VISUAL REPRESENTAION FOR AN XML BASED PROGRAMMING LANGUAGE

A Junior Project Presented to the Faculty of Computer and Informatics Engineering

In Partial Fulfillment of the Requirements for the Degree

Of Bachelor of Engineering in Software Engineering

by

Antwan Al-Haddad

Fadi Ghattas

Mohammad Tahawi

Under the supervision of

Dr. Ghassan El-Nemr

Contents

ABSTRACT

This project suggests a new programming language (GEN) represented with XML tags and a corresponding IDE based on the concept of visual programming,

Chapter one gives a brief history about programming languages and ends up with discussing the needed programming construct for any programming language.

Chapter two introduces XML, its importance, flexibility and the associated technologies with XML, and ends up with suggesting an XML structure for representing the suggested programming language.

Chapter three discusses the visual programming, its history, advantages and disadvantages and why it is a suitable solution for the suggested programming language.

Chapter four talks about IDE's, their role in the software development, their types, designs, advantages, the facilities they provide and the difficulties they face.

Chapter five introduces the IDE developed specially for GEN and gives a brief preview of the implementation and the technology used for it.

Chapter 1
a brief history of programming language

Computers are a big part of our modern world. We can the influence of this invention in all aspects of our daily life. But computers are just machines that understand very limited set of instructions, and without the proper mean of communication they are useless.

This is the motivation for a having a programming language, a way to tell computers what to do and how to do it, without a programming language this great invention would be just a worthless box.

1. Definition of a Programming language

A programming language is a formal language (a set of strings of symbols that may be constrained by rules that are specific to it) designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely. [1]

The earliest programming languages preceded the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, and still many are being created every year. Many programming languages require computation to be specified in an imperative form (i.e., as a sequence of operations to perform), while other languages utilize other forms of program specification such as the declarative form (i.e., the desired result is specified, not how to achieve it). [2]

The description of a programming language is usually split into the two components of syntax (form) and semantics (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard), while other languages, such as Perl 5 and earlier, have a dominant implementation that is used as a reference. [1]

2. A Brief History of Programming Languages

2.1. Machine Languages (The First Generation)

It is the first languages developed.

Even today, still the only language "understood" by computers.

Instructions (aka: "statements") in a machine language use only 0's and 1's.

(Naturally, this makes machine languages difficult for humans, although not for computers). [3]

Because the instructions are based upon the architecture of the computer, each model computer has its own, unique machine language.

(This means that machine language programs are not portable among different kinds of computers). [3]

Because the operations that a computer can actually do are very limited, and because each machine language instruction specifies exactly one machine-level operation, even very simple programs are quite lengthy. [4]

Disadvantages of machine languages:

1. Difficult for humans
2. Programs are not portable
3. Programs are quite lengthy

Typical machine language code:

0101 0011 1110 1000

0110 0011 1110 1010

0111 0011 1110 1100

The first part of each instruction specifies the operation to be done, and the second part specifies the memory location, or *address*, to be used [4] [3]

2.2.Assembly Languages "Low-Level Languages" (The Second Generation)

Assembly languages are considered low-level because they are very close to machine languages. Language for computers, microprocessors, microcontrollers, and other integrated circuits. [3]

It implements a symbolic representation of the binary machine codes and other constants needed to program a given CPU architecture ("Names instead of numbers are used to specify the operations to be done, and the memory locations involved"). [3]

This representation is usually defined by the hardware manufacturer, and is based on mnemonics that symbolize processing steps (instructions), processor registers, memory locations, and other language features typical equivalent assembly language code:

```
LOAD BASEPAY
ADD OVERPAY
STORE GROSSPAY
```

An assembly language is thus specific to a certain physical (or virtual) computer architecture. This is in contrast to most high-level programming languages, which, ideally, are **portable**. [5]

In software engineering, **porting** is the process of adapting software so that an executable program can be created for a computing environment that is different from the one for which it was originally designed (e.g. different CPU, operating system, or third party library). The term is also

used when software/hardware is changed to make them usable in different environments. [5]

Software is **portable** when the cost of porting it to a new platform is less than the cost of writing it from scratch. The lower the cost of porting software, relative to its implementation cost, the more portable it is said to be. [5]

And each assembly language instruction is translated into exactly one machine language instruction, assembly language programs also tend to be quite lengthy. [5]

A program called an **assembler** is used to translate assembly language statements into the target computer's machine code. The operation name (e.g., LOAD) is translated into the numeric code for the operation (e.g., 0101) and the *variable* name (e.g., BASEPAY) is associated with a particular numeric address in memory (e.g., 0011 1110 1000). [6]

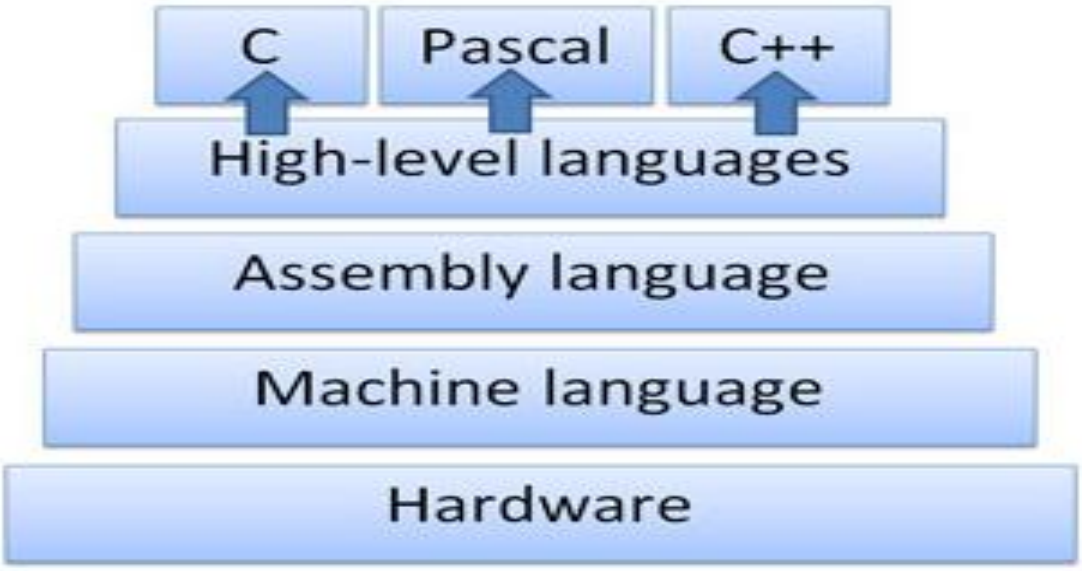
The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machine instructions and data.

This is in contrast with high-level languages, in which a single statement generally results in many machine instructions. [6]

Many sophisticated assemblers offer additional mechanisms to facilitate program development, control the assembly process, and aid debugging. [6]

2.3.Scripting Language "High-Level Languages" (The Third Generation)

Examples: FORTRAN, BASIC, COBOL, Pascal, Ada, C, C++, C#, Java



Use English words and familiar symbols to specify operations, so easiest for humans to use. [3]

"High-level language" refers to the higher level of abstraction from machine language. Rather than dealing with registers, memory addresses and call stacks, high-level languages deal with variables, arrays, objects, complex arithmetic or Boolean expressions, subroutines and functions, loops, threads, locks, and other abstract computer science concepts, with a focus on usability over optimal program efficiency. Unlike low-level assembly languages, high-level languages have few, if any, language elements that translate directly into a machine's native opcodes. Other features, such as string handling routines, object-oriented language features, and file input/output, may also be present. [3]

While high-level languages are intended to make complex programming simpler, low-level languages often produce more efficient code. [3]

Abstraction penalty is the border that prevents high-level programming techniques from being applied in situations where computational resources

are limited. High-level programming exhibits features like more generic data structures, run-time interpretation, and intermediate code files. [3]

Which often result in slower execution speed, higher memory consumption, and larger binary program size. [3]

For this reason, code which needs to run particularly quickly and efficiently may require the use of a lower-level language, even if a higher-level language would make the coding easier. [3]

In many cases, critical portions of a program mostly in a high-level language can be hand-coded in assembly language, leading to a much faster or more efficient optimized program. [3]

However, with the growing complexity of modern microprocessor architectures, well-designed compilers for high-level languages frequently produce code comparable in efficiency to what most low-level programmers can produce by hand, and the higher abstraction may allow for more powerful techniques providing better overall results than their low-level counterparts in particular settings. [3]

May be run on any computer that has a translation program -- a compiler (or, for some languages, an interpreter) -- that translates a program in a particular high-level language into the machine language of a particular computer.
For example, a C++ compiler on a Mac will translate C++ programs into the machine language of the Mac, while a C++ compiler on a PC will translate those same programs into the machine language of the PC. So high-level language programs are portable! [3]

Further, since high-level languages are not based upon a particular machine language, a single high-level instruction may be translated into several

machine language instructions.

So high-level language programs tend to be shorter than machine-language and assembly-language programs! [3]

2.4.General execution models for modern high-level languages

2.4.1. Interpreted languages: read and then executed directly, with no compilation stage. A program called an interpreter reads each program statement following the program flow, decides what to do, and does it. A hybrid of an interpreter and a compiler will compile the statement into machine code and execute that; the machine code is then discarded, to be interpreted anew if the line is executed again. Interpreters are commonly the simplest implementations, compared to the other two variants listed here. [7]

2.4.2. Compiled languages are transformed into an executable form before running. There are two types of compilation; the first one is *Machine code generation* where compilers compile source code directly into machine code. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages. See assembly language. [7]

the other compilation type is *Intermediate representation* when a language is compiled to an intermediate representation, that representation can be optimized or saved for later execution without the need to re-read the source file. When the intermediate representation is saved, it is often represented as byte code. The intermediate representation must then be interpreted or further compiled to execute it. Machines that execute byte code directly or transform it further into

machine code have blurred the once clear distinction between intermediate representations and truly compiled languages. [7]

2.4.3. Translated: A language may be translated into a lower-level programming language for which native code compilers are already widely available. The C programming language is a common target for such translators. See Chicken Scheme. [7]

But Note that languages are not strictly "interpreted" languages or "compiled" languages. Rather, language implementations use interpretation or compilation. For example, Algol 60 and Fortran have both been interpreted (even though they were more typically compiled). Similarly, Java shows the difficulty of trying to apply these labels to languages, rather than to implementations; Java is compiled to byte code and the byte code is subsequently executed by either interpretation (in a JVM) or compilation (typically with a just-in-time compiler such as HotSpot, again in a JVM). [7]

2.5.Auto Generated Programming Language (The Fourth Generation)

A fourth generation (programming) language (4GL) is a grouping of programming languages that attempt to get closer than 3GLs to human language, form of thinking and conceptualization. [7]

4GLs are designed to reduce the overall time, effort and cost of software development. The main domains and families of 4GLs are: database queries, report generators, data manipulation, analysis and reporting, screen painters and generators, GUI creators, mathematical optimization, web development and general purpose languages.

Also known as a 4th generation language, a domain specific language, or a high productivity language. [7]

2.5.1. Explain Fourth Generation (Programming) Language (4GL):

The natural-language, block-structured mode of the third-generation programming languages improved the process of software development. However, 3GL development methods can be slow and error-prone. It became clear that some applications could be developed more rapidly by adding a higher-level programming language and methodology which would generate the equivalent of very complicated 3GL instructions with fewer errors. In some senses, software engineering arose to handle 3GL development. 4GL and 5GL projects are more oriented toward problem solving and systems engineering [7]

4GLs are more programmer-friendly and enhance programming efficiency with usage of English-like words and phrases, and when appropriate, the use of icons, graphical interfaces and symbolical representations. The key to the realization of efficiency with 4GLs lies in an appropriate match between the tool and the application domain. Additionally, 4GLs have widened the population of professionals able to engage in software development. [7]

Many 4GLs are associated with databases and data processing, allowing the efficient development of business-oriented systems with languages that closely match the way domain experts formulate business rules and processing sequences. Many of such data-oriented 4GLs are based on the Structured Query Language (SQL), invented by IBM and subsequently adopted by ANSI and ISO as the standard language for managing structured data.

