Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR'S THESIS

Roman Betík

XML Data Visualization

Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková, Ph.D.

Study program: Programming

2010

I would like to thank to my supervisor RNDr. Irena Mlýnková, Ph.D. for leading my work. I would also like to thank to my parents and my sister for supporting me during my studies.

Contents

Title: XML Data Visualization
Author: Roman Betík
Department: Department of Software Engineering
Supervisor: RNDr. Irena Mlýnková, Ph.D.
Supervisor's e-mail address: irena.mlynkova@mff.cuni.cz

Abstract: The aim of the thesis is to implement a graphical editor designated to create and edit XML documents and documents in XML Schema language. The main part of the work is design and implementation of the program. Created application allows to edit (or create) XML documents using tools to modify particular graphical elements which represent elementary parts of the document (elements, text nodes etc.). It also includes support for easier creation of schemas in XML Schema language. It is possible to validate edited XML documents against DTD or XSD schemas. The resulting work can be saved back to its text representation or the visualization can be saved as an image. The work includes description of application's architecture and description of the most important parts of the source code.

Keywords: XML, XSD, visualization, planar graph, data, editor, XPath

Názov práce: Vizualizácia XML dát
Autor: Roman Betík
Katedra: Katedra softwarového inžinierstva
Vedúci práce: RNDr. Irena Mlýnková, Ph.D.
E-mail vedúceho práce: irena.mlynkova@mff.cuni.cz

Abstrakt: Cieľom práce je implementácia grafického editoru určeného na vytváranie a editáciu XML dokumentov a dokumentov v jazyku XML Schema. Hlavnou časťou je návrh a implementácia programu. Samotný program umožňuje editovať (prípadne vytvárať) XML dokumenty pomocou úpravy jednotlivých grafických prvkov reprezentujúcich elementárne časti dokumentu (elementy, textové uzly atď.). Takisto zahŕňa podporu pre jednoduchšie vytváranie schém v jazyku XML Schema. Editované XML dokumenty je možné validovať voči DTD alebo XSD schémam. Výslednú prácu je možné uložiť spätne do textovej podoby, prípadne si vizualizáciu uložiť ako obrázok. Práca obsahuje popis architektúry programu a najdôležitejších častí zdrojového kódu.

Kľúčové slová: XML, XSD, vizualizácia, rovinný graf, dáta, editor, XPath

Chapter 1

Introduction

XML language [2] is a commonly used format when sharing data. It is text-based so to create or edit XML documents an editor is needed. Most of the available editors for XML documents are text-based editors. The logical structure of XML documents enables us to go beyond the text representation and create applications which render XML documents in a much user-friendlier way.

The aim of this thesis is to create such an application. This application will be able to load existing XML documents or create new ones and display them in the form of planar graphs [3]. A user will be able to modify this layout by adding or removing elements of an XML document or by changing the visual parameters (like colour or shape) of particular entities. User will also be able to select elements using the language intented for querying XML data. The program will include classical functions every editor should have (detailed list of these feature is included in Chapter 3).

1.1 Structure of the Text

The second chapter describes basic concepts of XML language and XML schemas. This chapter also includes description of basic methods how to work with XML from the view of a programmer. The third chapter describes the goals and requirements of created application. The fourth chapter includes user documentation. The fifth chapter describes the architecture of the application and all major tools and libraries used when building it. The following chapter describes the main algorithms used in this application. The sixth chapter analyzes existing solutions. The seventh chapter summarizes the work. The last chapter, Appendix A, lists the contents of the attached CD.

Chapter 2

XML Basics

2.1 A Brief Introduction to XML Format

XML (Extensible Markup Language) is a general-purpose specification how to create new markup languages. It is extensible because XML allows us to create new markup elements. This format is text based, simple and very flexible.

The main purpose of XML (Listing 1 contains sample XML document) is to make sharing and distributing information between various independent systems easier. It is text based, easily parsed format which is easy to read even by humans. Nowadays it is used not only for storing information but also for data serialization, as data transport format and even for databases.

More about XML can be found at the website of the World Wide Web Consortium [1].

```xml
<?xml version="1.0" encoding="utf-8"?>
<Program>
  <Assemblies>
    <Assembly name="XmlStudio.Main">
      <File Source="MainWindow.xaml" />
      <File Source="MainWindow.xaml.cs" />
    </Assembly>
    <Assembly name="XmlStudio.Model">
      <File Source="Models/XmlModel.cs" />
      <File Source="Models/ModelFactory.cs" />
    </Assembly>
  </Assemblies>
  <Resources>
    <Image Source="Images/Icon.ico" />
  </Resources>
  <RequiredNetVersion version="4.0" />
  <Notes>
    <Note>XmlStudio's source code files.</Note>
  </Notes>
  <!-- This is a comment. -->
</Program>
```

Listing 1: Sample XML document.

There is a prolog at the very beginning of this sample XML document. Every XML file stars with a prolog. It has two parts: the XML declaration, which is shown in the sample (the minimal requirement is to specify the version) and Document Type Declaration (DTD). DTD is an optional part. More about DTD in the following sections. After prolog there is a root element with nested child elements.

Elements consist of so called tags. An element begins with a start-tag, e.g. `<Program>` and ends with an end-tag, e.g. `</Program>`. There can be text between these tags (as in the case of element `<Note>`) which is called the value of the element or another nested tags. When there is no text or no tags between these two tags, you can skip the end-tag and express the whole tag as an empty element, e.g. `<RequiredNetVersion />`. Element itself is identified by its name, in our sample document there is an element with the name `Program`.

Elements can have attributes. They appear after the name of the element and have the form of `attributeName="value"`. In our sample there is an attribute `name` on element `Assembly` with the value `XmlStudio.Main`. XML documents may also contain some comments which are inserted between `<!--` and `-->`.

This sample XML document is well-formed. The XML specification defines exactly what well-formedness is and it can be found here [2]. The key points are that the begin, end, and empty-element tags are correctly nested, with none missing and none overlapping. Element tags are case sensitive, so the start and end-tags must match exactly. There is only one root element which contains all the other elements.

2.1.1 XML and Namespaces

XML language includes the concept of namespaces. When we create documents, we usually create some kind of vocabulary of elements and attributes. If we want to combine more of these created “vocabularies” together in one document, we might encounter some difficulties with naming. To distinguish these different vocabularies, XML brings namespaces. This is solved with a separate standard [4].

Consider the XML fragments in Listing 2 and Listing 3.

```xml
<Company>
  <p>Microsoft Corporation</p>
  <p>Redmond</p>
</Company>
```

Listing 2: XML fragment 1

```xml
<Company>
  <Name>Microsoft Corporation</Name>
  <City>Redmond</City>
</Company>
```

Listing 3: XML fragment 2

If we tried to use these two fragments together in one document, we would get a conflict, because there are elements with the same names but a different structure. The solution of this problem is to put them into separate naming containers, as you can see in Listing 4. The first part belongs to the namespace identified by the uniform resource identifier (URI) [5] `http://www.microsoft.com/`. Because using this long name would be too difficult, there is a shortcut. The shortcut is using the prefix for the namespace. So for the namespace `http://www.microsoft.com/` there is prefix `m` which is declared on the root element and then used before the name of the elements that belong to this namespace.

```xml
<Companies xmlns:m="http://www.microsoft.com/"
           xmlns:n="http://www.hotmail.com/">
  <m:Company>
    <m:p>Microsoft Corporation</m:p>
    <m:p>Redmond</m:p>
  </m:Company>
  <n:Company>
    <n:Name>Microsoft Corporation</n:Name>
    <n:City>Redmond</n:City>
  </n:Company>
</Companies>
```

Listing 4: Naming conflict solved

2.2 XML Schema Definition

The structure of an XML document can vary a lot and therefore there are many different ways how to express the same information. This might not be a problem for people but this could prevent correct communication between different computer systems. Therefore it is a good practice to create a set of rules for your document which defines what elements and attributes are allowed. This way both communicating sides know what to anticipate and can prepare their application logic accordingly.

There are several languages designated for XML schema definition, for instance DTD [6] (Document Type Definition, Listing 5 contains sample DTD), XML Schema [7] (Listing 6 contains sample XSD), Relax NG [10] or Schematron [11]. Any of these languages itself would need many pages to describe so in case the reader is interested in these languages we leave him to find detailed information about these topics himself.

```
<!DOCTYPE Program [
    <!ELEMENT Program (Assemblies, Resources, RequiredNetVersion, Notes?)>
    <!ELEMENT Assemblies (Assembly+)>
    <!ELEMENT Resources (Image*)>
    <!ELEMENT RequiredNetVersion EMPTY>
    <!ATTLIST RequiredNetVersion version CDATA #REQUIRED>
    <!ELEMENT Notes (Note*)>
    <!ELEMENT Note (#PCDATA)>
    <!ELEMENT Assembly (File+)>
    <!ATTLIST Assembly name CDATA #REQUIRED>
    <!ELEMENT File EMPTY>
    <!ATTLIST File Source CDATA #REQUIRED>
    <!ELEMENT Image EMPTY>
    <!ATTLIST Image Source CDATA #REQUIRED>
]>
```

Listing 5: Sample DTD

The sample DTD contains schema declaration for the document in Listing 1. This DTD means that the root element must be element `Program`. Then there are definitions for elements and their content. The construct `<!ELEMENT ... >` defines the content of the element which name is after the keyword `ELEMENT`. There are various options you can use when declaring what content can be inside the element. The next construct is `<!ATTLIST ... >` which defines what attributes are valid for the specified element. For the complete instructions about how to write a DTD see its specification.

The sample in Listing 6 contains XSD schema for our sample XML document.

Every XML schema is also an XML document which only contains specially defined set of elements. As you can see in the sample, the root element of XML schema must be element `schema`. The rest of the schema are special elements and attributes which define it.

The simplest elements of XML schema language are the simple types. Our sample does not contain any simple type but in short, when element or attribute is of a simple type, it means that it can have only text content. But this text content can be constrained in various ways. Simple types are either built-in data types [8] or user defined data types [9]. On the other hand, when an element has been assigned a complex type, this element can contain other elements and also attributes (if the type specifies them). Our sample contains some complex types. For example element `Notes` has been assigned a complex type which contains a sequence of elements `Note`.

To describe every aspect of XML schema, we would need much more space. Therefore we leave the details out and if the reader is interested, the complete reference can be found in XML Schema specification [7].

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Program">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Assemblies">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Assembly" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="File" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="Source" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Resources">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Image" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="Source" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="RequiredNetVersion">
          <xs:complexType>
            <xs:attribute name="version" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="Notes">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Note" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 6: Sample XSD schema

2.3 XML Processing in Programming Languages

2.3.1 Document Object Model

Document object model (DOM) [12] is a language-independent interface for object oriented programming languages (or script languages). It provides developers with methods to access and modify the structure of XML documents. XML processor (or parser) generates virtual tree-like model of the entire document which is then stored in memory. This implies limits on its usage. XML documents which object model would not fit into memory cannot be accessed and modified using DOM. Therefore this method is suitable for smaller documents (approximately 0B - 10MB, the upper limit is determined by the implementation of DOM and available system memory). This interface is available for many languages, such as C++, Java, JavaScript, CORBA, C# etc. Figure 2.1 shows an example of DOM hierarchy of our sample in Listing 1.

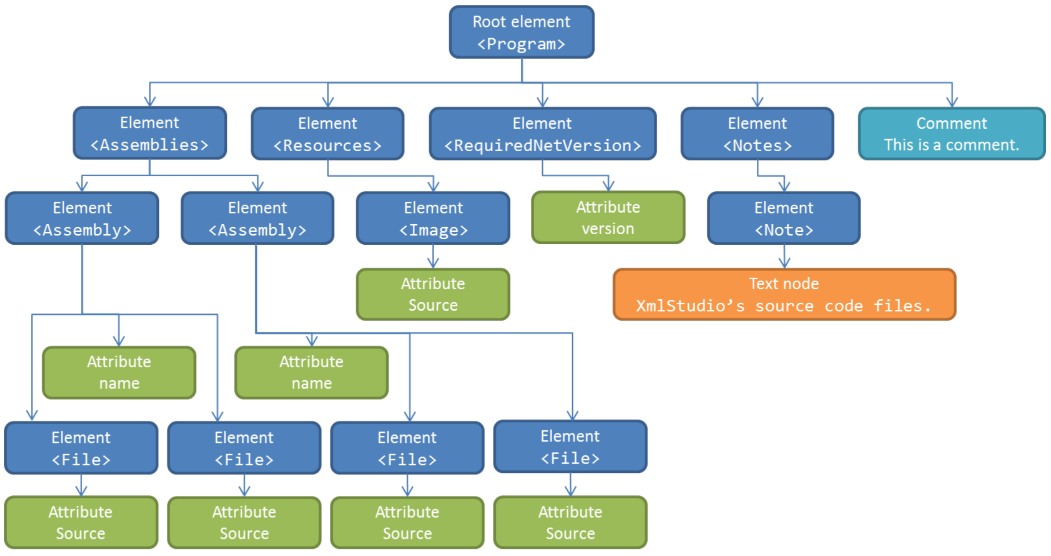


Figure 2.1: XML DOM tree hierarchy

2.3.2 Simple API for XML

Simple API for XML (SAX) [13] is a programming interface based on events. XML parser reads the document and when it encounters any significant items (start-tag, end-tag, text node etc.) it raises an appropriate event. A programmer needs to subscribe for these events using event handlers. These event handlers can be programmed to do any desired actions the programmer needs. This provides the developer with forward-only access to XML document. This also limits its usage. The programmer has to know what to do precisely because he cannot jump from one node to another randomly. The advantage is that the whole document does not have to be stored in memory and leaves much smaller memory footprint. This makes it a great tool for processing huge XML documents which can be hundreds or thousands of megabytes in their size while preserving small memory consumption. Its usage is intented mostly for batch processing. Listing 7 shows some events raised by SAX parser when parsing the sample file from Listing 1.

```
1) startDocument() - beginning of the document
2) startElement() Program - beginning of the element
3) startElement() Assemblies
4) startElement() Assembly - also provides attributes
5) ...
6) endElement() Assembly
7) startElement() Assembly
8) ...
9) endDocument() - end of parsing the document
```

Listing 7: Example of SAX events

2.3.3 Other Methods

DOM and SAX are the most common ways of accessing and processing XML documents programmatically. All other methods are just their variations. Differences are mostly in their implementation or programmatic usage.

2.3.4 XML & .NET Framework

.NET Framework [14] contains two main APIs which deal with XML. Their implementation is placed within `System.Xml` namespace.

The first API is based on the class `XmlDocument` [15], which implements the W3C Document Object Model (DOM) Level 1 Core and the Core DOM Level 2. This API contains more classes which model the DOM, like `XmlNode`, `XmlElement`, `XmlEntity`, `XmlAttribute` and so on. Each of these classes represents the corresponding XML part or entity and contains methods to modify and create these XML parts.

The second approach is similar to SAX. `XmlReader` and `XmlWriter` are classes which provide us with this SAX like API (to be precise more like Streaming API for XML, StAX [16] because it is pull-based). With these classes we are able to read and write documents that do not fit into memory at once.

There is also a third option, quite new technology called LINQ to XML [17]. But it is only rewritten object model for XML using old API and modified to be queryable using LINQ [18], so we can say it falls into the first category. Creation and manipulation of XML documents using this new API is easier and less complicated in comparison to `XmlDocument` and related classes. It has been placed in the namespace `System.Xml.Linq`. More about all of these classes can be found in MSDN library, in `System.Xml` section[1] and `System.Xml.Linq` section[2] namespaces.

2.4 XPath Language

XPath [19] is a language for addressing parts of an XML document. XPath uses expressions to select nodes in an XML document. The following list summarizes expressions that can be used:

- `nodename` – selects all child nodes of the named node

[1] http://msdn.microsoft.com/en-us/library/system.xml.aspx

[2] http://msdn.microsoft.com/en-us/library/system.xml.linq.aspx

- `/` – selects the root node
- `//` – selects nodes from the current node that match the selection regardless of their position
- `.` – selects the current node
- `..` – selects the parent node of the current node
- `@` – selects attributes

The following list shows some basic path expressions and their results:

- `Program` – selects all the child nodes of the `Program` element
- `/Program` – selects the root element `Program`
- `Assemblies/Assembly` – selects all `Assembly` elements that are children of element `Assemblies`
- `//File` – selects all `File` elements regardless of their position in the document
- `Program//File` – selects all `File` elements that are descendants of the `Program` element
- `//@Source` – selects all attributes named `Source`

Predicates

Predicates in XPath are used to find specific nodes. They are always embedded in square brackets. The next example shows some simple predicates:

- `/Program/Assemblies/Assembly[1]` – selects the first `Assembly` element that is the child node of the `Assemblies` element which is the child node of the `Program` element
- `Assemblies/Assembly[last()]` – selects the last `Assembly` element that is the child of the `Assemblies` element
- `//File[@Source]` – selects all the `File` elements that have an attribute `Source`

XPath Wildcards

XPath wildcards are used to select elements that are unknown.

