XML Structure Editor

A dissertation submitted to The University of Manchester for the degree of MSc in Information Systems Engineering in the Faculty of Engineering and Physical Science.

2009

Filippos Gournaras

School of Computer Science
The author would like to thank the supervisor, Christopher Harrison, for his support and guidance from the beginning to the end of this project.

To my parents Kostas and Vivi

for making it possible

and to Aggeliki for the inspiration
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>7</td>
</tr>
<tr>
<td>Declaration</td>
<td>8</td>
</tr>
<tr>
<td>Copyright Statement</td>
<td>8</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2. Background Information</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Mentor</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Cornell Program Synthesizer</td>
<td>11</td>
</tr>
<tr>
<td>3. Design</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Syntax directed editors versus conventional editors</td>
<td>13</td>
</tr>
<tr>
<td>3.1.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>3.1.2 BNF metalanguage</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3 How a BNF syntax directed editor works</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Human Computer Interface</td>
<td>17</td>
</tr>
<tr>
<td>3.2.1 Introduction</td>
<td>17</td>
</tr>
<tr>
<td>3.2.2 Human Computer Interface Literature Review</td>
<td>17</td>
</tr>
<tr>
<td>3.2.3 A BNF structured editor’s Human Computer Interface</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Building a prototype</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1 Prototyping in software development</td>
<td>21</td>
</tr>
<tr>
<td>3.3.2 A BNF editor prototype for the XML editor</td>
<td>22</td>
</tr>
<tr>
<td>3.3.3 From the prototype to the complete application</td>
<td>24</td>
</tr>
<tr>
<td>3.4 Design of the BNF editor</td>
<td>25</td>
</tr>
<tr>
<td>3.4.1 BNF data structure</td>
<td>25</td>
</tr>
<tr>
<td>3.4.2 BNF editor’s user interface</td>
<td>26</td>
</tr>
<tr>
<td>3.5 C# and Microsoft Visual Studio</td>
<td>29</td>
</tr>
<tr>
<td>4. Implementation of the BNF Editor prototype</td>
<td>31</td>
</tr>
<tr>
<td>4.1 BNF Editor’s Overview</td>
<td>31</td>
</tr>
<tr>
<td>The syntax checks and structure commands are dictated by the BNF language</td>
<td>31</td>
</tr>
<tr>
<td>4.2 BNF Editor’s main classes</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1 Tree Class</td>
<td>32</td>
</tr>
<tr>
<td>4.2.2 Regions Class</td>
<td>34</td>
</tr>
<tr>
<td>4.2.3 Worker Class</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Testing the main classes</td>
<td>36</td>
</tr>
<tr>
<td>4.3.1 Testing the Tree Class</td>
<td>36</td>
</tr>
</tbody>
</table>
4.3.2 Testing the Regions Class ......................................................... 38
4.3.3 Testing the Worker Class ......................................................... 38

4.4 Implementation of the Graphical User Interface .............................. 39
4.4.1 BNF Editor’s Main Window ......................................................... 39

5. Implementation of the XML Editor .................................................. 41
5.1 XML, XML Schema and Validation .............................................. 41
5.1.1 XML Document Object Model ..................................................... 41
5.1.2 Validation .................................................................................. 42
5.1.3 XML Schema Languages ............................................................. 43
5.2 XML Schema Parser ...................................................................... 44
5.2.1 XSD features included in the schema specific parser ................. 46
5.2.2 Implementation of the XML Schema Parser ................................ 47
5.3 XML Parser ................................................................................... 48
5.4 XML Serializer .............................................................................. 49
5.5 How the XML Editor works ........................................................... 50
5.6 XML Editor’s main classes ............................................................. 51

6. Evaluation, conclusions and future work ......................................... 53
6.1 Evaluation of the editors ............................................................... 53
6.1.1 Software Evaluation ................................................................. 53
6.1.2 Author Evaluation ................................................................. 55
6.1.3 User Evaluation .................................................................... 56
6.2 Conclusions ................................................................................ 57
6.2.1 Usability Issues ................................................................... 57
6.2.2 User View .......................................................................... 59
6.2.3 Conciseness and Minimalist Design ....................................... 59
6.3 Future Work ............................................................................... 60

User Guide ........................................................................................ 63
BNF Editor User Guide ....................................................................... 63
XML Editor User Guide ..................................................................... 67

APPENDIX A – Users Evaluation ....................................................... 71
APPENDIX B – BNF Editor’s Class Diagram .................................... 78
APPENDIX C – XML Editor’s Class Diagram .................................... 80
APPENDIX D – XML Schema Example ............................................. 82
References .................................................................................. 83

Word Count: 15896
Table of Figures

Figure 1 A hybrid approach for structured editors. Adapted from Waters (1982) ....... 18
Figure 2 Adding a new rule ................................................................. 18
Figure 3 Entering the Left and Right Hand Side of a Rule ........................................... 19
Figure 4 Expanding an $Expression$ placeholder ...................................................... 20
Figure 5 Ensuring syntactic correctness ..................................................................... 21
Figure 6 The prototyping incremental software development cycle. Source: (Pomberger and Bischofberger, 1993) ........................................................................................... 23
Figure 7 BNF tree structure example ........................................................................ 26
Figure 8 Regions ........................................................................................................ 27
Figure 9 Regions Tree Example ................................................................................. 28
Figure 10 A structure editor's structure ....................................................................... 31
Figure 11 Tree Class UML Class Diagram.................................................................. 33
Figure 12 Regional Classes ........................................................................................ 34
Figure 13 Worker Class .............................................................................................. 36
Figure 14 The initial BNF grammar tree..................................................................... 37
Figure 15 Expanding the Sequence results to new children in the BNF grammar tree 37
Figure 16 Deleting one of the Sequence children results to a child removal in the BNF grammar tree .............................................................................................................. 37
Figure 17 Clicking on the left hand side is recognized based on the region of the left hand side .................................................................................................................... 38
Figure 18 Clicking on the right hand side is recognized based on the region of the right hand side .................................................................................................................... 38
Figure 19 Editor's main window .................................................................................. 40
Figure 20 Document Object Model (Source: http://www.xml.com/1999/07/dom) ........ 42
Figure 21 DTD versus XSD ........................................................................................ 44
Figure 22 XSD schemas with and without element and type reference. Source: (Walmsley, 2001) ........................................................................................................ 46
Figure 23 Parsing XSD to PSVI .................................................................................. 48
Figure 24 Interactions between the application's components .................................... 50
Figure 25 XML Editor's DOKM and PSVI node classes .............................................. 52
Figure 26 User Interface Users Evaluation .................................................................... 56
Figure 27 Usability Users Evaluation .......................................................................... 57
Figure 28 A hybrid XML structure editor ..................................................................... 62
Figure 29 File menu .................................................................................................... 63
Figure 30 Add a new rule Window .............................................................................. 64
Figure 31 A Sequence expression placeholder denoted by the "$" symbol .................. 64
Figure 32 Expanding a Sequence Expression ............................................................ 65
Figure 33 Expanding an alternative expression .......................................................... 65
Figure 34 Expanding a symbol ................................................................................... 65
Figure 35 BNF Editor’s dynamic context menu ........................................................... 66
Figure 36 File Menu in the XML editor ........................................................................ 67
Figure 37 Loading an XML Schema .......................................................................... 67
Figure 38 XML Editor Context menu ........................................................................... 68
Figure 39 Selecting an element .................................................................................. 69

By Filippos Gournaras
Abstract

This report describes the process of designing and implementing an XML structure editor. The application to be developed is a syntax directed editor for XML designed for use as a teaching tool. The first sections of this report present and analyze literature on syntax directed editors and XML. This literature describes the theoretical basis for the application and also C# and Microsoft Visual Studio that will be used for its development. The project will follow the incremental system development method, meaning that a BNF editor prototype will be developed first to provide a working framework for a subsequent complete XML editor.
Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

i. Copyright in text of this dissertation rests with the author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the author. Details may be obtained from the appropriate Graduate Office. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.

ii. The ownership of any intellectual property rights which may be described in this dissertation is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

iii. Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the School of Computer Science.
1. Introduction

The aim of this MSc project is to produce an *XML structure editor*, specifically a *syntax-directed editor*, for use as a teaching tool. Structure editors are based on *grammars* that specify the legal *operators* and *expressions* allowed in *strings* in a *language*. By permitting only *correct* strings (e.g. in documents) *syntax errors* are eliminated and valuable feedback about available editing options is provided to the user. In a conventional editor (or *text editor*) the user provides sequences of characters as input and the text editor processes this input only at a presentation level. Conventional editors usually indent and colour elements that correspond to *keywords* of a programming language. Structure editors, however, *will allow the user only the editing options that are permitted for legal strings as defined by the grammar for a language*.

These features along with an intuitive graphical interface make structure editors appropriate for *novice users*. The pedagogical (or educational) value of structure editors as teaching tools for students has been acknowledged since the 1970’s. Structure editors encapsulate many benefits for the user, including construction of elements, elimination of typographic or other minor errors at the time strings are written, and more rapid debugging and testing of programs as there are no compilation errors (Zelkowtiz, 1990). All these benefits lead to an increase in productivity, especially for novice users.

