

Figures

Atanas Semerdzhiev, Kalin Georgiev, Petar Armyanov, Trifon Trifonov
Edition 0.1, 2024-11-04: Still a work in progress

Table of Contents

Introduction.....	1
Guide	2
Step 1. Figures.....	2
Step 2. Conversion to string.....	3
Step 3. Cloning	3
Step 4. String to figure conversion.....	4
Step 5. Input and output.....	5
Step 6. Abstract Factory	6
Step 7. Application	6
Step 8. Smart pointers (C++ specific)	7
Step 9. Reflection	7
Skills and Competencies.....	9

Introduction

In this project you will implement a program which works with 2D figures.

Initially, only a limited set of figures will be available to the end user (they are listed below). However, in the future this may change. It should be easy to add support for new figures to the program or to remove existing ones.

- ¥ 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.

Figures must be immutable. It is not necessary to be able to access their specific properties (i.e. the sides of a triangle, a circle's radius, etc.). It is enough to be able to calculate their perimeter.

All figures can be represented as a string. This is done by specifying the figure type and then listing its parameters, using the space character as separator. 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 allow the user to create a figure from its string representation, or to convert a figure to a string.

When the program starts, it asks the user to choose how they prefer to create new figures. All of the methods listed below must be available. Also, it should be easy to add new methods to the program.

- ¥ Random figures are created using a random number generator.

- ¥ STDIN figures are read from STDIN.

- ¥ File figures are read from a text file specified by the user.

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.

When a method of entering figures has been chosen, the user should be able to enter a list of figures. After the figures have been entered, the user should be able to:

- ¥ List all figures to STDOUT.

- ¥ Delete a figure from the list.

- ¥ Duplicate a figure in the list. The copy should be added to the end of the list.

- ¥ Store the resulting list back into a file.

Guide

Step 1. Figures

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`. For example:

Java

```
interface Figure {  
    double perimeter();  
}
```

C++

```
class figure {  
public:  
    virtual double perimeter() const = 0;  
    virtual ~figure() = default;  
};
```

Proceed to implement all figures. Make all of them immutable. For each figure type provide a constructor that receives its parameters. (i.e. the three sides of a triangle, a circle's radius, etc.).

Make sure that proper exceptions are thrown when a figure cannot be created due to incorrect parameters.

Cover all figures with unit tests. Consider the following:

- ¥ Ensure that when a figure is created with specific parameters, the resulting perimeter is correct. For example, if a triangle has sides of lengths 10, 20 and 30, then its perimeter must be exactly 60.
- ¥ Proper handling of the case when the user creates a figure with incorrect sides (e.g. a triangle with negative sides, a triangle where $a + b > c$ for some of its sides, etc.)
- ¥ Handling the case when calculating the perimeter results in an overflow (ideally will be handled early, in the constructor).



Virtual destructors in C++

When using C++, when you have a polymorphic base class, do not forget to add a virtual destructor. For more information check:

- ¥ [Guideline C.35: A base class destructor should be either public and virtual, or protected and non-virtual](#) | C++ Core Guidelines

Step 2. Conversion to string

Add a "to-string" method to each figure, which returns its string representation.

In some languages, such as Java or C#, there is 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 implement that functionality yourself.

For example, in C++, you can:

- ¥ add such a method to the base Figure class
- ¥ (what is, arguably, the better design) create your own base abstract class (interface) for things that can be converted to a string. For example:

```
class string_convertible {  
public:  
    virtual std::string to_string() const = 0;  
    virtual ~string_convertible() = default;  
};
```

Cover the to-string method with unit tests. Use it to also check whether the constructor of a figure sets appropriate values for its properties:

1. Create a figure with specific parameters (e.g. a triangle with sides 10, 20 and 30).
2. Convert it to a string.
3. Check whether the string is correct (in this case it should be "triangle 10 20 30").

Step 3. Cloning

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.

Write the appropriate unit tests to check that cloning works correctly.

Step 4. String to figure conversion

Use the [Factory](#) pattern to handle conversion from string to figure.

Create a class, which allows the user to create a figure from its string representation. Its interface may look something like this:

Java

```
class StringToFigure {  
    Figure createFrom(String representation);  
}
```

C++

```
class string_to_figure {  
public:  
    figure* create_from(std::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 other languages, 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.

Do not forget to write the appropriate unit tests to check that the factory works correctly. Note that:

¥ You can use the same representations that you employed in the tests of the to-string function to keep things consistent.

¥ You should include tests that explore what happens on incorrect input. For example:

- when the string is empty,

- when the string is missing some parameters, e.g. "tri angl e 10 20",
- when some of the parameters are ill-formed, e.g. "tri angl e 10 abc -30",
- when one or more of the parameters are negative, e.g. "tri angl e -10 20 30",
- when the name of the figure is incorrect, e.g. "unknown 10 20 30",
- and others.

Step 5. Input and output

Next, use the [Factory](#) pattern to handle figure creation from different sources.

Begin by creating a common interface for all factories. For example:

Java

