
        …
}

class FamilyCar extends Car {
        private Wheel[] wheels = new Wheel[4];
        private FamilyCarHood hood;
        …
}
```

Please implement the builder/director classes:

- *abstract class CarBuilder*
- *class MiniVanBuilder extends Builder*
- *class FamilyCarBuilder extends Builder*
- *class CarDirector* :

Write a simple test program.

3: Prototype

As we've discussed, the Prototype pattern is sometimes considered a competitor for the AbstractFactory pattern.
In a previous exercise, we've implemented the AbtractFactory pattern for creating mini-van or family-car parts.
Convert that solution so as to use the prototype pattern.

Consider

- Express your opinion on this approach versus the AbstractCarFactory.
- **Make an efficiency test !**

4(a): Adapter

The purpose of this exercise is to create an Adapter class that converts a 2-dimentional array of objects (2xn) into a java.util.Map .

Example
Such a 2D array may contain:

	Column 0 (KEY)	**Column 1 (VALUE)**
Row 0	Integer(111)	Employee (id = 111)
Row 1	Integer(555)	Employee (id = 555)
Row 2	Integer(222)	Employee (id = 222)

Declaration
The Adapter class should be declared as follows:

```
public class  MatrixMapAdapter  implements Map {
        public MatrixMapAdapter(Object[][] matrix)
        …
}
```

Notes:

- Please use the *Object adapter* technique.

- Interface *java.util.Map* requires a number of methods. Due to time limitations, we would only implement the following (other methods would get an empty implementation):
  - ***Object put(Object key, Object value)***
  - ***Object get(Object key)***
  - ***int size()***

  A much better (yet time-consuming) alternative would be: inherit from *AbstractMap* and implement only *entrySet(), put, get.*

- You may choose between two approaches:
  A) Memory consuming, but time efficient: Adapter would hold an instance of Map, which would be populated by the constructor of MtrixMapAdapter.
  B) Time consuming, but memory efficient: Adapter would only keep a reference to Object[][] . The *get* method would have to search through this array.

4(b): Adapter – level 2

Create an Adapter class which converts a list of Employees into a javax.swing.table.TableModel

Each employee would have:
- Id
- Name
- Salary

Example:
The List:

Yossi (id=1111, salary=9999)
Miri (id=5555, salary=22000)
Avi (id=2222, salary=16000)

The corresponding TableModel:

	Column 0 (ID)	**Column 1 (Name)**	**Column 2 (Salary)**
Row 0 (yossi):	1111	Yossi	9999
Row 1 (miri) :	5555	Miri	22000
Row 2 (avi) :	2222	Avi	16000

Notes:

1. Due to time limitations, your table model should only provide the following methods (the others would have an empty implementation):
   - int getRowCount()
   - int getColumnCount()
   - Object getValueAt(int row, int col)
   - void setValueAt(Object val, int row, int col)
   - String getColumnName(String colIndex)

2. Due to time limitations, your code is not required to be general – it is enough for it to support only the Employee class.
   However, could you suggest a way to write general code which would work for any class (assuming table columns correspond to class properties) ?

3. **Be aware of the problem of listeners** : ListModel is supposed to notify whenever its data changes. However, we shall not be able to detect changes make directly on the list - since it's a list, not a ListModel (For this exercise we shall assume users will not make direct modifications on the list . Another solution: require a ListModel). You may face similar problems when composition is involved.

5: Bridge

Consider the implementation of a simple book store, with methods for managing the inventory:

- **add book (given book)**
- **remove book (given its isbn)**
- **search by subject (returns a List of books)**

We predict 2 directions of evolution:

- Storage technique
  (book information may be stored in RAM, Flat file, Database …)
- Logical requirements (a computer-store would only allow computer books; a limited-size store would limit the number of books in inventory; some stores may offer discounts …)

Write a corresponding implementation, relying on the Builder UML Diagram.

Required classes:

- **class Book** :
  holds ISBN, title, price, subject
- **interface BookStorageImpl**
- **RAMStorageImpl implements BookStorageImpl** :
  stores books as a java.util.LinkedList
- (Other storage implementations, such as database or flat-file, will not be implemented due to time limitations)

- **abstract class AbstractBookStore** :
  parent of the logical hierarchy
- **class LimitedSubjectBookStore extends AbstractBookStore** :
  stores only books on some pre-determined subject, such as computers
- **class DiscountBookStore extends AbstractBookStore:**
  sells books for less than their recommended price. The amount of discount is determined when adding a book to inventory: *addBook(Book b, float discount)*

Notes:

- **Important: for this exercise, you are not required to allow combinations of logical requirements** (such as "computer-store-with-size-limit").
  This problem may be addressed in a following exercise.

- Of course, it would be advisible to rely on a general-purpose data storage layer, such as EJB's . However, this is not the purpose of this exercise.

6: Decorator

As mentioned in the “Bridge” exercise, our library does not yet allow users to combine various logical requirements, such as “computer-store-with-size-limit”.

Use the Decorator pattern to solve this problem.

Consider:
In this application, is the order of decorators important ? Do you like the behavior which occurs when wrapping the same store with multiple “subject-limitation” decorators?

7: Flyweight

The following class represents a circle:

