COMP 3512
Assignment 2

This assignment introduces the Command pattern. It can be used to implement "undoable" operations. We'll develop a program to "draw" & transform shapes. The program also allows us to undo & re-do a transformation.

1 The Command Pattern

The Command pattern is typically used in GUI toolkits. As an example, what happens when we click on a button depends on the particular application that uses it. The developer of the "button widget" must provide a means for the user to specify what to do, i.e., to specify the command to execute, when the button is clicked. In a procedural language, this can be achieved by having the application register a callback function. In an object-oriented language, we can use the Command pattern.

The Command pattern basically encapsulates a command into an object. In object-oriented languages, work is done by invoking methods on objects. A command therefore consists of an object (the receiver) & the operation to be performed on the object. The receiver & the operation are encapsulated into a command object. When a command object is "executed" (via its `execute()` method), the operation is performed on the receiver.

In the button example above, instead of registering a callback function, we associate a command object with the button. When the button is clicked, the associated command object is executed.

The Command pattern also facilitates the implementation of simple "undoable" operations. A command object may have an `unexecute()` method that performs the "inverse" operation on the receiver to undo what is done by the `execute()` method. When a command object is executed, it is put on a history list. By traversing the history list backwards (or forwards) & invoking the `unexecute()` (or `execute()`) method on each command object, we can undo (or re-do, i.e., undo the undo) one or more operations that have been performed.

In C++, we can have an abstract `Command` class:

```
class Command {
public:
  virtual ~Command() {}
  virtual void execute() = 0;
  virtual void unexecute() = 0;
};
```

In the above, `execute()` performs the operation on the receiver; `unexecute()` undoes that operation. Note that the receiver is not stored in a `Command` object — the type of a receiver is specific to each type of command.

As an example, consider an abstract `Shape` class with concrete derived classes. Suppose we want to encapsulate the operation of reflecting a `Shape` object about the x-axis. We can have a (concrete) class derived from the `Command` class:

```
class XReflectCommand: public Command {
public:
  explicit XReflectCommand(Shape *s = 0): shape_(s) {}
  virtual void execute();    // reflect shape_ about x-axis
  virtual void unexecute();  // undo reflection
private:
  Shape *shape_;             // receiver
};
```

Note that in this case, we can undo the reflection by reflecting the shape again about the x-axis.

2 The Main Program

The main program basically “draws” different shapes & allows the user to perform transformations on them & to undo & re-do those transformations.

We’ll only be “drawing” circles & triangles. For transformations. we’ll either translate a shape or reflect it about the x- or y-axis. These 3 types of transformations require the user to specify the shape to transform. Additionally, the translate operation needs the user to specify the “amount” to translate.

The program is implemented using a `DrawApp` object:

```
class DrawApp {
public:
  DrawApp();
  ~DrawApp();                  // destroy shapes & history
  void run();                  // loop to get user commands & execute them
  void addShape(Shape *s);
  bool undo();                 // undo
  bool redo();                 // redo

private:
  void draw() const;           // draw all shapes
  vector<Shape*>    shapes_;
  vector<Command*>  history_;  // history list
  // additional data members if necessary
};
```

Such an object contains 2 vectors: a vector of shapes to draw & a vector (the history) of the commands that have been executed. The history vector makes it possible to undo/re-do operations.

The `undo()` (respectively `redo()`) command traverses the history list backwards (respectively forwards) calling `unexecute()` (respectively `execute()`) on the command object. Both return a boolean indicating whether they succeed. If we are currently at the “beginning” of the history list, then we cannot undo, but may be able to re-do (if the history list is not empty). But if we are at the “end” of the history list, then we cannot re-do, but may be able to undo (again if the history list is not empty). If we choose to transform a shape, the history list is modified from our current position — basically all command objects from our current position are removed from the list & the new operation put on it. (Note: This depends on how we index into the history list.)

The `run()` method is basically a loop that repeatedly prints a prompt, reads a user command, executes it & re-draws all the shapes. If the command is invalid, nothing is done.

When the shapes are drawn, they are numbered starting from 1. This allows the user to specify a shape using that number. The following shows a sequence of valid user commands with comments (assuming that there are at least 3 shapes):

```
  x 2        # reflect shape 2 about the x-axis (operation 1)
  y 1        # reflect shape 1 about the y-axis (operation 2)
  t 3 (1,2)  # translate shape 3 by the amount (1,2) (operation 3)
  y 2        # reflect shape 2 about the y-axis (operation 4)
  u          # undo the last operation (i.e., undo operation 4)
  u          # undo another operation (i.e., undo operation 3)
  u          # undo yet another operation (i.e., undo operation 2)
  r          # re-do operation that we have just undone (i.e., re-do operation 2)
  x 1        # x-reflect shape 1 (this becomes operation 3 & changes history list)
  r          # does nothing since there are no operations to re-do
```

In the above, we have not shown the output of the program. See section 4 for a sample session. In general, the program ignores extra words after each command. In the above, we use the `#` sign to start a comment although it is not really necessary as extra words are ignored.

The program must be invoked with the name of a data file as a command-line argument & works with the shapes specified in that data file. (A sample data file will be provided.)

Note that we don't allow users to create & delete shapes within the program. This is to make it easier to implement undo/re-do operations.

Since we won't really be developing a grapical program, we "draw" a shape by displaying some information about the shape to standard error.

3 Shapes & Commands

We'll use an abstract `Shape` class with 2 concrete derived classes: `Circle` & `Triangle`.