```
interface FigureFactory {
    Figure create();
}
```

C++

```
class figure_factory {
public:
    virtual figure* create() = 0;
    virtual ~figure_factory() = default;
};
```

Now, create multiple factories, one for each source.

Create a RandomFigureFactory class, which creates a random figure with random parameters. It may look something like this:

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 these two factories cover all scenarios listed in the specification.

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

To test the factory that reads a figure from a stream, first create a string which contains the representation of one or more figures. Then create a stream from that string and pass it to the factory. Check that the factory creates the correct figure(s).

You may also want to check what happens when the stream is empty, or when it contains incorrect data. Also, if it will be possible to continue reading from the stream after an error occurs.

Testing the random factory will be more complicated. On one hand, when random number generation is taking place, you cannot predict the exact result of a single function call. Also, such tests are not deterministic and not reproducible and thus they break [the FIRST rule of unit testing](#).

On the other hand, this is probably not what one wants to test anyway. Instead, you should focus on what exactly "creating a random figure means". Determine :

- ¥ what the range of the random numbers should be
- ¥ what the distribution of the figures should be.
- ¥ and so on.

Then, create a test that will check if the random numbers generated are within the expected range and the distribution of figures corresponds to the specification.

A test may look like this:

- ¥ Determine a number N, which is large enough.
- ¥ Create N figures and store them in a collection, or simply count how many instances of a given figure have been created.
- ¥ For each figure check if its attributes are within the expected range.
- ¥ When the generation is over, check how many figures were created of each type. If the distribution is not what you expected, you may need to adjust the random number generator.

Step 6. Abstract Factory

Use the [Abstract Factory](#) pattern to allow the user to choose between the different input methods.

Create an abstract factory class. Add a function, which receives a string \tilde{N} 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.

Step 7. Application

Write an application, which uses the above-implemented classes to solve the program.

The program asks the user to choose the input method. The user enters it as a string. This string is then passed to the abstract figure factory and it returns a factory that can create figures from that source.

The program proceeds to 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 and then present the output to the user.
- ¥ To clone them you use the "clone" method.
- ¥ Deleting a figure in C++ will require you to consider a few details, such as adding virtual destructors. Using smart pointers will also help a lot (see the next step)
- ¥ 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.

Step 8. Smart pointers (C++ specific)

In the examples above, in all C++ excerpts we assumed that figures are returned as raw pointers. Switch to using smart pointers to manage the memory of the figures. This will make the code safer and easier to maintain.

- ¥ Make all figure factories return `std::unique_ptr<figure>` instead of `figure*`.
- ¥ Make the abstract factory return `std::unique_ptr<figure_factory>` instead of `figure_factory*`.

The caller who receives the unique pointer can easily convert it to a shared pointer, if needed.

For more information on smart pointers, see Section 4. Smart Pointers (Items 18–22) of [\[effective-modern-cpp\]](#).

Step 9. Reflection

Use reflection to automatically register all figure types in the factory. This way, you can add new figures without changing the factory.

In Java and C# you can proceed as follows.

When creating a figure from its string representation (say, "triangle 10 20 30"):

1. First, read the figure type. In the example above this is "triangle"
2. Then, capitalize its first letter. In our example you will get "Triangle".
3. Now search whether there is a class with that name that is a subtype of `Figure`. If so, then we can assume that this is a valid figure. Otherwise (if there is no such class, or it is not a descendant of `Figure`), this is not a valid figure type.
4. Proceed to read the figure's parameters from the string and invoke the proper constructor.

structor to create a figure of that type.

- a. In Java, to make it easier, use `invokeConstructor` from Apache Commons' `ConstructorUtils` <https://commons.apache.org/proper/commons-lang/api/docs/org/apache/commons/lang3/reflect/ConstructorUtils.html>
- b. If using C#, check this answer on Stack Overflow: <https://stackoverflow.com/a/3255716>

To create a random figure, one can come up with a similar idea:

1. Enumerate all subtypes of `Figure`. Pick a random one.
2. Check its constructors for one, which has one or more `double` parameters (and nothing else but `double`). Note the number of parameters.
3. Generate that many random numbers and invoke the constructor.

After accomplishing this, you will be able to add or remove figures without having to alter the figure-creating factories in any way.

Skills and Competencies

Programming concepts:

- ¥ Assertions
- ¥ Exception handling
- ¥ Exception safety guarantees
- ¥ Invariants
- ¥ OOP principles
- ¥ Polymorphism
- ¥ SOLID
- ¥ Streams
- ¥ TDD
- ¥ Unit testing

Design Patterns:

- ¥ [Abstract Factory](#)
- ¥ [Factory](#)
- ¥ [Prototype](#)