Most 4GLs contain the ability to add 3GL-level code to introduce specific

system logic into the 4GL program.

The most ambitious 4GLs, also denoted as Fourth Generation Environments, attempt to produce entire systems from a design made in CASE tools and the additional specification of data structures, screens, reports and some specific logic. [7]

2.5.2. Different types of 4GLs exist:

2.5.2.1. **Table-driven (codeless) programming:** Usually running with a runtime framework and libraries. Instead of using code, the developer defines his logic by selecting an operation in a pre-defined list of memory or data table manipulation commands. In other words, instead of coding, the developer uses Table-driven [8] algorithm programming (See also control tables that can be used for this purpose). A good example of this type of 4GL language is PowerBuilder. These types of tools can be used for business application development usually consisting in a package allowing for both business data manipulation and reporting, therefore they come with GUI screens and report editors. They usually offer integration with lower level DLLs generated from a typical 3GL for when the need arise for more hardware/OS specific operations. [8]

2.5.2.2. **Report-generator programming languages:** Take a description of the data format and the report to generate and from that they either generate the required report directly or they generate a program to generate the report. See also RPG [8]

2.5.2.3. **Forms generators:** Manage online interactions with the application system users or generate programs to do so. [8] [9]

More ambitious 4GLs (sometimes termed fourth generation environments)

attempt to automatically generate whole systems from the outputs of 'CASE' tools, specifications of screens and reports, and possibly also the specification of some additional processing logic. [8] [9]

2.5.2.4. **Data management 4GLs:** Such as SAS, SPSS and Stata provide sophisticated coding commands for data manipulation, file reshaping, case selection and data documentation in the preparation of data for statistical analysis and reporting. [8] [9]

Some 4Gl languages: [10]

- **Creators & General use / versatile:** Omnis Studio GUI, Genero Studio
- **General use / versatile:** XBase++, PLL (Paul Lozano Language), GeneXus (Knowledge-based Multi-Platform Development Tool)
- **Database query languages:** SQL, Genero BDL
- **Data manipulation, analysis, and reporting languages:** Genero
- **Mathematical optimization:** AIMMS
- **GUI application development:** Genexus Database-driven
- **Screen painters and generators:** Oracle Forms, Genero Screen Designer
- **Web development languages:** ActiveVFP, Genero Web Client

2.6.Artificial Intelligence Programming Language (The Fifth Generation)

A fifth generation (programming) language (5GL) is a grouping of programming languages build on the premise that a problem can be solved, and an application built to solve it, by providing constraints to the program (constraint-based programming), rather than specifying algorithmically how the problem is to be solved (imperative programming). [11]

In essence, the programming language is used to denote the properties, or logic, of a solution, rather than how it is reached. Most constraint-based and logic programming languages are 5GLs. A common misconception about 5GLs pertains to the practice of some 4GL vendors to denote their products

as 5GLs, when in essence the products are evolved and enhanced 4GL tools. Also known as a 5th generation language. [11]

2.6.1. Explains Fifth Generation (Programming) Language (5GL):

The leap beyond 4GLs is sought by taking a different approach to the computational challenge of solving problems. When the programmer dictates how the solution should look, by specifying conditions and constraints in a logical manner, the computer is then free to search for a suitable solution. Most of the applicable problems solved by this approach can currently be found in the domain of artificial intelligence. [11] [12]

Fifth-generation languages are designed to make the computer solve a given problem without the programmer. This way, the programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them. Fifth-generation languages are used mainly in artificial intelligence research. [12]

Prolog, OPS5, and Mercury are examples of fifth-generation languages.

PROLOG (acronym for PROgramming LOGic) is an example of a Logical Programming Language. It uses a form of mathematical logic (predicate calculus) to solve queries on a programmer-given database of facts and rules. [12]

Considerable research has been invested in the 1980s and 1990s, into the development of 5GLs. As larger programs were built, it became apparent that the approach of finding an algorithm given a problem description, logical instructions and a set of constraint is a very hard problem in itself. During the 1990s, the wave of hype that preceded the popularization of

5GLs and predictions that they will replace most other programming languages, gave way to a more sober realization. [12]

Simple table show all generation of Programming languages differences, [2]**:**

Low-Level Languages		High-Level Languages		
First generation languages	Second generation languages	Third generation languages	Fourth generation languages	Fifth generation languages
machine dependent		machine independent		
can control the hardware precisely		cannot control the hardware precisely		
non English-like		English-like		
procedural languages		declarative languages		
can directly run by a computer		must be translated into machine codes before it can be executed by a computer		

2.7.Programming Language Theory.

Programming language theory (**PLT**) is a branch of computer science that deals with the design, implementation, analysis, characterization, and classification of programming languages and their individual features. It falls within the discipline of computer science, both depending on and affecting mathematics, software engineering and linguistics. [13]

The design of a programming language relay on a theoretical model (or models), these models differs according to the programming paradigm the language is adopting.

As an example, the functional programming paradigm relays heavily on lambda calculus and Combinatory logic, while imperative languages are based on Turing machine, more recent languages are using multiple paradigms, and so they rely on more than one model.

As a result of that, different programming languages are just different *Syntactical* representation for similar *Symantec*.

Most programming languages have, among others, five constructs: assignment, variable declaration, sequence, test, and loop. These constructs form the imperative core of the language. [14]

2.7.1. Assignment and variables

The imperative paradigm reflects the underlying hardware model used in machines these days, that is the Von Neumann Architecture, in which we have a processing unit, a memory unit and input /output peripherals. The assignment construct is a statement that binds a *variable* to a *value*, variables are an abstraction of *memory cells* where the variable *na*me is bound to the *address* of the memory cell and the *value* is represented by the *bit pattern* in the cell.

2.7.2. Variable Declaration

Before being able to assign values to a variable x, it must be declared, which associates the name x to a location in the computer's memory. Variable declaration is a construct that allows the creation of a variable and adding it to the current environment, to be looked up and used in subsequent statements and expressions as long as the variable is in scope.

In Caml, variable declaration is written as let x = ref t in p and it isn't necessary to explicitly declare the variable's type. It is not possible in Caml to declare a variable without giving it an initial value. In C, like in Java, declaration is written {T x = t; p}. It is possible to declare a variable without giving it an initial value, and in this case, it could have any value.

2.7.3. Test A test is a construct that allows the creation of a statement composed of a boolean expression b and two statements p1 and p2. In Java, this statement is written

if (b) p1 else p2. To execute the statement if (b) p1 else p2 in a state s, the value of expression b is first computed in the state s, and depending on whether or not its value is true or false, the statement p1 or p2 is executed in the state s.

In Caml, this statement is written if b then p1 else p2.

In C, it is written as it is in Java.

2.7.4. Loop A loop is a construct that allows the creation of a statement composed of a boolean expression b and a statement p. In Java, this statement is written while (b) p. To execute the statement while (b) p in the state s, the value of b is first computed in the state s. If this value is false, execution of this statement is terminated. If the value is true, the statement p is executed, and the value of b is recomputed in the new state. If this value is false, execution of this statement is terminated. If the value is true, the statement p is executed, and the value of b is recomputed in the new state... This process continues until b evaluates to false.

This construct introduces a new possible behavior: non-termination. Indeed, if the boolean value b always evaluates to true, the statement p will continue to be executed forever, and the statement while (b) p will never terminate. This is the case with the instruction

```
int x = 1;
while (x >= 0) {x = 3;}
```

To understand what is happening, imagine a fictional statement called skip; that performs no action when executed. You can then define the statement

while (b) p as shorthand for the statement

if (b) {p if (b) {p if (b) {p if (b) ...

else skip;}

else skip;}

else skip;}

else skip;

So a loop is one of the ways in which you can express an infinite object using a finite expression. And the fact that a loop may fail to terminate is a consequence of the fact that it is an infinite object.

In Caml, this statement is written while b do p.

In C, it is written as it is in Java.

2.7.5. Sequence A sequence is a construct that allows a single statement to be created out of two statements p1 and p2. In Java, a sequence is written as {p1 p2}. The statement {p1 {p2 { ... pn} ...}} can also be written as {p1 p2 ... pn}. To execute the statement {p1 p2} in the state S, the statement p1 is first executed in the state S, which produces a new state S`. Then the statement p2 is executed in the state S`. In Caml, a sequence is written as p1; p2. In C, it is written the same as it is in Java.

2.7.6. Input and Output An input construct allows a language to read values from a keyboard and other input devices, such as a mouse, disk, a network interface card, etc. An output construct allows values to be displayed on a screen and outputted to other peripherals, such as a printer, disk, a network interface card, etc.

Those five constructs plus the input/output construct are enough for any programming language to simulate Turing machine and so solve any problem that is computationally solvable.

The problem developers face is the variety of the different *syntax* programming language called to represent the same *Symantec*.

Defining a universal representation for those constructs will result in a universal cross platform easy to use programming language.

The next chapter gives an introduction to XML and the flexibility it provides, and shows the advantages of using XML for representing a programming language.

Chapter 2
Flexibility in programing Languages (XML)

After the briefly discussion about programing languages in the last chapter it's became obvious that the majority of programing languages are sharing the same Symantec with difference syntax and since the code is data it can be easy to serialize that code into portable source code representation to reach the desired universal, portable programing language.

This need promote us to provide a brief introduction about the cross platform that offer all the technique needed which is (XML language).

1. Extensible Markup Language (XML)

XML is the acronym for Extensible Markup Language and has been developed by W3C (the World Wide Web Consortium), XML is an ideal format for storing structured data intended for publishing or exchange between different applications, XML derives from SGML (Standard Generalized Markup Language) [15].

What makes XML flexible is that it allows the end user to specify custom tags and to associate semantic information to source code text [16].

An XML document has a logical and a physical structure. The logical structure allows the document to be divided into units called elements. These elements can contain other elements in turn thus allowing for a complex logical structure to be defined [17].

For example, to describe a book we need a book element, a title element and an author element. Also, this book will have a unique ISBN number that could be stored as an attribute to the book element. Here is how this would be expressed in XML:

```
<book ISBN="123456789">
<title>The story of my life</title>
<author>George Thomas</author>
<author>Tom Jhones</author>
</book>
```

2. Schemas and validation

2.1. Document Type Definition (DTD)

Document Type Definition, or DTD in short, is a technology directly related to XML documents. It provides a way of defining the logical structure of the XML document. The logical structure contains all the elements that can be used and describes how they can be used in relation to each other. This results in a document hierarchy. Without a DTD, an XML document can only be checked to determine if it is well formed. This means that every start tag is followed by a corresponding end tag [18]. The use of a DTD allows us to check for validity of the XML document. Everything that is in the XML document must conform to the DTD specification. Therefore we are able to enforce certain restrictions on how the XML document can be composed and this makes it easy to create applications that process these XML documents. The DTD contains a number of declarations [19]. Each declaration can be one of the following declaration types:

- **ELEMENT**

  Elements are the basic contents of an XML file and correspond to user defined tags (i.e. book, author) as illustrated in the example above. We can have empty elements, or elements that contain other elements or text.

- **ATTLIST**

  A list of attributes associated with a particular element can be declared using ATTLIST. Every attribute has a name, a type and a default value.

- **ENTITY**

  Entities are used to avoid repetition in XML documents.
  They are declared once and can be referred to many times.
  Both internal and external entities are allowed in DTDs.
  Internal entities are defined within the current DTD, while external entities reside in a separate DTD.