```
public class Circle {
        private double x;               // x-coordinate of center
        private double y;               // y-coordinate of center
        private double r;               // radius
        private double area;            // area of this rectangle

        /** Constructor */
        public Rectangle(double x, double y, double r){
                this.x=x; this.y=y; this.r=r;
                area=Math.PI *r*r;      // calculated once, for efficiency
        }

        /** Area of this circle */
        public double getArea(){
                return area;
        }

        /** Distance between CENTERS of circles */
        public double distance(Circle c2){
                double dx= x-c2.x ,  dy= y-c2.y;
                return Math.sqrt(  dx*dx + dy*dy);
        }

        /** Does this circle intersect (overlaps) with c2 */
        public boolean intersects(Circle c2){
                return distance(c2) < (r + c2.r);
        }
}
```

Convert this class so as to use the Flyweight pattern, according to the following instructions:

A) Change properties *x,y* so as to make them extrinsic.
   Leave the radius to remain intrinsic (what is the benefit of that ?)
   Note how this changes the interface of some methods, making them less elegant.

B) Write a CircleFactory class. This class would hold a map of circles , keyed by their radius. It will also provide the following method:

```
Circle  getCircle(int  r) {
                // If circle with required R already exists, return it.
                // Otherwise create new circle, store it in map, and return it.
}
```

C) Currently, circles are never garbage-collected. Could you suggest a solution?

8: Strategy

Implement a method for listing files inside a given directory:

File[] listFilesInDir(String dirName, java.io.FileFilter filter, java.util.Comparator comp)

Where:

- ***dirName*** is the path to an existing directory, e.g. "c:/work"

- ***filter*** is an implementation of the following interface:

```
public interface  java.io.FileFilter {
        boolean accept (File  f) ;
}
```

  The *listFilesInDir* method should only print files that are accepted by filter. For testing, please write a concrete filter that only accepts files with ".txt" extension.

- ***comp*** is an implementation of the following interface:

```
public interface  java.util.Comparator {
        int compare(Object  x,  Object  y);
        // returns:
        // - negative value if  x is considered "less than" y
        // - positive value if  x > y
        // - zero  if  x is considered "equal to" y
}
```

  The *listFilesInDir* method should sort the files using the given comparator. For testing, please write a concrete comparator which compared files according to the alphabetical order of their name.

Technical Hints:

A) To sort an array, one may rely on Arrays.sort(Object[], Comparator)

B) Obtaining files inside a directory:

```
File  dir=new File ("c:/work");
File[] files = dir.listFiles();
OR:
File[] files = dir.listFiles(FileFilter  filter);
```

9: Observer

We shall attempt to write an observer which would be notified whenever there's a change to the "lastModified" attribute of a file.

Approach #1 (level 1):
One solution may be to create a sub-class of *java.io.File* , which would maintain a list of listeners and notify them whenever the method *File.setLastModified* is called.

Required classes:

- *class FileEvent:*
  Holds information on a file-change event, including properties such as:
  - Name of the file which has been changed (String)
  - timestamp indicating when change occurred (long)

- *interface FileListener:*
  Declares the callback method which all observers (listeners) must implement:
  - *public void fileAttributesChanged(FileEvent fe)*

- *class NotifyingFile extends java.io.File:*
  A sub-class of File, which holds a list of listeners, and overrides *setLastModified()* so as to notify all listeners when there's a change.
  - list of listeners
  - *public void addFileListener(FileListener listener)*
  - *public void removeFileListener(FileListener listener)*
  - *public boolean setLastModified(long timestamp)*

- A short test example.

Consider:
* Supposing a user changes the file's timestamp by directly editing the file (e.g. using notepad). Will Your listeners be notified of the change ?

* Consider which changes should be introduced in order to notify observers of other changes, such as making the file "read-only".

Approach #2 (level 2):

Modify your *NotifyingFile* so that observers would be notified even if a user edits the file directly (e.g. notepad), without invoking the *setLastModified()* method.

To facilitate this, create a thread which checks the timestamp of the file every fixed number of seconds. If timestamp has changed, it would notify listeners.

10: Visitor

The lecture introduced a spell-check visitor on classes Chapter, Sentence, word. Write another visitor which counts the **number of appearances of each word** in a chapter.

Example:

For the following chapter:

It was **a** fine spring morning.
The sun was shining.
There was not **a** could in **the** sky.

Output should be (in any order):

Word "a" appears twice
Word "the" appears twice
Word "was" appears 3 times.
Words that appear once:
It, fine, sprint, morning, sun, shining, there, not, could, in, sky

Notes:

- It is recommended that your visitor keep a Map of all words it has encountered so far, and for each word - the number of appearances counted so far.
  Once visitor finished the entire visit on the chapter, you could print out the map it has accumulated.
- Performance and memory consumption are not an issue for this exercise.
- Words may be printed out in any arbitrary order.