- `*` – matches any element node
- `@*` – matches any attribute node
- `node()` – matches any node (of any kind)

XPath Axes

XPath axis defines a set of nodes relative to the current node. The following list summarizes them:

- `ancestor` – selects the ancestors of the current node (parent, granparent, etc.)
- `ancestor-or-self` – selects all ancestors of the current node and the current node itself
- `attribute` – selects all attributes of the current node
- `child` – selects all children of the current node
- `descendant` – selects all descendants of the current node
- `descendant-or-self` – selects all descendants of the current node and the current node itself
- `following` – selects everything after the end tag of the current node
- `following-sibling` – selects all siblings after the current node
- `namespace` – selects all namespace nodes of the current node
- `parent` – selects the parent node of the current node
- `preceding` – selects everything before the start tag of the current node
- `precesing-sibling` – selects all siblings before the current node
- `self` – selects the current node

For better overview of XPath axes, see Figure 2.2.

Location Path Expression

Axes are used in location paths. They are absolute or relative. An absolute location path starts with a forward slash (`/`), a relative without the slash. The location path consists of one or more steps, e.g. absolute location path might look like this: `/step/step/step/...`. Every step consists of an axis, a node test (which identifies a node within axis) and zero or more predicates. The syntax for one step is as following: `axisname::nodetest[predicate]`. For example `child::text()` selects all text child nodes of the current node.

The result of an XPath expression can be a node set, a string, a boolean or a number.

For complete information about XPath language, see the specification of the XPath language [19].

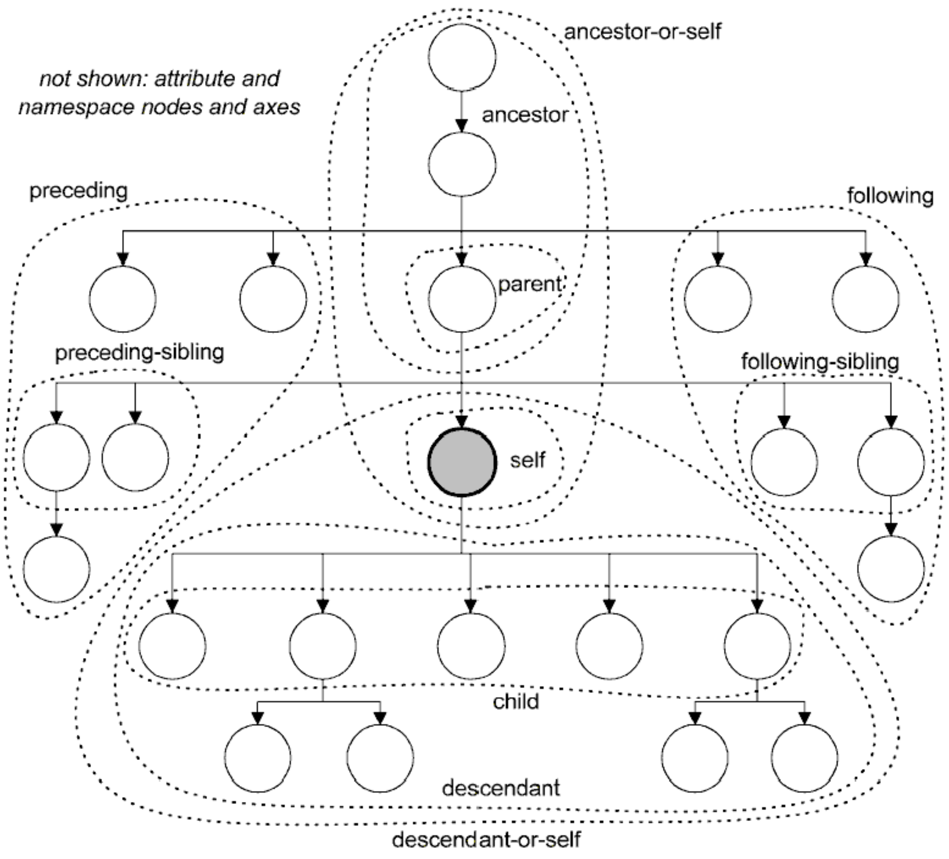


Figure 2.2: XPath axes

Chapter 3

Application Goals

Application created as an objective of this thesis is called XmlStudio. From now on when refering to this application, we use the name XmlStudio.

Main goals of XmlStudio application are the following points:

- XML data visualization.
- Modifications of the visualized structure.
- Modifications of appearance of particular visualized elements.
- Element selection.
- XML Schema editing and creation.
- Storing of the edited documents.
- Working with large documents.

3.1 XML Data Visualization

XmlStudio is an application for graphical manipulation, visualization and editing of XML documents. Graphical means that the user is not forced to edit the source code of the document but he gets a visual representation of the XML tree. Every element of a document, if it is an element or a text node, is represented with a separate graphical entity and these are connected according to the structure of the document. These connections emphasize the parent-child relationships.

XmlStudio creates the visual representation of XML document using simple interconnected graphical elements layed out on canvas. XmlStudio uses simple algorithm (which renders XML document as a tree) to layout the tree structure of XML document so the user gets a clear view of how the structure looks like. This view makes it much easier to grasp it.

3.2 Modifications of the Visualized Structure

XmlStudio enables editing of every element, its properties like name or namespace and its attributes. It is possible to add or remove elements from the document thus allowing modifications of the tree hierarchy. Whole subtrees can be copied or moved to another place in the document's tree. Single elements can be moved in their layer between their siblings (because order of elements is significant).

3.3 Modifications of Appearance of Particular Visualized Elements

Another goal is to provide users with easy modifications of rendered elements, mostly changing the shape or colours of selected items. Any shape, background and border colour can be modified as the user wishes.

3.4 Element Selection

XmlStudio offers selection filter to select any set of elements in currently rendered tree. There is a language for addressing parts of an XML document called *XPath* [19]. XmlStudio uses this language for element selection.

3.5 XML Schema Editing and Creation

XmlStudio is also able to create and edit schemas in XML Schema language. Using predefined sets of allowed elements and attributes, XmlStudio makes it easier for users to create or edit schemas. XML Schema language is one instance of the XML. Module which works with XSD benefits from all the tools available for generic XML documents so when user learns how to use tools for generic XML editing, learning XSD part is really easy (assuming the user knows XSD).

3.6 Storing of the Edited Documents

XML documents (or XSD schemas) can be saved back to the text format, exported to various image formats and sent to the printer.

3.7 Working with Large Documents

The last but important objective is to be able to work with very large documents which cannot be loaded into memory at once. XmlStudio makes it possible via simple expansion method (described in Chapter 6) which allows to load only certain parts of the document. All these features are described in more detail in the following chapters.

Chapter 4

User Documentation

XmlStudio offers tools to create and edit XML documents and XSD schemas. The following sections describe in detail what and how you can achieve with XmlStudio.

4.1 User Interface

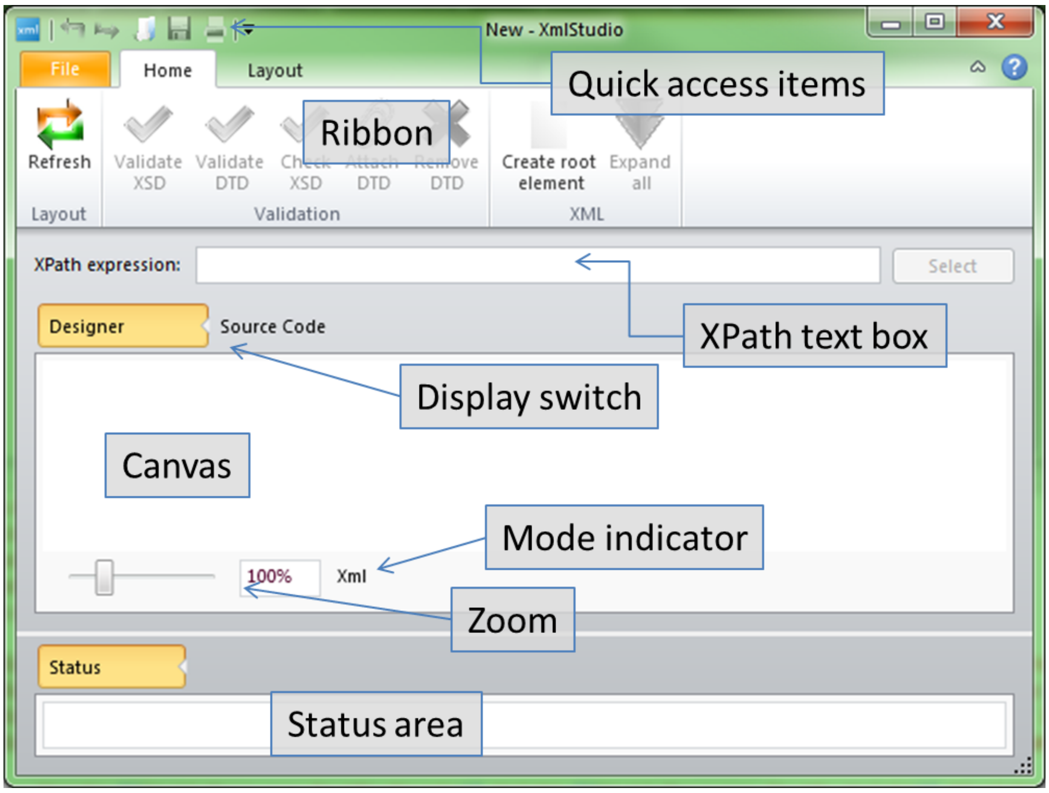


Figure 4.1: Main window

Figure 4.1 shows the main window of XmlStudio. It contains the following parts:

- Quick access items – the most frequently used commands can be added here
- Ribbon – the main toolbar

- XPath text box – input area for XPath queries
- Display switch – switches between `Designer` mode and `Source code` mode; `Source code` mode shows the source code of currently loaded document, however, it is read-only
- Canvas – the main working area
- Mode indicator – indicates whether the current document is being edited in `XML` or `XSD` mode
- Zoom – enables to zoom-in or zoom-out the canvas
- Status area – warnings and error messages from validation (DTD or XSD) are displayed here

4.1.1 The Ribbon

XmlStudio uses special interface called ribbon [31] where toolbars are placed on tabs. Ribbon was introduced by Microsoft [1] in Microsoft Office 2007 [2]. XmlStudio uses open source implementation called Fluent Ribbon Control Suite [29] which is targeted for Windows Presentation Foundation applications [26].

The ribbon has a special part called backstage which you can see when you click on tab File. Backstage area of XmlStudio is shown in Figure 4.2.

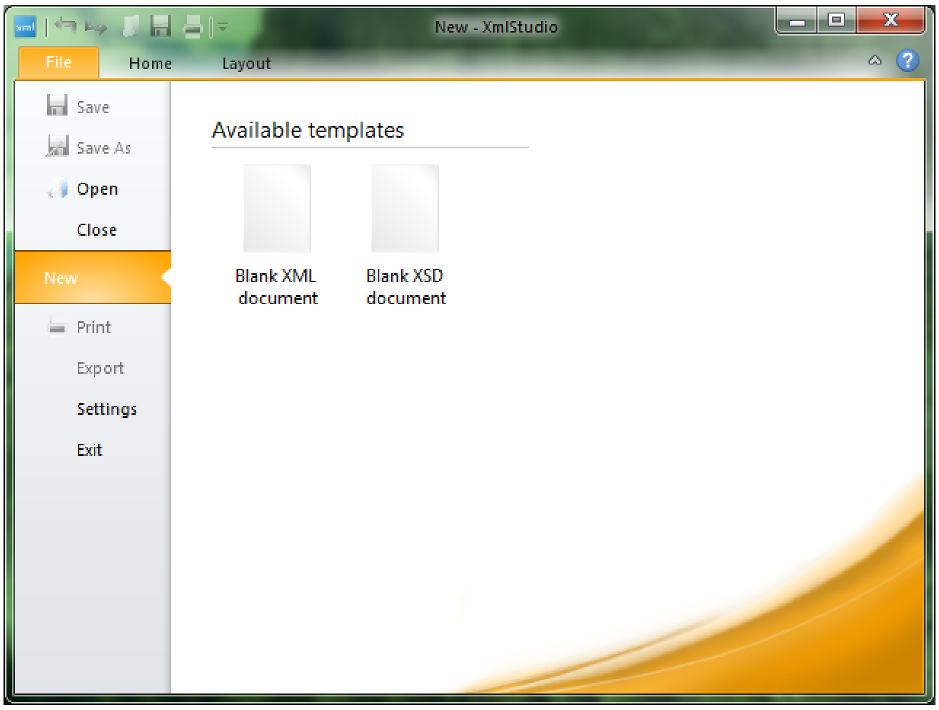


Figure 4.2: Backstage

There are several buttons:

[1]`http://www.microsoft.com/`
[2]`http://office.microsoft.com/`

- `Save` – saves the document to a file
- `Save As` – saves the document to a file with a chosen name
- `Open` – opens the document
- `Close` – closes the current document
- `New` – shows options to create either blank XML document or blank XSD document
- `Print` – shows dialog window with printing options
- `Export` – shows dialog window with options to export the current document to various image formats
- `Settings` – shows Settings dialog
- `Exit` – terminates the application

The Home tab (see Figure 4.3) contains the following commands:

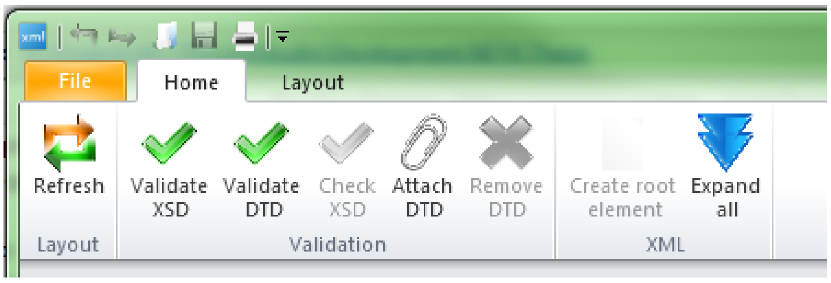
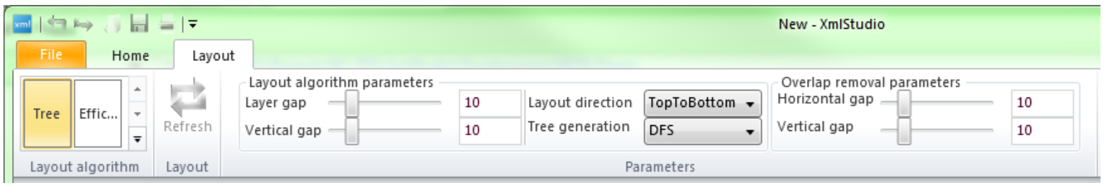


Figure 4.3: The Home tab

- `Refresh` – recalculates and redraws the rendered graph
- `Validate XSD` – tries to validate the document against XSD; it validates successfully only if XSD is attached inside the document
- `Validate DTD` – tries to validate the document against DTD; it validates successfully only if DTD is attached inside the document
- `Check XSD` – this command is disabled in XML mode
- `Attach DTD` – allows to attach external DTD to the current document, note that if there was DTD already specified, this replaces it
- `Remove DTD` – removes attached DTD
- `Create root element` – this command is available only when there is no root element in the document (when the document is empty); if enabled, it provides the user with dialog to create new root element
- `Expand all` – if this command is enabled, it makes it possible to expand all collapsed nodes in the document (warning: use this command only for small documents, expanding too many nodes leads to poor performance)

The Layout tab contains some basic settings for layout algorithms used in XmlStudio. Figure 4.4 contains settings for Simple tree layout algorithm.

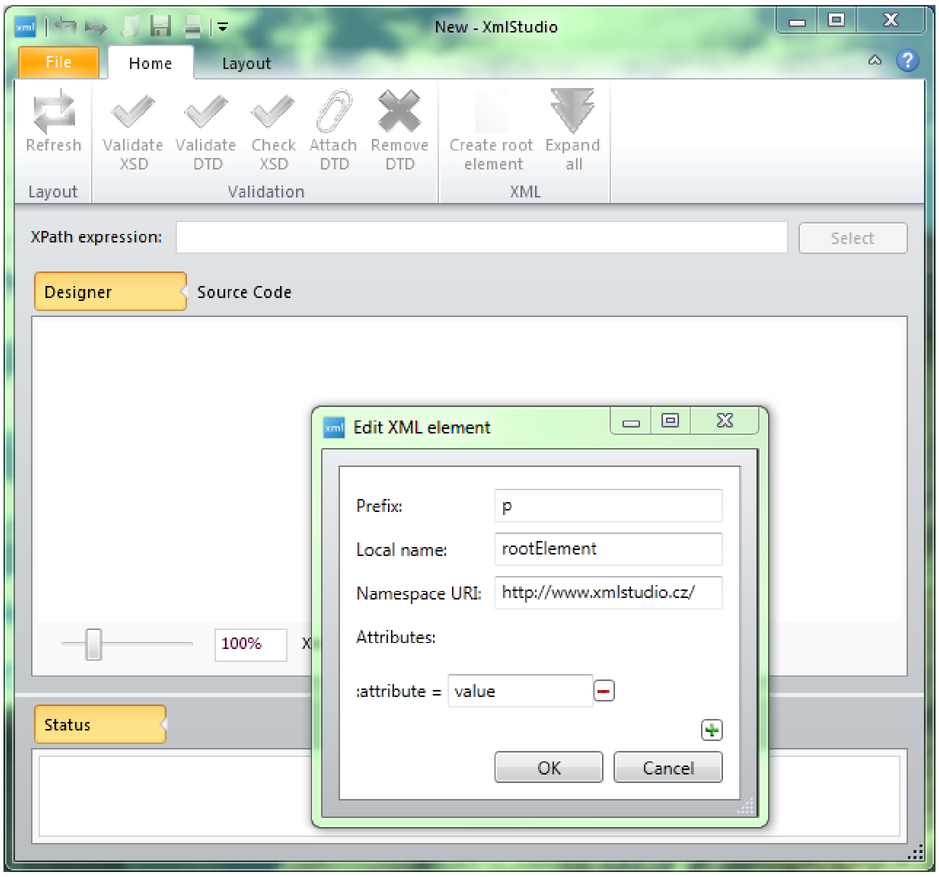


Layer gap 10
Vertical gap 10
Layout direction TopToBottom
Tree generation DFS
Horizontal gap 10
Vertical gap 10

Figure 4.4: The Layout tab

4.2 XML Document Creation

With XmlStudio you can create new XML documents. You need to follow simple steps to achieve it. First you need to start XmlStudio. Then you can start inserting new elements. Right-click on the canvas and select `Create root element` or click on `Create root element` command. The dialog, where you can enter the name (prefix, namespace) of the element, add some attributes if desired, pops up. When you are done with setting the properties of this new element, click OK and the element will appear on canvas. Figure 4.5 shows dialog with the new root element and Figure 4.6 shows the inserted root element.