Since its introduction EXtensive Markup Language (XML) has been considered as HTML’s successor. XML was originally created by the World Wide Web Consortium (W3C) “to meet the challenges of large-scale electronic publishing” (W3C, 2009). The main characteristic of XML that differentiates it from HTML is the ability for information publishers to create their own tags or use shared tags. In the past decade XML has developed into a valuable tool for content management, documentation, databases and a basic component of Asynchronous JavaScript and XML (AJAX) technology that has heavily influenced web development.
2. Background Information

This chapter is a study of the literature available on syntax directed editors and syntax directed editing environments. The most historically significant structure editors and their strengths and weaknesses are discussed.

2.1 Introduction

The first generations of syntax directed editors had to overcome huge difficulties, especially in the design of their user interfaces. As structure editors had never been implemented before (at that time), they invoked great interest and some attempts were successful. This section provides a literature review of the most significant of the syntax directed editors that have been developed.

2.2 Mentor

Mentor was an ambitious project that was developed in the late 1970’s and aimed to provide an interactive programming environment. In its essence Mentor was a program for manipulating structured data and was based on Mentol, a tree manipulating language. With Mentol, data was represented in an abstract syntax tree (AST), which was used for structure manipulation.

The Mentor system formed the basis for Mentor-Pascal, an interactive programming environment for Pascal developed by the creators of Mentor. Mentor-Pascal uses a multi-pass approach for programming that consists of the following passes: during the program input, the program is checked for syntactic correctness, and then the program is checked so that every identifier has been declared (before its use) by the “scoper”. Subsequently, the type checker takes over to check whether all the programming operations are well-formed (a form of semantic checking rather than syntax checking). Then, the program is checked for runtime errors and finally Mentor-Pascal attempts to check if the
A program is corresponding to the computation intended by the programmer with the help of a semi-automatic theorem prover (Donzeau-Gouge et al., 1980).

Even if Mentor did not implement successfully all the features its creators intended, it most significant contribution to the first generations of structure editors was the fact that it “proved” that the creation of an interactive programming environment is feasible and provides many advanced features for programmers.

**2.3 Cornell Program Synthesizer**

The Cornell Program Synthesizer (CPS) (Teitelbaum and Reps, 1981) was one of the first syntax-directed programming environments that succeeded in capitalizing on the benefits of structure editors. First used in classes at Cornell University in 1979, the high level of abstraction provided by CPS, combined with “top-down” program development support led to its immediate success. Two years later CPS was used by 3000 students per year in various universities.

CPS was based on its syntax directed editor which guaranteed syntactically correct programs for the host programming language. Program creation was simplified with the use of templates, placeholders and phrases. Templates and phrases are the main programming elements, while placeholders notate positions requiring additional code. Templates are predefined “code snippets” that contain phrases and placeholders. Phrases include all the assignments, declarations, expressions and variables and are entered directly by the user. However, input in phrases is syntactically checked and errors have to be corrected manually.

Programs are created by the insertion of templates and the replacement of placeholders with either templates or valid phrases. Template insertion is done through commands and insertion of phrases directly by text. The user can access all the templates, phrases and placeholders for modification with the cursor, which is moved with the terminal’s cursor keys. The cursor can only move over elements that are allowed to be modified.
This top-down programming approach allows instant execution and rapid testing at any stage of development. CPS also provides debugging in the form of variable monitoring and single-step execution. These features were implemented to promote the usefulness of CPS as a teaching tool for novice users and students.

The decades that have elapsed since the introduction of CPS in the family of syntax directed editors have made CPS’ user interface obsolete; however the design principles and development goals can still inspire and provide insight into structure editors. The basic principles of CPS’ design are (Teitelbaum et al., 1981):

- Specialization is facilitated with the use of templates that can be inserted through commands. Specialization eliminates the need for memorization, typographic errors and time consuming code writing.
- Constraint is what differentiates CPS from a traditional editor. The user can insert data and modify the program only in positions that it is meaningful and syntactically correct. Therefore, templates are predefined and the cursor is restricted to positions that modifications are permitted.
- Consistency refers to a simple and consistent user interface. A small set of well-defined fundamental rules in the user interface leads to gentle learning curve for novice users.
- Manual control of the program creation process. CPS’ creators preferred error detection rather than error detection. This choice was made to avoid the confusion involved in automatic error correction, and
- Immediate visual response means instant visual feedback for every action taken by the user. The programming environment detects errors as soon as every action is completed. Syntactic errors and illegible actions will be immediately highlighted and correction will be demanded.
3. Design

This chapter focuses on the design of the applications that will be implemented in this project. It includes a presentation of the program’s structure and features. As the project will follow the prototyping development approach, it will be explained how the prototype will be built, how the prototype will evolve into the final application and what are the expected benefits of this approach.

3.1 Syntax directed editors versus conventional editors

3.1.1 Introduction

The differentiation between syntax directed and conventional editors lies on the ability to incorporate semantics into document authoring or program writing. As described in the introductory section of this paper, the syntax directed editor has knowledge of the document syntax allowing only valid actions and options. This section describes through examples using Backus-Naur Form (BNF) the basic principles of a syntax editor (Scowen, 1993).

3.1.2 BNF metalanguage

BNF is a syntactic metalanguage that can define a language by providing rules (i.e. production rules) that can produce all members of the set of strings that form the language. Production rules consist of a left hand side and a right hand side separated by the := symbol. These rules mean that the left hand side can be replaced with what is on the right hand side. The right hand side can consist of symbols and expressions. Expressions are either alternatives or concatenations of symbols or expressions. Symbols that cannot be further decomposed are called terminals. The following example displays a BNF production rule, which can be translated as $S$ can be replaced by a concatenation of $\text{Choice1}$ or $\text{Choice2}$ and $\text{Terminal}$.
3.1.3 How a BNF syntax directed editor works

In contrast to conventional editors, in which creating a rule is done by simply entering a new line and a sequence of characters, in a structure editor creating a rule is done by inserting a new template like the one in the example that follows:

Example: \[S \rightarrow <\text{Choice1} >\mid <\text{Choice2}>\] <Terminal>

In this example \(<\text{Symbol}>\) and \(<\text{Expression}>\) are placeholders that need to be replaced by the actual symbols and expression. For expanding the \(<\text{Expression}>\) placeholder the structured editor provides the following template options:

- \(<\text{Symbol}>\)
- \([<\text{Expression}>\mid <\text{Expression}>]\)
- \([<\text{Expression}>\text{ <Expression}>]\)

This combination of strictly predefined and restricted input aims to achieve the two basic goals of a structure editor. Firstly, this method ensures correct syntax and secondly it enables time saving and intuitive production rule creation.

However, a structure editor’s great strength lies in modification. In an existing BNF grammar, modifying a symbol or an expression necessitates changes in all the succeeding production rules that contain the symbol or expression that have been modified.
This example provides a BNF grammar for integer and decimal numbers. It is obvious that this grammar cannot represent negative integer numbers. This problem can be addressed by changing the `<integer>` production rule to `<unsigned integer>` and adding a new rule for `<integer>`. Normally, using a conventional editor this would invoke a series of tedious copy-paste, deleting and inserting actions. A structure editor, however, can simplify the process. Changing `<integer>` to `<unsigned integer>` produces the following grammar in a single step:

Example:

```
<integer> ::= <digit> | <integer> <digit>
<decimal fraction> ::= . <integer>
<decimal number> ::= <integer> | <decimal fraction> |
<integer> <decimal fraction>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```
Now we can add a new integer rule that will include negative integers:

Example:

```
<integer> ::= <unsigned integer> | - <unsigned integer>
              | + <unsigned integer>
<unsigned integer> ::= <digit> |
                      <unsigned integer> <digit>
<decimal fraction> ::= . <unsigned integer>
<decimal number> ::= <unsigned integer> |
                       <decimal fraction> |
                      <unsigned integer> <decimal fraction>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Exploiting a structure editor’s unique features when deleting a rule or symbol involves automatically updating the whole set of production rules. Continuing from the previous examples, deletion of the `<decimal fraction>` non-terminal symbol will result to the following set of rules:

