# Object Oriented Design - Exercise 1

### Design, Code and Test Using Structural and Traversal Design Patterns in Java

In this exercise you will:

- Design the first version of an XML pretty-printer
- Produce a set of UML class and sequence diagrams to document your design
- Experience design patterns in actual Java code
- Write a covering set of unit tests
- Experience the use of Eclipse, JUnit and Ant

The deadline for this exercise is Thursday, March 17th, 2005.

## XML Document Syntax

In this exercise you only need to support a small subset of the XML format standard. An XML document is a string which starts with a <Tag> and end with the same </Tag>. Inside the tag could be a text value, or other tags. For example:

```
<Book id = "123" status = "available">
    <Name>Refactoring</Name>
    <Author>
        <FirstName>Martin</FirstName>
        <LastName>Fowler</LastName>
        <!-- Here's an example of a list within a list -->
        <OtherBooks>
            <BookName>UML Distilled</BookName>
            <BookName>Analysis Patterns</BookName>
        </OtherBooks>
    </Author>
</Book>
```

XML documents follow these rules:

- A tag name must be start with an English letter, and may continue only with letters, digits or underscore (_) characters.
- Opening and closing tags must match (i.e. `<Name>John</FirstName>` is illegal).
- There must be exactly one root tag (i.e. not starting with a tag is illegal, and so is having two or more top-level tags).
- Tag names are case sensitive (i.e. `<name>Me</NAME>` is illegal).
- There is no limit to the depth of the tags hierarchy.
- The same tag can be used more than once (as with the `'BookName'` tag in the above example).
- Each tag can contain either a text value (like the `FirstName` tag) or other tags (like `Author`), but you do not need to support mixed tag (for example "`<a>111<b>222</b>333</a>`"). For general knowledge, mixed tags are supported by the XML standard.

Two special document elements in the above example are comments and attributes:

- A comment looks like a tag that starts with `<!--` and ends with `-->`. Comments do not have a closing tag, do not start a new hierarchy level, and have no other effect.
- An attribute is written as attrname = "attrvalue" inside the tag, after the tag name. A tag can have several attributes, but it cannot have two attributes with the same name.

In this exercise, you do not have to support the self-closing tags syntax such as `<Name value = "Fowler"/>` which the XML standard supports.

## Requirements

Write an executable program called 'xmlp' that can read an XML file, and output it (in this version) as an HTML file. The program takes as command-line arguments a filename, and two other optional arguments:

```
xmlp -nc -na filename
```

The program parses the XML document in the given file, and reports a detailed error and exits if it is not a valid XML document. Otherwise, it creates a file called `filename.html` (where 'filename' is the name of the input file), which includes the contents of the XML document in a more readable way:

- Each text element is described by a 'name: value' line
- Each list element is described by a 'name:' line, followed by its contents in the next line, indented by another tab
- Comments are described by a '\\ comment text' line
- Attributes are described by a 'name = value' string enclosed by parentheses, in the same line of the element they belong to.
- Comments and attributes should be written in *italics*.
- The title of the HTML page should be the input file name.

For example, if the XML document describing a book, in the previous section, were the contents of a file called `abook.xml`, then running `xmlp abook.xml` should produce a file called `abook.xml.html` with these contents:

```
<HTML>
<HEAD>
<Title>abook.xml</Title>
</HEAD>
<BODY>
Book: <i>(id = 123)</i> <i>(status = available)</i><br/>
   Name: Refactoring<br/>
   Author:<br/>
      FirstName: Martin<br/>
      LastName: Fowler<br/>
      <i>// Here's an example of a list within a list</i><br/>
      OtherBooks:<br/>
          BookName: UML Distilled<br/>
          BookName: Analysis Patterns<br/>
</BODY></HTML>
```

When displaying this file in a web browser, it will look like:

```
Book: (id = 123) (status = available)
    Name: Refactoring
    Author:
        FirstName: Martin
        LastName: Fowler
        // Here's an example of a list within a list
        OtherBooks:
            BookName: UML Distilled
            BookName: Analysis Patterns
```

The `-na` command-line argument implies that XML attributes should not appear in the output, and the `-nc` arguments implies that comments should not appear in the output. Both options may

appear together and in any order, but always before the input file name.

The program should not write anything to the standard output, and should write to the standard error in cases of error. Errors include inability to open the input file or create the output file, parsing errors of the input file, unsupplied input file and so forth. If the output file already exists, it should be overwritten.