- **NOTATION**

  Notations are used to refer to data that is not in XML format.
  Notations can also be linked with entities by using the NDATA keyword.
  The following is a sample DTD that can be used to enforce the logical structure of the example presented previously in the XML section. By examining the DTD we see that a book must have one title and at least one author. Also a book has a unique ISBN number as an attribute. Both the title and author elements contain character data that stores the information.

```
<!ELEMENT book (title,author+)>
<!ATTLIST book ISBN CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

2.2. XML Schema (XSD)

A newer schema language, described by the W3C as the successor of DTDs, is XML Schema, often referred to by the initialism for XML Schema instances, XSD (XML Schema Definition). XSDs are far more powerful than DTDs in describing XML languages. They use a rich datatyping system and allow for more detailed constraints on an XML document's logical structure. XSDs also use an XML-based format, which makes it possible to use ordinary XML tools to help process them [19].

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="book">
<xs:complexType>
<xs:sequence>
<xs:element ref="title"/>
<xs:element maxOccurs="unbounded" ref="author"/>
</xs:sequence>
<xs:attribute name="ISBN" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="title" type="xs:string"/>
<xs:element name="author" type="xs:string"/>
</xs:schema>
```

3. Working with XML documents (XML Parser)

The World Wide Web Consortium has defined a standard interface for accessing XML files called Document Object Model (DOM).

Another interface called Simple API for XML (SAX) has been developed by members of the XML-DEV. And a newer technology than the others we discussed was developed (STAX).

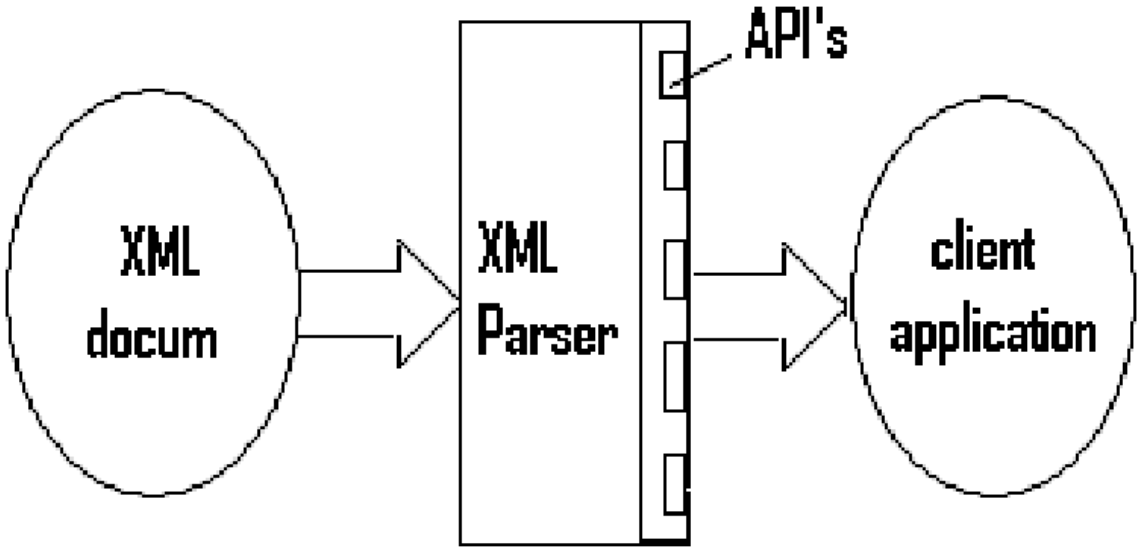


Figure 1

3.1.Document Object Model (DOM)

DOM is a language neutral interface for allowing programs to access and update the structure and style of documents. DOM is a tree-based API that generates an internal tree structure of the document and allows an application to navigate and manipulate the tree. Every document is composed of Nodes and a variety of node types are defined in the DOM Core specification [20]. The DOM tree for the XML example presented previously is shown in Figure 2. Methods for manipulating the tree or its components are provided by the specific parser implementation. The advantage of this interface is that the complete document exists in memory and it can be easily processed and manipulated. The disadvantage is that working with large documents imposes a large memory requirement.

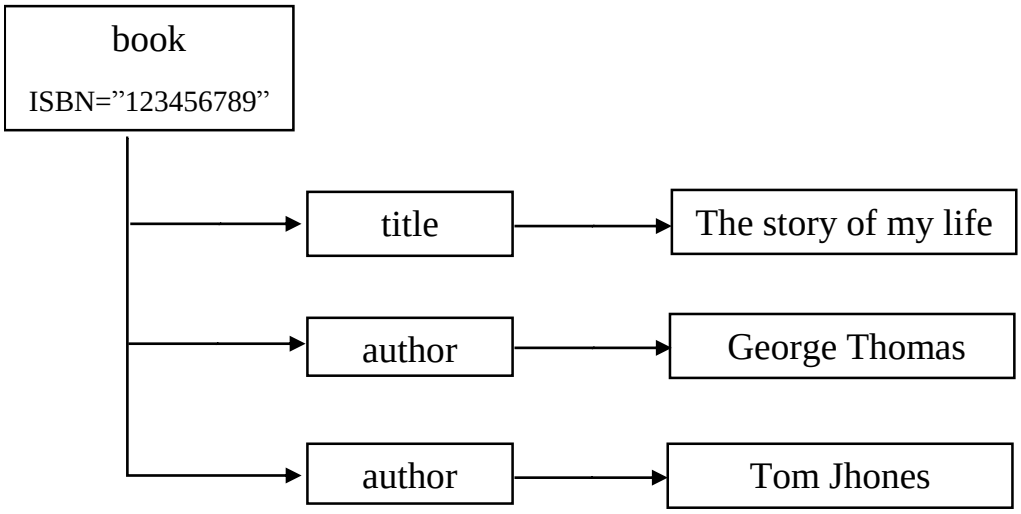


Figure 2

3.2.Simple API for XML (SAX)

SAX is an event-based API that reports event as it is parsing the XML document. An application that handles the events can be build to perform various tasks. The key difference when compared to DOM is that SAX does not built a tree representation of the XML file in memory [18]. Therefore, SAX is more convenient for tasks that do not require the complete tree to be present in memory. A typical scenario in which SAX would be a better choice occurs when for example, we need to search a 10MB file to count the number of “Author” elements. The system requirements by SAX are minimal when compared to DOM [21].

3.3.Stax

StAX is a newer technology then the others we discussed and it is the only one with a JSR (JSR-173).

Parsing with StAX look like parsing with SAX. Again StAX does not store anything to memory and the document is read from beginning to end once.

However in SAX, your event handler is called by SAX when an event occurs. In StAX, you ask StAX to continue to next event [22].

3.4.Comparing the Parsing Approaches

Each of the three approaches discussed offers advantages and disadvantages and is appropriate for particular types of applications [22]. Table 1 compares the three parsing approaches.

Parsing Approach	Advantages	Disadvantages	Suitable Application
DOM	Ease of use, navigation, random access, and XPath support	Must parse entire document, memory intensive	Applications that modify structure and content of an XML document, such as visual XML editors
SAX	Low memory consumption, efficient	No navigation, no random access, no modification	Read-only XML applications, such as document validation
STAX	Ease of use, low memory consumption, application regulates parsing, filtering	No random access, no modification	Data binding, SOAP message processing

Table 1. DOM, SAX, and STAX Comparison

4. Using XML Towards Potable Source Code

After reviewing XML language and related Document Type Definition language and the existing interfaces for working with XML documents. And the benefits which we get from the flexibility in the XML.

It’s the time use all of that towards the desired goal.

4.1. Serialize objects as XML (serialization)

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization. Figure 3 shows the overall process of serialization.

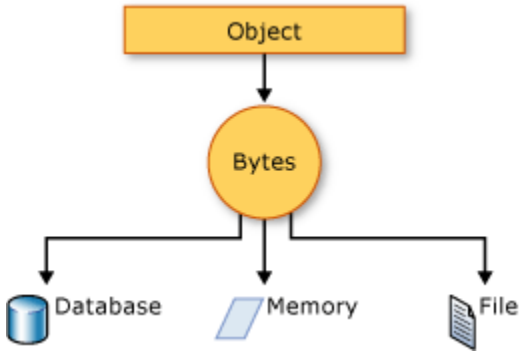


Figure 3

The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory [23].

4.1.1. Binary or XML Serialization

Either binary or XML serialization can be used. In binary serialization, all members, even those that are read-only, are serialized, and performance is enhanced. XML serialization provides more readable

code, as well as greater flexibility of object sharing and usage for interoperability purposes.

- Binary Serialization Binary serialization uses binary encoding to produce binary streams which may be stored in files or transported to other systems [24].

XML Serialization XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML [20].

4.1.2. Programing languages support

In this section we won't go throw the implementation of serialization in difference framework as much as is some clear example about serializing object to XML in this difference framework.

- **XML Serialization with .Net**

  In .Net we can serialize object to XML using

  The System.Xml.Serialization namespace which contains classes that are used to serialize objects into XML format documents or streams [23].

  For example we want to serialize an object of a class that contain data about city [25].

  First of all, I am going to have a custom class that represents a unit that contains data about a city:

```
public class City
{
   public string Name { get; set; }
   public string State { get; set; }
   public string County { get; set; }
   public bool IsCountyCenter { get; set; }
   public bool IsStateCapital { get; set; }
   public int Population { get; set; }
}
```

Instantiate the class and values to the existing properties:

```
City city = new City();
city.Name = "Independence";
city.State = "KS";
city.County = "Montgomery";
city.IsCountyCenter = true;
city.IsStateCapital = false;
city.Population = 10000;
```

The resulting XML file will look like this:

```
<?xml version="1.0"?>
<City xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Independence</Name>
  <State>KS</State>
  <County>Montgomery</County>
  <IsCountyCenter>true</IsCountyCenter>
  <IsStateCapital>false</IsStateCapital>
  <Population>10000</Population>
</City>
```

Which can be deserialized back to the same object.

- **XML serialization with java**

  Java offers many technique we can use to serialize java object to XML or vise-versa one of them is XMLEncoder and XMLDecoder which are classes in the java.beans package [26].

  Another way is JAXB (Java Architecture for XML Binding) which provides mechanism to marshal (write) java objects into XML and unmarshal (read) XML into object [27].

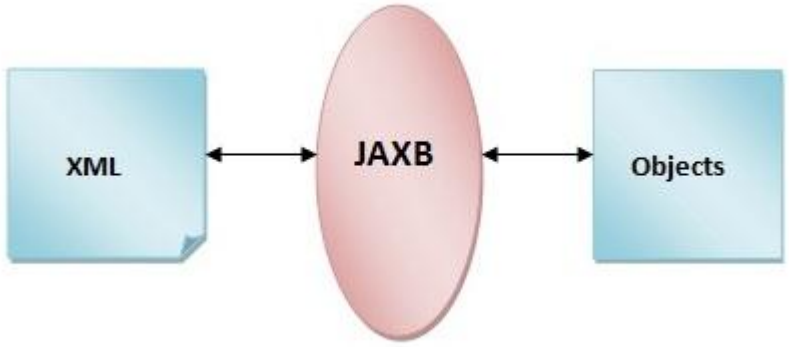


Figure 4

For example we have employee class which contain some data about an employee:

```
    @XmlRootElement
public class Employee {
private int id;
private String name;
private float salary;
public Employee() {}
public Employee(int id, String name, float salary) {
super();
this.id = id;
this.name = name;
this.salary = salary;
}
@XmlAttribute
public int getId() {
return id;
}
public void setId(int id) {
this.id = id;
}
```

```
@XmlElement
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
@XmlElement
public float getSalary() {
return salary;
}
public void setSalary(float salary) {
this.salary = salary;
}
}
```