Prefix: p
Local name: rootElement
Namespace URI: http://www.xmlstudio.cz/
Attributes:
:attribute = value

100%

Figure 4.5: Root element creation

With root element added, there are several step we can take:

- Edit properties of just added element.
- Add child element(s).

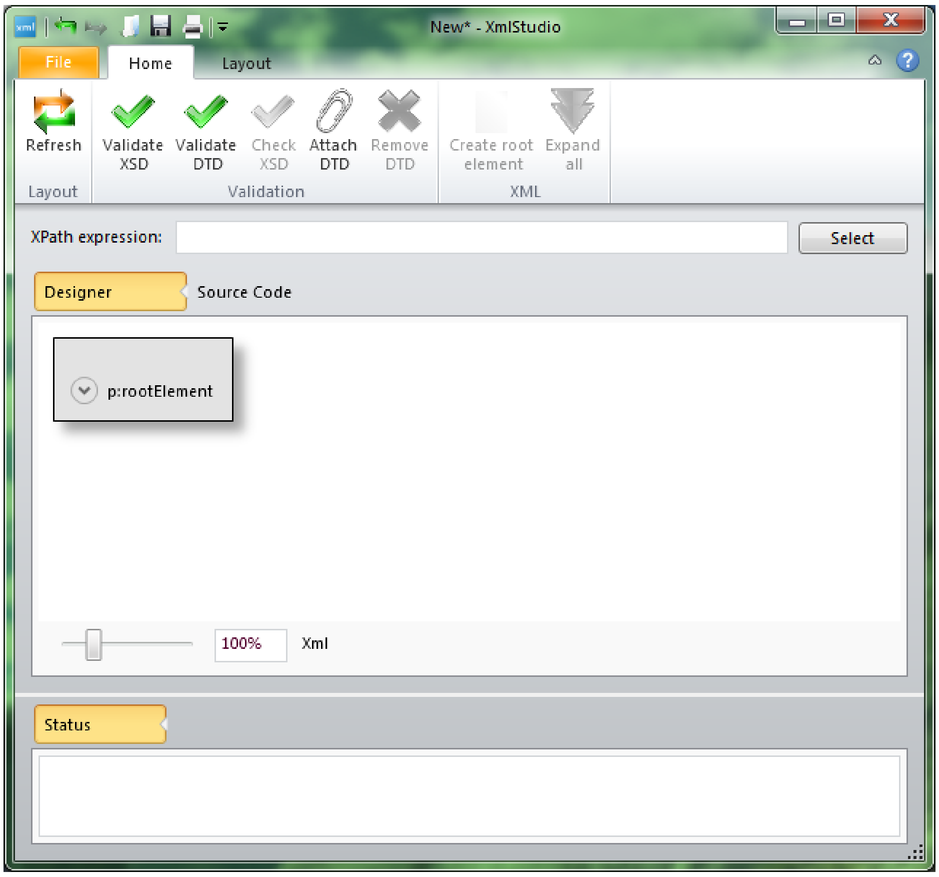


Figure 4.6: Added root element

- Add child text node(s).
- Change colours or shape of the visualized element.
- Remove the element.

4.3 Loading Existing XML Documents

To open an existing XML document, click on `Open` command from the backstage area. After selecting the file, the root element of the existing document is loaded in XmlStudio. This situation is shown in Figure 4.7. If the mouse cursor is over the root element, the small `plus` button is visible. This button serves to expand the element. If the element contains less than some number of child elements (this number can be set in settings of XmlStudio), the element is expanded right after clicking on this button. If there is more than the set number, the dialog pops up. In this dialog you can specify the range of child elements to expand. This range is expressed as a string, e.g. to expand first ten child elements and then elements from twenty to thirty, the range should be `1-10,20-30`. The dialog also displays the number of child elements of the element to be expanded. All this is shown in Figure 4.8. To expand all child nodes simply leave the text box empty.

If you did not expand all children you still can do so by clicking the `plus` button and specifying the remaining range or leaving the text box empty (which will expand all child nodes). You can also collapse the expanded node by clicking the `minus` button.

When you know the document contains smaller number of elements (e.g. less than few hundreds), you can skip expanding single elements and you can use the `Expand all` command from the ribbon. This will load the whole document into XmlStudio and render every node. Use this command with caution because loading huge documents will practically freeze the application.

Huge documents should be expanded with caution. You should expand only those branches or elements which you need to work with. All other parts of the document can stay collapsed.

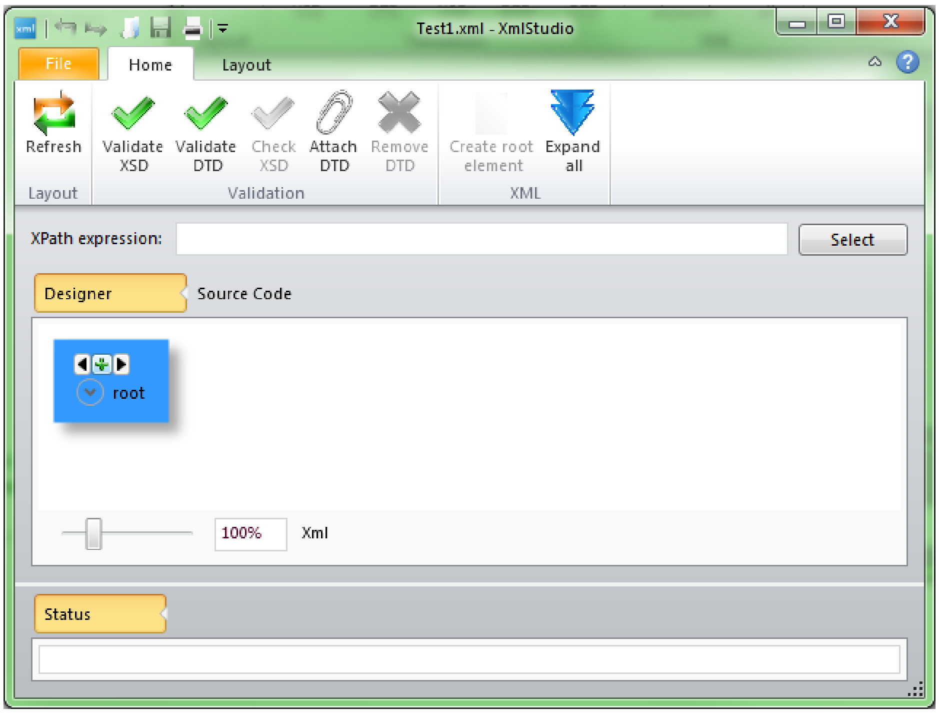


Figure 4.7: Opening an existing file with root element loaded

4.4 Editing Properties

To edit properties just right-click on the element and select `Edit`. The same dialog window as for adding root element pops up. There you can edit whatever properties you wish, then confirm your changes with `OK` button or cancel them with `Cancel` button. The second approach is to use the contextual tab which opens when a node is selected. This tab is shown in Figure 4.9. To edit the element, select it and then click `Edit` button in ribbon.

4.5 Adding Child Nodes

To add a new child element, right-click on the element and select `Add child element` from the context menu. Dialog box appears (similar to the one when editing an element). Type in the necessary data and confirm your intention

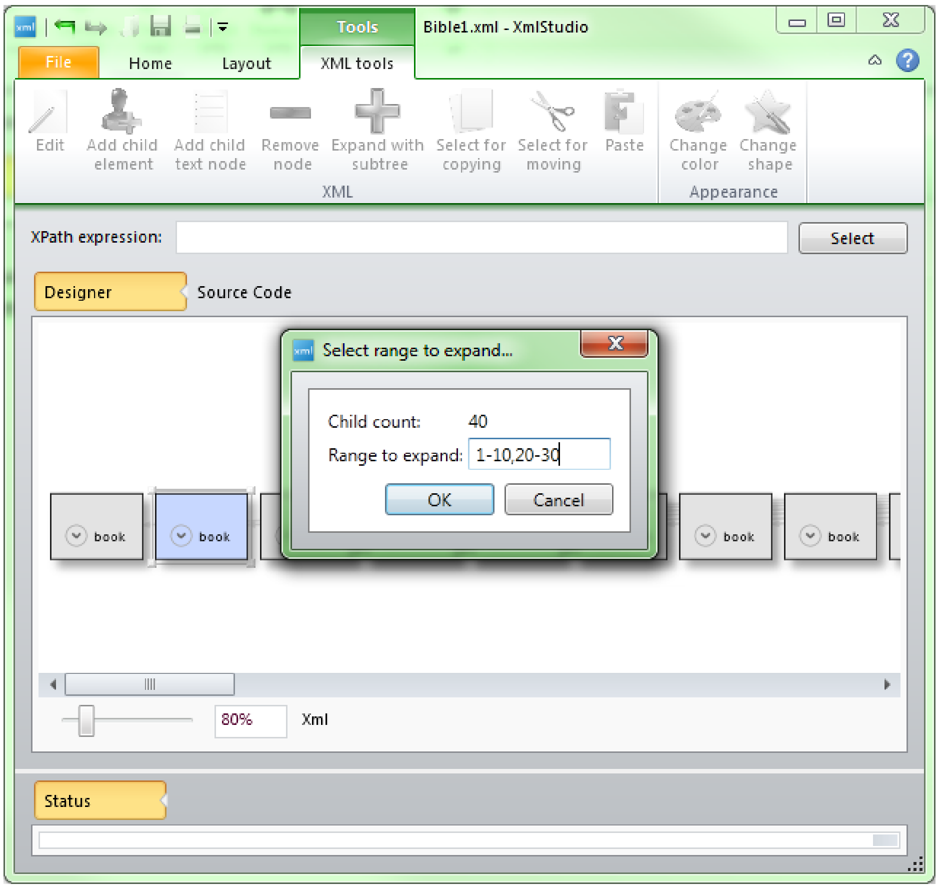


Figure 4.8: Expanding an element

by clicking `OK`. Figure 4.10 shows the added child element on canvas. The child element is connected to its parent with a line ended with an arrow. This empasizes the parent-child relationship. You can also use command from the contextual toolbar in ribbon as in the previous case. Use the `Add child element` button after selecting the desired element.

To add new text element, simply select `Add child text node` from the context menu of the parent element. In the dialog box insert the text which should be the content of the new text node and click `OK`. The new text node appears under its parent. The same can be done using the `Add child text node` command from the ribbon.

4.6 Changing Appearance

To change the colour or shape of selected items (elements or text nodes), first select them either by dragging the rectangle around them or by clicking on them while holding the `Ctrl` key. When you have selected all items you want, right click on any of them and choose `Change colors` or `Change shape` from the context menu (depending on what you want to change). The dialog box will appear. There you either change the colours or the shape of the element. As usual, confirm your action by clicking `OK`. Figures 4.11 and 4.12 show these two dialogs. Another way of doing the same is choosing the appropriate commands from the ribbon as in previous cases.

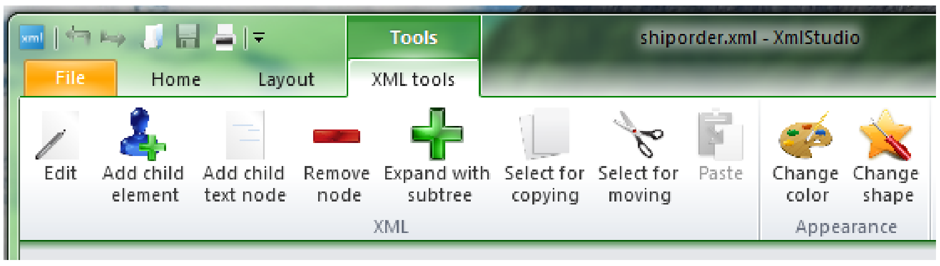


Figure 4.9: Tab with contextual tools

4.7 Node Removal

To remove the node (element or text node), select `Remove node` from the context menu. There will be a dialog asking you if you want to delete the whole subtree with the selected node. If you choose yes, the element and all its descendant nodes will be removed from the document. If you choose no, only the selected node will be removed and the child nodes of this node will be added where the removed node resided. If removed node had more than one child, all of these children will be moved to the higher level and become the child nodes of the former parent of the removed node. The same can be achieved using the command `Remove node` from the ribbon.

When the document consists of many elements, there are three functions which can help with copying or moving the whole subtrees.

4.8 Copying and Moving Subtrees of Nodes

The first two operations require two steps. The first step is to select the root element of the subtree which you want to copy (choose `Select for copying` from the context menu or from the ribbon) or move (choose `Select for moving` from the context menu or the ribbon). Then select the element which should be the destination of the operation, the new parent of copied (moved) subtree. Then choose `Paste subtree` from the context menu (or `Paste` from the ribbon) and the subtree will be copied (or moved) to the desired destination. Figure 4.13 and Figure 4.14 show the process of moving a subtree and Figure 4.15 shows the result.

The third operation is changing the order of a node among its siblings. Every visualized item has two small arrows (Figure 4.16) visible when the mouse is over it. Using these two arrows, it is possible to change the order of this node. It can be moved one item to the left or to the right. The content of the moved node does not change, all descendant nodes are moved with its parent.

4.9 Node Selection

XmlStudio benefits from XPath language and uses it to select nodes in the loaded document. To select nodes according to your XPath query, type in the query to the text box as shown in Figure 4.17 and click `Select`. The nodes selected by the query will get selected.

Tip: if you want to change colours or shapes of more items at once (of items

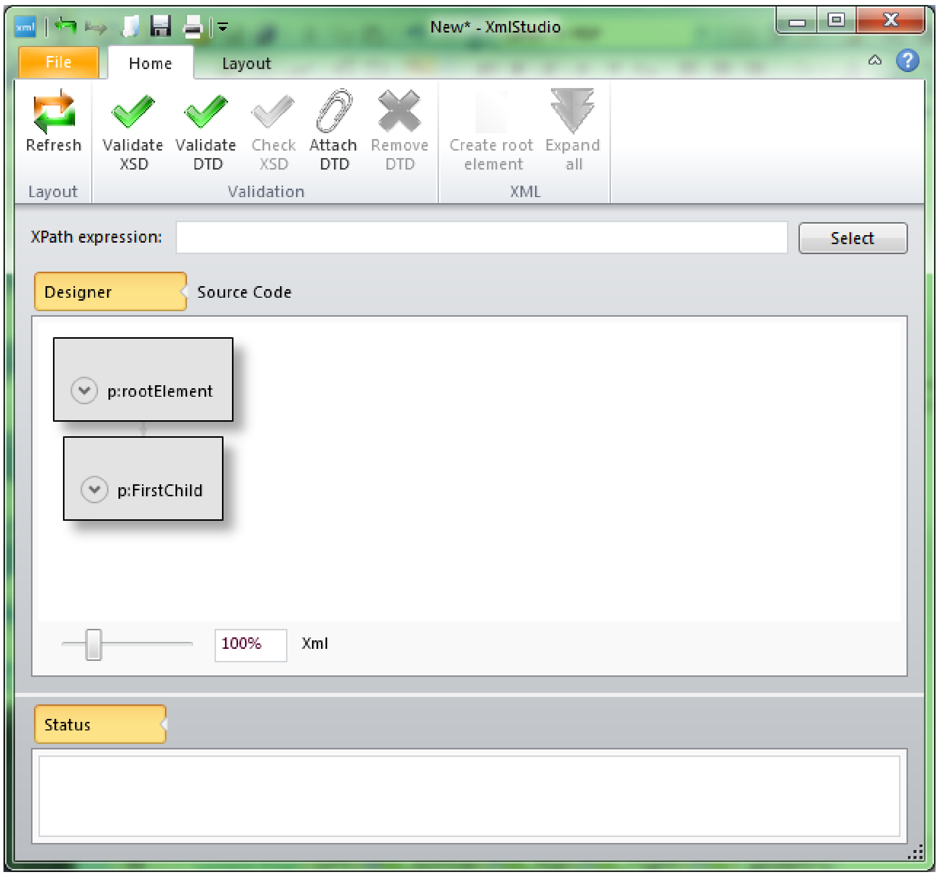


Figure 4.10: Added child element

that can be a result of an XPath query), write the query, select those items and then you can easily change visual properties of those items.

4.10 DTD Validation

There are two ways to validate existing document against DTD. If the current document already contains DTD, the validation can be done using the `Validate DTD` command from the ribbon. This will scan the document and report any errors or warnings to the status area.