Example:

```
<integer> ::= <unsigned integer> | - <unsigned integer>
              | + <unsigned integer>
<unsigned integer> ::= <digit> |
                      <unsigned integer> <digit>
<decimal number> ::= <unsigned integer> |
<decimal fraction> | |
<unsigned integer> <decimal fraction>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Once `<decimal fraction>` is deleted, the right hand side of `<decimal number>` rule needed to be updated by removing all the `<decimal fraction>`. The results is a valid set of BNF rules, even if `<decimal number>` with the deletion of `<decimal fraction>` has degenerated to an integer. It is important to note that for the final
outcome the author is still responsible. A structure editor can only guarantee syntactic correctness.

3.2 Human Computer Interface

3.2.1 Introduction

One of the most important drawbacks of a structure editor is the feeling from the user’s viewpoint that the editor is “holding back” the writing process. Users find conventional editors more “convenient”, mainly because of the high level of freedom provided. This hindrance for structure editors can be neutralized by a cleverly designed user interfaces and this crucial role of the user interface highlights its importance in this project.

3.2.2 Human Computer Interface Literature Review

Waters proposed a hybrid form for structure editors (1982) arguing that a writing process based exclusively on commands and templates restricts the users and feels unnatural to them. Waters also proposed a basic architecture for structure editors that allows them to integrate a textual viewpoint with the strict parse tree viewpoint.

This hybrid architecture is presented in Figure 1. As the diagram illustrates, the editor uses two representations for the document. The first is a textual representation, for which text commands are used, while the second is a parse tree representation, for which structure editor commands are used. Syntax checks are available for the parse tree representation. The pretty printer’s role is to present the parse tree as text to the user and the parser’s role is to update the parse tree according to the text commands.
3.2.3 A BNF structured editor’s Human Computer Interface

This section will demonstrate how a structure editor for BNF would work via screenshots that display the main features and options of a structure editor. The following screenshots show the process of writing the BNF example grammar for decimal integer representation that was used in the previous section. The first screenshot displays how to add a new production rule in an empty document.

![Screenshot of BNF Syntax Directed Editor](image)

Figure 2 Adding a new rule
All the basic actions for authoring a document in a structure editor are accessible through a menu. As it has already been described in the previous sections, a structure editor does not allow the user to input data freely; it has to ensure syntactical correctness. The “adding a new rule action” can be accessed from the main right-click menu and it will invoke another menu for left and right hand side input, as displayed in the following screenshot:

![Figure 3 Entering the Left and Right Hand Side of a Rule](image)

Through this second menu, the user can input the left hand side symbol of the rule and choose the type of right hand side of the rule, choosing between expression, and terminal symbol followed by expression. The result of this action is a production rule with <integer> as left hand side symbol and a placeholder for expression in the right hand side of the rule. The $expression$ placeholder can be expanded –again through a menu- as shown in the next screenshot (Figure 4). Selecting “expanding the placeholder” with an alternative or sequence expression will result in an additional placeholder that will need to be expanded, while selecting a symbol will lead to a second menu for the symbol input, similar to the one used for the input of the left hand side of a rule in Figure 3.
Using the expression placeholder menu to complete the <integer> rule (as has already been shown) results to the automatic addition of the <unsigned integer> production rule. A non-terminal symbol in BNF is not allowed without a corresponding production rule. To preserve syntactic correctness, the editor adds a default production rule for the unexpanded <unsigned integer> symbol that contains an expression placeholder as its right hand side. This behavior of the editor results to the following screenshot:
3.3 Building a prototype

3.3.1 Prototyping in software development

Prototyping is a development methodology that derives from conventional engineering disciplines. In such engineering disciplines, prototyping means the creation of a model that will be used for subsequent mass production. In contrast to conventional engineering disciplines, in software development prototyping means the creation of a basic version of the software that implements at least the most important features, and which can subsequently be incrementally modified to include further functionality.

Connell and Shafer (1989) in their structured rapid prototyping methodology defined the prototype as a functional version of the software that can be produced with minimal effort and which is flexible enough for the modifications needed to be transformed to a complete system. Its main goal is to present and implement the key characteristics of the complete system.
The intermediate step of a prototype version has proved to be invaluable, especially for interactive applications, for two main reasons (Floyd, 1984):

- The wide scope of complicated applications does not allow the concurrent development of all the features. Phase-oriented development would narrow the scope by dividing the application to components that would be developed separately. However, this approach does not guarantee that these components will finally integrate with each other successfully. A prototype narrows the scope by only developing the basis for the final application. After a fully working prototype has been developed, adding the more advanced features will only need minor additions and is guaranteed to be successful.

- A programmer usually knows how to build successfully an application only after finishing it. Prototyping helps programmers to capitalize on the experience gained during the process of building the prototype, thus leading to a more stable and polished final version.

3.3.2 A BNF editor prototype for the XML editor

This project will follow the incremental system development method, which is based on the idea of gradually expanding a basic solution in order to solve a more complex (but, equally importantly, similar) problem. This decomposition into distinct steps will simplify and accelerate the design and implementation of the project. Figure 6 presents the prototyping software development cycle. This figure successfully illustrates the main advantage of prototyping, which is the ability to build the complete system based on a “known to be working” prototype.
A BNF editor prototype will be developed for the needs of this project. An XML editor is a complex application that would be very hard to implement on the first try. On the other hand, a BNF editor encapsulates all the basic features that an XML editor needs, avoiding more advanced features – like the XML validator – in the first stages of development. What is more significant is that BNF and XML share the same data structure (i.e. they are both inherently tree structured) This development approach has significant advantages:

- BNF is a basic yet mathematical language that provides a solid framework for the transformation to XML.
- The simplicity of BNF can help in simplifying the design and implementation process in the first stages eliminating the need complicated features and options. With the prototype, these can be addressed in the final stage, based on well designed and finalized mechanisms.
- Design and implementation can be accelerated by the fact that any refactoring, which may be required in the first stages of development, involves BNF and will be faster and easier to apply.
• BNF and XML can both be represented using tree data structures, thus ensuring an easy transition from the prototype to the final application.
• The experience gained in the process of building the prototype can be re-applied directly to the process of building the final application, leading to significant improvements.

3.3.3 From the prototype to the complete application

One of the most significant reasons for following the prototyping approach in this project is the capability to evaluate the prototype. Instead of evaluating the application after its completion when time resources will be scarce, the evaluation of the prototype will provide useful feedback from volunteers at a stage when modifications will be simpler, less time consuming and easier to plan.

The final stage of the development will be the transition from the prototype to the complete application. Both BNF and XML share a common data structure representation, thus simplifying the transition process. Additionally, all the major features available in the BNF editor, such as the refactoring options, will only require minor modifications and more importantly the prototype will provide the user interface for the final application.

However, the final XML editor will require enhanced features compared to the prototype. First of all, what differentiates XML from BNF in terms of document authoring is the validation of XML documents against a DTD or XML Schema. A valid XML document is a document that conforms to the structural constraints imposed by the DTD or XML Schema. A syntax directed editor for XML has to ensure syntactical correctness, which in this case is both the well-formedness and the validity of the XML document. The final stage of creating the XML editor will introduce to the application all the enhanced features.
3.4 Design of the BNF editor

3.4.1 BNF data structure

As it has already been described in the previous sections, the first stage of this project will be the BNF editor prototype. This section describes the design and implementation decisions for the prototype.

Even though BNF resembles a list of rules (i.e. textually), in its essence it is a tree structure with multiple children for each parent node. In this tree structure, every rule is a tree node with two children: the left and the right hand side of the rule. The right hand side is also a tree node, whose children can be sequences, alternatives or symbols. Every sequence or alternative is also a tree node containing a sequence, an alternative or symbols.

This BNF data structure can be explained using one of the previous BNF examples. The following BNF rules can be represented with the tree structure in Figure 7. Every rule is a tree node and every leaf of the tree is a symbol.

Example:

\[
\text{<integer> ::= <unsigned integer> | - <unsigned integer>}
\]
\[
| + <unsigned integer>
\]
\[
\text{<unsigned integer> ::= <digit> |}
\]
\[
\text{<unsigned integer> <digit>}
\]
3.4.2 BNF editor’s user interface

The BNF editor is based on a simple user interface. All the actions are accessible via a right mouse button context menu. Depending on the position of the mouse pointer on screen, the program can determine the element of the BNF that the user is pointing at. This function enables the program to present a different menu for every element depending on its type. For example, a right-click on a symbol will present the user the options to erase, modify or replace the symbol, while a right-click on “empty space” will present the option to add a new rule.

From a design and implementation viewpoint, the method used for determining the element pointed at by the user is of crucial importance. The BNF editor features a “regions” data structure that represents the on-screen content of the program. Every element of the BNF grammar is assigned its on-screen coordinates, thus a region of the screen is marked as containing this element. The region is a box surrounding the element. The calculation of the dimension of this region is possible by calculating the length of the element and knowing the dimensions of each character. So, each element is assigned
the coordinates of its bottom left and top right corner of its region; they suffice for calculating the region box that surrounds it.

Figure 8 displays a production rule and its regions. The rule itself, each side of the rule, as well as every symbol that is contained is a region in their own right. This means that the rule is represented as a larger region (green box in figure) containing the regions of each side of the rule (orange and light blue boxes in figure). The regions of each side contain the expressions’ regions (blue boxes).

