COMP 3512

Assignment 3

In this assignment, we'll look at the Visitor pattern. We'll be developing a program to "draw" shapes. The program can also save shapes to a file & re-create them from the file using a simple factory.

1 Shapes

We'll use an abstract `Shape` class with 3 concrete derived classes: `Circle`, `Rectangle` & `Triangle`. These classes in turn use a `Point` class. We have already looked at some of these classes in lecture.

The `Shape` class contains (pure virtual) methods to "draw" & save a shape:

```
class Shape {  // ABC
public:
  virtual ~Shape() {}
  virtual void draw() const = 0;
  virtual void save(std::ostream& = std::cout) const = 0;
  // see below for an additional method
};
```

The 3 concrete derived classes implement the above pure virtual methods together with some additional methods. (See provided files).

The following illustrate the output of the `draw` method:

```
[C: (1,-2), 3]
[R: (2,-3), (4,-5)]
[T: (5,-6), (7,-8), (2,1)]
```

The first line indicates a circle with center (1,-2) & radius 3. The second indicates a rectangle with 2 corners at (2,-3) & (4,-5). The last line indicates a triangle with the 3 vertices (5,-6), (7,-8) & (2,1).

When the above 3 shapes are saved to a file (using the `save` method), the content is as follows:

```
circle
(1,-2)
3
rectangle
(2,-3)
(4,-5)
triangle
(5,-6)
(7,-8)
(2,1)
```

Note that this is slightly different from what we did in lecture. In order to make the data file portable between different compilers, the different `save` methods do not use `typeid`. Rather, each `save` method writes a hard-coded type name (`circle`, `rectangle` & `triangle`).

Make sure your output formats for `draw` & `save` match exactly the formats shown above.

2 The Visitor Pattern

The various shape classes demonstrate a classic OO design with concrete derived classes implementing or overridng virtual functions in an abstract base class.

However, there is a disadvantage to this design — if ever we need to add a new operation on shapes, we'll need to add new methods in all the different shape classes. We may, for example, want to add a "translate" operation that moves a shape by a certain displacement. To do this, we'll need to add `translate` methods to `Shape` & its derived classes. This will necessitate the recompilation of all the affected classes.

The Visitor pattern solves this problem. To make it easier to add new operations, each of the shape classes has a `accept` method that accepts a visitor (of class `ShapeVisitor` in our case). When a "shape visitor" visits a shape object, it performs operations on that object (e.g. translates it by a certain displacement). Different types of "shape visitors" (deriving from `ShapeVisitor`) perform different operations on shapes.

The `ShapeVisitor` class has different "visit" methods to visit different types of shapes:

```
class ShapeVisitor {
public:
  virtual ~ShapeVisitor() {}

  virtual void visitCircle(Circle *s) = 0;
  virtual void visitTriangle(Triangle *s) = 0;
  virtual void visitRectangle(Rectangle *s) = 0;
  // ... more methods if there are more shapes
};
```

In the above, the "visit" methods are pure virtual. Concrete derived classes need to implement these methods to perform specific operations on circles, triangles & rectangles.

Each of the shape classes has an `accept` method that accepts a `ShapeVisitor`. This method is basically declared as:

```
  virtual void accept(ShapeVisitor&);
```

In this assignment, we'll have 3 types of "shape visitors": `TranslationVisitor`, `XReflectionVisitor` & `YReflectionVisitor`. They all inherit from `ShapeVisitor`.

A `TranslationVisitor` translates a shape by a certain "displacement" (which technically is a vector but is represented by a `Point` in our case) when it visits the shape. Each point in the shape is translated by the same amount. As an example, if the displacement is (1,2), the point (3,4) is translated to (4,6). This is basically addition of points.

An `XReflectionVisitor` reflects a shape about the x-axis when it visits the shape. This means that each point in the shape is reflected about the x-axis. Under this type of reflection, the point (1,2) is mapped to (1,-2) (i.e. the x-coordinate remains the same, the y-coordinate is negated).

Analogously, a `YReflectionVisitor` reflects a shape about the y-axis when it visits the shape, i.e., each point in the shape is reflected about the y-axis. For example, the point (1,2) is mapped to (-1,2) (i.e. the y-coordinate remains the same, the x-coordinate is negated).