Create a new object:

```
Employee emp=new Employee(4090789,"Antwan Al Haddad",50000)
```

the xml generated will look like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<employee id="4090789">
   <name>ANTWAN AL HADDAD</name>
   <salary>50000.0</salary>
</employee>
```

5. XML based representations for programming languages

XML-based markup languages offer great flexibility as well as the ability to represent documents as DOM annotated trees. The objective is to develop XML-based program representations in which the corresponding DOM trees represent source code information at the same level as this is provided by an AST abstraction of the CST (concrete syntax tree, or, parse tree) generated by a language parser.

The proposed representation is simple but detailed enough to represent the complete syntax of a specific programming language [28] in the form of a DTD schema and annotated source code in the form of a DOM tree. In addition, the DTD schema is extensible to allow for new entities and attributes to be added to model analysis results obtained by source code analysis tools [29].

For example, representations for programming languages that use XML as the modeling environment provide several advantages. Namely, these are:

- **Easy to understand and manipulate**

  By keeping the representation of the language as close as possible to the grammar of the language no extra learning burden should be added to the tool developers. Since the grammar is something that

developers are already familiar with, the representations should approximate the grammar as much as possible [30].

- **Extensible**

  The need to accommodate for the evolving programming languages requires that the representation is equally easy to change. The representation should also be flexible enough to allow for other representations to be developed based on it as extensions [28].

- **Widely supported**

  The success of a program representation is based on how well it is supported. The development of APIs that enable developers to easily read, store and change the information based on the suggested representations is very important. These APIs should be readily available for a variety of platforms [29].

- **Human Readable**

  Given the nature of the software maintenance tasks, it is necessary for the format to be easily readable. Even though most processing will be done automatically by the tools, enabling a developer to read the information directly will make the development task easier [29].

5.1. Annotating Source Code using XML

5.1.1. Mapping ASTs to DTDs

Given a specific programming language with a need to define a representation in which every valid source code document can be mapped to. This mapping from ASTs to XML trees can be accomplished by define a method to map the grammar of the programming language to a Document Type Definition (DTD). The mapping at this level will guarantee that all possible syntax trees

defined by the grammar (and therefore any source code program), can be mapped to XML trees defined by the corresponding DTD. One of the requirements when defining new XML representations is to make them as easy to use as possible.

In this context, there is a set of general transformation rules that assist in implementing a good mapping from a grammar to a DTD. Below, a list of transformation rules for defining such mappings is presented.

- Non-terminal grammar symbols are mapped to elements.
- Sequences of non-terminals are mapped to model sequence groups (i.e. using the ”,” symbol).
- Choices of non-terminals are mapped to model choice
- Groups (i.e. using the ”j” symbol).
- Non-Terminals that contain sequences of characters
- Are mapped to attributes.
- Optional non-terminals are mapped to model optional
- Choice groups (i.e. using the ”?” symbol).
- Lists of non-terminals are mapped to model set groups (i.e. using the ”*” or ”+” symbol).
- Terminals are mapped to attributes.
- Choices of terminals are mapped to named group attributes.
- Sequences of terminals are mapped to distinct attributes.

The use and combination of these transformation rules allow us to generate very elaborate mappings from grammars to DTDs. Once a grammar is mapped to a DTD, the ASTs for a specific program can be mapped to XML files. These XML files can then be used in place of the ASTs or the original source files for maintenance tasks.

Example Mappings In order to demonstrate how the previous rules are used there are two simple examples. In the first one, there is a combination of the rule for sequences of non-terminals and the rule for optional non-terminals. The resulting element declaration states that the element a contains an element b which is optionally followed by element c.

The second example is a demonstration of the rule for mapping lists of nonterminals to model groups is used with in conjunction with the rule for mapping terminals to attributes. It's obvious from the grammar rule that b is an artifact for expressing the fact that can occur many times. The element declaration captures this concept by simply using the "+" symbol. The terminal is mapped to an attribute declaration by storing the literal in a string called value. In the table below the mappings for both examples are show

Grammar	DTD
a : b Copt	<!ELEMENT a (b,c?)>
a : b Literal b : c b c	<!ELEMENT a (c+)> <!ATTLIST a value CDATA>

5.1.2. Java Markup Language (JavaML)

The generation of a program representation for Java is based on a parser generator tool called Java Compiler Compiler or JavaCC in short [JavaCC]. This tool was initially developed by Sun Microsystems and it is the most popular parser generator for Java. A parser generator is a tool that uses a

BNF grammar as input and generates source code that can parse any instance of the BNF grammar. The popularity of JavaCC is most probably due to the grammar for Java that is shipped with the tool. The majority of Java source code parsers are built using JavaCC and the Java 1.1 grammar. The Java 1.1 grammar was developed by "Sriram Sankar" at Sun Microsystems and a copy of this grammar can be found in the distribution of JavaCC package [29].

Below we present a small example of a Java source file and its corresponding JavaML representation as it is automatically generated by adding semantic actions in the Java parser generated by JavaCC.

Java source code

```
public class Car{
int color;
public int getColor()
{return color;}
}
```

JavaML representation

```
<ClassDeclaration Identifier="Car">
<FieldDeclaration>
<PrimitiveType Type="int"></PrimitiveType>
<VariableDeclaratorId Identifier="color"/>
</FieldDeclaration>
<MethodDeclaration Identifier="getColor">
<ResultType>
<PrimitiveType Type="int"/>
</ResultType>
<Block>
<ReturnStatement>
<PrimaryExpression>
<Name Identifier="color"></Name>
</PrimaryExpression>
</ReturnStatement>
</Block>
</MethodDeclaration>
</ClassDeclaration>
```

And there is many languages parser for different languages like CPPML for C++ and OOML [30].

6. GEN XML Representation

As mentioned in before, it is enough for a language to have only few construct to be able to solve any computationally solvable problem.

Those constructs are:

- Variable declaration
- Conditional statement
- Looping statement
- Function call

And as discussed in this chapter, XML is a good candidate for representing (serializing) those programming constructs.

The following is the XML tags representing the building blocks of GEN.

- Conditional Statement:

  a conditional statement consist of a Boolean condition, then clause and else clause, GEN provides conditional statement as follows:

  <function fname=”if” condition=”<Boolean expressino>”>

  <then>

  <!—then block->

  </then>

  <else>

  <!—else block->

  </else>

  </function>

- Looping statement:

  Looping statement consist of a condition as a Boolean expression and a body that will keep being executed until the condition is false, GEN provides that in the form of while tag,

  <function fname=”while “ condition = “<Boolean expression>” >

  <!—body ->

  </function>

- Variable Declaration:

  Variable declaration consist of a variable name and a variable type to be allocated in the memory, GEN provide as follows:

  <function fname=”declare” name =”<variable name>” type= “<variable type>” />

- Function call

  A function is a collection of statement that carry a specific task, GEN provides function call as follows:

  <function fname=”<function name>” param1=”name1” param2 =”name2” ….$param_n$ = “$name_n$ ” />

Chapter 3
A Visual Representation

After reviewing the main problems of using GEN scripts and listing famous IDE's with the facilities they provide, none prove to be a solution for the main problem GEN has as an XML-based programming language.

This call for the need of a new tool to be used for GEN scripts, but before looking for a solution or an approach to solve the problem, let us briefly review what are the main problem we are facing, and why current tools are not sufficient.

1. The problem

   Choosing XML to represent GEN code gives it a lot of flexibility, just Looking at XML gives the general idea of the data (or code in our case) right away, but when there is a need for looking into details reading XML tags is very confusing for the human eye.

   For example, if when writing code the person forgets a closing angle brackets '>', or a closing double quotes ' "', the parser will generate strange error messages. Understanding these error messages and correcting them is a tedious and time consuming procedure, and this add s cost to development in terms of debugging and maintenance.

   Using XML to store code enables the automation of code, and gives a lot of facilities as discussed in the previous chapter. But working with it directly is not productive and therefor there is a need for another layer or representation between the user and the code, to overcome this issue.

1. **Current solutions:** what most IDE's offer is just ease of editing using auto complete features or intellisense, this increases the productivity of during the implementation phase of the software development life cycle, but the problem persists when doing code review, debugging or maintenance . Plus there is no support for the particular syntax of gen.

2. Possible solutions:

2.1. **Customizing an already existing editor**

the main drawback of using available editors is the lack of support for GEN scripts, an XML document can be a legal XML but not a correct GEN script However, it is possible to extend the functionally of some editor to add support to GEN, an example of that is the VIM editor.

2.1.1. **Vim**

Vim is a highly configurable text editor built to enable efficient text editing. It is an improved version of the Vi editor distributed with most UNIX systems and it is distributed free [31].

Vim offers a lot of functionalities and extended features [32], such as:

- Syntax extensions

  Vim allows controlling indentation and syntax-based color coding of text and gives the user many options to define this automatic format. The user also can customize style of indentation.

- Programmer assistance

  Although Vim doesn't try to provide all programming needs, it offers many features normally found in Integrated Development Environments (IDEs).

- Scripting and plug-ins

  It is possible for the user to develop his own Vim extension or download plug-ins from the Internet.

  Session context

  Vim keeps session information in a file, .viminfo. which helps in solving the "Where was I?" problem when revisiting and editing a file. The user can define how much and what kind of information to sustain across sessions. For example, the user can define how many "recent

documents" or last-edited files to track, how many edits (deletions, changes) to remember per file, how many commands to remember from the command history.

- Keyword completion

  Vim allows the user to complete partially typed words with context-sensitive completion rules. For example, Vim can look up words in a dictionary or in a file containing keywords specific to a language.

  It is possible for the user to write their own scripts for Vim, and in that way expand the functionality of the editor. This is a really strong addition to the feature set of Vim, because it would normally have required coding in a lower-level language and recompilation of the editor in order to add even simple features.

  Vim plugin can be written in VIML, a scripting language designed specifically to extend the functionality of the editor to match the user needs.

One disadvantage this approach has is Vim is different from a standard editor most users are used to, and this will limit the use of GEN.

2.2. Customizing an already existing XML editor

since GEN is XML based language, we can take advantage of that by extending an XML editor, by coming up with a DTD (Document Type Definition) or an XML Schema, and writing GEN will be writing XML that is validated against an XML schema, that will hold the information about GEN tags and possible attributes for each one, possible types for each attribute, and so on.

This will provide a Syntax validator for GEN, where errors can be detected

right away and this will increase the productivity and lessen the time spent looking for errors due to simple typos like a missing character or missing enclosing tag and so on.

A disadvantage that this solution has is, GEN is not stable yet, it is still evolving and changing, and that will cause the DTD or the XML schema to be rewritten and modified every time an update happens with GEN.

Those two approaches will solve only one aspect of the problem and that is *writing* the code, but the readability of the code is still a problem.

3. One more approach to the problem:

before thinking more about a solution, let us take a look at the characteristic of a GEN.

it is an interpreted language, simple with very small set of programming constructs. A good candidate solution for this is *Visual Programming*:

a programming paradigm that was a famous researching topic till the mid 90's, but after that the Computer Science community stopped giving it that much of attention.

3.1. What is Visual Programming**:** A programming language that uses a visual representation like graphics, drawings, animation or icons, partially or completely. [33]

That doesn't mean it is trying to replace text, but text is not the only way to represent code.

3.2. Why Visual Programming:

The main advantage of using VPL is *code abstraction* [34], raising the level of code abstraction to visual level helps the programmer see the relationship between different “shapes” or blocks in the code.

Second of all, it increases the productivity rate because of an enhanced and improved communication between different programmers in a team.

3.2.1. **Common misconception:**

Visual Basic and the entire Microsoft Visual family™ are not, despite their names, visual programming languages. They are textual languages which use a graphical GUI builder to make programming decent interfaces easier on the programmer. but since representing the code itself is still done through text *and only* text, it is still a pure textual language.

3.2.2. **Limitation of VPL:**

one of the limitation of Visual Programming is known as the *Deutsch Limit* [35], referring to a comment made by Peter Desutsch:

“Well, this is all fine and well, but the problem with visual programming languages is that you can't have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system?”

Of course with a small language like GEN, this is not a problem.

3.2.3. **Benefits of VPL:**

Visual programming addresses one of important issue in software engineering. It enables users to master the complexity of programming by visualizing it. The raising of the abstraction to the visual level reveals semantic relationships among program entities and makes it more understandable. [34]

In the real world application, Baroth and Hartsough [36] reported on the use of LabView in their workplace. They had two group of developers to create a telemetry analyzer. One group was asked to use LabView and another group was asked to use C programming. After about eight weeks of work, they found that the visual programming group had exceeded the original requirements while the C programming group had not completed the original requirements. They cited several advantages of visual programming such as its flexibility in the design process, improvement of communication between users and programmers, and shorter period to train programmer to master the language. Besides, they have observed increased productivity through a reduction in software development time as communication between the customer, the developer and the computer is facilitated by the visual programming tools used.

4. Implementing the solution:

Technologies used to implement the solution are the C# programming language in the .NET framework and WPF.

4.1. **C#:** (pronounced C sharp) is a new programming language designed for building a wide range of enterprise applications that run on the .NET Framework [37]. An evolution of Microsoft C and Microsoft C++, C# is simple, modern, type safe, and object oriented. C# code is compiled as managed code, which means it benefits from the services of the common language runtime (CLR). These services include language interoperability, garbage collection, enhanced security, and improved versioning support. C# is a language in its own right [38]. Although it is designed to generate code that targets the .NET environment, it is not part of .NET. Some features are supported by .NET but not by C#, some other features of the C#

language are not supported by .NET (for example, some instances of operator overloading).

4.2. **Dot Net Framework:** Microsoft .NET Framework (pronounced dot net) is a software framework developed by Microsoft that runs primarily on Microsoft Windows [39]. It includes a large library and provides language interoperability (each language can use code written in other languages) across several programming languages. Programs written for .NET Framework execute in a software environment (as contrasted to hardware environment), known as the Common Language Runtime (CLR), [40] an application virtual machine that provides services such as security, memory management, and exception handling. The class library and the CLR together constitute .NET Framework.

Microsoft .NET Framework's Base Class Library provides user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications [40]. Programmers produce software by combining their own source code with .NET Framework and other libraries. .NET Framework is intended to be used by most new applications created for the Windows platform. Microsoft also produces an integrated development environment largely for .NET software called Visual Studio.

4.3. WPF: Windows Presentation Foundation (WPF) provides developers with a unified programming model for building rich Windows smart client user experiences that incorporate UI, media, and documents. [41]

Windows Presentation Foundation (WPF) is a next-generation presentation system for building Windows client applications with visually stunning user experiences. With WPF, you can create a wide range of both standalone and browser-hosted applications.

The core of WPF is a resolution-independent and vector-based rendering engine that is built to take advantage of modern graphics hardware. WPF extends the core with a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2-D and 3-D graphics, animation, styles, templates, documents, media, text, and typography. WPF is included in the Microsoft .NET Framework, so you can build applications that incorporate other elements of the .NET Framework class library.

The following are some of the most dramatic changes that WPF ushers into the Windows programming world [42]:

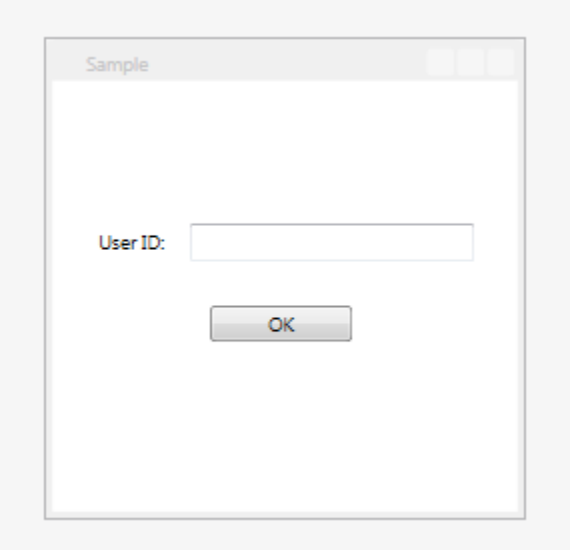
- A web-like layout model: Rather than fix controls in place with specific coordinates, WPF emphasizes flexible flow layout that arranges controls based on their content. The result is a user interface that can adapt to show highly dynamic content or different languages.
- A rich drawing model: Rather than painting pixels, in WPF you deal with primitives—basic shapes, blocks of text, and other graphical ingredients. You also have new features, such as true transparent controls, the ability to stack multiple layers with different opacities, and native 3-D support.
- A rich text model: WPF gives Windows applications the ability to display rich, styled text anywhere in a user interface. You can even combine text with lists, floating figures, and other user interface elements. And if you need to display large amounts of text, you can use advanced document display features such as wrapping, columns, and justification to improve readability.

- Animation as a first-class programming concept: In WPF, there's no need to use a timer to force a form to repaint itself. Instead, animation is an intrinsic part of the framework. You define animations with declarative tags, and WPF puts them into action automatically.
- Support for audio and video media: Previous user interface toolkits, such as Windows Forms, were surprisingly limited when dealing with multimedia. But WPF includes support for playing any audio or video file supported by Windows Media Player, and it allows you to play more than one media file at once. Even more impressively, it gives you the tools to integrate video content into the rest of your user interface, allowing you to pull off exotic tricks such as placing a video window on a spinning 3-D cube.
- Styles and templates: Styles allow you to standardize formatting and reuse it throughout your application. Templates allow you to change the way any element is rendered, even a core control such as the button. It has never been easier to build modern skinned interfaces.
- Commands: Most users realize that it doesn't matter whether they trigger the Open command through a menu or through a toolbar; the end result is the same. Now that abstraction is available to your code, you can define an application command in one place and link it to multiple controls.
- Declarative user interface: Although you can construct a WPF window with code, Visual Studio takes a different approach. It serializes each window's content to a set of XML tags in a XAML document. The advantage is that your user interface is completely separated from your code, and graphic designers can use professional tools to edit your XAML files and refine your application's front end. (XAML is short for Extensible Application Markup Language).

The following code snippet is an example of XAML code for a GUI.