This scheme is necessary for the program to provide the correct options for every click. For instance, a right click inside the region of `<unsigned integer>` (blue box in figure) should bring up the menu options available for a symbol that lies within a right hand side of a rule, while a right click inside the right hand side region, but not inside the region of a symbol should bring up the menu options available for the right hand side of a rule. Similarly, a right click inside the region of a rule, but outside of the regions of its left and right hand sides should bring up the menu options available for rules.

To achieve this function, regions are represented by a tree data structure. Each node of the tree is a region that is contained in the region of its parent node and may contain one or more regions as its children nodes. The nodes also contain
the coordinates for each region. The regions illustrated in figure 8 form the following tree shown in figure 9:

![Regions Tree Example](image)

**Figure 9 Regions Tree Example**

Based on this tree, the program can easily determine the active element (or, more precisely, the currently selected region). If the right mouse button click occurs at \((x, y)\) coordinates, the program starts a depth-first search at the regions tree looking for regions that contain the coordinates \((x, y)\). The path to the region in the deepest level determines the menu options that will be available to the user.

For instance, if right mouse button click’s coordinates are \((x, y)\) with \(x_2 < x < x_5, y_2 < y < y_2\), the program will find that the click occurred inside the region of root, subsequently, rule 1 and finally right hand side of rule 1. These regions describe the right hand side of the first rule but not a specific symbol, thus revealing the intention of the user to modify the right hand side of the rule as a whole and not a specific element inside it.
3.5 C# and Microsoft Visual Studio

The project will be implemented in C# and Microsoft Visual Studio. This programming platform will allow the development of an application with intuitive graphical user interface that will provide a rich user experience. As it has already been analyzed, the success of a syntax directed editor depends significantly on the convenience provided to the user. C# and Microsoft Visual Studio are modern, sophisticated and well documented tools widely used in the development of commercial applications. C# is an object oriented programming language created to build and improve upon the success of C and C++ languages.

“C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, and Java programmers.”

(Hejlsberg et al., 2003)

Stefik and Bobrow (1986) have successfully summarized the basic concepts of object oriented programming:

- The use of objects combines data and procedures in the same data structure
- Objects interact with each other via messages. Objects respond to the messages they receive by using their methods.
- Objects are categorized to classes and instances. Classes make data abstraction possible, as they represent a data type and all its procedures and instances are objects in a class.
- Specialization allows inheritance between classes, thus simplifying the creation of similar objects.

The main benefits of object oriented programming include minimal effort to understand and maintain software, responsiveness to events (such as mouse
clicks), reusability of components and extensibility in the form of sub-classing and customizing existing components (Fayad and Schmidt, 1997).

Microsoft Visual C# is an industry-leading powerful and efficient solution for developing software for Windows and was chosen for the project as it provides full access and exploitation of the Windows API system. The fact that Visual C# is an event-driven programming language, allowing events to affect the main flow of the program, makes it a fitting choice for applications with graphical user interfaces.
4. Implementation of the BNF Editor prototype

This chapter describes the implementation process for the prototype BNF editor. The program’s main software components are presented in attempt to explain how the prototype works. This chapter also presents the main classes that implement these software components, as well as an overview of the graphical user interface of the program.

4.1 BNF Editor’s Overview

The basic principles of a BNF structure editor have already been discussed in the design of the BNF editor. The basic software components implemented for the application are:

- The BNF grammar tree that is the tree data structure representation of the BNF grammar.
- The pretty printer that is responsible for printing (i.e. unparsing) the BNF grammar to the user in a concise and precise textual format.
- The parser that is responsible for presenting to the user only the “legal” options depending on the position of the active element of the grammar.

![Figure 10 A structure editor’s structure](image)

The syntax checks and structure commands are dictated by the BNF language.
4.2 BNF Editor’s main classes

A diagram of all the classes implemented for the BNF editor can be found in Appendix B. The next sections of this chapter will describe the most basic classes implemented for the BNF editor that will also be reused for the final application.

4.2.1 Tree Class

Both BNF grammars and XML can be represented in memory by a tree data structure. Moreover, the regions representing each element’s position on-screen can also be represented by a tree structure. Since C# does not provide a built-in tree class, it had to be implemented from scratch. The same tree class should be used for both BNF and XML; therefore the class implemented takes advantage of generics (also termed parameterized types) to provide a common tree class that can be used for any tree data structure. Generics allow a class to define a data structure without actually committing to an actual data type for elements of the data structure. This eliminates the need to implement a different tree class for BNF, XML and regions. The same tree class can be used without defining the data type of each node. The data type of each tree can be defined when each tree gets instantiated. Generics are a very powerful feature and the main reason why C# was chosen for the project: the same tree class will be used for both the BNF and the XML editor, without rewriting the code, ensuring a fast and bug-free transition from the prototype to the final application.

Figure 11 shows the UML Class Diagram for the Tree class. The base class for the tree class is the tree node class. Every tree node has a parent node, a list of its children nodes (which is a tree node list), a region (that represents its position on screen) and a value. The value is a generic type. Depending on the application the value can be either a BNF rule or an XNL node.

The main methods of the tree node class are:

- Add_down : Adds a child node to the tree node.
- Add_right: Adds a sibling node to the tree node.
- Calculate_region: calculates the coordinates of the region of the node
- Active: this method returns true if the user is has clicked inside the region of the tree node

These methods suffice for all the necessary operations for each tree node. The TreeNodeList class handles complements the TreeNode class with all the necessary operations for the list of each node’s children. The TreeNodeList is a subclass of the built-in List<> class that implements a generic list.

Finally, the Tree class is merely a subclass of the TreeNode class. The only additional methods are a print method that prints to the screen the whole tree and a method that calculates the region of the tree and all of its nodes.

![Figure 11 Tree Class UML Class Diagram](image)
4.2.2 Regions Class

As it has been described while discussing the design of the prototype each element of the BNF grammar on screen is surrounded by a “box” that can be calculated by a set of (x, y) coordinates. The first coordinates (x, y) are the bottom left corner of the box and the second (X, Y) are the top right corner of the box. From these coordinates the program can easily calculate whether the cursor position is inside the region of an element and thus which element is being clicked on. The classes that implement regions are shown in figure 12 below:

![Image of Regional Classes](image)

**Figure 12 Regional Classes**

*Regional* is the class that implements the region of each element. It contains an instance of coordinates (box) and a method to print the region of an element used for debugging purposes.

*Coords* class is a class that implements the coordinates of the box that surrounds each element.
Mouse_xy is a class that represents the coordinates of the mouse pointer on screen.

Finally, MyGlobals is a global class that includes two properties:

Charlength and height are the length and height in pixels of a single character on screen respectively.

### 4.2.3 Worker Class

The worker class is the class that implements all the features of the editor. Figure 13 shows the UML class diagram for the worker class. The worker class contains click_x and click_y that are the coordinates of the last mouse click and grammar which is the BNF grammar tree represented by the Tree class described previously.

The role of this class is to act as an intermediary between the graphical user interface and the tree structure that represents the BNF grammar. The following methods collect input data from the user and pass it to the BNF tree:

- Add_rule
- Expand_alternative
- Expand_sequence
- Expand_symbol

All of these methods create a new rule or regular expression that will be added or replace an existing element of the BNF grammar, based on the Tree class methods.
4.3 Testing the main classes

The classes described in the previous sections of this report are the main software components. The functionality and stability of the application rely on the effectiveness of these main classes.

4.3.1 Testing the Tree Class

The Tree class implements a tree data structure. In the application the tree is the document’s data structure. Therefore, the Tree class needs to implement methods for adding and removing children (or subtrees), as well as providing access to the interface of each node. The Tree class was tested for these operations using the tree view panel of the application. In each editor the Tree is bound to the tree view panel that presents to the user the structure of the
document. When a node is added or removed the tree view shows the updated Tree.

The figures that follow show the testing of the Tree class. Initially the BNF grammar tree includes one rule with a sequence as a right-hand expression. Expanding the sequence to two new sequences is translated into adding two sequence children to the initial sequence node. The updated tree is shown in the left-side tree view panel. When the newly added sequence child is removed, the result is removal of the sequence child from the tree.

![Figure 14 The initial BNF grammar tree](image1)

![Figure 15 Expanding the Sequence results to new children in the BNF grammar tree](image2)

![Figure 16 Deleting one of the Sequence children results to a child removal in the BNF grammar tree](image3)
4.3.2 Testing the Regions Class

The Regions class is responsible for calculating the on-screen coordinates for each expression. When the user presses the right-click mouse button the application relies on the Regions to determine which expressions the user was clicking on, namely the “active” expressions. The context menu will present to the user the options available for all the “active” expressions. The following figures show how the program identifies the active expression based on the Regions class. Clicking on the left-side of a rule or the right-hand side of a rule is shown in the context menu. In figure 18 the program also identifies the sequence expression that the user clicked on.

![Figure 17 Clicking on the left hand side is recognized based on the region of the left hand side](image1)

![Figure 18 Clicking on the right hand side is recognized based on the region of the right hand side](image2)

4.3.3 Testing the Worker Class

The Worker class is a class that operates as an intermediary between the graphical user interface and the data structures used for the representation of the document. As is shown in the previous screenshots, this class is working...
properly, facilitating the adding, removing and expanding operations on the data structures of the application based on the actions of the user via the graphical user interface.

The remainder of the application is tested in the User Guide Appendix.

4.4 Implementation of the Graphical User Interface

4.4.1 BNF Editor’s Main Window

