Task 1

Design Patterns: Abstract Factory, Factory, Prototype

Introduction

Implement a program which works with 2D figures. It should be possible to easily add support for new figures to the program or remove existing ones. As part of the current task, implement support at least for the following:

- `Triangle` — has three sides, specified by their length.
- `Circle` — has a radius.
- `Rectangle` — has two sides, specified by their lengths.

For all figures it should be possible to calculate their perimeter.

A figure can be represented by a string. This is done by listing its type, followed by its parameters. For example:

- `"triangle 10 20 30"` — a triangle with sides 10, 20, and 30.
- `"circle 2.2"` — a circle with a radius of 2.2.
- etc.

Your solution should be able to create a figure from its string representation, or to convert a figure to a string.

When the program starts, it should ask the user to choose how to enter a list of figures. It should be possible to use the methods specified below, and it should be easy to add new methods.

- Random — creates a list of random figures. The program asks the user how many figures to generate. After that it generates that many random figures with random properties.

- STDIN — the figures are read from STDIN. The user specifies their count and enters them manually.

- File — the figures are read from a text file.

For the last two options you don't have to invent new syntax for entering figures. Instead, you can use the standard string representation for figures.

After the figures have been entered, the user should be able to:

- List them to STDOUT.

- Delete a figure from the list.

- Duplicate a figure in the list.

- Store the resulting list back into a file.

Hints

Create a hierarchy to represent the figures. First, create an abstract base class or interface `Figure`. Use it as a base for all figures. It should have an operation called `perimeter`. Cover all figures with appropriate unit tests.

Use the Prototype design pattern to make it possible to clone a figure without knowing its type (i.e. polymorphic cloning). In some languages, such as Java and C#, there is a specific interface (`Cloneable` in Java and `ICloneable` in C#) that must be implemented, and a specific function that you must override. In other languages, such as C++, you have to provide that yourself by either adding a `clone` method to the base `Figure` class, or by implementing your own `Cloneable` class.

Add a "to-string" method to each figure, which returns its string representation. Here, again, some languages, such as Java or C#, have a standard method, that has to be overridden for that purpose (`toString` in Java and `ToString` in C#). Others, such as C++ will require you to add that method to the base `Figure` class, or to create your own base class for things that can be converted to a string.

Next, use the Factory pattern to handle figure creation from different sources.

Create a factory, which allows the user to create a figure from its string representation. Note that you don't have to parse the string manually. In some languages, you can create a stream from the string and use that to input the data. For example, in C++ you can use `std::stringstream` (here is how). In others, different classes may be used to achieve the same result. For example, see the following sources for Java:

- Scanner | Java 7 API
- Java User Input | w3schools
- Java Scanner | Baeldung
- BufferedReader vs Console vs Scanner in Java | Baeldung.

In the text we refer to that functionality as a "stream", but this does not need to exactly correspond to a specific class name in the language of your choice. For example, you may be using `Scanner` in Java, `std::stringstream` in C++ and so on.

Create a `RandomFigureFactory` class, which creates a random figure with random parameters.

Create a `StreamFigureFactory` class, which creates figures from a stream. Make it so that it receives a stream in its constructor. Every time a figure needs to be created, it should attempt to read the data for it from the stream. A simple way to implement this is to read a line of text from the stream and then pass it to the string-to-figure factory that you have already created.

Note that the latter two factories cover all scenarios listed in the specification.

- You can use the random factory to read a sequence of random figures.
- You can use the stream factory to read a sequence of figures either from STDIN, or from a file.

Use the Abstract Factory pattern to allow the user to choose between the different input methods. First, derive a common interface/base class for all factories. Then, equip the abstract factory with a `create` function that receives a string — the name of the input type that will be used. The abstract factory should create the appropriate figure factory and return it to the caller.

The program can use the newly-created figure factory to read N figures.

After reading the figures, the program's functionality relies on operations that are universal to all figures:

- To list them you use the "to-string" operation.
- To clone them you use the "clone" method.
- Deleting them in C++ will require you to consider a few details, such as adding virtual destructors. Using smart pointers will also help a lot.
- Storing the figures in a file should also rely on the "to-string" operation.

Unless there is a difference in how you display the figures, you can use the same code to both print to STDOUT, and to store a sequence of figures to a file. Similarly to how we organized the input, use streams, instead of hard-coding the output target. Then either pass a stream attached to STDOUT, or to a file.