Software Engineering
2016-17
Lab session 5
Adapters and polymorphic compositions

Objective

1. Implement the refactoring "Extract method".
2. Understand the necessity and functionality of the `Adapter` design pattern.
3. Test and implement a polymorphic composition.

Instructions

In this session we are given two (fake) libraries for converting a LibreOffice document to PDF and HTML. Each library has its own interface that differs in several aspects:

- Method name and nomenclature
- Text in string or char* format
- Use of namespaces
- Error management using return values or throwing exceptions

We will use adapters to provide a common interface that allows us to treat the two converters in the same way, in spite of their differences. The module `LibreOfficeTools` for converting documents to PDF is located in the file `libLibreOffice2Pdf.hxx`, and the module `OO_WarGeneration` for converting to HTML is located in `libLibreOffice2Html.hxx`, both in the `externalLibs` directory.

Refactoring "Extract method"

Programmers often add a comment at the beginning of a long section of code that explains what the code does (for example, `/* New formats are generated here */`). This is a symptom that it would be better to place the section of code in its own method, whose name reflects the comment (for example, `generateNewFormats`). The method containing the code will be easier to understand and it is sufficient to look up the method definition to obtain details about the implementation. Smaller methods are easier to understand and enable simplifications via "early returns".

To extract a method we have to pay attention to the data flow, identifying which data is needed as input and which data is modified. This helps us define the arguments of the new method. We can also perform this refactoring in reverse: substitute a method call with the content of the method. It is safe to try this back and forth since the tests provide us with the security that we are not breaking anything.

Guide

The lab session consists in implementing the new functional tests in `ConverterTests.cxx`. To facilitate the implementation of the tests we provide the following guide:

1. From inside the method `addWork` of the class `MeltingPotOnline`, extract a method `findAuthor` that takes the name of an author as input and returns a reference to the corresponding instance of `Author`, throwing an exception if the instance is not found. Add a call to the method `findAuthor` in `addWork`.

2. Go over your code and identify other sections of code that could form their own methods.

3. Implement the conversions using a fake:

   - Fail and pass the test `testConvertersByDefault_withoutOriginals`.
   - Fail and pass the test `testConvertersByDefault_withOneOriginal` in the obvious way: creating the files directly in the constructor using string literals that can be stored in an array, as in the following example:

```
const char* fakeGenerated[] = {
    "generated/An author - A work [multiple HTML files].war",
    "generated/An author - A work [printable].pdf",
    ...
    0 // This zero is needed to stop the loop
};
for ( int i = 0; fakeGenerated[i]; ++i )
    std::ofstream newfile( fakeGenerated[i] );
```

- Once the test passes, we can substitute the common parts of the string, each time checking that the test keeps passing. Only the suffixes should remain in `fakeGenerated`.
- Refactor: Extract the above code into an auxiliary method `generateConversions`.
- The test now passes, but the implementation is a fake since it does not use the conversion libraries to create the files, which are instead empty.

4. Adapt the library `OO_WarGeneration` using a class called `HtmlConverter`. Guide the implementation using the following unary tests (`HtmlConverterTests`, that you have to create on your own). Copy the `setUp`, `tearDown` y `createOriginalFile` methods from the functional tests.

   - `testConvert_generateFile`: The following code

     ```
     HtmlConverter converter;
     createOriginalFile( "Original.odt" );
     converter.convert( "originals/Original.odt", "generated/Prefix" );
     ```

     should create a file `generated/Prefix [multiple HTML files].war`. Use the method `LibFileSystem::listDirectoryInOrder` to test the name of the files in the `generated` directory. To pass the test it is sufficient to create an instance of `std::ofstream` with the name of the file as parameter. We can then refactor the code using the parameters of the `convert` method.
   - `testConvert_generateContent`: this test is basically the same as the previous test, but instead of testing that the file exists, we test the content of the file using the method `LibFileSystem::fileContent`. Our fake library adds the following text to the file:

     ```
     War file generated from 'originals/Original.odt'\n
     ```

     Pass the test by first inserting the string literal into the method `convert`, and then refactoring the code so that the `OO_WarGeneration` library is called.
   - `testConvert_withInexistentOriginal`: test the case in which we cannot perform conversion because the original does not exist. The library returns -1 but our interface expects an exception with the message `The original file does not exist`.
   - `testConvert_polymorphicCall`: the class `HtmlConverter` has to inherit from a class `Converter` that defines a common interface with `PdfConverter`. To test the polymorphic call it is necessary to create a polymorphic variable of type `Converter`, assign an instance of `HtmlConverter` to the variable, and call the method `convert`.