The BNF Editor prototype was implemented with the transition to the final XML Editor in mind. This is the reason why the main window (presented in figure 19) is split in two parts. The first part consists of a tree view of the BNF grammar and the second part is the main part of the structure editor where the user can insert and modify the rules of the grammar. The tree view was found necessary so that the user can have a quick overview of the grammar and its structure even when the list of rules gets long and complicated. The tree view uses colours to identify different elements of the BNF grammar. Rules are denoted by light green, while sequence expressions by blue, alternative expressions by dark green and symbols by orange. The user can quickly get acquainted with the colourisation scheme and identify easily the regular expressions within a rule.
Figure 19 Editor’s main window
5. Implementation of the XML Editor

This chapter presents the implementation of the final application of the project, namely the XML editor. Before the mechanics of the application can be explained, the focus is on the most important aspects of XML technology and how the application exploits them. Finally, this chapter describes the XML editor’s main classes and how the classes developed for the prototype can be reused.

5.1 XML, XML Schema and Validation

5.1.1 XML Document Object Model

The Document Object Model (DOM) is a platform and language independent interface that provides access to the structure and content of HTML and XML documents. DOM is used by programmers to navigate access and modify elements of XML documents.

The DOM is based on a hierarchical tree structure of nodes, as illustrated in figure 20. The document is represented by a Document node and every element by an Element node. The main components of each node are its name, value, attributes and type and each node is linked with its parent, children and siblings (Marini, 2002).
The DOM in this project is used in two different methods of the program. As will be described in a following section, all the editing and authoring actions derive from an XML Schema validation. An XML Schema is a document written in XML, which contains rules that describe an XML document. In this context the DOM is used to parse the XML Schema and extract all these rules from it. Additionally, the DOM tree was chosen as the data structure to represent the XML document processed by the user.

5.1.2 Validation

XML documents need to be well-formed. Well-formedness in XML means compliance to a basic set of rules. An XML document must contain one root element, elements have to be properly nested, end tags must not be omitted and attributes have to be quoted. Well-formedness is necessary and guarantees that the XML document can be parsed to a DOM tree; however it is not enough for XML to fulfill the needs that led to its creation.
The notion of well-formedness was extended to validation, so that XML documents can be meaningfully parsed with the aim to extract not only data but “information” as well. Schema languages have been developed to support validation against a schema. Schemas are documents that define the structure and content of XML documents. An XML document can be validated against a schema only if it conforms to the rules that the schema contains. These rules usually define the elements and their attributes and datatype (Nicola and John, 2003).

As has been already been described, an XML document can be parsed to a DOM tree. However, validation provides additional information about the content and semantics of the document’s nodes. This information can be added to its node of the DOM tree, thus creating a Post Validation Schema Infoset (PSVI). PSVI mainly contains the legible types and values for each node and contributes to providing semantic data for applications based on XML (Coen et al., 2004).

5.1.3 XML Schema Languages

One of the most important characteristics of XML is its ability to be self-descriptive. XML’s self-descriptiveness is facilitated by associating an XML document with schemas, such as Document Type Definition (DTD), XML Schema Definition (XSD), Schematron and Relax NG. DTD’s and XSD’s are the most common schemas and are two different means of specifying the structure, content and semantics of an XML document. For this project only DTD and XSD were considered because at the present time they are by far the most common schemas for XML.

DTD was the first major schema language introduced for DTD and is widely used. DTD is inherited from SGML (Standardised General Markup Language) and contains rules that define the elements of an XML document and their attributes. However, the inability to support namespaces and its weak datatype system soon highlighted the need for a more powerful schema language for XML.
XSD became a W3C Recommendation on 02 May 2001 as a successor to DTD. XSD include extensive support for datatypes and is written in XML. Thus an XML document can be described by another document also written in XML.

The example below (figure 21) shows a DTD and an XSD for a simple XML document describing a book. Even in this simple case it is evident that XSD can provide more tools for defining structure and content in XML.

<table>
<thead>
<tr>
<th>DTD</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;!ELEMENT book (title,author)&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!ELEMENT title (#CDATA)&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!ELEMENT author (#CDATA)&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!ELEMENT isbn (#PCDATA)&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!ATTLIST book isbn CDATA &gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;xsd:element name=&quot;book&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td><a href="">xsd:complexType</a></td>
</tr>
<tr>
<td></td>
<td>&lt;xsd:element name=&quot;title&quot; type=&quot;xsd:string&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;xsd:element name=&quot;author&quot; type=&quot;xsd:string&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;xsd:attribute name=&quot;isbn&quot; type=&quot;xsd:integer&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xsd:complexType&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;xsd:simpleType name=&quot;isbnType&quot; type = &quot;xsd:integer&quot;/&gt;</td>
</tr>
</tbody>
</table>

Figure 21 DTD versus XSD

For the validation of the XML documents written in this project XSD was preferred over DTD for the following reasons:

- DTD uses non-XML syntax
- XSD datatype systems is far more extensive than DTD
- DTD does not support namespaces
- PSVI produced by DTD is weak compared to XSD

An example XML Schema can be found in Appendix D.

### 5.2 XML Schema Parser

The XML Schema forms the basis of all the editing and authoring features in the XML editor. The user is only permitted to perform actions and operations that comply with the chosen Schema. A pre-requisite for such an XML editor is a dynamic menu that can provide only legible additions and alterations to the XML file, namely a dynamic menu based upon the XSD. Extracting all the necessary
information from the XSD can be achieved through parsing the XSD into a PSVI representation in memory by an XSD parser.

The XSD parser implemented in this project is a DOM parser that parses the XSD into a DOM tree with additional information about the data types, the attributes, the minimum and maximum occurrences and the semantics of each node of the XSD, namely a PSVI. Once an XSD is loaded into the application the XSD parser will instantly create the PSVI. The advantages of this approach are that XSD parsing is fast for reasonably long XSD’s and happens only once. After the PSVI data structure representation has been created, it is presented to the application that will then provide the user with all the tools needed to create a well-formed and valid XML document. The well-formedness and the validity of the document are guaranteed by the PSVI, since only XML instances of the source schema can be created by the editor (Matsa et al., 2007).

A full XSD parser can be very complicated and exceeds the time limitations and purpose of this project. Instead of implementing an unfinished and error prone full XSD parser, the application is based on a schema specific XSD parser. By imposing constraints on the source schemas, the resulting XSD parser is more reliable and easier for the user to comprehend. The basic constraint on the source schemas is that the exclusion of element reference and element type attribute.

XSD allows elements to be declared as references to other elements or custom types through the `xsd:type` and `xsd:ref` attributes of an element. Both these features are excluded from the XSD parser of the XML editor. **However, it should not be assumed that the XSD parser is less powerful because of these exclusions.** A schema with equivalent functionality can be built even without element and type references. The only difference is that schemas that do not exploit these XSD features usually tend to contain duplicate code. The following example in figure 22 shows two equivalent schemas, the first one includes `xsd:type` and `xsd:ref` attributes, while and the second does not.
5.2.1 XSD features included in the schema specific parser

The XSD parser implemented is able to parse the following XSD features:

- **xs:schema** that is always the root element of an XML schema
- **xs:element** that declares XML elements. These elements can either be simple or complex types. Simple types are elements whose value can only be one of the built-in xs data types (xs:string, xs:integer, xs:date, etc.) On the other hand complex types are elements that include other elements and/or attributes. While simple types can be easily parsed to an element of the appropriate data type in the PSVI, complex types’ subtree needs to be further parsed. Both xs:simpleType and xs:complexType are supported.
- Indicators: **xs:all**, **xs:sequence**, **xs:choice**, declaring a sequence without specific order, an ordered sequence and an alternative expression of elements respectively. **MaxOccurs** and **minOccurs** define the minimum and maximum times an element or an attribute can be occur.

---

Figure 22 XSD schemas with and without element and type reference. Source: (Walmsley, 2001)
• The type attribute that declares the data type of the value of an element or attribute. The built in types of xs:string, xs:integer, xs:data, xs:date, xs:Boolean are all supported. The type attribute is important, as it is the means to check whether the input of the user conforms to the schema and can be used to prevent errors in the input of a value.

These features are the most common features used in an XML schema and form a solid basis for creating meaningful XML documents.

5.2.2 Implementation of the XML Schema Parser

An XML document can be seen as a depth-first serialization of a tree (Lu et al., 2006). The XML Schema is an XML document with an xs:schema root element. All the elements contained in the schema can be seen as trees of other indicators, elements and attributes and are subtrees of the root element. This view of a schema dictates the implementation of a depth first parser.

The XSD parser parses elements in a depth-first manner and works recursively. The parser contains parsing methods for every XSD feature (i.e. xs:element, xs:complexType). As the parser meets and XSD feature (that is a subtree but can be seen as a tree itself) it calls the appropriate parsing method to parse the subtree. Once the parser reaches the leaf of the initial tree, parsing is complete.

The output of the parser is the XSD PSVI, namely the Infoset of the XML schema that will be used by the application for the creation of a validated ML document. Figure 23 shows the PSVI produced by the parser for a simple schema. It can be easily noted how the XML tree-like structure of the schema can be parsed into a tree.

As the parser starts with the root element of the document, every time it finds an element it calls the element parsing method and passes the element’s sub tree to it. Working recursively in a depth-first manner, the parser continues with the parsing of the elements until it reaches a leaf. All the additional information
extracted in the process of parsing is stored in the PSVI to be used by the application during editing and authoring of the XML document.

### 5.3 XML Parser

The XML document edited in the editor is represented in memory by a *DOM tree* as it has already been explained. The XML parser is the program component that is responsible for transforming the structure commands of the user to actions on the DOM tree.

The application presents to the user a list of actions that is generated dynamically based on the source schema and the “active element” of the XML document. These dynamically created menus produce actions needed to be performed on the DOM tree that contain both textual input from the user and syntactic input that derives from the graphical user interface. *In that sense the parser is not a typical textual parser that needs to parse strings of characters to a data structure but a method that can transform a combination of text input and user actions to an updated DOM tree.*