If the document does not contain DTD declaration, you can add one using the `Attach DTD` command. It will prompt you to locate the file with DTD (please note that the current version of XmlStudio requires the DTD to be located in the same directory as the XML document) and it will automatically add the DTD declaration. Then the command `Validate DTD` works as in previous case.

4.11 XML Schema Validation

To validate the document against XSD, there already has to be XSD specified inside the XML document. This can be achieved using the standard editing tools available in XmlStudio (adding attributes) so there is no special tool for this case. Then using the command `Validate XSD` you get all the errors and warnings

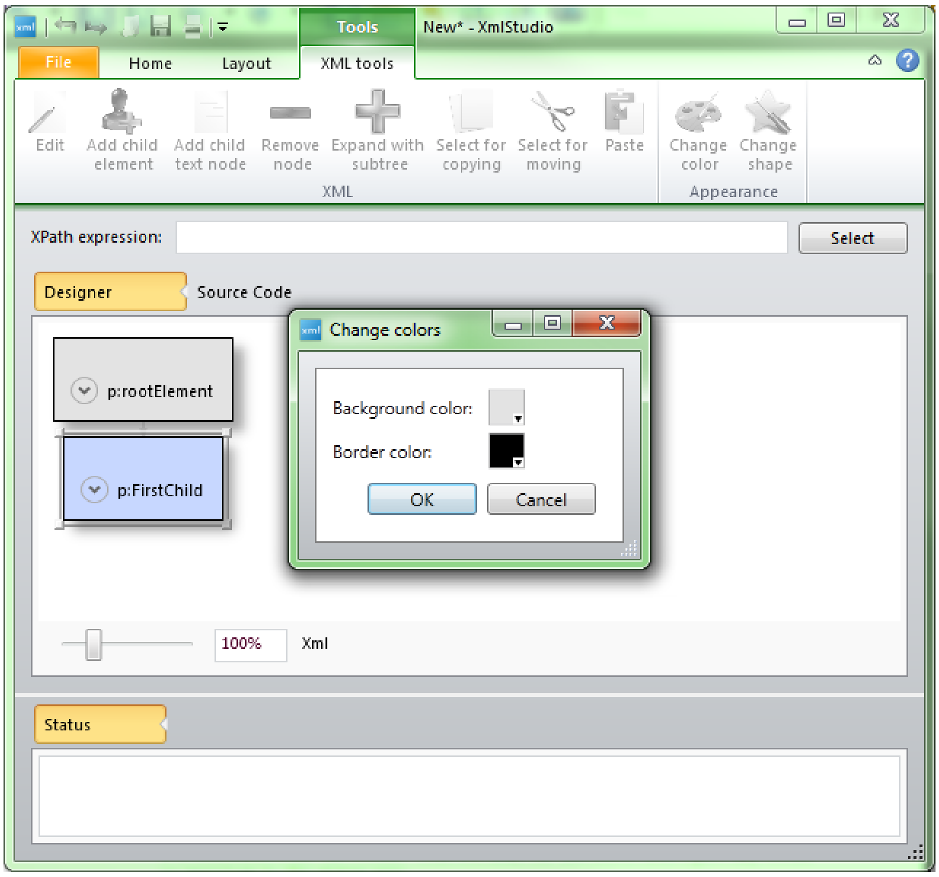


Figure 4.11: Dialog for changing the colours

violating the XSD.

4.12 Export

The visualized XML document can be saved to one of the following image formats: JPEG, BMP, PNG, TIFF, GIF, XPS. To export the current visualization, click on the `Export` command in the backstage area.

4.13 Printing

The visualized XML document can also be sent to the printer. To do so, click on the `Print` command in the backstage area.

4.14 Settings

XmlStudio offers some preferences that can be changed. The Settings dialog is displayed after clicking on `Settings` item in the backstage area. The dialog (see Figure 4.18) contains three tabs: General, Colors and XML. General tab contains the language selection. The user interface of XmlStudio is available in three languages: American English, Slovak and Czech. The next tab, Colors

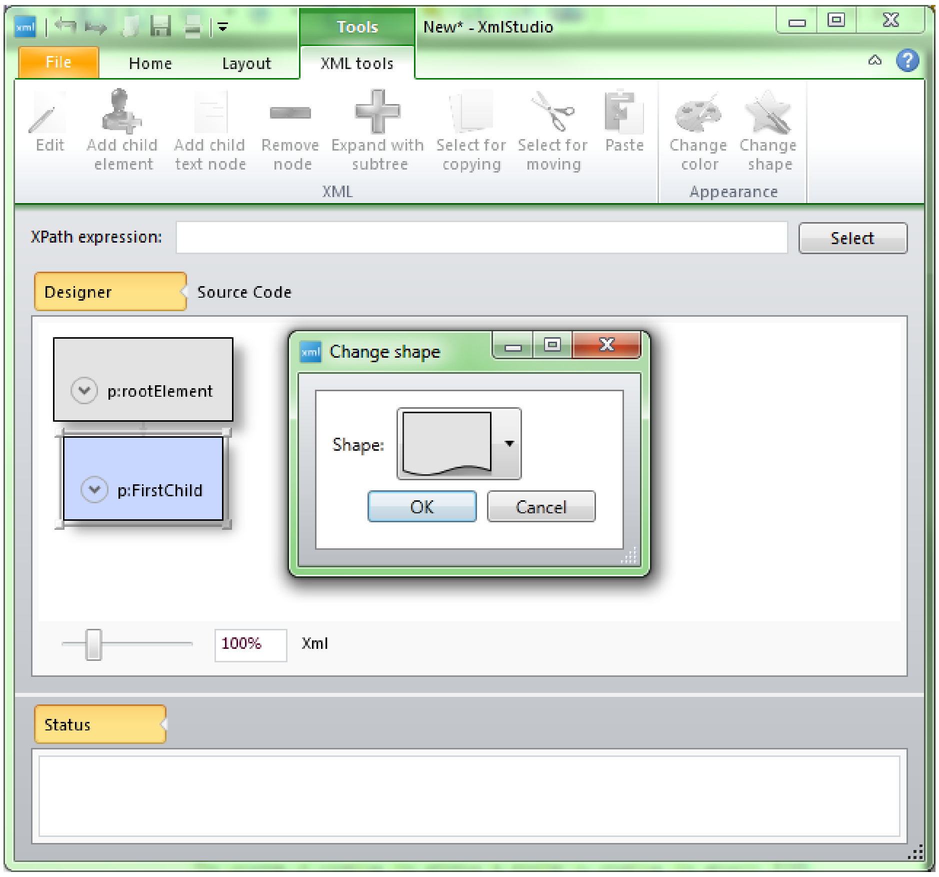


Figure 4.12: Dialog for changing the shape

(Figure 4.19), provides settings for default colours used by the application when rendering elements and text nodes.

4.15 XML Schema Creation and Editing

4.15.1 Creation

To create XML schema using XmlStudio, you need to create new blank document by clicking `New` in the backstage area and selecting `Blank XSD document`. XmlStudio is now in XSD mode and you can start creating new XML schema.

The process of creating the schema is similar to creating the generic XML document. First you need to add the root element. XmlStudio fills the name, prefix and namespace of the element for you because the root element must be the element `schema` in the namespace `http://www.w3.org/2001/XMLSchema`. The prefix on the other hand can be changed to anything you want, default is `xs`. After adding the root element, you can add child elements. The context menu gives you the list of all the elements that can be inserted as a child of the element `schema`. The example for the element `schema` can be seen in Figure 4.20.

This way the XML schema can be created from scratch. However, the context help currently does not take into account already inserted elements, so it always offers all the available elements as if the parent element had no children.

XmlStudio offers a tool to check whether the current document is the correct

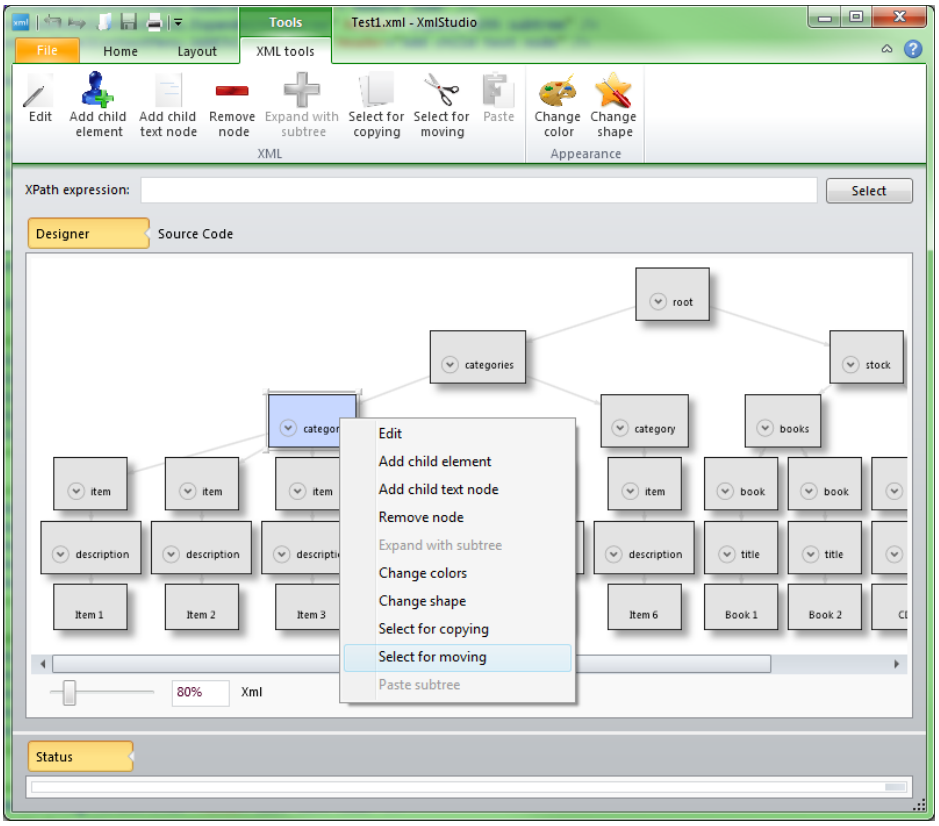


Figure 4.13: Moving the subtree, the first step

XSD schema. This tool is available from the ribbon and is called `Check XSD`. All the errors and warnings from this tool are reported to the status area.

4.15.2 Editing

If you open an existing XSD, it is loaded and displayed in the same manner as the generic XML document. You can expand and collapse elements as desired. You can also edit the element and their attributes, edit the structure by adding or removing elements etc. The only difference is that if you edit elements in XSD mode, the editing dialog offers you customized environment.

Figure 4.21 shows a dialog window when editing element `restriction`. Dialog box editing attribute `base` contains the list of available data types that can be set as a base type. This list contains built-in data types and every named data type already defined in this schema. The similar value lists are available also when editing element `attribute` (attributes `ref` and `type`), `attributeGroup` (attribute `ref`), `element` (attributes `ref`, `type` and `substitutionGroup`), `extension` (attribute `base`), `group` (attribute `ref`), `keyref` (attribute `refer`), `list` (attribute `itemType`) and `union` (attribute `memeberTypes`).

The rest of the tools works the similar way as in XML editing mode (including appearance changing, exporting, printing, etc.).