3 The Shape Factory

We have seen this in lecture. To read shapes stored in a file, we use a `ShapeFactory` class:

```
class ShapeFactory {
public:
  explicit ShapeFactory(std::istream& in);
  Shape* create();
  // ...
};
```

The `create` method reads the type name of a shape from its associated input stream & then calls the appropriate constructor. Each of the classes `Circle`, `Triangle` & `Rectangle` has a constructor that takes an input stream as an argument. (See provided header files.) This constructor reads the appropriate data members from the input stream. `create` returns the null pointer on failure.

4 The Main Program

The program must be invoked with the name of a data file as a command-line argument. This file may be non-existent. But if it does exist, it should be a file containing shape data (written by the `save` method). In that case, the program reads in the shapes, stores & displays them & then waits for user input after displaying a prompt (`>`) *to standard error*. If the specified file does not exist, the reading & displaying of shapes is skipped.

User can repeatedly issue commands until the end-of-file key is entered. When end-of-file is encountered, the program saves all its current objects to the file specified on the command-line. If the file did not exist before, it is created. (However, if there are no current objects, this creation of the data file may be skipped.) If the file did exist, its content is now completely overwritten. The program then exits.

The following shows an interaction with the program. Each line that starts with a `>` is a command issued by the user. (However, the `>` is the prompt printed by the program.) The other lines are the output of the program (written to standard output). The output assumes that there are no shapes at the start:

```
> c circle (1,-2) 3  # create circle with specified centre & radius
[C: (1,-2), 3]
>  c rectangle (2,-3) (4,-5)
[C: (1,-2), 3]
[R: (2,-3), (4,-5)]
> c triangle (5,-6) (7,-8) (2,1)
[C: (1,-2), 3]
[R: (2,-3), (4,-5)]
[T: (5,-6), (7,-8), (2,1)]
> x
[C: (1,2), 3]
[R: (2,3), (4,5)]
[T: (5,6), (7,8), (2,-1)]
> y
[C: (-1,2), 3]
[R: (-2,3), (-4,5)]
[T: (-5,6), (-7,8), (-2,-1)]
> t (10,-10)
```

```
[C: (9,-8), 3]
[R: (8,-7), (6,-5)]
[T: (5,-4), (3,-2), (8,-11)]
> d
[C: (9,-8), 3]
[R: (8,-7), (6,-5)]
[T: (5,-4), (3,-2), (8,-11)]
```

Note that the program processes user input line by line & for each line, it just reads in sufficient information to carry out the command & ignores the rest of the line. This is shown in the very first command above.

As can be seen, `c` is used to create & add a shape, `x`, `y` & `t` are used to transform all current shapes & `d` is for drawing all shapes. Note the "arguments" to the `c` & `t` commands.

Invalid commands (including those that have invalid arguments) should be ignored. No error message is needed.

Sample input/output files will be provided. Make sure your program output matches exactly those in the output files provided.

5 Submission

This assignment is due at 10pm, Wednesday, December 3, 2008. Submit your source file(s) to `In` in the directory:

```
\COMP\3512\a3\set<X>\<id\_name>\
```

where <X> is your set and <id_name> is your student ID & name separated by an underscore (e.g. `a00000001_SimpsonHomer`). Do not use spaces to separate your name. We'll basically go into each <id_name> directory & compile your files using something like `g++ *.cpp`, but with additional switches. So make sure that you put your actual source files in that directory & not, for example, just a zip file. If you ever need to submit more than one version, name the folder of each later version with a version number after your name, e.g., `a00000001_SimpsonHomer_v2`.

This assignment will be marked mainly based on testing the required features. In order to be able to test your program at all, it must at least be able to load shapes from a file & draw them.

If your program does not compile, you may receive zero for the assignment. Otherwise, the grade breakdown is approximately as follows:

Code clarity	10%
Reading/writing shapes from/to files†	20%
Drawing shapes	15%
Translation⋆	15%
x-reflection⋆	15%
y-reflection⋆	15%
Error-handling	10%

†requires `ShapeFactory` for reading
⋆requires user input, draw & visitor