5.4 XML Serializer

The XML serializer of the program is equally important to the XSD parser. Once the schema PSVI has been created, the application presents to the user all the available authoring and editing options. The XML document created by the user is represented in memory as a DOM tree and the user needs to interact with this tree while authoring the XML document, therefore it has to be visible to the user in the editor at all times. The visualization of the DOM tree creates the need for serialization of the DOM tree to XML.

The serializer is based on the same principles as the parser. The XML document can be seen as a tree structure and all that is needed to transform the DOM tree to XML is to define a simple transformation for each node of the DOM tree. For the needs of this project the simplest transformation was chosen. Every element node in the DOM tree is printed in the editor in pure XML syntax, which means that every element is declared by its start and end tag that contain all its children and the elements are separated by line breaks.

This serialization was chosen because it offers flexibility and simplicity. Simplicity will help novice users comprehend XML syntax faster, while still maintaining flexibility by making a more complex transformation available through an EXtensible Stylesheet Language Transformation (XSLT) that can transform well-formed XML documents to a more convenient or stylish format.
5.5 How the XML Editor works

The previous sections of this report have described the main components of the application, namely the schema parser, the PSVI tree, the parser, the DOM tree and the serializer. Figure 24 illustrates the interactions between these main components that form the backbone of the application.

The only component which the user can view and control is the text presented in the editor. However, the way the user can interact with it and the set of “legal” and thus available interactions with it is controlled by the DOM tree of the target XML document and the PSVI tree of the source schema.

The PSVI tree is generated as soon as the source schema is loaded into the application. Similarly, every time a new document is created, its DOM tree is initialized. All the user’s controls are based on the textual representation of the DOM tree produced by the serializer. Whenever the user wishes to perform an action on the document or an element of the document the “active” element will be identified and after the PSVI tree is consulted only the syntactically correct options will be presented to the user. After the action has been performed, the parser will update the DOM tree and the serializer will present the updated XML document to the user.
The cooperation of the parser, the serializer and the schema parser guarantees that documents created by the user validate against the source schema and that even novice users will be accurately and intuitively guided to create valid XML documents.

5.6 XML Editor’s main classes

The XML Editor’s main classes are directly inherited from the prototype BNF Editor (c.f. actual re-use of an earlier prototype). All of the data structures involved in the implementation are tree data structures: both the PSVI tree and the DOM tree can be implemented as specializations of the generics class Tree developed for the prototype. The Regions class also derives from the prototype, while the worker class that implements, the serializer and the XML parser is based on the worker class in the prototype with some minor modifications. The only major addition to the prototype was the validation of the produced file to a schema, which is also built upon the Tree generic base class.

The reuse of the main classes of the application was of crucial significance to the success of this project. The implementation of two different applications from scratch would exceed the time and complexity scope of an MSc project. However, developing the final application based on the classes already implemented from the prototype minimized the amount of source code needed to be written and guaranteed a “known to be working” environment with minimal testing needs, thus making the project feasible.

The classes that implement the DOM and PSVI nodes can be seen in the figure 25. These two classes provide a data structure that contains all the special properties of each tree node. The properties related to the structure of the tree are implemented by the Tree class that has been already presented.
A full UML diagram of all XML Editor’s classes can be found in Appendix C.
6. Evaluation, conclusions and future work

This chapter provides an insight into the overall success of the project. The prototype and the final application are evaluated by the author and the user, in an attempt to evaluate syntax directed environments and their implementation process. One of the most important part of this project are the conclusions drawn and the knowledge gained while working on the project and these will be presented and compared to results available in literature. Finally, this chapter explains future work that could take this project one step further and how the structure editing environment that was developed can be improved.

6.1 Evaluation of the editors

6.1.1 Software Evaluation

The success (or otherwise) of software projects is often measured in terms of whether the project has exceeded its budget or estimated delivery time. In the software industry budget and time are considered as the main metrics for success in software development. However, an application delivered on time and within the initially allocated budget does not guarantee “success”. The question of the quality\(^1\) of the software is raised and can be seen from various viewpoints.

\(^1\) The term “quality” is problematic in the context of software, i.e. software is not “engineered" in the same sense that conventional artefacts are engineered. One basis for this premise is that physical artefacts (in conventional engineering disciplines) are constructed from physical materials via the application of physical processes, whereas, software is not. Never-the-less, software can be "engineered" in the more conventional sense of the term if it is systematically constructed from a formal description of its properties. The use of simple grammars (to specify languages as sets of strings) provides one example of a simple formal specification that is suitable for subsequently systematically engineering software that satisfies the grammar specification, e.g. when constructing a parser (or unparser) as in the project described in this report. Similarly, but more expressively, modern software design notations, e.g. UML, have now progressed to being "formal" in the sense that they have been themselves formalised (albeit in a less than satisfactory manner, in the view of some observers...).
Rubey and Hartwick (1968) proposed a framework for *quantitative evaluation* of software that relies on the measurement of a proposed set of *attributes* and *metrics* that are considered to represent the “quality” of a software product. The evaluation according to this quantitative approach is based on seven axes:

- The logical correctness of the program
- Absent of interference between program entities
- Optimized performance in terms of computational power and memory usage
- Intelligibility of the program, referring to the ability to read and comprehend the program’s source code with the aid of comments, consistent program design, meaningful method names etc.
- Ease of modification of the program
- Ease of use and learning of the program

Boehm *et. al.*(1976) expanded the above model and proposed their own model that is focusing on two major factors: *as-is-utility* and *maintainability*. As-is-utility measures the portability, reliability, efficiency and accessibility of the program, while maintainability refers to testability, understandability and modifiability.

For the purposes of this project some characteristics that need to be present in modern software components were considered out of the scope of the project. The applications developed were built for a “Windows-only” environment and portability (to other environments) was never intended. The questionnaires written for the evaluation of the applications developed for the project will follow the model of Boehm *et. al.*(1976) adjusted to the context of syntax directed editors. The questionnaires collected can be found in Appendix A.
6.1.2 Author Evaluation

The main objective of this project was to build an XML structure editor that would be able to provide a fast and error-safe way to write a valid (or “legal”) XML document. As is shown in the XML editor User Guide, this objective was satisfied.

The overall success of a structure editor relies on a wide range of characteristics: these include (but are not restricted to) editing functionality, responsiveness, and tests of input and textual representation.

The XML editor is functional and ensures valid input when data is entered by the user. Moreover, the textual representation with the help of the tree view of the XML document is concise and at the same time provides visual aids, such as text highlighting and elision of parts of the document that the user is not interested in.

Continuing with the evaluation, the XML editor has been tested for completeness, reliability and usability. It was found to perform extremely well for medium sized schemas and XML files. However, as the size of the XML files increases (e.g. over approximately 500 lines), especially for XML schemas that define elements with extensive depth, the responsiveness of the application slows down. This is due to the fact that the algorithms designed for the main components of the application are recursive and based on the DOM model, whose performance is known to decrease significantly for larger XML files. Optimization to avoid exploring subtrees of the data structures that are not needed in specific actions would significantly improve performance. Additionally, the built-in .NET richTextBox class was found expensive in terms of computing “power” and could be replaced by a custom textbox class. Optimization and a custom textbox class would significantly improve performance, but were not completed due to insufficient time resources.
6.1.3 User Evaluation

After the completion of the application, the XML editor was given to ten users for evaluation along with a questionnaire to complete. Due to time limitations of the evaluation period, only seven questionnaires were returned on time. These questionnaires can be found in Appendix A.

As expected, the user interface was found to be “user friendly” and the application did not pose serious problems to novice users. Most of the users found the application easy to use and would use it for XML authoring even though they would normally prefer an alternative commercial application for that job. However, this was expected since XML editor is “competing” with existing XML editors that have been under development for years.

The most important problems noted by users was the lack of XML documentation in the application, inadequate feedback for the operations the editor is computing in the background and reduced responsiveness for large documents. All these problems reported were expected and agree with the author’s evaluation of the program. They are well known weaknesses of the program due to time limitations of the project. The future work section explains how these problems would be solved, if extra time was spent on the project.

![User Friendly Interface](image)

**Figure 26 User Interface Users Evaluation**
Figures 26 and 27 show the results of the users’ evaluation for the questions relevant to the “friendliness” of the user interface and the “usability” when writing an XML document. The majority of the users found the XML editor easy to use and the user interface friendly.

6.2 Conclusions

6.2.1 Usability Issues

The target users of syntax directed environments usually includes novice users and students that require help for the creation of formal documents for programming languages. The most significant difference between a syntax directed and a conventional editing environment is the former’s internal structured representation of the document that defines a “legal” (i.e. semantically meaningful given the currently selected item at the user interface) set of actions that the user is confined to. Because of this constraint the usability of structure editors is one of the most important issues in their design and implementation. The success of a structure editor lies in its ability to support a high level of interaction between the user and the editor.