The `Shape` class contains (pure virtual) methods to draw & transform a shape. The transformations are translations & reflections about the x- & y-axes.

```
class Shape {  // ABC
public:
  virtual ~Shape() {}
  virtual void draw() const = 0;

  // translate by amount delta, e.g. if delta is (1,2), the point (3,4) is transformed
  // to (4,6)
  virtual void translate(const Point& delta) = 0;

  virtual void xreflect() = 0; // reflect about x-axis, e.g. (1,2) -> (1,-2)
  virtual void yreflect() = 0; // reflect about y-axis, e.g. (1,2) -> (-1,2)
};
```

The 2 derived classes implement the above pure virtual methods.

There are 3 types of `Command` objects representing the 3 kinds of transformations — `TranslateCommand`, `XReflectCommand` & `YReflectCommand`. They are all derived from the `Command` class. We've already encountered the `Command` & the `XReflectCommand` classes. The `YReflectCommand` & `TranslateCommand` classes are similar to the `XReflectCommand` class & they all have an associated shape.

Partial implementation of the shape classes & partial headers for the command classes are in the file `a2files.zip`.

4 Sample Session

Assuming that the content of the input data file is:

```
C (1, 2) 3
T (2, 3) (4, 5) (6, 7)
C (-2, 3) 5
```

then the following shows a sample session:

```
1: [C: (1,2), 3]
2: [T: (2,3), (4,5), (6,7)]
3: [C: (-2,3), 5]
> x 1
1: [C: (1,-2), 3]
2: [T: (2,3), (4,5), (6,7)]
3: [C: (-2,3), 5]
> y 2
1: [C: (1,-2), 3]
2: [T: (-2,3), (-4,5), (-6,7)]
```

```
3: [C: (-2,3), 5]
> t 3 (2,-1)
1: [C: (1,-2), 3]
2: [T: (-2,3), (-4,5), (-6,7)]
3: [C: (0,2), 5]
> u  # undo translation
1: [C: (1,-2), 3]
2: [T: (-2,3), (-4,5), (-6,7)]
3: [C: (-2,3), 5]
> x 4  # invalid command - no shape 4
> u  # undo y-reflection
1: [C: (1,-2), 3]
2: [T: (2,3), (4,5), (6,7)]
3: [C: (-2,3), 5]
> a  # invalid command
> u  # undo x-reflection
1: [C: (1,2), 3]
2: [T: (2,3), (4,5), (6,7)]
3: [C: (-2,3), 5]
> r  # redo x-reflection
1: [C: (1,-2), 3]
2: [T: (2,3), (4,5), (6,7)]
3: [C: (-2,3), 5]
> y 1
1: [C: (-1,-2), 3]
2: [T: (2,3), (4,5), (6,7)]
3: [C: (-2,3), 5]
> r  # nothing to redo
> u  # undo y-reflection of shape 1
1: [C: (1,-2), 3]
2: [T: (2,3), (4,5), (6,7)]
3: [C: (-2,3), 5]
>
```

Note that

- the shapes are displayed to standard error (including the leading numbers);
- the prompt (`>`) is printed to standard output;
- when a command is invalid, it is simply ignored: no error message is printed & the shapes are not redrawn;
- when a command does nothing, the shapes are not redrawn;
- the program exits on end-of-file.

5 Additional Information & Requirements

We've provided a `ShapeFactory` class for creating shapes. It can be used to load shapes from a data file.

For simplicity, you can put all the command classes are in `Command.h` & `Command.cpp`. Implementation of `DrawApp` should go in `DrawApp.cpp`.

You'll need to submit all necessary files so that your program builds correctly. This would include the provided files (suitably modified).

To facilitate testing, your program must be able to run in "debug mode". If your program is compiled with the macro `DEBUG` defined, it must print the following additional information to standard output:

- a plus character (`+`) every time a command object (of any of the 3 types) is dynamically allocated;
- a minus character (`-`) every time a command object (of any of the 3 types) is deallocated;
- the size of the history list every time a command object is added to it (i.e., every time a valid `x`, `y` or `t` command is executed).

Each of the above output should be followed by a newline character.

6 Submission

This assignment is due at 11pm, Thursday, November 29, 2012. Submit your source files in a zip file to `In` in the directory:

```
\COMP\3512\a2\set<X>\
```

where `<X>` is your set. Your zip file should be named `<name_id>.zip`, where `<name_id>` is your name & student ID separated by an underscore (for example, `SimpsonHomer_a12345678.zip`) Do not use spaces to separate your last & first names. Your zip file should unzip directly to your source files (without creating any directories). We'll basically compile your files using

```
g++ -ansi -W -Wall -pedantic *.cpp
```

after unzipping.

If you need to submit more than one version, name the zip file of each later version with a version number after your ID, e.g., `SimpsonHomer_a12345678_v2.zip`. In that case, we'll only mark the version with the highest version number. (If there are multiple versions with the same highest version number, your submission will be regarded as invalid & will not be marked.)

Do not submit rar files. You may receive a score of 0 if you don't submit a zip file.

If your program does not compile, again you may receive a score of zero for the assignment. (We'll be using g++ version 4.)

This assignment will be marked based mostly on testing the required features. Note that in the following approximate grade breakdown, some items depend on others. For example, in order to be able to test your program at all, it must at least be able to load shapes from a file; in order to test if transformations work correctly, the program must be able to accept valid commands & display shapes; to check whether undo & redo are implemented correctly requires debug mode, etc.

Code clarity	10%
Load & display shapes	10%
User input validation	10%
Transformations	30%
Undo/Redo	30%
Other `DrawApp` features	10%

Note that your program must use the Command pattern to implement the undo/redo operations as described in this write-up.