     Note: whenever you define a base class with virtual methods, you should include a virtual destructor method (usually empty).

5. Adapt the library `LibreOfficeTools` using a class called `PdfConverter`. The unit tests are analogous to `HtmlConverterTests`, but the implementation is different. Things to take into account:

   - The components of the library are inside the namespace `LibreOfficeTools`.
   - In case of error, you have to catch the exception in the library and throw the exception expected by our library. This is done using a `try/catch` clause (look at the functional tests to see how it works).
   - The exception in `LibreOfficeTools` does *not* inherit from `std::exception`.
   - The PDF converter has two modes: watermark or printable. Both the name and the content of the generated file is different depending on the mode. It is necessary to test the PDF converter in both modes. By default, the mode is 'printable' and the watermark is empty. The conversion is only done in watermark mode after calling the method `PdfConverter::activateWatermark( string )` with a non-empty string as argument.
   - To fail the polymorphic test without breaking the test of `HtmlConverter`, you can alter the signature of `convert` such that the method does not override the method in `Converter`. For example, an argument of the method could be a string copy instead of a constant reference.

6. Next, create a class `ConverterGroup` that manages the list converters (multiple polymorphic composition). We add the converters in the following way:

```
converterGroup.add( "html" );
converterGroup.add( "pdf_print" );
converterGroup.add( "pdf_mark" );
```

   In this way the specific converter type is hidden from the client (`MeltingPotOnline`). The converter group defines its own method `convert` that recursively calls the components. The unit tests in `ConverterGroupTests.cxx` follow the same scheme that we have used in other multiple compositions (`Author` to `Work` and `Collection` to `Work`), but at some point it is necessary to introduce polymorphism.

   - `testConvert_withHtmlConverter`: composition with 1 `HtmlConverter`. Implement delegation.
   - `testConvert_withoutConverter`: composition with 0.,1 `HtmlConverter`. Manage optionality.
   - `testConvert_withPrintablePdfConverter`: polymorphic composition with 0.,1 `Converter`. Make it polymorphic and add a conditional statement in the method `add`.
   - `testConvert_withWatermarkPdfConverter`: polymorphic composition with 0.,1 `Converter`. Add the new type.

- `testConvert_withHtmlAndPdfConverters`: polymorphic composition with 0..* `Converter`. Multiple cardinality.
- `testConvert_withUnknownConverter`: if we try adding a converter to `.doc`, an exception is thrown with message `Unsupported format`.

7. Once this infrastructure is in place, we can refactor the code and replace the fake implementation in `MeltingPotOnline`.

   - Duplicate: add an attribute of type `ConverterGroup` to `MeltingPotOnline`.
   - Fill: in the constructor, add all necessary converters to the converter group using the list of suffixes from `generateConversions`.
   - Change: instead of calling the method `generateConversions`, we instead call the method `ConverterGroup::convert` when a new work is added.
   - Cleanup: delete the implementation `generateConversions` which is no longer needed.

8. Optional: fail and pass the third and last functional test, which adds converters according to a configuration file whose name is passed as argument to an alternative constructor of `MeltingPotOnline`. Hint: since the content of the configuration file consists of words without spaces, the alternative constructor can parse the file using the following code:

```
std::ifstream config( configFile.c_str() );
std::string format;
while ( config >> format )
{
        // add the format
}
```

What to hand in?

The classes implemented in the `src` directory of your Git repository, and the unit tests in `src/unitTests`. You should also send an email to your lab session teacher in the format previously indicated.

When is the deadline?

Prior to the next lab session.