Khwaja and Urban (1993) discussed the main usability issues present in syntax directed environments. Their study highlights the importance of actual user feedback on the interface of the structure editors. Users tend to think in different
ways and structure editors should provide alternative ways of performing actions when writing a document. Linearity and an enforced construction discipline can pose serious usability problems for large groups of users. A method to overcome these hindrances is to design the structure editor for advanced users also. A structure editor’s success can be measured by the advanced user acceptability. While novice users can be taught to think and write in a way compatible with the editing environment, advanced users have already set up their own habits for authoring a formal document and would be more than willing to dismiss an editing environment that does not provide flexibility. In addition to flexibility advanced users require features that can simplify and shorten the creation of a document that replace a conventional editor.

The XML structure editor developed in this project has been successful in this area. Writing an XML document that has to be validated against a schema requires tedious work. The user of a conventional editor would have to always consult the schema about the legible elements and content and at the same time avoid typographic and syntax errors such as missing end tags and correct spelling of elements’ and attributes’ names. In contrast to this time and attention consuming approach the XML editor can offer syntactic correctness by providing valid structure and correct content for the document without undermining the user’s own preferences for writing an XML document. The XML editor does not enforce a linear approach to writing the XML document. The user can choose which elements to insert in an order that would match the user’s habit, as long as certain rules are followed (for instance, children cannot be inserted before their parents have been inserted). An experienced user should be able to write an XML document within minutes, a task that otherwise would require much more time. On the other hand, novice user should find themselves getting faster as they learn more about XML authoring. This makes the XML editor accessible to both novice and advanced users.

Another important issue related to the usability of a structure editor is whether the program enforces a top-down only approach for document creation. Ideally, according to Neal and Szwilus (1990), a structure editor should support a
bottom-up approach “as well” to provide more freedom to the user and the applications implemented for the project follow this suggestion.

### 6.2.2 User View

Every structure editor is based on the data structure representation of the output document. In this sense, the text present to the user as the document is merely a textual representation (or unparsing) of the in-memory data structure. How this data structure is presented to the user affects the accessibility and usability of the environment. The most common approaches are textual and tree form representations. Welsh et al. (1991) argued that neither a tree form or a textual representation on their own can provide a “rich” user experience. A combination of these two representations can solve the problem. However, it depends on the format of the output document how they should be combined.

For both the applications developed for this project the textual and tree form representations of the document structures were used separately. While the area of the editor that the user is working on uses only a textual representation, the user can has access to a tree form overview of the document’s structure in a separate area. Ideally, editing options would also be for the tree form overview; however, time restraints did not allow this feature to be implemented. As the working document increases in size with the insertion of new elements, the tree view becomes invaluable. The user has the option to make parts of the document “collapse” or “become visible” in the tree view, thus focusing on the part of the document that the user is editing.

### 6.2.3 Conciseness and Minimalist Design

A structure editor’s nature inevitably imposes constraints on the user and the user’s actions. Structure editors are usually based on menus and commands rather than plain text input. As a result, additional menus and windows are needed for data input compared to a conventional editor. Conciseness and minimalist design are necessary as an antidote to extensive menus and pop-up
windows that can slow down the responsiveness of the program and confuse
the user (Usability inspection et al., 1994).

The BNF editor prototype did not fully succeed in providing a concise and
minimalist user interface. The insertions of new rules, as well as expanding
regular expressions, all required the user to input data through new windows,
different for each operation. While the overall interaction with the environment
was “smooth”, the user would usually prefer a more concise and faster way to
build a BNF grammar and would need time to learn the outline of the pop-up
windows.

The experience gained from the prototype was exploited for the design of
the XML editor. All the actions can be instantly performed from the main
window with the exception of data input for the values of elements that
will bring up a new window. However, this window was designed to be
simple and easily learnt by the user, consisting only of a textbox and a
button. The concise design is one of the most important features of the
application, making the creation of an XML document simple and fast.

6.3 Future Work

Although the XML editor that was developed has been successful in all the
significant areas that are required for a viable structure editing environment, the
application can be further improved to provide a “richer” user experience and to
support a complete introduction to XML technology for novice users.

Firstly, the editor could be improved by adding features to the program that
would help novice users to understand XML and had to be kept out of the scope
of this project due to time limitations. XML documentation could be added for
the users to refer to whenever they need a quick look at the more “theoretical”
aspects of XML. Moreover, a “console box” that would explain why the structure
editor will or will not allow an operation could be added.

Secondly, the main improvement would be the “hybridization” of the structure
editor. This would mean a structure editor that would allow the user to alter the
data of the document by interacting directly with the text presented by the
application. The user would be able to delete an element, by actually deleting it, or add the value of an element by typing the value. In order to preserve the strengths of a structure, a hybrid structure editor should be able to parse and unpars the text. The parser is the software component that is able to parse the text entered by the user into the program's data structure. The unpars er executes the same operation in the opposite direction. The unpars er creates a textual representation of the data structure in a manner such that, if the text was parsed again, it would output the same data structure. A parser and an unpars er operating in cooperation can ensure syntactical correctness and that illegible input or text from the user is discarded.

Finally, the XML editor could be easily modified to support XSLT. XSLT, as has already been explained, is able to transform an XML document to more stylish or comprehensible format. XML, XML schemas and XSLT’s are the most significant concepts behind the use and application of XML in software development. The ability to “toggle” between the source XML document and the XSLT formatted output would help novice users gain more quickly knowledge of the usefulness and importance of XML. At the same it would help more experienced users to have a complete overview of the document being authored.

The following figure discusses the architecture of the XML editor modified to support the proposed improvements. The application already has the underlying “mechanics” to support such a modification. The implementation of the XSLT serializer and of the unpars er could be added to the application without conflicting with the features and methods already implemented.
Figure 28 A hybrid XML structure editor
User Guide

BNF Editor User Guide

Opening the application will bring up the application’s main window. Before the user can start writing the BNF grammar, the user must create a new file from the file menu as shown in figure 29.

![Figure 29 File menu](image)

All the editing options are available through a context menu. When starting a new grammar the only available option is to add a rule. Clicking on the add rule options will invoke a new popup window asking for the basic information for the rule: the left hand symbol, the right hand terminal symbol and the type of the regular expression of the right hand side of the rule. The cancel button will close the window and return to the editor, while the “add rule” button, adds the rule to the grammar and then returns to the editor.
Adding a new rule will insert the rule to the BNF grammar; however the regular expression of the right hand side will remain a placeholder – that can be identified by the symbol “$” instead of “<” and “>” at the beginning and end of the regular expression – until it is expanded by the user to a proper regular expression. A sequence placeholder is displayed in figure 31.

Expanding the sequence in figure 31 will bring up the expanding a sequence window as it can be seen in figure 32.

All placeholders have to be expanded in a valid BNF grammar. The program can distinguish between an expanded regular expression and a placeholder and will provide the option to expand the placeholder depending on its type. Each type of regular expression placeholder brings up a different new window for expanding the expression.

Expanding the sequence in figure 31 will bring up the expanding a sequence window as it can be seen in figure 32.
Figure 32 Expanding a Sequence Expression

The window for expanding an alternative expansion is similar, while expanding a symbol will bring a much simpler window as can be seen in the following figures.

Figure 33 Expanding an alternative expression

Figure 34 Expanding a symbol
Adding new rules and expanding regular expressions is how the user can write the BNF grammar. Which option is available to the user is handled by the application and presented to the user via the context menu. Right mouse button clicking on an unexpanded sequence expression will bring up a sequence expanding window while, right mouse button clicking on a symbol expression will bring up the symbol menu.

The dynamic context menu is showcase in the next figures. The application can identify the element that the user is pointing at and dynamically create the appropriate context menu. Clicking on the left hand side of a rule gives options only for the left side of the rule. The dynamic context menu can be seen in figure 35:

![Figure 35 BNF Editor's dynamic context menu](image)

For every rule or expression in the grammar the user a delete option is also available to the user. Deleting a rule or a regular expression will affect the whole BNF grammar as the structure editor will calculate all the necessary modifications in the grammar invoked by the deletion. For instance, deleting a symbol will delete the symbol from all the rules of the grammar and the rule that has this symbol in the left hand side.
This chapter will present the main features of the program and show how the user can interact with the application in order to create an XML document. This section is based on testing using the XML schema that can be found in Appendix D as the source XML schema.

First of all, before starting authoring the XML document, the user has to create a new file through the file menu:

![Figure 36 File Menu in the XML editor](image)

Then the user has to choose the source XML schema for the XML document. Clicking on the “Browse for Schema” menu button, the user can browse for the XML schema through the file browser menu of Windows. After the file is selected the file address will appear in the address bar as illustrated in the following screenshot:

![Figure 37 Loading an XML Schema](image)
After the user has selected the XML Schema file, clicking on the Load Schema button next to the address bar will load the schema into the application. This action will create the PSVI in memory that will be the basis for all the actions and options available to the user. Loading the XML schema will also result to the appearance of the XML document’s root element.