```
<Window x:Class="DataDrivenWPF.Sample"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Sample" Height="300" Width="300">
           <Grid>
               <Label Height="28" Margin="24,88,0,0" Name="label1"
                     VerticalAlignment="Top" HorizontalAlignment="Left"
                      Width="57">User ID:</Label>
               <Button Margin="99,0,104,106" Name="cmdOK" Height="23"
                     VerticalAlignment="Bottom">OK</Button>
               <TextBox Height="23" Margin="87,90,28,0" Name="txtUserID"
                     VerticalAlignment="Top" />
           </Grid>
</Window>
```

Which will result in the following window.



Here we can see how WPF uses XAML to write the code for the GIU, each tag is just a serialization for an object of a class that represents a UI element.

5. Conclusion:

The solution proposed to overcome the difficulties of GEN development is to have a graphical IDE where each language construct will be represent by a different symbol.

Writing code will be simply dragging symbols and arranging them in a flow chart representing the logic to be coded.

Later on, the IDE will be in charge of writing the equivalent code (i.e XML tags) and producing the final source code.

Chapter 4
Integrated Development Environment

This chapter demonstrates the goal of the IDEs and role that IDEs played in the programing evolution by showing the different type of IDEs and what it used for and their advantage and disadvantage.

1. What programmers used before IDEs?

The time when a developer wrote an application in a text editor, saved it, exited the editor, ran the compiler, wrote down the error messages on a pad of paper, and then traced back through the code again. [43]

This was in the days before IDE's and real-time code checking and one-click links to errors being taken for granted [43]

2. History of the ide's and there Evolution on the time

IDEs initially became possible when developing via a console or terminal. Early systems could not support one, since programs were prepared using Flowcharts, entering programs with punched cards (or paper tape, etc.) before submitting them to a compiler. Dartmouth BASIC was the first language to be created with an IDE (and was also the first to be designed for use while sitting in front of a console or terminal). Its IDE (part of the Dartmouth Time Sharing System) was command-based, and therefore did not look much like the menu driven, graphical IDEs prevalent today. However it integrated editing, file management, compilation, debugging and execution in a manner consistent with a modern IDE. [44]

Maestro I is a product from Softlab Munich and was the world's first integrated development environment 1975 for software. Maestro I was installed for 22,000 programmers worldwide. Until 1989, 6,000 installations existed in the Federal Republic of Germany. Maestro I was arguably the world leader in this field during the 1970s and 1980s. Today one of the last Maestro I can be found in the Museum of Information Technology at Arlington. [44]

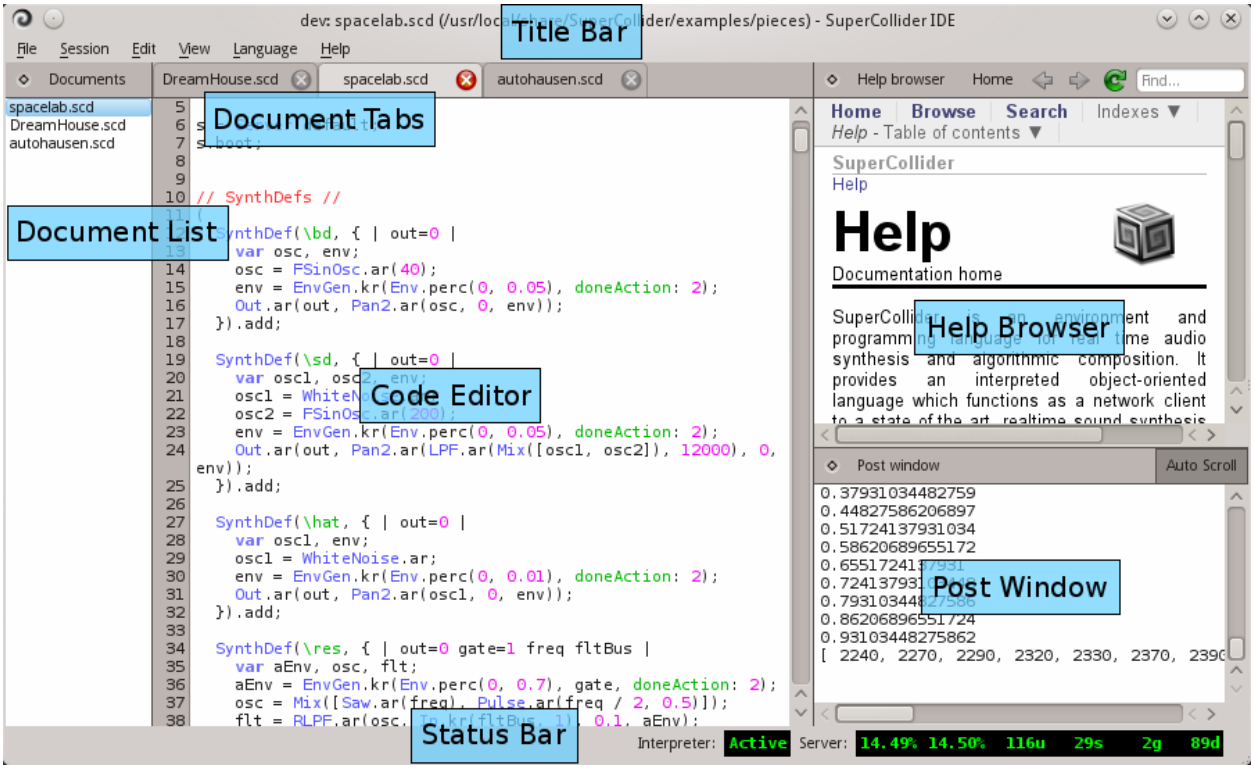
One of the first IDEs with a plug-in concept was Softbench. In 1995 Computer woche commented that the use of an IDE was not well received by developers since it would fence in their creativity. [44]

3. Role of IDE's in Development

Computer-assisted software engineering (CASE) tool known as the integrated development environment (IDE): is computer software that generally consists of a source code editor, a compiler or interpreter (or both), build-automation tools, and a debugger. Many IDEs also have a version control system or tools for building a graphical user interface (GUI) ideally, use of an IDE should result in greater productivity compared with the use of multiple single purpose CASE tools for program development, such as a text editor and a separate compiler like we menschen in the intro. [45]

4. Contents of IDE

Here is a picture that describe an IDE and what should be contacted (SuperCollider IDE) [46] [47]



Graphical IDEs encompass functions that include code browsing and editing, compilation, debugging, and basic project management capabilities. More advanced IDEs provide additional capabilities such as static analysis, performance monitoring, and other tools to enable developers to understand system bottlenecks and use of system resources. Another popular feature built into advanced IDEs is collaborative development support, fostering code sharing among development team members [46] [47]

5. The goal of using IDE

In the Software Engineering Body of Knowledge (SWEBOK), goals that a software development environment (which includes IDEs) should meet. Such environments should [45]

1. Reduce the cognitive load on the developer.
2. Free the developer to concentrate on the creative aspects of the process.
3. Reduce any administrative load associated with applying a programming method manually.
4. Make the development process more systematic.

6. Facilities and advantages an IDE gives

1. Syntax highlighting (e.g. for Haskell, Cabal, Literate Haskell, Core, etc.)
2. Macros (e.g. inserting imports/aligning/sorting imports, aligning up text, transposing/switching/moving things around) [48]
3. Type information (e.g. type at point, info at point, type of expression)
4. Intelligence/completion (e.g. jump-to-definition, who-calls, calls-who, search by type, completion, etc.)
5. Project management (e.g. understanding of Cabal, configuration, building, installing, package sandboxing)
6. Interactive REPL (e.g. GHCi/Hugs interaction, expression evaluation and such)
7. Knowledge of Haskell in the GHCi/GHC side (e.g. understanding error types, the REPL, REPL objects, object inspection)
8. Indentation support (e.g. tab cycle, simple back-forward indentation, whole area indentation, structured editing, etc.)
9. Proper syntactic awareness of Haskell (e.g. with a proper parser and proper editor transpositions a la the structured editors of the 80s and Isabel et al)
10. Documentation support (e.g. ability to call up documentation of symbol or module, either in the editor, or in the browser)
11. Debugger support (e.g. stepping, breakpoints, etc.)
12. Refactoring support (e.g. symbol renaming, hlint, etc.)

13. Templates (e.g. snippets, Zen coding type stuff, filling in all the cases of a case, etc.)
14. Automatic completion: As programmer typing, the editor will try to find possible continuations of the code, and pop up a list of suggestions, so can quickly insert one instead of typing the whole text. This is called *auto completion* and is a great aid towards writing code faster and with less errors. In general, this works for class and method names
15. Method call assistance: The editor helps when writing arguments of a method by showing a complete list of the method's arguments and their default values, as long as it manages to figure out exactly which method that is. This is called method call assistance. It helps to know exactly which argument the programmer typing, and reduces the need to look into documentation.
16. Class library navigation**:** Often it is very useful to make the programmer able to jump directly to a place in a file, where a particular class or method is implemented, or to find all the places where a class or a method name is used. The IDE offers practical ways to achieve that.

7. Types of IDE's

And there are two types of IDE's [49]: lumberjack IDE and magician IDE:

7.1. Lumberjack IDE**:**

This type of IDE provides all sorts of labor-saving devices to perform typing/processing that programmer could do but do not want to do because it is boring, repetitious and/or time consuming. Examples? Automatically indenting code. Automatically adding end-tags in XML documents. Automatically knowing where to go in the online manuals to look up information based on where the cursor is, right now.

7.2. Magician IDE:

This type of IDE provides all the above and more. At the press of a button it will generate big chunks of code for you that you can compile and run. It is capable of optimizing your code behind the scenes in all sorts of clever ways. It gives a visual view of the code and data so that programmer don not actually have to know all the details about how things are stored and inter-related. It is capable of pulling all aspects of projects that programmer have together in one place, freeing him to get on with that task at hand.

8. Categories of IDE's

In terms of IDE technology trends, tools for independent software vendors (ISVs) today fall into three major categories: [47]

8.1. Includes long-standing suppliers of proprietary development tools who have invested many man-years in building their own integrated environment (e.g., GreenHills Multi, MetroWerks CodeWarrior, and of course Microsoft's ubiquitous VisualStudio product line).

8.2. The second group includes companies that create customizations or plug-ins to Microsoft Visual Studio to enable product or application-specific development (e.g., LynuxWorks VisualLynx, Mentor Graphics code|lab, and Texas Instruments CodeComposer Studio).