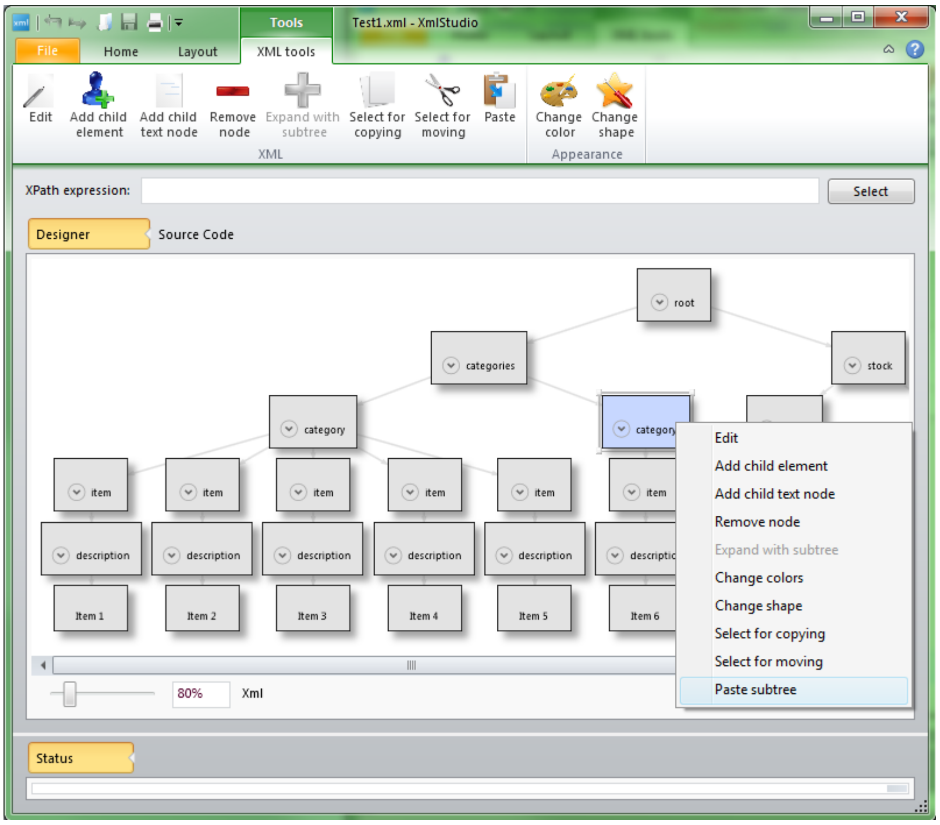


Figure 4.14: Moving the subtree, the second step

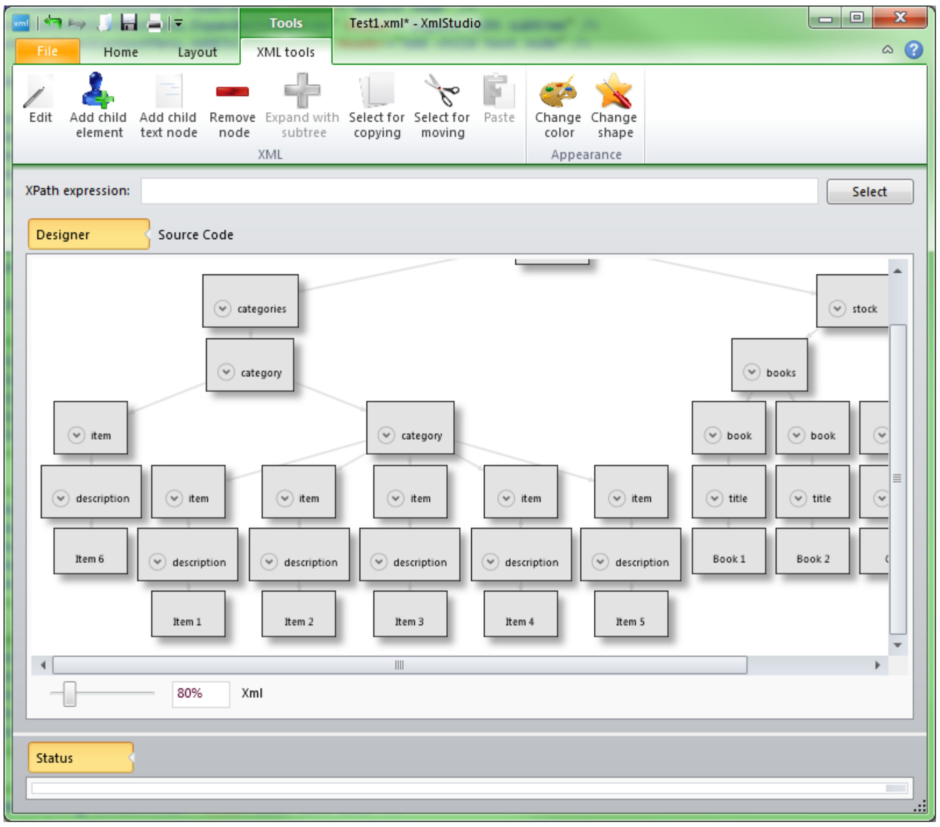


Figure 4.15: Result of moving the subtree

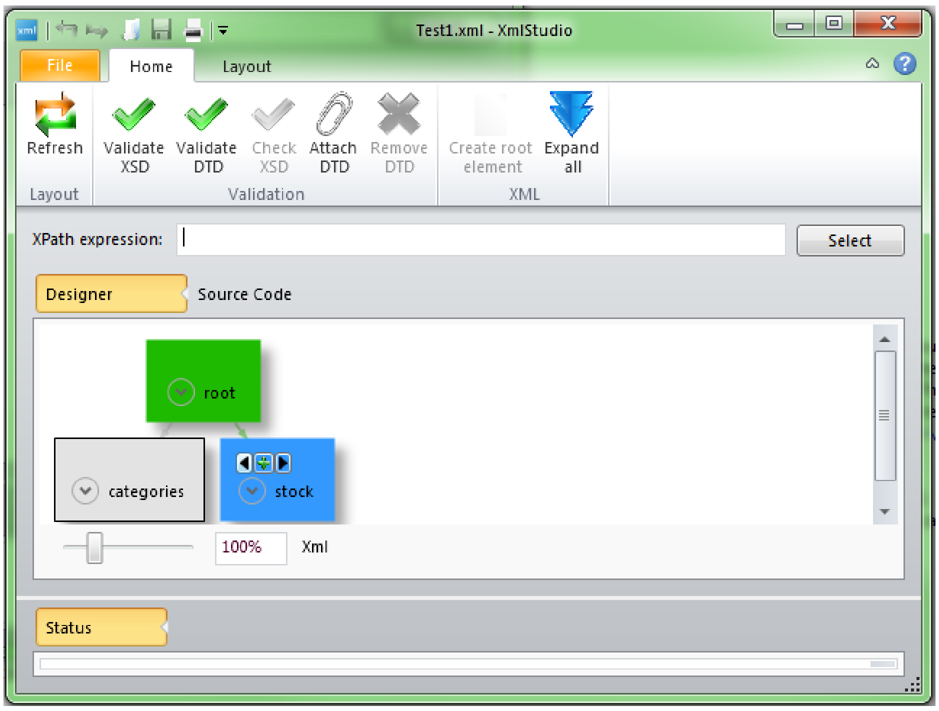


Figure 4.16: Arrows to change the element order

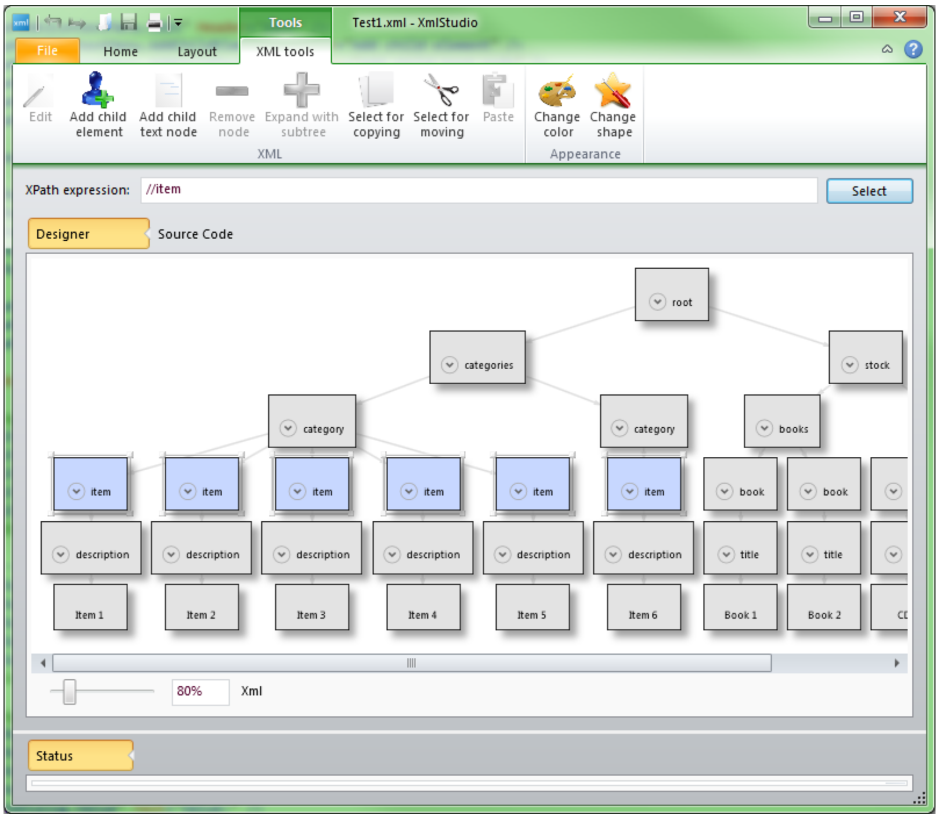
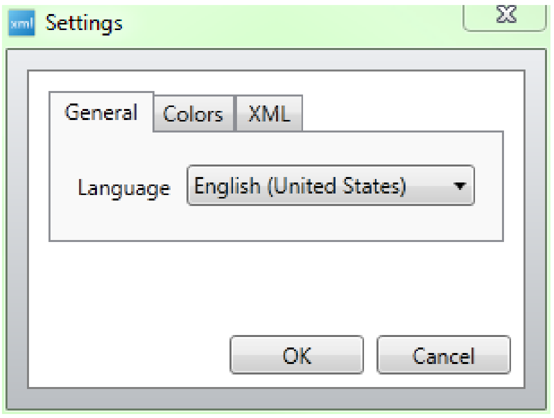
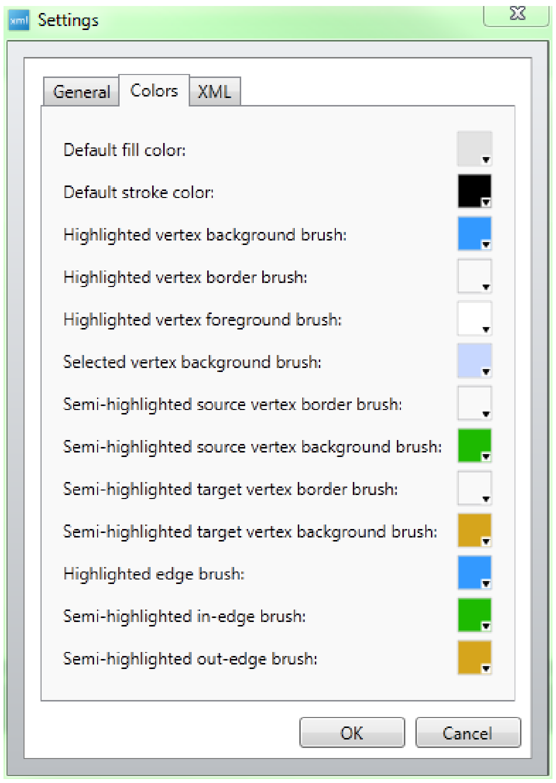


Figure 4.17: Selecting nodes using XPath language



Language: English (United States)

Figure 4.18: Settings



Default fill color:	⬜
Default stroke color:	⬛
Highlighted vertex background brush:	🟦
Highlighted vertex border brush:	⬜
Highlighted vertex foreground brush:	⬜
Selected vertex background brush:	🟦
Semi-highlighted source vertex border brush:	⬜
Semi-highlighted source vertex background brush:	🟩
Semi-highlighted target vertex border brush:	⬜
Semi-highlighted target vertex background brush:	🟨
Highlighted edge brush:	🟦
Semi-highlighted in-edge brush:	🟩
Semi-highlighted out-edge brush:	🟨

Figure 4.19: Colors tab

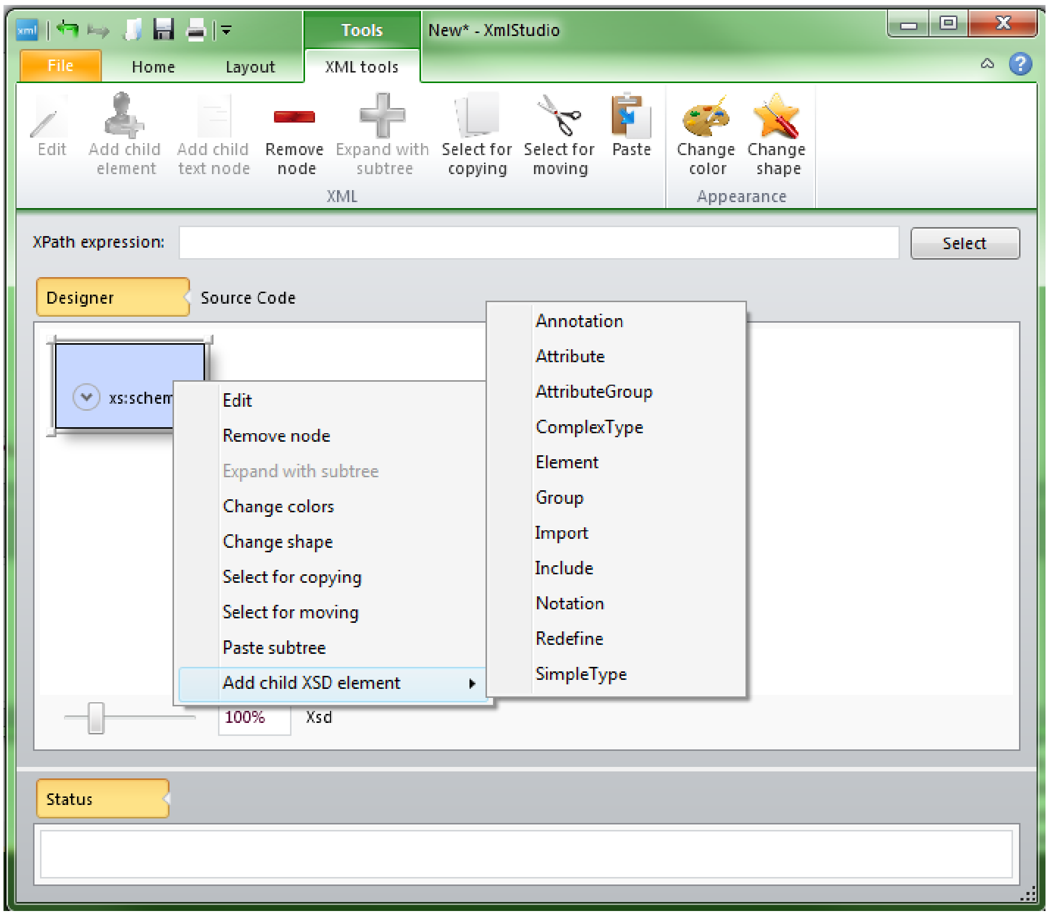
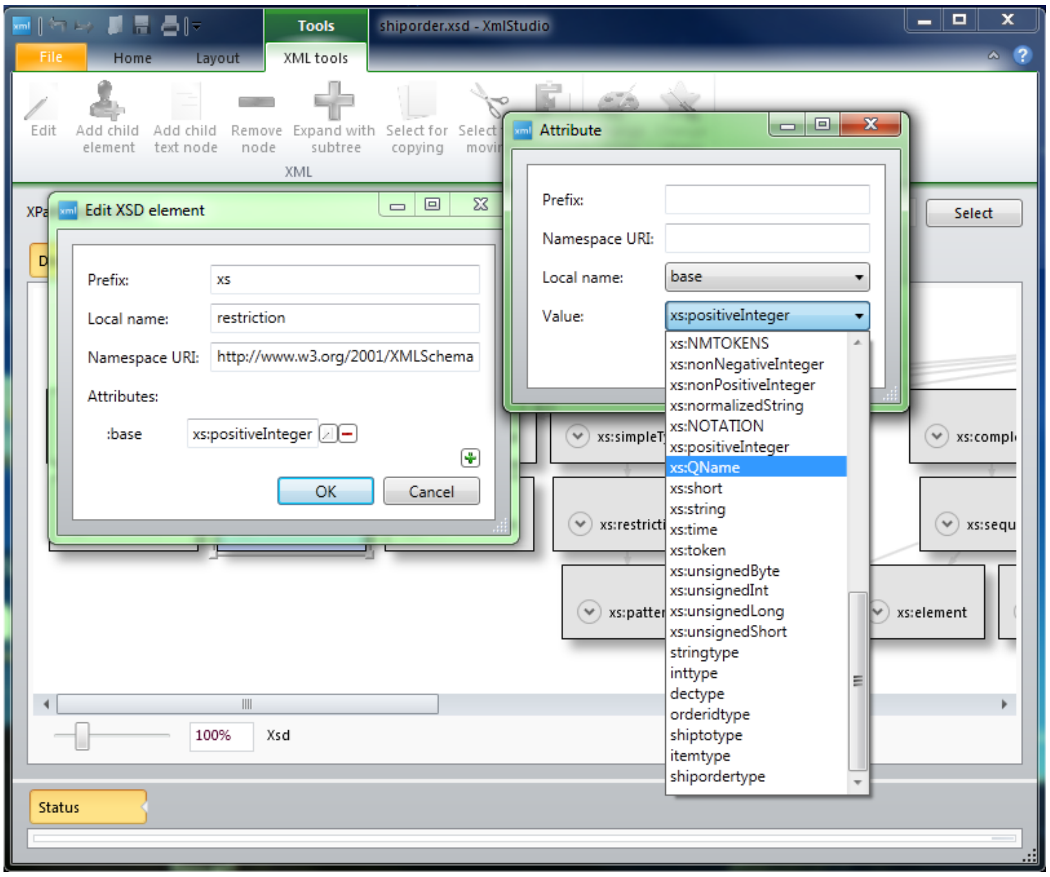


Figure 4.20: Available children of schema element



Edit XSD element

Prefix: xs

Local name: restriction

Namespace URI: http://www.w3.org/2001/XMLSchema

Attributes:

:base xs:positiveInteger

Attribute

Prefix:

Namespace URI:

Local name: base

Value: xs:positiveInteger

Figure 4.21: Editing in XSD mode

Chapter 5

Architecture of the Application

5.1 Introduction

The architecture of XmlStudio is based on separation of concerns. It means that the whole application is divided into set of modules where every module offers an encapsulated set of functions. This approach helps to keep the entire application maintainable during its life and easier to comprehend by other programmers.

When writing small applications using this technique, it might seem that it prolongs the development time (application is more complex than it would be when written the most straightforward way). It is often true but in the longer horizon any major changes require much more time and effort because the application was not written to be extendable. When deciding how to design XmlStudio, we took these questions into consideration and tried to make the best of it.

5.2 Overview of the Architecture

XmlStudio is written in C# language using .NET Framework 4 [1] as a development platform. The application consists of three main parts (assemblies) as shown in Figure 5.1. These modules are the result of used design patterns. The overall architecture is inspired by the MVC (Model-View-Controller) and the MVVM (Model-View-ViewModel) design patterns described in [23] and [24].

5.2.1 Model-View-Controller Design Pattern

Model-View-Controller is one of the compound design patterns used in software development. This pattern requires three parts: Model, View and Controller as you can see in Figure 5.2. The main advantage of this pattern is object decoupling. It means that Model does not depend on the View (user interface/presentation layer) or the Controller and vice versa. In other words if the programmer decides to change the presentation layer of the application he does not need to do any major changes in the Model (if the pattern is followed properly).

This design pattern however requires some overhead in the application which is mostly caused by the need for every call from the user interface to go through

[1] `http://www.microsoft.com/net/`

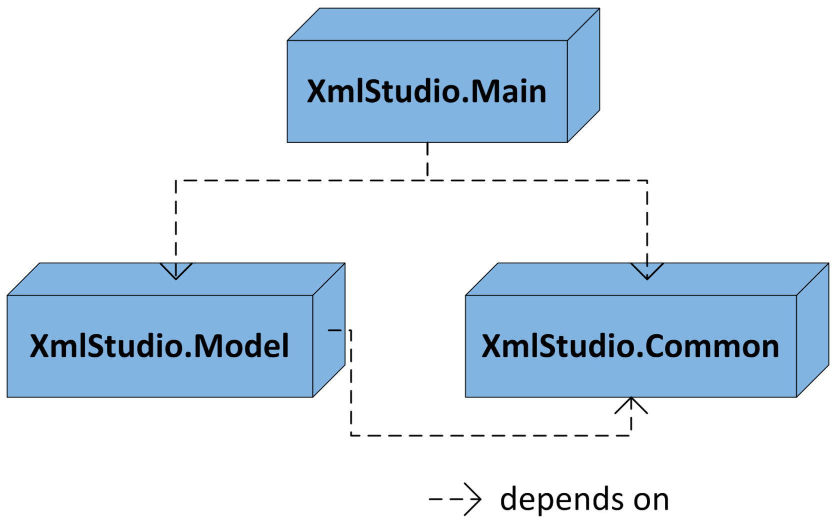


Figure 5.1: Main program modules

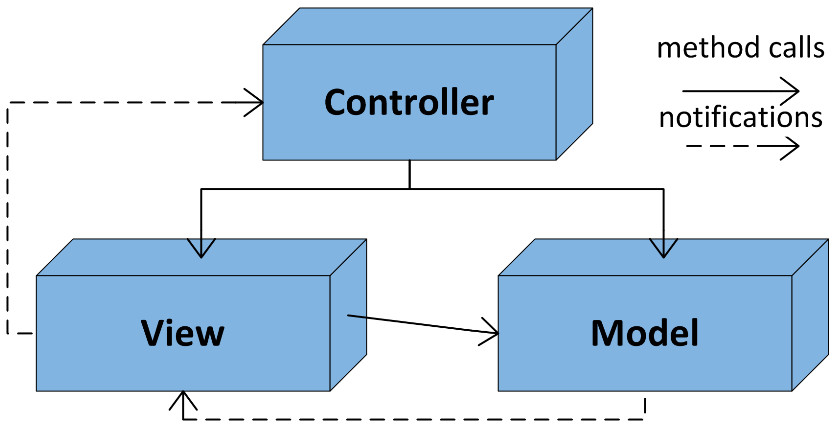


Figure 5.2: Model-view-controller

the Controller and then to the Model. But the benefits of this pattern overweight the disadvantages. It provides us with three independent units and guides us to write better encapsulated objects. It means easier future changes in the code and easier maintenance of the application.

The Model

The Model holds all the state data and application logic. It does not depend on the View or the Controller. On the other hand it provides View and Controller with the interface which they can use to communicate with it. Model communicates with the View using the observer pattern. It notifies the View about changes in itself and it can either send it related data along with the notification or the View pulls the data out of the Model on its own. Model does not need to know anything about the Controller. Simply said, it does not depend on the Controller at all.