## Design

While this exercise can be easily programmed within a single class, this won't work since this `xmlp` is only a first version, so it is crucial to maintain an open mind with respect to possible future requirements. Consider the following possibilities:

1. It may be required to produce output in other formats - either different HTML formats, text formats, LaTex or others.
2. It may be required to read different data formats of hierarchical data - relational tables with foreign keys, special-purpose formats and others, and be able to produce the same output for them.
3. It may be required to support other kinds of XML document elements besides texts, attributes and comments.
4. It may be required to dynamically define parameters of the output - for example, the fonts and colors in which different elements of the output are displayed.
5. It may be required to insert the HTML output into specified locations inside existing HTML pages (e.g, html code without the html, body .. tags), to display the output in a site which has frames, backgrounds, sounds, etc.
6. It may be required to modify the input document before pretty-printing it: for example, sorting the elements, changing tag names to uppercase, and so forth.

You must design your program so that it is easy to add code that implements the above requirements. Assume that you are the one who will actually have to code it - that's how it usually is in "real life". For each of the above requirements write an explanation in your *README* file, not more than three sentences long, which explains how it should be coded. For example:

> Requirement: It may be required to define filters on which parts of the input document get printed. For example, new command-line arguments can dictate that only the simple (non-hierarchical) elements get printed, that only elements that start with a given string get printed, and so forth.
>
> Solution: Write an Iterator for each kind of filter, whose `next()` method will move to the next element for which the filter is true. Such iterators are implemented as Decorators of other iterators, which easily enables to dynamically combine different filters and does not require changing or recompiling existing code.

It is important that each solution you present will be at most three sentences long. The intention is to enforce the use of design patterns vocabulary rather than elaborating specific class and object relationships.

## Code & Unit Test

This exercise intends you to divide your time equally between actual coding and between design, writing UML diagrams, and answering the above six design questions. Coding should be done in Java, using the Eclipse development environment. You should use the standard libraries to their full extent - use the standard streams, strings, data structures and so forth - but you may not use any existing XML parsers (DOM, SAX, etc.). With a proper design, this exercise is quick and simple to code.

It is also required that you submit unit tests to test your work. Organize your unit tests into classes by subject, and write a method for each small test. Each test should be self-validating - that is, know by itself whether it has passed or failed. Writing unit tests should be an integral part of coding, and is essential when code must be changed in newer versions. You will have a chance to estimate the convenience of unit tests in exercise 3. Until then:

- Read the following article about unit testing as part of coding and the build process.
- You must use the JUnit Unit Testing Framework for your unit tests.

Unit tests are used during development - you are most encourage to try doing test-first programming in this exercise - and for regression testing. To support this, the system's full build process must run all unit tests in addition to just building it. You must provide an ant file.
ant is a tool for building a project on which we will review in the next Tirgul.

The code you submit must be built with no compiler warnings, and pass all unit tests.

One metric for measuring the usefulness of a set of unit tests is called coverage, which means the percentage of your code that the unit tests actually run. Coverage of 90% or above is considered good, and you should aim to that goal.

This is an advanced course, so there is no intention to take points for coding style or naming conventions - the emphasis is on proper design. However, you are as always expected to write clear code with a consistent style.

## Submission

Our course uses the department's regular submission system, which means that your first step is to register for the course. Once that's done, submit a tar file as usual, with the following contents:

- All program source code
- All unit tests source code should be in a sub-directory called *test*
- The README file, with the usual contents (IDs, logins and full names, descriptive list of files and features) and answers to the six possible extensions in the design section above. The README file should also describe parts of the design or design choices that are not evident from reading the UML diagrams.
- An ant build.xml file that has tasks that will be explained in the next Tirgul.
  (Do not worry, creating a build.xml with ant should take no more then an hour, and is independant from the rest of the project)
- UML class diagrams, describing all classes in the program. There can be one or several diagrams, as long as they are readable.
- UML sequence diagrams of two non-trivial object interactions of your choice.
- All UML diagrams should be built using the plugin of Eclipse, and should be converted to jpeg (using right click on the diagram, and choosing the jpeg conversion). The diagrams should be submitted with the project as xmlpUcd.jpg and xmlpUsd.jpg (class and sequence diagrams).
- Eclipse .project and .classpath files , a script to run the program (an example in guidelines)

Submission is electronic only - do not submit any printouts or hand-written papers.

## How to Start

- Read these general guidelines for Object-Oriented design (especially 'Design Guidelines')
- Re-read about the Composite, Decorator, Iterator and Visitor design patterns
- Design the program; start with the data structures, then the major operations, and finish with the "main" program. Create UML class and sequence diagrams (see also this tutorial) as you go.
- Read about unit testing and the JUnit framework before starting to code.
- Code the program in the same three steps, writing unit tests in parallel with code. That is, write a test class for building the data structure, then a test class for the main operations, then a test class for the main program.
- If during the coding and testing process you decide to change the design, maintain the UML diagrams to reflect the changes.
- Register to the course and then submit the exercise according to the above instructions.
- For general knowledge only, you can also read the XML and XSLT standards.

Good luck!