8.3. The third, newest, and most dynamic ISV IDE strategy is to build on a shared open source code base like the Eclipse Project. Evolved from IBM's WebSphere Developer Studio, Eclipse combines code contributed by IBM and dozens of Open Source community members. The growing list of companies supporting, contributing to and leveraging Eclipse includes Ericsson, HP, IBM, Intel, MontaVista Software, QNX, Samsung, SAP, TimeSys, and many others.

9. Rules for choosing an IDE

Not all integrated environments are created equal; integration for its own sake adds little to a positive developer experience. An effective IDE should meet or exceed the quality of the sum of its integrated parts.

First-time IDE users or developers experienced with a particular IDE and making a transition should consider the following: [47]

1- Many IDEs are at least partially open. Ensuring that IDE have customize key functions to suit the programmer particular needs, e.g., substituting programmer favorite editor for the default, invoking a different compiler revision or an entirely different compiler without breaking integration of point-and-click error tracking, and integrating resident revision control tools

2- Some IDEs are host-specific (e.g., Visual Studio runs only on Microsoft Windows workstations). If your team uses a mix of workstation types (Windows, Linux, Solaris) or even different versions of the same host (Windows2000 vs. WindowsXP, Red Hat vs. SuSE Linux, Red Hat 8.0 vs. 9.0, etc.), making sure the IDE supports appropriate activities on each type of workstation.

3- No IDE is completely seamless and no IDE vendor is immortal. Ensuring that IDE lets the team member's access project components using CLI tools and that project data and code itself can be exported to familiar formats and used with legacy "make" tools.

4- Ensuring that a new IDE offers a clean migration path either from legacy CLI build paradigms or from formats employed by incumbent environments.

5- Coming from a CLI environment, some IDEs exert unexpected control over your development project. For example, some IDEs by default hide intermediate file types (e.g., relocatable object files), forcing you either to

find new ways to work or to go outside the IDE to preserve your prior work-flow

6- Impressive and attractive IDEs may accomplish the goals of a vendor giving a demo, but take pains to understand how they address your projects and practices. Also, IDEs present themselves as a collection of views, perspectives, or states; transition among these views is often non-intuitive. Built-in documentation with real-use scenarios is essential.

10. Difficulties IDEs suffer from in programing?

10.1. From the point of view of a programmer

Examples of critical items on the Software Usability Measurement Inventory (SUMI) for experienced and Novice developers. [45]

- **Experienced developers:**
  1. There is too much to read before you can use some elementary features
  2. never learn to use all that is offered in this software
  3. It feel safer if programmer use only a few familiar commands
  4. It feel in command of this software when using it (disagree)
  5. It takes too long to learn the software commands
  6. It keep having to go back to look at the guides
- **Novice developers:**
  1. There is never enough information on the screen when it's needed
  2. It takes too long to learn the CASE tool features
  3. Learning to operate this software initially is full of problems
  4. Learning how to use new functions is difficult
  5. It feel safer using only a few familiar commands
  6. prefer to stick to the facilities that know best

10.2. From point view of ide's types:

And as we mashed Previse there is to types of ide's magician IDE and lumberjack IDE

Doesn't the magician IDE sounds great? What's not to like?

Well, a lot actually. If the programmer go in this route may end up not understanding how his own application is organized "under the hood". And may lose visibility of all the dependencies in the project has because the IDE does it all for him. May find that the only way you can talk to database is through the "wizard screens" because those screens generated so much magic code that programmer can no longer connect to the database unaided...

The magician IDE comes at a significant price: understanding. The lumberjack IDE on the other hand, is less flashy but at least programmer know what it his doing and could - if required - do it himself.

[49]

11. Challenges in the design of usable IDEs

There are several reasons why it is difficult to design usable IDEs. These tools are typically very complex computer applications with large numbers of different kinds of functionalities. These functionalities often appear in quite different modes or contexts that include source code editing, dynamic debugging, file linking and management, and the viewing of relations between structural elements of the application being developed (i.e., program comprehension), among others. Each context may require somewhat different functionality representation or user interface organization. For example, it may be beneficial to design the basic user interface for a source code editor to resemble that of a standard word processor, which capitalizes on the developer's familiarity with this type of general office productivity tool for manipulating

text. In contrast, a reusable components browser may need an advanced user interface that exhibits different visualizations to facilitate program comprehension. This requires a much higher level of integration of textual and graphical representation of program elements, such as classes, packages, or data structures. It is no simple task to design a user interface for an integrated application that is equally consistent and usable across many different contexts or modes. [45]

It is also more difficult to apply certain design principles from the human–computer interaction (HCI) area to a domain like IDEs, which should support both the cognitive processes of individual developers and collaboration among members of a development team. [45]

12. Description of GEN IDE

- Dealing and generating block of code by just dropping some shape in the canvas so the most of code will be generated automatically and even though we can still write and insert some code by hand throw text fields like when we specify and put the conduction of a while loop or if statement.
- Generate portable code because is XML-BASE code generator.
- give us the ability of Project management because the output of our ide will be some shape content whit other forming some form look like the UML diagrams form.
- Reduce the amount of time that need it to write code and save us the time to correct the syntax's errors because there is already block of code saved in and linked with some shape so when the we drop some shape in the canvas the correct code will be generate saving us the time of correcting the syntax's errors and we called that the ability of Proper syntactic

awareness.

- Reusability: it ease to make changes in the code just by changing the links between the shape's and adding some more shape's and link's

Some situation writing code by hand is more efficient than generate it because there is some special station we must change the form in block of code to do the specific job and we can't do all the special cases and save them in sort of shape's it will be hundreds of shapes and thousands of rules that specify how the shapes must link with each other.

And for that the programmer lose his control on the generated code and the ides become the controller of the generated code.

And from all that we put our ide in the types of Magician IDE because is do some the work for the user behind the seen by linking the block of code with each other and generate automatically code.

Chapter 5
Implementing the Solution

1. The Model

we begin by modeling the statements in GEN, all the statements share the following properties:

- Id: attribute which is a unique identifier for each statement.
- Fname: attribute to store the type of the statement, to be used later by the interpreter.

Because of that the model will consist of a base class called *Statement*, which has the common properties shared by all the statements in GEN,
plus the method *Code*, which is responsible for generating the code for the GEN statement, and returning that as an XML node to be included in the final source code.

This method was declared to be *abstract*, which will force each class that inherits *Statement*, to implement its own version of *code*.

Then we will have a class for each statement in GEN, adding more statements to the IDE will be easy, all what should be done is inheriting the base class, and implementing the *Code* method.

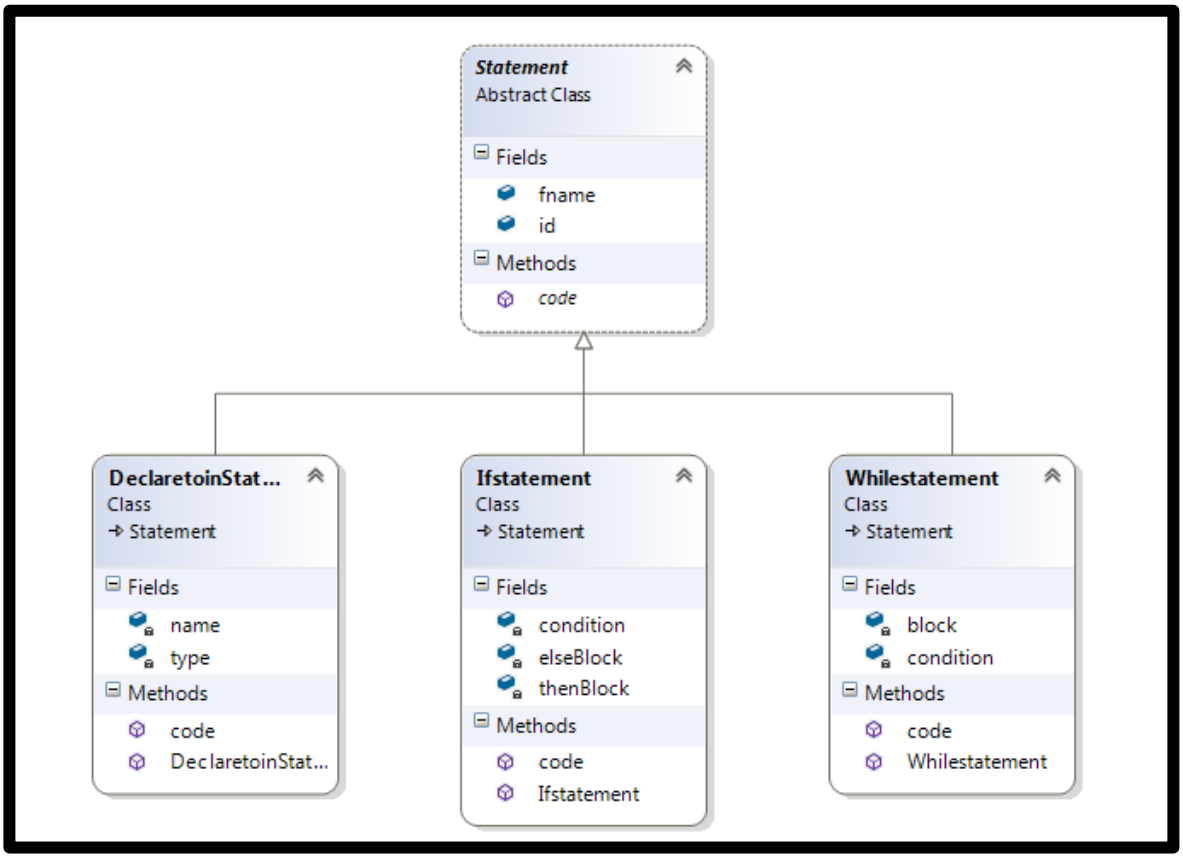
Current classes are:

1.1. If Statement: with the following attribute:
- 1.1.1. Condition: representing the Boolean condition of the if statement
- 1.1.2. elseBlock: list of statements representing the body of the else clause
- 1.1.3. thenBlock: list of statements representing the body of the then clause

1.2. While Statement:
- 1.2.1. Condition: Boolean condition for the while loop
- 1.2.2. Body: list of statements representing the body of the loop

1.3. Declaration Statement:
- 1.3.1. Type: type of the variable being declared
- 1.3.2. Name: the name of the variable

The following is a diagram representing the hierarchy of those classes:



The following are code snippet for the mentioned classes above,