The Controller

The Controller 'sits' in the middle between the Model and the View. It receives user inputs and decides what it means to the Model. It usually calls some Model methods. Controller then communicates back with the View and notifies it to change its state. Relation between the View and the Controller could be best described in the strategy pattern [25]. The View cares only about presentation layer while the Controller decides what the user actions mean. The Controller is the strategy for the View. If we change the Controller we could get another strategy.

The View

The View could be described as an example of composite pattern. User interface elements form the composition and create the whole graphical user interface. These units are main user input points.

5.2.2 Model-View-ViewModel Design Pattern

Model-View-ViewModel [24] is a design pattern that originated in Microsoft[2]. It is targeted for new user interface (UI) development platforms (like Windows Presentation Foundation, WPF [26] used in XmlStudio).

This patterns attempts to benefit from both MVC as well as from the Extensible Application Markup Language (XAML) [27] and WPF and their databinding possibilities. This pattern tries to remove all the code behind from the View layer and moves it to the ViewModel layer. View contains only bindings to the ViewModel thus provides better separation of concerns.

The Model is practically the same as in MVC pattern. The View refers to all the graphical user interface (GUI) elements such as windows, buttons, text boxes, labels etc. The ViewModel is a "model of the view" which means that it is an abstraction of the View. It servers as a data binding between the Model and the View. Simpy said it holds data ready to be displayed in the user interface.

The following sections describe the main assemblies used in XmlStudio.

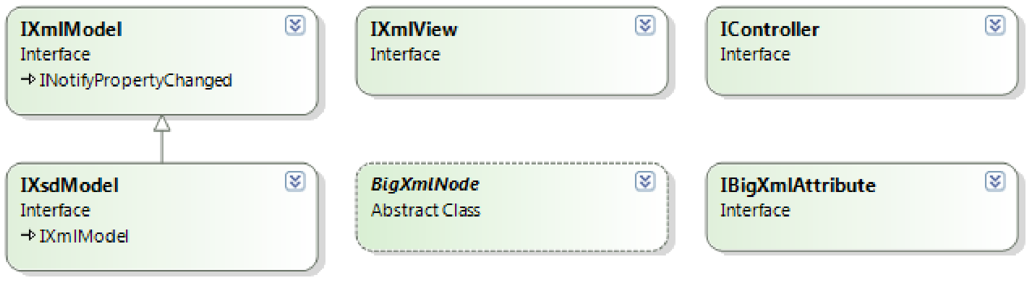


Figure 5.3: Base common classes and interfaces

[2]http://www.microsoft.com/

5.2.3 XmlStudio.Common

`XmlStudio.Common` is an assembly which contains common classes and interfaces used by both `XmlStudio.Main` and `XmlStudio.Model` assemblies. The most important classes and interfaces are shown in Figure 5.3. Every rectangle represents a class or an interface. The line ended with an empty arrow means that the class (or interface) where the line begins inherits from the class (interface) where the line ends. In our case, the interface `IXsdModel` inherits from the interface `IXmlModel`.

IXmlModel

This interface defines properties, methods and events which should be implemented by the class which serves as an application model. It contains all the necessary methods for generic XML part of XmlStudio. Some members of the `IXmlModel` interface can be seen in Listing 8.

```
public interface IXmlModel : INotifyPropertyChanged {
    // Properties
    bool CanUndo { get; }
    bool CanRedo { get; }
    bool Modified { get; }
    string DocumentName { get; }
    string Path { get; }
    int NodeCount { get; }
    ApplicationMode Mode { get; set; }
    XmlDocumentType DocumentType { get; }

    // Some methods
    void Open(string name);
    void Close();
    void Save();
    void Save(string path);
    BigXmlNode GetRootElement();
    void SetDocumentType(string rootElementName, string publicId, string systemId,
      string internalSubset);
    void RemoveDocumentType();
    void ExpandNode(BigXmlNode node, string range = "", bool wholeSubtree = false);
    void ExpandAll();
    // and so on...

    // some events
    event EventHandler<NodeExpandedEventArgs<BigXmlNode>> NodeExpanded;
    event EventHandler<NodeCollapsedEventArgs<BigXmlNode>> NodeCollapsed;
    event EventHandler<RawXmlChangedEventArgs> RawXmlChanged;
    event EventHandler<DocumentNameRequiredEventArgs> DocumentNameRequired;
    event EventHandler<NodeUpdatedEventArgs<BigXmlNode>> NodeUpdated;
    event EventHandler<SubtreeUpdatedEventArgs<BigXmlNode>> SubtreeUpdated;
    event EventHandler<NodeAddedEventArgs<BigXmlNode>> NodeAdded;
    event EventHandler<NodeRemovedEventArgs<BigXmlNode>> NodeRemoved;
    // and so on...
}
```

Listing 8: IXmlModel interface

IXsdModel

This interface inherits from the `IXmlModel` interface and defines new additional methods required for XSD part of XmlStudio. Some members of the `IXsdModel` interface can be seen in Listing 9.

```
public interface IXsdModel : IXmlModel {
    // some of the IXsdModel members
    string XsPrefix { get; }
    void CheckXsd();
    // ...
    BigXmlNode CreateXsdElement(string name);
    IEnumerable<string> NamedAttributes { get; }
    IEnumerable<string> NamedSimpleTypes { get; }
    IEnumerable<string> NamedComplexTypes { get; }
    IEnumerable<string> NamedAttributeGroups { get; }
    IEnumerable<string> NamedGroups { get; }
    IEnumerable<string> NamedElements { get; }
    IEnumerable<string> NamedKeyElements { get; }
    IEnumerable<string> NamedUniqueElements { get; }

    IEnumerable<XsdBuiltInDataType> ListBuiltInDataTypes();
    IEnumerable<BigXmlNode> ListNamedAttributes();
    IEnumerable<BigXmlNode> ListNamedSimpleTypes();
    IEnumerable<BigXmlNode> ListNamedComplexTypes();
    IEnumerable<BigXmlNode> ListNamedAttributeGroups();
    IEnumerable<BigXmlNode> ListNamedGroups();
    IEnumerable<BigXmlNode> ListNamedElements();
    IEnumerable<BigXmlNode> ListNamedKeyElements();
    IEnumerable<BigXmlNode> ListNamedUniqueElements();
    // ...
}
```

Listing 9: IXsdModel interface

IXmlView

`IXmlView` interface defines few methods to be implemented by the View of the application (in case of XmlStudio, the window). Members of the `IXmlView` interface can be seen in Listing 10.

```
public interface IXmlView {
    Point GetCurrentMousePosition();
    void Close();
    void AddMessageToStatus(string message);
    void ExportCanvas();
    void PrintCanvas();
    void ChangeMode(ApplicationMode mode);
    void InitXmlMode(string path);
    void InitXsdMode(string path);
    void SetProgressBarVisibility(bool visibility);
    void SetProgressBarValue(int value);
}
```

Listing 10: IXmlView interface

IController

`IController` interface defines all the methods used from the side of the View to control the application. Some of them can be seen in Listing 11.

```
public interface IController {
    void ValidateXsdAsync();
    void CancelValidateXsd();
    void ValidateDtdAsync();
    void ValidateDtd();
    void CancelValidateDtd();
    void SetDocumentType(string rootElementName, string publicId, string systemId,
      string internalSubset);
    void RemoveDocumentType();
    // ...
    void CheckXsd();
    BigXmlNode CreateXsdElement(string name);
    // ...
}
```

Listing 11: IController interface

BigXmlNode

`BigXmlNode` is an abstract class which serves as a data transfer object [30] between the Model and the View. It encapsulates properties and methods of XML nodes so it can be easily displayed in the View. It includes properties like Name, Prefix, NamespaceURI, Parent, Value, Attributes etc. All the details are covered in source code of XmlStudio.

5.2.4 XmlStudio.Model

Assembly `XmlStudio.Model` contains implementation of Model, Controller and other helper classes which we can denote as data layer.

Data layer consists of classes which can be seen in Figure 5.4. These classes encapsulate functionality to read from XML documents and write to them. They are the core of the XmlStudio. They provide the Model and higher layers with data from hard drive and take care of writing it back.

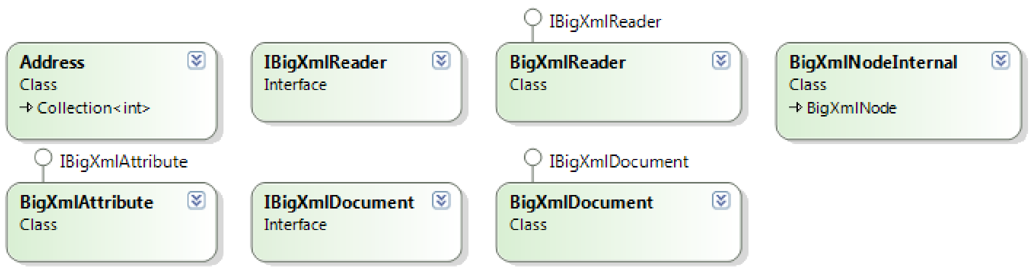


Figure 5.4: Data layer classes

Another set of classes shown in Figure 5.5 implements interfaces for Model and Controller and also contain helper factory classes to create Model and Controller.

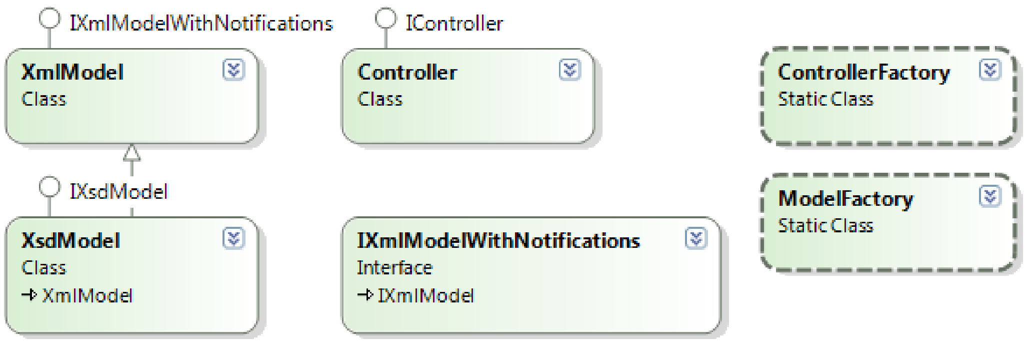


Figure 5.5: Model, controller classes

Next few classes in Figure 5.6 serve for XSD manipulation (`XsdModel`), creation of `XsdModel` (`XsdFactory`) and the last class (`XsdAttributeFactory`) is a helper to list attributes of XSD elements.

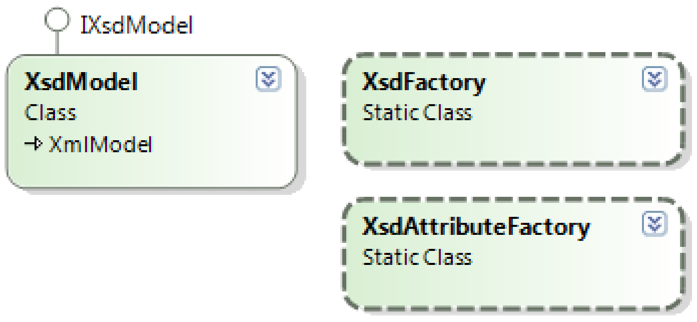


Figure 5.6: XSD classes

Important classes are classes inheriting from the abstract class `XmlModelAction`. These classes are actions which manipulate with the structure of XML document. All of these actions are undoable. XmlStudio uses Undo Framework [21] to implement the undo and redo functionality. Every action needs to implement two methods shown in Listing 12. Undo Framework contains all other necessary plumbing so the use of these actions is simple as shown in Listing 13.

5.2.5 XmlStudio.Main

Classes in assembly `XmlStudio.Main` are mostly GUI elements like windows, various controls and other helper classes.

GraphSharp

XmlStudio uses open source project GraphSharp [28] for rendering and layouting the visualized elements. GraphSharp is a graph layout framework. It contains various layout algorithms and a GraphLayout control for use in WPF applications. XmlStudio uses some of the layout algorithms to display the loaded document (details are covered in Chapter 6).

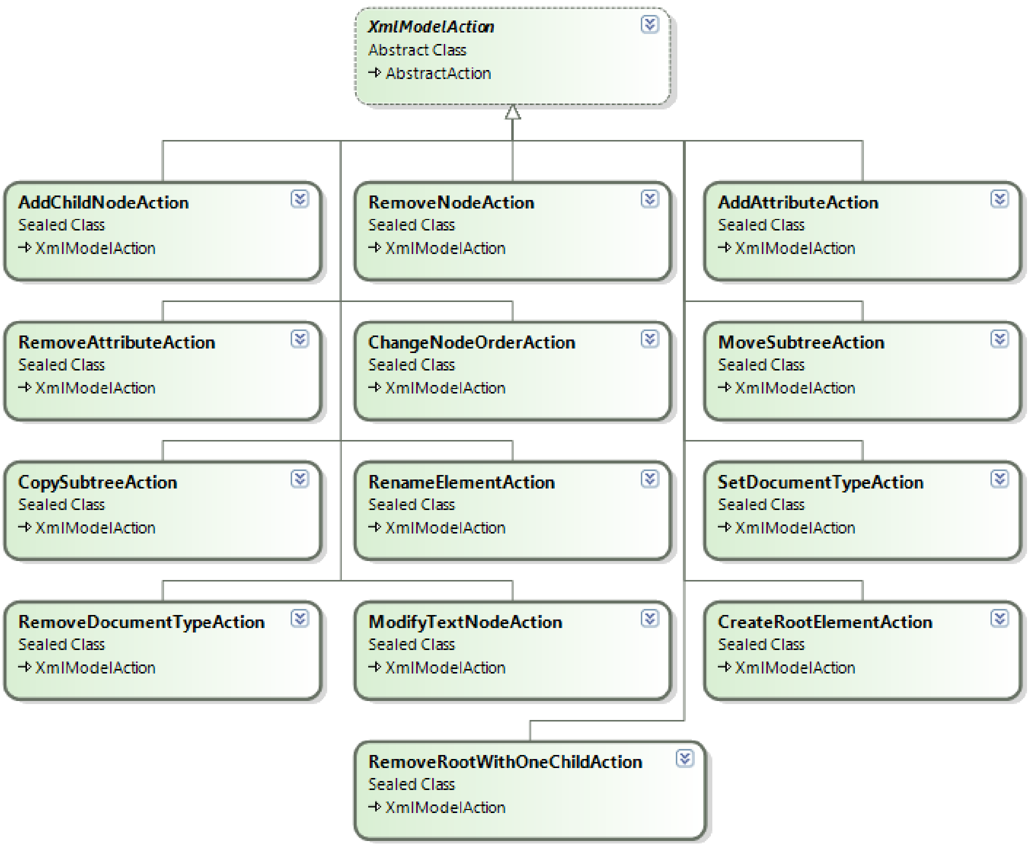


Figure 5.7: Action classes

Fluent

According to [29], “Fluent Ribbon Control Suite is a library that implements an Office-like (Microsoft® Office Fluent™ user interface) for the Windows Presentation Foundation (WPF).” XmlStudio benefits from this library and uses it for its main window and toolbar. Figure 5.8 shows an example of Fluent user interface.

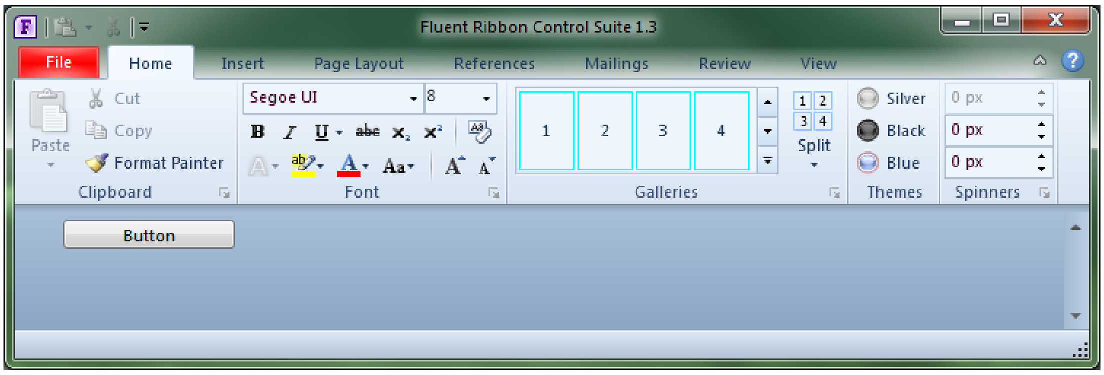


Figure 5.8: Fluent user interface

```
public abstract class AbstractAction : IAction {
    //...

    /// <summary>
    /// Override execute core to provide your logic that actually performs the action
    /// </summary>
    protected abstract void ExecuteCore();

    /// <summary>
    /// Override this to provide the logic that undoes the action
    /// </summary>
    protected abstract void UnExecuteCore();

    //...
}
```

Listing 12: AbstractAction class

```
// ...
var action = new RenameElementAction(this, nodeInternal, name);
this.actionManager.RecordAction(action);
// ...
```

Listing 13: Action usage

Tomers.WPF.Localization

XmlStudio supports multiple languages in its user interface. The library which provides it with all the required classes is called `Tomers.WPF.Localization` [20].