Now the user can continue with the authoring of the XML document. All the legible actions available for the selected element are presented to the user via a dynamically created context menu. The next screenshot displays the context menu for the root element with an example schema loaded:

![Figure 38 XML Editor Context menu](image)

The context menu in this example will provide the user the ability to add a sequence of children. When an element is a sequence of elements, like in this example, the user will be able to enter the sequence of all the children with one single click. In the same way the user can insert a sibling. Inserting a sibling, child or attribute will only be available to the user if all the conditions set by the schema are fulfilled. In the above example the root element cannot have a sibling, since that would invalidate the XML document, so the “Insert Sibling” option in the menu remains empty. The same principle applies when the number of children or attributes has exceeded the maximum occurrences defined in the XML schema. Inserting the sequence in the above example will insert the children element in the root element and update the text presented to the user. At this point, it should be noted that the active element is always highlighted in blue, as shown in figures 38 and 39.
The tree view on the left panel of the XML editor presents a quick overview of the XML document’s structure. After the elements are inserted into the XML document the user has the ability to enter or alter the value of the element. Selecting the “Set Value” option from the context menu will bring up a new window where the user can input the value. The value must conform to the data type defined in the XML schema in order to be accepted. This simple window is shown in figure 40:

Using the context menu and inserting children, sibling, attributes and values of elements the user can create well-formed and valid XML documents. The
following screenshot shows an XML document written in the XML Editor that is valid against the example schema used.

![XML Document]

Figure 41 Valid XML Document created in the editor
APPENDIX A – Users Evaluation

Name/Age/Occupation: Kostas/20/Student

Q1. What is your level of experience with XML?
   A. Novice (never heard of XML before)
   B. Intermediate (basic knowledge about XML)
   C. Advanced (Used XML for applications)

Q2. How easy was to write an XML document using the XML Editor?
   A. Very easy (encountered no or very few problems)
   B. Intermediate (encountered problems, but none of them was serious)
   C. Not easy at all (did not manage to create an XML document)

Q3. Which description fits best the graphical user interface?
   A. User-friendly (smooth learning curve)
   B. Standard (steep learning curve)
   C. Unhelpful (actions and operations were inaccessible)

Q4. How you rate the overall performance of the XML Editor?
   A. Very good (I would actually use it for XML editing)
   B. Mediocre (Working but I would prefer an alternative application)
   C. Poor (I would avoid using it at all)

Q5. What improvements would you suggest?
   A method to provide feedback from the application to the user.

Q6. What was the biggest problem you experienced?
Name/Age/Occupation: Andreas/26/Programmer

Q1. What is your level of experience with XML?
A. Novice (never heard of XML before)
B. Intermediate (basic knowledge about XML)
C. Advanced (Used XML for applications)

Q2. How easy was to write an XML document using the XML Editor?
A. Very easy (encountered no or very few problems)
B. Intermediate (encountered problems, but none of them was serious)
C. Not easy at all (did not manage to create an XML document)

Q3. Which description fits best the graphical user interface?
A. User-friendly (smooth learning curve)
B. Standard (steep learning curve)
C. Unhelpful (actions and operations were inaccessible)

Q4. How you rate the overall performance of the XML Editor?
A. Very good (I would actually use it for XML editing)
B. Mediocre (Working but I would prefer an alternative application)
C. Poor (I would avoid using it at all)

Q5. What improvements would you suggest?

Q6. What was the biggest problem you experienced?
The program was not fast enough for large documents.
Name/Age/Occupation: Tim/22/2nd Year Eng. Student

Q1. What is your level of experience with XML?
   A. Novice (never heard of XML before)
   B. Intermediate (basic knowledge about XML)
   C. Advanced (Used XML for applications)

Q2. How easy was to write an XML document using the XML Editor?
   A. Very easy (encountered no or very few problems)
   B. Intermediate (encountered problems, but none of them was serious)
   C. Not easy at all (did not manage to create an XML document)

Q3. Which description fits best the graphical user interface?
   A. User-friendly (smooth learning curve)
   B. Standard (steep learning curve)
   C. Unhelpful (actions and operations were inaccessible)

Q4. How you rate the overall performance of the XML Editor?
   A. Very good (I would actually use it for XML editing)
   B. Mediocre (Working but I would prefer an alternative application)
   C. Poor (I would avoid using it at all)

Q5. What improvements would you suggest?
   I would suggest including XML documentation in the application and including a view of the XML schema.

Q6. What was the biggest problem you experienced?
   Having to use another application for viewing the XML schema.
Name/Age/Occupation: Vassiliki/23/1st Year Computer Science Student

Q1. What is your level of experience with XML?
   A. Novice (never heard of XML before)
   B. Intermediate (basic knowledge about XML)
   C. Advanced (Used XML for applications)

Q2. How easy was to write an XML document using the XML Editor?
   A. Very easy (encountered no or very few problems)
   B. Intermediate (encountered problems, but none of them was serious)
   C. Not easy at all (did not manage to create an XML document)

Q3. Which description fits best the graphical user interface?
   A. User-friendly (smooth learning curve)
   B. Standard (steep learning curve)
   C. Unhelpful (actions and operations were inaccessible)

Q4. How you rate the overall performance of the XML Editor?
   A. Very good (I would actually use it for XML editing)
   B. Mediocre (Working but I would prefer an alternative application)
   C. Poor (I would avoid using it at all)

Q5. What improvements would you suggest?

Q6. What was the biggest problem you experienced?
   Learning XML.
Name/Age/Occupation: George/29/Mechanical Engineer

Q1. What is your level of experience with XML?
A. Novice (never heard of XML before)
B. Intermediate (basic knowledge about XML)
C. Advanced (Used XML for applications)

Q2. How easy was to write an XML document using the XML Editor?
A. Very easy (encountered no or very few problems)
B. Intermediate (encountered problems, but none of them was serious)
C. Not easy at all (did not manage to create an XML document)

Q3. Which description fits best the graphical user interface?
A. User-friendly (smooth learning curve)
B. Standard (steep learning curve)
C. Unhelpful (actions and operations were inaccessible)

Q4. How you rate the overall performance of the XML Editor?
A. Very good (I would actually use it for XML editing)
B. Mediocre (Working but I would prefer an alternative application)
C. Poor (I would avoid using it at all)

Q5. What improvements would you suggest?
A tool that would provide tips for what to do next.

Q6. What was the biggest problem you experienced?
I would prefer to be able to type text myself in some occasions, for instance to correct the value of an element.
Q1. What is your level of experience with XML?
A. Novice (never heard of XML before)
B. Intermediate (basic knowledge about XML)
C. Advanced (Used XML for applications)

Q2. How easy was to write an XML document using the XML Editor?
A. Very easy (encountered no or very few problems)
B. Intermediate (encountered problems, but none of them was serious)
C. Not easy at all (did not manage to create an XML document)

Q3. Which description fits best the graphical user interface?
A. User-friendly (smooth learning curve)
B. Standard (steep learning curve)
C. Unhelpful (actions and operations were inaccessible)

Q4. How you rate the overall performance of the XML Editor?
A. Very good (I would actually use it for XML editing)
B. Mediocre (Working but I would prefer an alternative application)
C. Poor (I would avoid using it at all)

Q5. What improvements would you suggest?
A consol-like textbox to provide comments for the operations taking place.

Q6. What was the biggest problem you experienced?
Name/Age/Occupation: Anthony/25/Marketing Student

Q1. What is your level of experience with XML?
A. Novice (never heard of XML before)
B. Intermediate (basic knowledge about XML)
C. Advanced (Used XML for applications)

Q2. How easy was to write an XML document using the XML Editor?
A. Very easy (encountered no or very few problems)
B. Intermediate (encountered problems, but none of them was serious)
C. Not easy at all (did not manage to create an XML document)

Q3. Which description fits best the graphical user interface?
A. User-friendly (smooth learning curve)
B. Standard (steep learning curve)
C. Unhelpful (actions and operations were inaccessible)

Q4. How you rate the overall performance of the XML Editor?
A. Very good (I would actually use it for XML editing)
B. Mediocre (Working but I would prefer an alternative application)
C. Poor (I would avoid using it at all)

Q5. What improvements would you suggest?
Help documentation for users not acquainted with XML would be useful.

Q6. What was the biggest problem you experienced?
Had problems understanding the structure of an XML schema.
APPENDIX B – BNF Editor’s Class Diagram

This appendix displays the class UML diagram of the BNF editor. The main classes are explained in the main body of this report. The following figure displays all the classes of the BNF editor.

The first part of the class diagram is the regular expression class. Regular expressions in BNF can be symbols, sequence expression or alternative expressions. Moreover, each of these classes is regular expression and should be able to replace one of the other classes in the BNF grammar tree data structure. For this reasons, regular expressions have been implemented as an abstract class with symbols, sequence and alternatives as classes that inherit from the base regular expression class.

IsPrintable and IsRegion are interfaces that allow these classes to be used in a generics class, such as the Tree class and provide an interface for accessing these classes’ methods.

All the dialog boxes that interact with the user are implemented as separate windows that inherit form the built-in Window class and exploit the built-in user controls available in .NET Visual Studio.

The class diagram follows in the next page.
Figure 42 BNF editor’s UML Class Diagram
APPENDIX C – XML Editor’s Class Diagram

This appendix displays the class UML diagram of the XML editor. The main classes are explained in the main body of this report. The following figure displays all the classes of the XML editor.

IsPrintable and IsRegion and IsXML are interfaces that allow classes to be used in a generics class, such as the Tree class and provide an interface for accessing these classes’ methods.

All the dialog boxes that interact with the user are implemented as separate windows that inherit form the built-in Window class and exploit the built-in user controls available in .NET Visual Studio.

The class diagram follows in the next page.
Figure 43 XML classes UML diagram
APPENDIX D – XML Schema Example

For the needs of this report the following XML Schema was used as an example schema to describe the basic characteristics of the XML Schema language, how it is used in the XML Editor and as the source XML Schema for the User Guide. The source of the XML schema is W3C’s website.

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.shiporders.com">

  <xs:element name="shiporder">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="orderperson" type="xs:string"/>
        <xs:element name="shipto">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="address" type="xs:string"/>
              <xs:element name="city" type="xs:string"/>
              <xs:element name="country" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="item" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="note" type="xs:string" minOccurs="0"/>
              <xs:element name="quantity" type="xs:positiveInteger"/>
              <xs:element name="price" type="xs:decimal"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="orderid" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
References


international conference on World Wide Web. Banff, Alberta, Canada, ACM.