```
using System;
using System.Text;
using System.Windows;
using System.Xml;

namespace DiagramDesigner
{
    public  abstract class Statement
    {

        public string id;
        public string fname;

        public abstract XmlElement code();


    }
}
```

The *DeclarationStatement* Class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;

namespace DiagramDesigner
{
    class DeclaretoinStatement : Statement
    {

        string name ;
        string type  ;

        public DeclaretoinStatement() {
            fname = "Declare";
            name = "foo";
            type = "bar";

        }

        public override XmlElement code()
        {
            XmlDocument doc = new XmlDocument();
            XmlElement Function;


            Function = doc.CreateElement("Function");
            Function.SetAttribute("fname", fname);
            Function.SetAttribute("id", id);

            Function.SetAttribute("type", type);
            Function.SetAttribute("name", name);


            return Function;

        }
    }
}
```

The *ifStatement* Class:

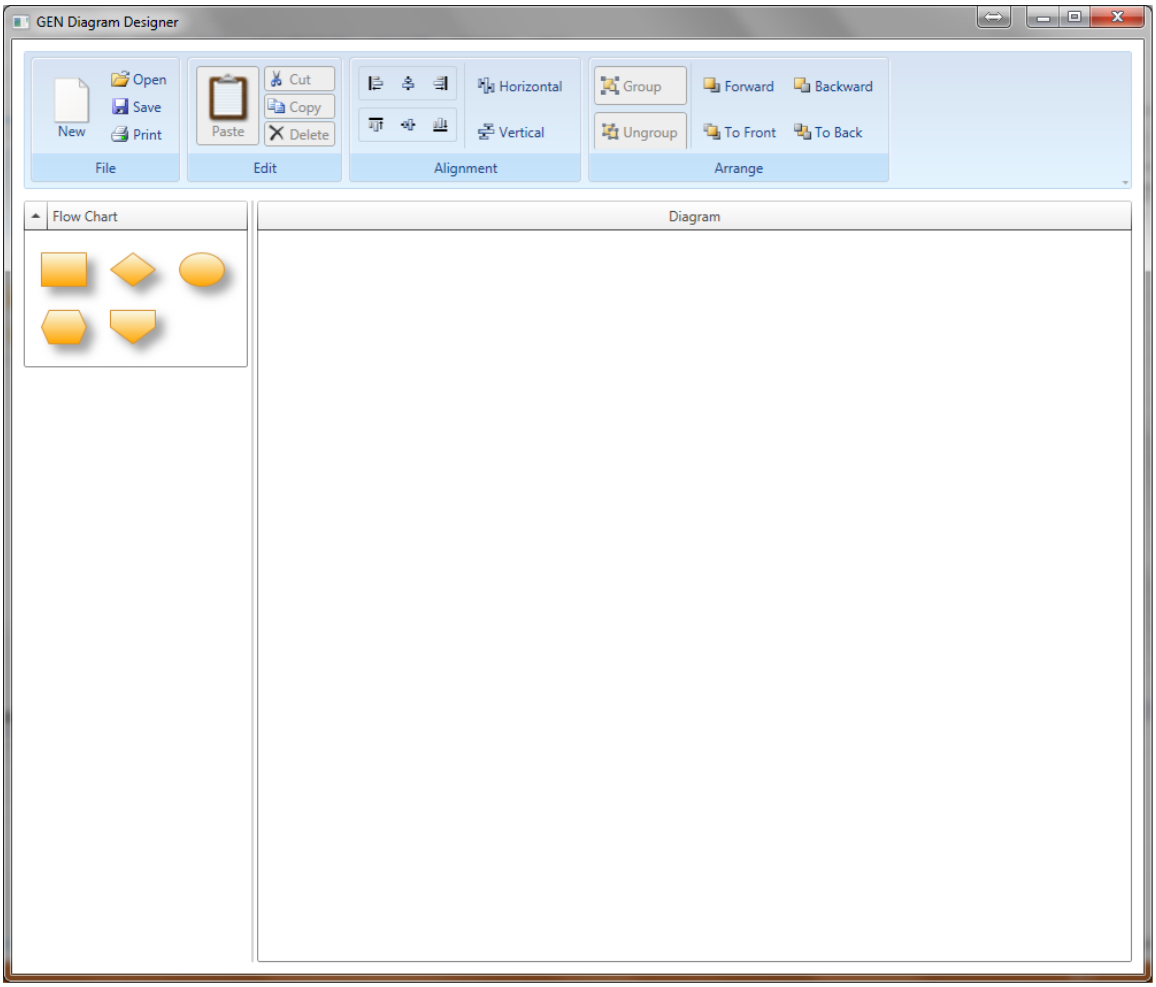
```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;

namespace DiagramDesigner
{
    class Ifstatement : Statement
    {
        string condition ;
        List<Statement> thenBlock;
        List<Statement> elseBlock;

      public Ifstatement() {
            thenBlock = new List<Statement>();
            elseBlock = new List<Statement>();
            fname = "if";
            id = "123";
            condition = "foo == bar";
        }

         public  override XmlElement code() {
            XmlDocument doc = new XmlDocument();
            XmlElement Function;
            XmlElement Then;
            XmlElement Else;

            Function = doc.CreateElement("Function");
            Function.SetAttribute("fname", fname);
            Function.SetAttribute("id", id);
            Function.SetAttribute("condition", condition);
            Then = doc.CreateElement("Then");
              foreach (Statement s in thenBlock) {
                XmlNode ChildNode = doc.ImportNode(s.code(), true);

                Then.AppendChild(ChildNode);
            }
            Function.AppendChild(Then);
            if (elseBlock.Capacity != 0) {
                Else = doc.CreateElement("Else");
                foreach (Statement s in elseBlock) {
                    XmlNode ChildNode = doc.ImportNode(s.code(), true);
                    Else.AppendChild(ChildNode);
                }
                Function.AppendChild(Else);
            }

            return Function;
        }
    }
}
```

The *whileStatement* Class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;

namespace DiagramDesigner
{
    class Whilestatement : Statement
    {
        string condition;
        List<Statement> block;

        public Whilestatement() {
            block = new List<Statement>();
            id  = "123";
            fname = "while";
            condition =  "foo == bar";
        }

        public override XmlElement code(){
            XmlDocument doc = new XmlDocument();
            XmlElement Function;
            XmlElement Do;

            Function = doc.CreateElement("Function");
            Function.SetAttribute("fname", fname);
            Function.SetAttribute("id", id);
            Function.SetAttribute("condition", condition);

            Do = doc.CreateElement("do");
              foreach (Statement s in block)
            {
                XmlNode ChildNode = doc.ImportNode(s.code(), true);
                Do.AppendChild(ChildNode);
            }

            Function.AppendChild(Do);
            return Function;

        }
    }
}
```

2. Execution at run time

This is the main window of the IDE:



The user drag a symbol from the right, and drop it into the canvas on the left. when the drag drop operation is complete, a new *DesignerItem* object is created.

The *DesignerItem* class contains the needed information to keep track of the flow chart items in the diagram, plus a *Statement* object to hold the statement that the flow chart symbol represents.

The process of determining the correct derived class of statement and allocating the object is done through the **Method Factory** design patter with the *factory* object, implemented using the *singleton* pattern.

3. Design Pattern

3.1. Definition of Design Patterns:

Design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

3.2. Uses of design pattern:

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

3.3. Types of design patterns

3.3.1. **Creational design patterns** This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

- Abstract Factory: Creates an instance of several families of classes.
- Builder: Separates object construction from its representation
- Factory Method: Creates an instance of several derived classes
- Object pool: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

- Prototype: A fully initialized instance to be copied or cloned
- Singleton: A class of which only a single instance can exist

3.3.2. **Structural Design pattern:** This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

- Adapter Match interfaces of different classes
- Bridge: Separates an object's interface from its implementation
- Composite A tree structure of simple and composite objects
- Decorator Add responsibilities to objects dynamically
- Facade A single class that represents an entire subsystem
- Flyweight A fine-grained instance used for efficient sharing
- Private Class Data Restricts accessor / mutator access
- Proxy An object representing another object

3.3.3. **Behavioral design patterns :**This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

- Chain of responsibility A way of passing a request between a chain of objects.
- Command Encapsulate a command request as an object
- Interpreter A way to include language elements in a program
- Iterator Sequentially access the elements of a collection
- Mediator Defines simplified communication between classes
- Memento Capture and restore an object's internal state
- Null Object Designed to act as a default value of an object
- Observer A way of notifying change to a number of classes
- State Alter an object's behavior when its state changes
- Strategy Encapsulates an algorithm inside a class

- Template method Defer the exact steps of an algorithm to a subclass
- Visitor Defines a new operation to a class without change

-

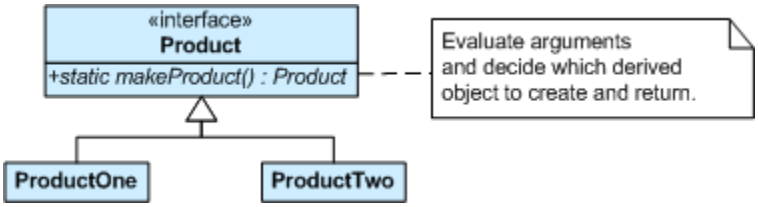
Leads to efficient solutions: The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

Factory Method Design Pattern

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

An increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive names.

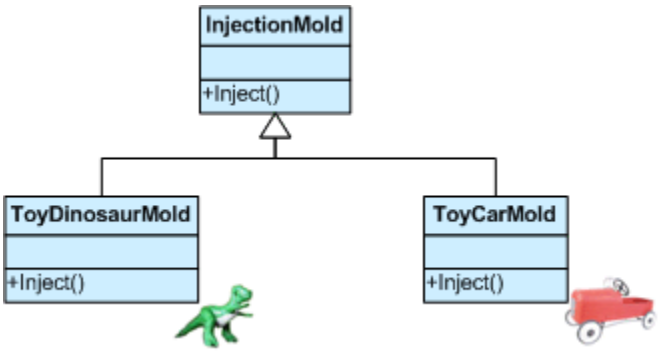


Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

Check list

1. If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a `static` factory method in the base class.
2. Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate.
3. Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.
4. Consider making all constructors `private` or `protected`.



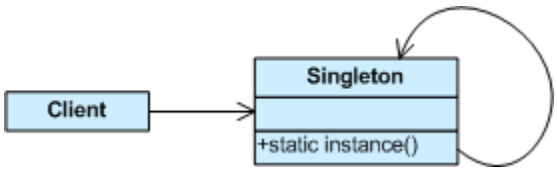
This structural code demonstrates the Factory method offering great flexibility in creating different objects. The Abstract class may provide a default object, but each subclass can instantiate an extended version of the object.

Singleton Design Pattern

Intent

Ensure a class has only one instance, and provide a global point of access to it, Encapsulated "just-in-time initialization" or "initialization on first use".

Structure



Make the class of the single instance responsible for access and “initialization on first use”. The single instance is a private static attribute. The accessor function is a public static method.

Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, “The President of the United States” is a global point of access that identifies the person in the office.

Check list

1. Define a private static attribute in the “single instance” class.
2. Define a public static accessor function in the class.
3. Do “lazy initialization” (creation on first use) in the accessor function.
4. Define all constructors to be protected or private.
5. Clients may only use the accessor function to manipulate the Singleton.

Applying Design pattern in the IDE implementation

Using the Factory Method design pattern:

This pattern was used for creating the corresponding statement for each flow chart symbol.

StatementFactory class