The translated expressions and texts are placed in separate XML files, each for one language (culture). This library also provides a markup extension for XAML so that the translation can be done inside the declarative markup which defines the user interface, not in the code behind. The simple example of its usage in XAML code can be seen in Listing 14.

We need to mention two important things about the usage of this localization library. The first one is the need to declare the `Translate.Uid` attribute on every element which we want to translate. This attribute defines a key which we later use in XML document with translations. The value of the translated property is replaced with the code `{loc:Translate Value}`, where `Value` is the default value to be shown when the key is missing from the XML file. The example of how this XML file with translated values might look is shown in Listing 15. You can notice that sometimes there is more than one value declared for the same key, this is particulary useful when we want to translate more than one property of the GUI element, e.g. properties `Text` and `ToolTip` of a GUI control. The library of course handles the look up of the appropriate values from the XML file and the proper binding to GUI elements.

The ViewModels

Another important classes in the `XmlStudio.Main` assembly are the ViewModels. There are several of them, the following listing summarizes them.

- `XmlAttributeCollectionViewModel` - ViewModel for the collection of at-

```
<Window x:Class="XmlStudio.Dialogs.EditXsdElementDialog"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        loc:Translate.Uid="EditXsdElementDialog.WindowTitle"
 xmlns:loc="clr-namespace:Tomers.WPF.Localization;assembly=Tomers.WPF.Localization">

<!-- ... -->

<TextBlock Height="{StaticResource TextBlockHeight}" Grid.Row="0"
    x:Name="prefixTextBlock" Text="{loc:Translate Prefix:}"
    loc:Translate.Uid="EditXsdElementDialog.prefixTextBlock" />
<TextBlock Height="{StaticResource TextBlockHeight}" Grid.Row="1"
    x:Name="localNameTextBlock" Text="{loc:Translate LocalName:}"
    loc:Translate.Uid="EditXsdElementDialog.localNameTextBlock"  />
<TextBlock Height="{StaticResource TextBlockHeight}" Grid.Row="2"
    x:Name="namespaceURITextBlock" Text="{loc:Translate NamespaceURI:}"
    loc:Translate.Uid="EditXsdElementDialog.namespaceURITextBlock"  />
<TextBlock Height="{StaticResource TextBlockHeight}" Grid.Row="3"
    Grid.ColumnSpan="2" Text="{loc:Translate Attributes:}"
    loc:Translate.Uid="EditXsdElementDialog.Attributes" />

<!-- ... -->

</Window>
```

Listing 14: Tomers.WPF.Localization example

```
  <!-- Menu items -->
  <Value Id="Menu.File" Header="File" />
  <Value Id="Menu.File.New" Header="New" ToolTip="New" />
  <Value Id="Menu.File.New.Xml" Header="Xml Document" />
  <Value Id="Menu.File.New.Xsd" Header="Xsd Schema" />
  <Value Id="Menu.File.Open" Header="Open" ToolTip="Open" />
  <Value Id="Menu.File.Save" Header="Save" ToolTip="Save" />
```

Listing 15: Localization file example

tributes.

- `XmlAttributeViewModel` - ViewModel for one attribute.
- `XmlElementViewModel` - ViewModel for XML elements.
- `XmlNodeViewModel` - base ViewModel for all XML nodes.
- `XmlTextViewModel` - ViewModel for text nodes.
- `XsdAttributeCollectionViewModel` - ViewModel for the collection of attributes when editing XSD document.
- `XsdAttributeViewModel` - ViewModel for one attribute when editing XSD document.
- `XsdElementViewModel` - special ViewModel for elements when editing XSD document.

- `XmlViewModel` - wrapping ViewModel which contains all data for the GUI when editing generic XML documents.
- `XsdViewModel` - wrapping ViewModel which contains all data for the GUI when editing XSD documents.

The rest of the classes in this assembly are various dialogs, user controls and helpers. All the resources like images, icons, XAML templates etc. are contained within this assembly. The result of compiling this assembly is the executable file which is the entry point of the whole application.

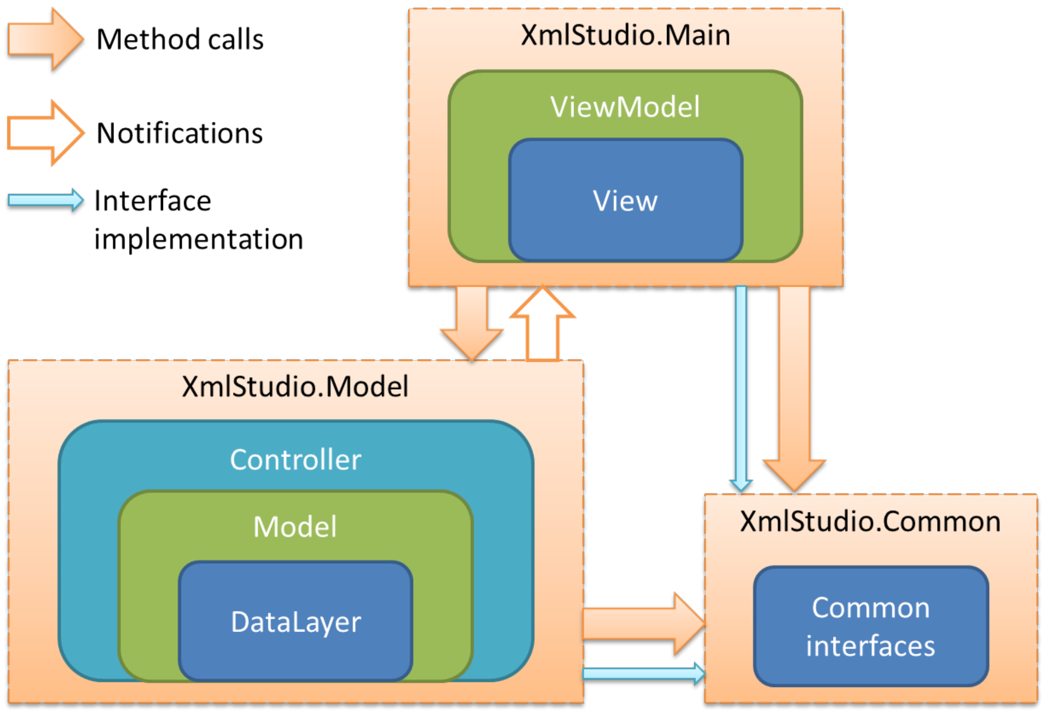


Figure 5.9: Logical view of XmlStudio

5.2.6 Summary

To sum up the architecture of XmlStudio, we have to emphasize the division of the whole application into smaller modules. Figure 5.9 best illustrates this. The core resides in the `XmlStudio.Model` assembly. There are classes which form the data layer used for XML data retrieval. A little higher above this data layer are the model classes which encapsulate the retrieved data and create an object model for the XML data. It also includes methods for manipulation with this object model. This assembly also contains the implementation of the Controller. The `XmlStudio.Common` contains common interfaces and helper classes. The last assembly, `XmlStudio.Main` is the graphical user interface for the application.

Chapter 6

Important Algorithms

6.1 Handling Large Documents

XmlStudio offers a feature to load large XML documents. However, these documents cannot be loaded in memory at once, because it would consume too much memory and would make performance of the application really poor. Therefore we had to come with a solution that enables users to work with large documents but in a way that is bearable both from the side of memory consumption and the side of performance.

The solution implemented in XmlStudio has very simple idea: enable user to see only those parts of the document which he needs to work with. Other parts of the document can be hidden (thus not loaded, not consuming any memory).

XML documents can be regarded as trees [32]. Trees with one vertex designated as the root are called rooted trees. XML documents can be seen as these rooted trees. Let every element be a vertex of the tree. Let the root element of the XML document be the root of the tree. The parent-child relationships form edges in the tree. Because every element can have only one parent, there cannot be any cycles in this graph [33] (every tree is also a graph). Therefore this graph is a tree. The DOM also benefits from this observation.

We model the XML documents exactly that way. Furthermore, we consider text nodes to be vertices of this tree as well. To be able to read only some parts of the document, we need some kind of addressing. XmlStudio uses very simple method where every element (or text node) has its number (actually order number). This order number is relative to its parent. This method can be best expressed with a picture. Figure 6.1 shows these address numbers.

Then if you want to get the address of a certain node, you follow the path from the root element straight to the node which address you want, e.g. the last element `File`, which is the child of the second element `Assembly`, would have the address `[1, 1, 2, 2]`. This address is internally represented as an array of integers. Every node loaded in XmlStudio has this address assigned and is also used to retrieve child elements and later for saving.

The core of retrieving nodes from the underlying file is implemented in class `BigXmlReader` which implements interface `IBigXmlReader` and resides in assembly `XmlStudio.Model`. There are several methods used when expanding nodes. Listing 16 shows the method signatures of the most important ones.

The method `GetRootElement()`, as its name suggests, gets the root element

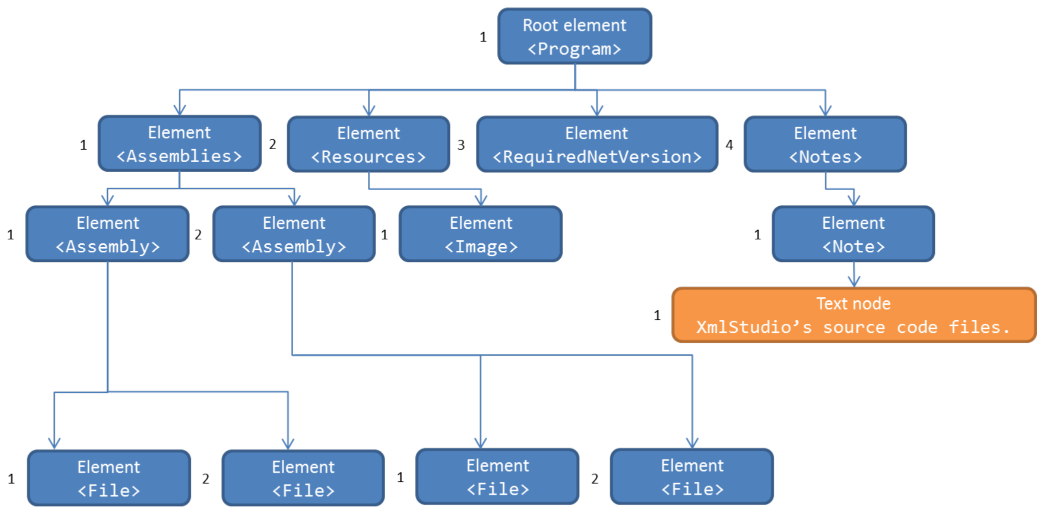


Figure 6.1: Addressing in XmlStudio

```
BigXmlNodeInternal GetRootElement();
int GetChildCount(int[] address);
int GetChildCount(XmlReader reader);
XmlReader NavigateToAddress(int[] address);
IEnumerable<BigXmlNodeInternal> GetChildNodes(int[] address, IEnumerable<int> range,
    int childCount)
IEnumerable<BigXmlNodeInternal> GetChildNodes(int[] address, string range);
```

Listing 16: Method signatures of important methods from `BigXmlReader` class

of the XML document. It uses `XmlReader` from `System.Xml` namespace to navigate through the document and when it is positioned at the root element, it creates object representation of it (wrapped in class `BigXmlNodeInternal` and then counts the number of child elements. This count is done by the second method, the `GetChildCount()`. This method has two overloads, the first one gets an address as an array of integers, the second one gets a reference to opened `XmlReader` instance which is positioned at the element which child count to get. The second one is the actual doer of counting. The first one uses the other method, `NavigateToAddress()`, to get an opened `XmlReader` at the right address and then calls the second overload of the `GetChildCount()`. This second overload of `GetChildCount()` reads the document using the reader passed as a parameter and counts elements and text nodes. When it encounters the end element at the right depth (which equals to the depth when it started counting) in the document, the count is finished. Then it returns the count. The method `NavigateToAddress()` takes an address as a parameter. It reads from the `XmlReader` and tries to get to the right element at a certain depth. For example, consider the document shown in Listing 17.

When we want to navigate to address `[1, 3, 2]`, where the element `note` resides, the `NavigateToAddress()` method needs to do the following steps. Firstly, get to the root element, skip the next two elements (`orderperson` and `shipto`), then continue reading at the third element, which is element `item`. After that, skip the first child element `title` to finally get to the second element, which is the final position. The numbers in the address indicate how many elements it is necessary to skip and then indicates when to dive to the "lower" depth by reading

```xml
<?xml version="1.0" encoding="utf-8"?>
<shiporder>
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

Listing 17: Sample XML document to show how `NavigateToAddress()` works

from the `XmlReader`.

This method of addressing nodes and loading only particular ones is used when user wants to expand a single node. This single node has its address so we can use `BigXmlReader` to read all of its child nodes. This can be repeatedly used for every node in XML document.

The `GetChildNodes()` methods are used to retrieve the collection of nodes from the specified address. The range parameter denotes which child nodes to retrieve. One overload takes also a parameter `childCount`. This is used for performance reasons. When a node is retrieved from the file, `BigXmlReader` also counts the number of its child nodes. This way, when retrieving child nodes of this node, there is no need to count it again.

There is also an option to expand the whole document at once. This is not implemented by consecutively calling `GetChildNodes()` on every address in document. This would require to start reading from the beginning of the document for every single node. When there are few hundreds of nodes, this would cause unnecessary overhead. Therefore this feature has its own method in `BigXmlReader`. It is method called `GetAllFromAddress()`. This method reads the whole subtree of the document starting at the specified address, which is passed in as a parameter. It returns the reference to the root of this loaded subtree. This way, only one pass through the file is necessary for the entire subtree. If starting at the root, the method returns the whole document.

`BigXmlReader` class contains some additional methods, most of them are helper methods. The one worth mentioning is the method `WriteChildNodesFromAddress`. This method is used when saving the document to a file. It takes an address of the node to save and the reference to `XmlWriter` instance as parameters. It navigates to the right address in the original file and writes all descendant nodes from this point on. However, this methods is only a part of the whole saving process, the remaining parts are described in the following section.

6.2 Saving the Document

Saving the partially loaded document can be quite tricky. The situation is not so complicated when the nodes from the original file are not being moved around. It means they have still the same address so it is easy to find them in the original file and write them back as needed. The problem arises when some nodes are moved in the document and their addresses change. This becomes a little more complicated when there are changes in the moved subtree. To solve the situation we came with the method `Save` on `BigXmlNodeInternal` class. This class is an internal representation of a node in the document. This method is recursivelly called on every node in loaded document so the whole document is written to the file.

Every node in XmlStudio has two addresses: `OriginalAddress` and `Address`. The `OriginalAddress` is the address where the node was when it was retrieved from the file. The `Address` is the actual address in the document. If we compare these two addresses, we know if the node has been moved. If it was not expanded, we know where to look for its children which we need when saving the document. If the `OriginalAddress` is `null`, we know that the node has not been loaded from the file but created in XmlStudio.

XmlStudio uses a simple trick to save the file. It writes the current document to a temporary file. The original file is finally replaced with the new one. In case anything happens during the save operation, the original file stays intact.

The actual saving starts at the root of the document and continues in the depth-first search (DFS [34]) manner. The signature of the `Save` method is shown in Listing 18.

```
void Save(XmlWriter writer, IBigXmlReader reader, bool updateAddresses,
    BackgroundWorker report);
```

Listing 18: Method signature of of the `Save` method

The `XmlWriter` is being passed from one recursive call to another which causes that nodes are being written in the correct order.

The important part of the whole saving process is how a single node is being saved. If the node is an element, the start-tag is written to the `XmlWriter`. Then the attributes (if there are any) are written. If the current node is fully expanded, the `Save` is recursivelly called on every child node of this node. Finally, the end-tag is written. If the node is collapsed, it means it was not loaded from the file so now it can be loaded and written to the temporary file. If the node is partially expanded, the expanded nodes get written the normal way and the collapsed node are loaded from the original file. The last case is when the node is a text node. In this case only the value is written. The final step is to update the address of the node.

When the document is saved, another file is created along with it. This file has an extension `.xmlStudio`. This file is an XML document with saved colours and shapes of the rendered items. When a user saves the file, this file is automatically created. This file contains the list of all the nodes that were loaded into XmlStudio. This list contains the address of the node and then its visual properties. These properties are wrapped in `VisualizationInfo` class within

`XmlViewModel` class. This class is serialized into this file. When the main XML or XSD document is being loaded, XmlStudio tries to locate this `.xmlStudio` file and when it succeeds, the values saved in this file are deserialized. XmlStudio keeps track of these values in private property of the `XmlViewModel` class called `LoadedVisualizationInfo`. When the user expands the node, XmlStudio checks this property to see if there are any changes in the appearance. If there are, the default appearance values are replaced with the values loaded from the `.xmlStudio.` file.

6.3 Layout Rendering

XmlStudio uses an open source library called GraphSharp [28] to layout the elements in canvas. This library provides eight different layout algorithms to layout graphs. It includes the Fruchterman-Reingold algorithm [35], Kamada-Kawai algorithm [36], ISOM algorithm [37], Simple Tree layout algorithm, Sugiyama layout algorithm [38] etc.

Simple Tree Layout Algorithm

This simple layout algorithm renders the input tree as a layered tree. The resulting composition consists of a certain number of layers (which equals to the height of the tree - which is the longest path from the root of the tree to the leaf). The resulting layout might look similar to the sample shown in Figure 6.2.

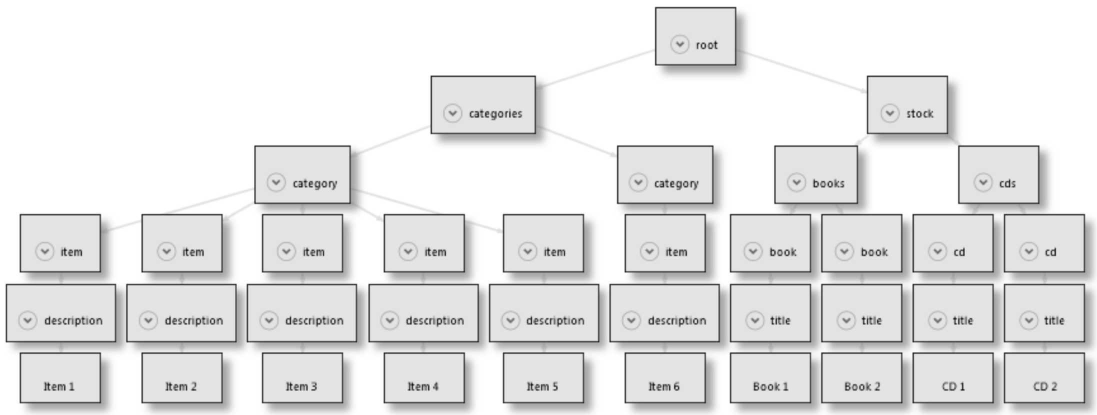


Figure 6.2: Example of a resulting layout of the simple tree algorithm

Let the tree in XmlStudio be a directed acyclic graph [40]. We show this algorithm in the form of C# source code. This code consists of three methods, the first one is the main method that controls the layouting called `InternalCompute`, the other one does the actual layouting (calculation of positions), it is called `CalculatePosition`. The third one is the method `AssignPositions` which adjusts the final positions. There is also a method `GenerateSpanningTree`, but this only creates a spanning tree [39] from the input graph. The input graph in XmlStudio is always a tree, so this method does not change the input graph (but this layout algorithm can be used for general graphs as well). The source code for the `InternalCompute` method is shown in Listing 19 and the source code for the `CalculatePosition` method is shown in Listing 21.

