R. Hinze, S. Franck Summer Semester 2005

Software Engineering

http://proglang.informatik.uni-freiburg.de/teaching/swt/2005/

Exercise Sheet 11

Deadline: July 5th, 2005

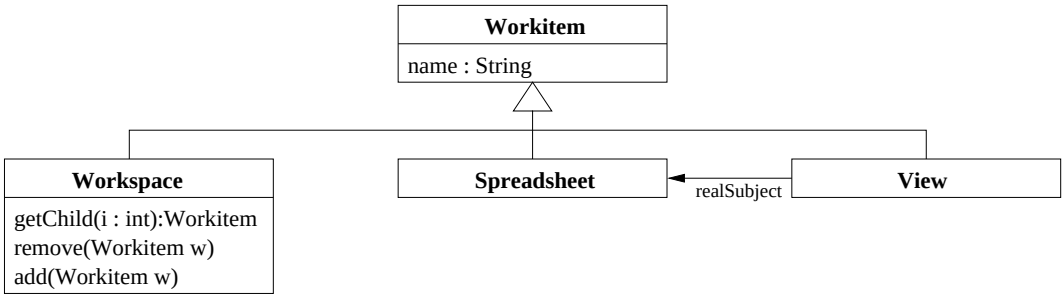
Exercise 1 – Design Patterns (I): (4 points)
In the second and third exercise sheet you have been working with "Marvel". Marvel is a spread sheet application which offers so-called "Workspaces". These workspaces may contain arbitrary many base documents (spreadsheets) as well as other workspaces.
Which design pattern can be used to implement this structure easily, especially if you want the structure to be extendable?
Give the interfaces of the classes (no full blown implementation).
You don't have to create fancy objects - it is sufficient if they have a name and some basic management methods.

Solution:
Since we have to describe some structure, we have a look at the structural patterns (Decorator, Composite and Proxy). We want to model bags (workspace) of content, whereas the content may be something basic (spreadsheet) or another bag (workspace). This structure can be implemented easily using the **Composite** pattern.
The class interfaces are:

```
abstract class Workitem {
  String name;

  public String getName()         {[...]}
  public void   setName(String n) {[...]}
}

class Spreadsheet extends Workitem {
  public Spreadsheet(String n) {[...]}
}

class Workspace extends Workitem {
  private Vector items;

  public Workspace(String n) {[...]}

  public Workitem getChild(int i)    {[...]}
  public void     add(Workitem n)    {[...]}
  public void     remove(Workitem n) {[...]}
  public Iterator getChildren()      {[...]}
}
```

Exercise 2 – Design Patterns (II): (3 points)
We also want to have `Views`, which are projections of `Spreadsheets`. A `View` shows some part of the data of a `Spreadsheet`.
How can the `Views` be added to the existing structure? Which design pattern can be used in implementing the `View`?
Draw a class diagram of the current state of the object's structure.

Solution:
Basically a `View` is inserted as a baseobject into the Composite structure.
Since the `View` is just a projection of a `Spreadsheet`, it can be implemented using the **Proxy** pattern. The `View` is the Proxy to the Spreadsheet. It manages the access to the Spreadsheet by returning only the part projected out. The `View` does not store any data itself, so there's no redundancy.



Exercise 3 – Design Patterns (III): (3 points)
A lot of functionality will be added to this structure. This process will be evolutionary, and we don't want the basic structure to change with every new method.
So we want the interface to come up with a minimal set of operations that allows us to extend the structure without interfering with the base classes. But we also don't want to put all this methods into the client (the classes which use the structure).
Which design pattern implements a third alternative?
What do the interfaces look like after using this design pattern? How would you implement a `count` function, which counts all objects in a workspace recursively (e.g. also counts objects in subordinate workspaces)?

Solution:
The **Visitor** pattern allows you to define new operations without changing the classes of the elements on which it operates.
To make our structure work with the visitor pattern, we add the function `accept` to all classes of our structure:

```
abstract class Workitem {
  String name;

  public String getName()         {[...]}
  public void   setName(String n) {[...]}

  public abstract Object accept(Visitor v);
```

```
}

class Spreadsheet extends Workitem {
  public Spreadsheet(String n) {[...]}

  public abstract Object accept(Visitor v) {
    return v.visitSpreadsheet(this);
  }
}

class View extends Workitem {
  public View(String n, Spreadsheet s) {[...]}

  public abstract Object accept(Visitor v) {
    return v.visitView(this);
  }
}

class Workspace extends Workitem {
  private ArrayList items;

  public Workspace(String n) {[...]}

  public Workitem getChild(int i)    {[...]}
  public void     add(Workitem n)    {[...]}
  public void     remove(Workitem n) {[...]}
  public Iterator getChildren()      {[...]}

  public abstract Object accept(Visitor v) {
    return v.visitWorkspace(this);
  }
}
```

In addition, we need the `Visitor`-Class:

```
abstract class Visitor {
  public abstract Object visitSpreadsheet(Spreadsheet s);
  public abstract Object visitView(View v);
  public abstract Object visitWorkspace(Workspace w);
}
```

The counting functionality is implemented as a Visitor subclass:

```
class CountVisitor {

  public CountVisitor() {  }
```

```
  public Object visitSpreadsheet(Spreadsheet s) {
    return new Integer(1); // we can't use int, since we
                           // have to return an Object!
  }
  public Object visitView(View v) {
    return new Integer(1);
  }
  public Object visitWorkspace(Workspace w) {
    Integer  sum      = new Integer(0);
    Iterator children = w.getChildren();
    while (children.hasNext()) {
      sum = add(sum, ((Workitem)(children.next())).accept(this));
    }
    return sum;
  }

  private Integer add(Integer fst, Integer snd) {
    // we can't add two Integers directly, but have to retrieve
    // their int-Values first.
    return new Integer(fst.intValue() + snd.intValue());
  }
}
```