```
void InternalCompute() {
    if(Parameters.Direction == LayoutDirection.LeftToRight
        || Parameters.Direction == LayoutDirection.RightToLeft) {
        //change the sizes
        foreach(var sizePair in sizes.ToArray())
            sizes[sizePair.Key] = new Size(sizePair.Value.Height,
                sizePair.Value.Width);
    }

    if(Parameters.Direction == LayoutDirection.RightToLeft
        || Parameters.Direction == LayoutDirection.BottomToTop)
        direction = -1;
    else
        direction = 1;

    // compute a spanning tree from the graph - graphs in XmlStudio
    // are always trees so the result of this phase is the same graph
    GenerateSpanningTree();

    // first layout the vertices with 0 in-edge degree
    foreach(var source in spanningTree.Vertices
                .Where(v => spanningTree.InDegree(v) == 0))
        CalculatePosition(source, null, 0);

    // then the others
    foreach(var source in spanningTree.Vertices)
        CalculatePosition(source, null, 0);

    // assigns the positions to vertices
    AssignPositions();
}
```

Listing 19: Source code of the `InternalCompute` method

The `InternalCompute` method calls the preparation methods and then the calculation. This algorithm has some parameters (e.g. layout direction, vertex and layer gaps, spanning tree generation algorithm etc.) but we ommit these details for the sake of simplicity.

There are some variables worth mentioning. The first one is the `layers` collection. This collection contains information about layers, specifically its `Size`, collection of vertices, the `NextPosition` and the `LastTranslate`. The `NextPosition` property is the position where the center of the next added vertex would be. The `Size` of the layer is the maximum height of a vertex in this layer plus the layer gap (one of the parameters of the algorithm). The `LastTranslate` value is the latest `translate` value of the node in the current layer. This `translate` value is described in the following paragraph.

The preparation phase of the algorithm is the computation of the spanning tree. Then the core calculation begins. The basic principle is a DFS traversal of the tree. When the `CalculatePosition` reaches the leaf of the tree, the position of this node is returned to its caller. However, this position might later change (during the `AssignPositions` method).

Every node has a helper data stored in the `VertexData` class instance. There are three fields stored in here: the parent node, the amount of translate and finally

```
void AssignPositions() {
    double layerSize = 0;
    bool changeCoordinates = (Parameters.Direction == LayoutDirection.LeftToRight
      || Parameters.Direction == LayoutDirection.RightToLeft);

    foreach(var layer in layers) {
        foreach(var v in layer.Vertices) {
            Size size = sizes[v];
            var d = data[v];
            if(d.parent != null) {
                d.position += data[d.parent].translate;
                d.translate += data[d.parent].translate;
            }
            VertexPositions[v] =
            changeCoordinates
            ? new Point(direction * (layerSize + size.Height / 2.0), d.position)
            : new Point(d.position, direction * (layerSize + size.Height / 2.0));
        }

        layerSize += layer.Size;
    }
    if(direction < 0)
        NormalizePositions();
}
```

Listing 20: Source code of the `AssignPositions` method

the position. The important field is the `translate` value. It indicates the difference between adjacent nodes (parent-child) in the tree (specifically the difference of their positions). When the nodes beying layed out have all the same size, the `translate` value would be always zero. But if the sizes differ, there might occur a situation when a parent node is wider than its child. Because the child gets its position sooner, the parent would not fit above it (because it is wider). Therefore the `translate` value indicates that the child node must be moved to the right (or any appropriate direction according to set options). When the position of a node is computed, the value is returned to the caller and when all its child nodes are positioned, this parent node is placed right in the middle above them. This traversal continues until the root element has all its children positioned. Finally the root itself is positioned above its children.

The `AssignPositions` method does the adjustments of positions according to the `translate` values and according to the algorithm parameters (orientation etc.). Its source code is shown in Listing 20.

```
protected double CalculatePosition(TVertex v, TVertex parent, int l) {
    if(data.ContainsKey(v))
        return -1; //this vertex is already layed out

    while(l >= layers.Count)
        layers.Add(new Layer());

    var layer = layers[l];
    var size = sizes[v];
    var d = new VertexData { parent = parent };
    data[v] = d;

    layer.NextPosition += size.Width / 2.0;
    if(l > 0) {
        layer.NextPosition += layers[l - 1].LastTranslate;
        layers[l - 1].LastTranslate = 0;
    }

    layer.Size = Math.Max(layer.Size, size.Height + Parameters.LayerGap);
    layer.Vertices.Add(v);
    if(spanningTree.OutDegree(v) == 0) {
        d.position = layer.NextPosition;
    } else {
        double minPos = double.MaxValue;
        double maxPos = -double.MaxValue;
        //first put the children
        foreach(var child in spanningTree.OutEdges(v).Select(e => e.Target)) {
            double childPos = CalculatePosition(child, v, l + 1);
            if(childPos >= 0) {
                minPos = Math.Min(minPos, childPos);
                maxPos = Math.Max(maxPos, childPos);
            }
        }

        if(minPos != double.MaxValue)
            d.position = (minPos + maxPos) / 2.0;
        else
            d.position = layer.NextPosition;
        d.translate = Math.Max(layer.NextPosition - d.position, 0);

        layer.LastTranslate = d.translate;
        d.position += d.translate;
        layer.NextPosition = d.position;
    }
    layer.NextPosition += size.Width / 2.0 + Parameters.VertexGap;
    return d.position;
}
```

Listing 21: Source code of the `CalculatePosition` method

Chapter 7

Other Solutions

There are many XML editors available. One of them is an application called Stylus Studio 2010 [41]. This application suite contains many XML related features like XML Editing tools, XQuery tools, XSLT tools, XML Schema tools, DTD tools and many more. We are interested mostly in XML Editing tools and XML Schema tools.

XML Editing Tools offer many features, here is the brief overview of them:

- Multiple synchronized XML editing views: tree view, text view, a robust XML-aware editor
- XML grid view
- Sense:X: Intelligent XML editing feature that provides XML sensing, XML tag completion, syntax coloring
- Integrated XML Schema/DTD validator
- XML differencing
- XML well-formedness checker and XML indenter
- XML canonicalizer
- XML generator
- XML document wizards
- XML code folding
- XML spell checker
- etc.

XSD Tools support XSD editing, conversion, validation, documentation generation etc. Included XSD editor is a visual editor that supports modular XSD component design, refactoring etc.

Another application offering similar functionality is XMLSpy [42]. This application suite is an XML editor, development environment for modeling, editing, transforming and debugging XML-related technologies. Some of the key features are summarized in the following list:

- Intelligent XML editor
- Graphical XML editing and text-based XML editing views
- XML validation with advanced error handling
- XPath auto-completion and XPath Analyzer window
- Visual XML Schema editor
- Database integration
- Very large file support
- and many more...

There are another XML editors which offer similar functionality like XML Pro [43], EditiX [44] or Oxygen XML Editor [45]. Table in Figure 7.1 summarizes the main features of mentioned XML editors and compares it with XmlStudio:

Feature	**Stylus Studio**	**XmlSpy**	**EditX**	**Oxygen XML Editor**	**Xml-Studio**
Textual XML Editing	yes	yes	yes	yes	no
Visual XML Editing	grid view	grid view	form view	tree view	planar graph view
XML Schema Validator	yes	yes	yes	yes	yes
DTD Validator	yes	yes	yes	yes	yes
XPath Evaluator	yes	yes	yes	yes	yes
Undo & Redo	yes	yes	yes	yes	partial
Textual XSD Editing	yes	yes	yes	yes	no
Visual XSD Editing	yes	yes	yes	yes	yes
Large file support	partial	yes	no	partial	yes
XML Visualization Exporting	no	no	no	no	yes

Figure 7.1: Feature comparison

XmlStudio cannot compete with commercial solutions because these solutions are developed by development teams, not by a single person. But XmlStudio shows one possibility how the XML editing could be done in a little different way and offers simple editing tools which should be enough for simple scenarios.

Chapter 8

Summary and Conclusion

The aim of this thesis was to create a graphical XML and XSD editor. Created application XmlStudio offers tools to create and edit generic XML documents. These documents are shown as visualized trees. This visualization can be modified, but not only the document itself but also the colours, shapes or positions of any elements. XML documents can be checked against DTD or XML schemas. User can also query loaded parts of the document using XPath language. The visualized documents can be exported to various image formats including JPEG, PNG and BMP.

XML schemas can also be edited using this application. Using context help on XSD elements, the user does not need to remember the names of the elements available, XmlStudio remembers them instead. The same applies for the attributes and also for some values (like built-in or user defined types, groups, etc.).

Both XML and XSD documents are being saved with the visualization, so when the user loads the document, all the colours and shapes are restored.

XmlStudio is capable of opening and editing huge XML files using the simple expansion method described in this thesis.

However, XmlStudio cannot and does not try to compete with professional solutions which are on the market nowadays. Those solutions are being developed by the whole teams of professional developers, architects and other people involved in the application development. XmlStudio tries to show how XML editing can be done in a different way and can be used for example for creating visualizations of simple XML documents or editing XML schemas without writing all the markup.

Of course, there are many ways how XmlStudio could be enhanced. The following list shows some ideas which could make XmlStudio a better application:

- Allow simultaneous text and visual editing of XML and XSD documents.
- Add syntax highlighting to the source code view.
- Improve user-friendliness of element editing (allow also direct element editing on canvas).
- Allow users to better customize the appearance of visualized elements.
- Optimize performance of the application when there are many (thousands) of elements on canvas.

- Add more bulk editing operations.
- Add DTD editor.
- Allow inserting elements via dragging them on canvas.
- Add support for XSLT.
- Improve contextual help for XML schemas.

Bibliography

[1] World Wide Web Consortium, http://w3c.org/.

[2] Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C, November 2008, http://www.w3.org/TR/REC-xml/#sec-well-formed

[3] Planar graph, Wikipedia, http://en.wikipedia.org/wiki/Planar_graph

[4] Namespaces in XML 1.1 (Second Edition), W3C, August 2006, http://www.w3.org/TR/xml-names11/

[5] Uniform Resource Identifier, Wikipedia, http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

[6] Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C, November 2008, http://www.w3.org/TR/REC-xml/#dt-doctype

[7] XML Schema Part 0: Primer Second Edition, W3C, October 2004, http://www.w3.org/TR/xmlschema-0/

[8] Biron, P., Malhotra, A.: XML Schema Part 2: Datatypes - Built-in Data Types, W2C, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#built-in-datatypes

[9] Biron, P., Malhotra, A.: XML Schema Part 2: Datatypes - Simple Type Definition, W2C, May 2001, http://www.w3.org/TR/xmlschema-2/#rf-defn

[10] RELAX NG Specification, ISO/IEC 19757-2:2008, December 2001, http://www.relaxng.org/spec-20011203.html

[11] Schematron Specification, ISO/IEC 19757-3:2006, 2006, http://www.schematron.com/spec.html

[12] Document Object Model, W3C, January 2005, http://www.w3.org/DOM/

[13] Simple API for XML, November 2001, http://www.saxproject.org/

[14] Microsoft .NET Framework, Wikipedia, http://en.wikipedia.org/wiki/.NET_Framework

[15] XmlDocument Class, MSDN Library, http://msdn.microsoft.com/en-us/library/system.xml.xmldocument.aspx

[16] Streaming API for XML, Wikipedia, http://en.wikipedia.org/wiki/StAX

[17] LINQ to XML, MSDN Library, http://msdn.microsoft.com/en-us/library/bb387098.aspx

[18] Box, D., Hejlsberg A.: .NET Language-Integrated Query, MSDN Library, February 2007, http://msdn.microsoft.com/en-us/library/bb308959.aspx

[19] Clark, J., DeRose, S.: XML Path Language, W3C, November 1999, http://www.w3.org/TR/xpath/

[20] Shaham, T.: WPF Localization - On-the-fly Language Selection, October 2007, http://blogs.microsoft.co.il/blogs/tomershamam/archive/2007/10/30/wpf-localization-on-the-fly-language-selection.aspx

[21] Ossenkov, K.: Undo Framework, July 2009, http://undo.codeplex.com/

[22] XAML, MSDN Library, http://msdn.microsoft.com/en-us/library/ms752059.aspx

[23] Model-View-Controller design pattern, Wikipedia, http://en.wikipedia.org/wiki/Model-view-controller

[24] Model-View-ViewModel design pattern, Wikipedia, http://en.wikipedia.org/wiki/Model_View_ViewModel

[25] Strategy Pattern, Wikipedia, June 2010, http://en.wikipedia.org/wiki/Strategy_pattern

[26] Windows Presentation Foundation, MSDN Library, http://msdn.microsoft.com/en-us/library/ms754130.aspx

[27] XAML Overview (WPF), MSDN Library, http://msdn.microsoft.com/en-us/library/ms752059.aspx

[28] GraphSharp - graph layout framework, June 2009, http://graphsharp.codeplex.com/

[29] Fluent Ribbon Control Suite, May 2010, http://fluent.codeplex.com/

[30] Data Transfer Object, Wikipedia, http://en.wikipedia.org/wiki/Data_transfer_object

[31] Ribbon (computing), Wikipedia, http://en.wikipedia.org/wiki/Ribbon_(computing)

[32] Tree (graph theory), Wikipedia, http://en.wikipedia.org/wiki/Tree_(graph_theory)

[33] Graph (mathematics), Wikipedia, http://en.wikipedia.org/wiki/Graph_(mathematics)

[34] Depth-first search, Wikipedia, http://en.wikipedia.org/wiki/Depth-first_search

[35] Fruchterman-Reingold Algorithm, Network Workbench, https://nwb.slis.indiana.edu/community/?n=VisualizeData.Fruchterman-Rheingold

[36] Kamada-Kawai Algorithm, Network Workbench, https://nwb.slis.indiana.edu/community/?n=VisualizeData.Kamada-Kawaii

[37] Meyer, B.: Self-Organizing Graphs - A Neural Network Perspective of Graph Layout, January 1998, page 253, http://www.springerlink.com/content/lau0ugxt04fp1e8b

[38] Eiglsperger, M., Siebenhaller, M., and Kaufmann, M.: An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing, J. Graph Algorithms Appl., 9(3), 305-325, 2005.

[39] Spanning tree, Wikipedia, http://en.wikipedia.org/wiki/Spanning_tree

[40] Directed Graph, Wikipedia, http://en.wikipedia.org/wiki/Directed_graph

[41] Stylus Studio - XML Editor, XML Data Integration, XML Tools, Web Services and XQuery, http://www.stylusstudio.com/

[42] XMLSpy - XML Editor for Modeling, Editing, Transforming, & Debugging XML Technologies, http://www.altova.com/xmlspy.html

[43] Vervet Logic XMLPro - XML Editor, http://www.vervet.com/products.php

[44] EditiX XML Editor, http://www.editix.com/

[45] Oxygen XML Editor, http://www.oxygenxml.com/

Chapter 9

Appendix A

This thesis contains an attached CD with an installer of XmlStudio, source code of XmlStudio, HTML documentation generated from the source code and an electronic version of this document.

The disc contains the following directories:

- `SourceCode` – contains the complete source code in the form of solution in Visual Studio 2010
- `Documentation` – contains HTML documentation generated from comments in the source code
- `Installer` – contains the installer of XmlStudio with all required dependencies
- `Thesis` – contains this text in PDF format