```
public class StatmentFactory
  {

      private static StatmentFactory instance;
      private Dictionary<String, IStatmentFactory> dict;



      protected StatmentFactory()
      {
          dict = new Dictionary<string, IStatmentFactory>();
          dict.Add("If", new IfStamtentFactory());
          dict.Add("While", new whileStamtentFactory());
          dict.Add("Declare", new DeclarationStamtentFactory());

      }

      public static StatmentFactory StatementFactoryInstance() {

          if (instance == null)
              instance = new StatmentFactory();

          return instance;

      }

      public Statement Get(String name)
      {
          return  dict[name].createStatement();
      }

  }
```

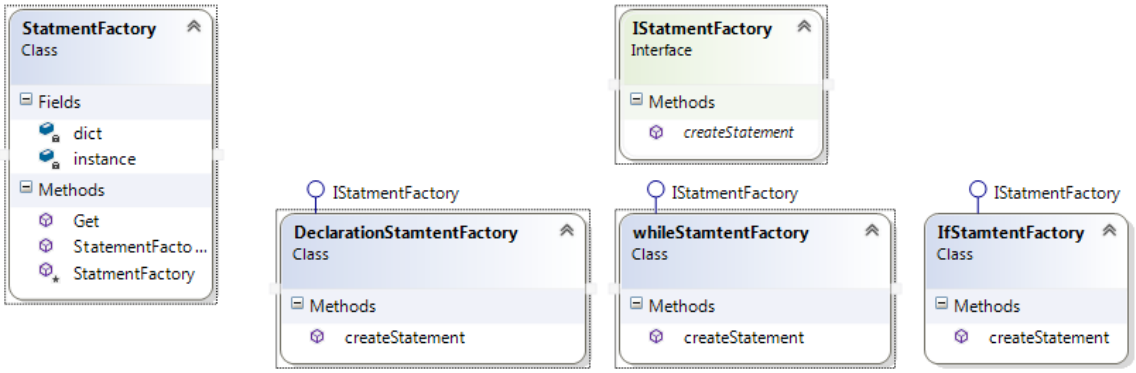
In writing this class the singleton design pattern was used, and this will be discussed later in the chapter.

The *StatementFactory* class, has a dictionary mapping each name statement name, to the appropriate factory object.

The *Get* method, is passed a string representing the statement name, the method use the name as an index for the dictionary and find the correct factory object, and

then it call the *createStatement* method, that will return an object representing the statement required.

Using this patter will make extending the IDE easy, in the case of adding a new statement, all what needs to be done is just add a new factory class, and map the name of the new statement to an object of the class factory.
The following is a diagram showing the hierarchy of the *IStatementFactory* family of classes:



The following code snippet is for the *IStatementFactory* interface and the implementing classes

```
using System;
using System.Collections.Generic;
using System.Text;

namespace DiagramDesigner
{
    public interface  IStatmentFactory {
         Statement createStatement();
    }

    public class IfStamtentFactory:IStatmentFactory {

        public Statement createStatement() {
            return new Ifstatement();
        }
    }

    public class whileStamtentFactory : IStatmentFactory {
        public Statement createStatement() {
            return new Whilestatement();
        }
    }

    public class DeclarationStamtentFactory : IStatmentFactory {
        public Statement createStatement() {
            return new DeclaretoinStatement();
        }
    }
}
```

Using the Singleton design pattern:

To apply this pattern, a unique object of the class is created , and the constructor is made private.

Instead, the class will supply a method for creating an instance of the class, that method, will check for the presence of the object and return it, or create a new one in case there wasn't any instance of it before.

This will guarantee having only one unique object of the class and provide a global access for it anywhere in the code.

Future Work:

- Give the IDE the functionality of reading GEN script and generating the corresponding flow chart diagram.
- Writing an XSD for the language to check the validity of GEN codes, (not yet possible because the language is not stable and still evolving)
- Adding the functionality of rearranging the flowchart Items in the canvas to enhance the user experience.

Bibliography

[1] wikipedia.org. (2013, December) wikipedia.org. [Online]. https://en.wikipedia.org/wiki/Programming_language

[2] Maurizio Gabbrielli, *Programming Languages: Principles and Paradigms (Undergraduate Topics in Computer Science)*.: Springer, 2010.

[3] Greg shaw. users.cis.fiu.edu. [Online]. http://users.cis.fiu.edu/~shawg/2210/languages.doc

[4] Michal L. Scott, *Programming Languages and Pragmatics 3rd Edition*.: Morgan Kaufmann, 2009.

[5] princeton.edu. princeton.edu. [Online]. http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Assembly_language.html

[6] wikipedia.org. (2013 , October) wikipedia.org. [Online]. http://en.wikipedia.org/wiki/Porting

[7] Cory Janssen. techopedia.com. [Online]. http://www.techopedia.com/definition/24308/fourth-generation-programming-language-4gl

[8] princeton.edu. princeton.edu. [Online]. http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Fourth-generation_programming_language.html

[9] wikipedia.org. (2013, October) wikipedia.org. [Online]. https://en.wikipedia.org/wiki/Fourth-generation_programming_language

[10] Robert W. Sebesta, *Concepts of Programming Languages, 10th Ed*.: Addison-Wesley, 2012.

[11] Cory Janssen. techopedia.com. [Online]. http://www.techopedia.com/definition/24309/fifth-generation-programming-language-5gl

[12] wikipedia.org. (2013, October) wikipedia.org. [Online]. https://en.wikipedia.org/wiki/Fifth-generation_programming_language

[13] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Programming_language_theory

[14] Gilles Dowek, *Prinnciples of Programming Languages*. London: Springer, 2009.

[15] (2013, Nov.) Wikipedia. [Online]. http://en.wikipedia.org/wiki/XML

[16] Brian Benz and John R. Durant, *XML Programming Bible*.: Wiley, 2003.

[17] (2013, Dec.) Extensible Markup Language (XML) 1.0. [Online]. http://www.w3.org/TR/1998/REC-xml-19980210

[18] Aaron Skonnard and Martin Gudgin, *Essential XML Quick Reference*.: Pearson Education, 2002.

[19] JOE FAWCETT, LIAM QUIN, and DANNY AYERS, *Beginning XML*. Indiana: Wrox, 2012.

[20] Bipin Joshi, *Beginning XML with C# 2008 From Novice to Professional*.: Apress, 2008.

[21] Bill Evjen and Kent Sharkey, *Wrox Profissional xml*.: Wrox, 2007.

[22] Ajay Vohra and Deepak Vohra, *Pro XML Development with Java Technology*.: Apress, 2006.

[23] (2013, Dec.) Microsoft. [Online]. http://msdn.microsoft.com/en-us/library/ms233843.aspx

[24] (2013, Dec.) BinarySerialization. [Online]. http://www.blackwasp.co.uk/BinarySerialization.aspx

[25] (2013, Dec.).NET Zone. [Online]. http://dotnet.dzone.com/articles/serializing-and-deserializing

[26] (2013, Dec.) XMLEncoder. [Online]. http://docs.oracle.com/javase/7/docs/api/java/beans/XMLEncoder.html

[27] Sonoo Jaiswal. (2013, Dec.) javatpoint. [Online]. http://www.javatpoint.com/jaxb-tutorial

[28] Hrvoje Simic and Marko Topolnik, "Prospects of encoding Java source code in XML ,".

[29] Ademar Aguiar, Gabriel David, and Greg Badros. JavaML2. [Online]. www.fe.up.pt/~aaguiar/docs/JavaML2.pdf

[30] (2013, Dec.) Markup_languages. [Online]. en.wikipedia.org/wiki/Markup_languages

[31] Home Page. [Online]. http://www.vim.org/

[32] Arnold Robbins, *Learning the Vi and Vim*.: O'Rilley, 2008.

[33] Ricardo Baeza-Yates. Computer Science Departement - University of Chilie. [Online]. http://users.dcc.uchile.cl/~rbaeza/cursos/vp/todo.html

[34] Rodina Ahmad, "VISUAL LANGUAGES: A NEW WAY OF PROGRAMMING," *Malaysian Journal of Computer Science,* vol. 12, no. 1, pp. 76-81, June 1999.

[35] FAQs. [Online]. http://www.faqs.org/faqs/visual-lang/faq/

[36] E. Baroth and C. Hartsough, *Visual Programming in The Real World*. Englewood Cliffs: Prentice-Hall, 1994.

[37] Microsoft. (2013, November) MSDN. [Online]. http://msdn.microsoft.com/en-us/library/aa287558(v=vs.71).aspx

[38] Bill Evjen Christian Nagel, *Professtional C# 2012 and.NET 4.5*.: John Wiley & Sons, 2013.

[39] Wikiepdia. (2013, November) Wikiepdia. [Online]. http://en.wikipedia.org/wiki/.NET_Framework

[40] Hoang Lam,*.NET Essentials*.: O'rielly, 2002.

[41] Microsoft. (2013, November) MSDN. [Online]. http://msdn.microsoft.com/en-

us/library/ms754130(v=vs.110).aspx

[42] Matthew MacDonald, *Pro WPF 4.5 in C#*, 4th ed.: Apress.

[43] Andy Patrizio. (2013, February) mendix. [Online]. http://www.mendix.com/think-tank/the-history-of-visual-development-environments-imagine-theres-no-ides-its-difficult-if-you-try/

[44] Wikipedia. (2013, October) Wikipedia. [Online]. https://en.wikipedia.org/wiki/Integrated_development_environment#History

[45] Ahmed Seffah Rex Bryan Kline. (2005, July) psychology.concordia.ca. [Online]. http://psychology.concordia.ca/fac/kline/Library/ks05.pdf

[46] sccode.org. sccode.org. [Online]. http://doc.sccode.org/Guides/SCIde.html

[47] Jacob Lehraum and Bill Weinberg. (2004, july) eetimes.com. [Online]. http://www.eetimes.com/document.asp?doc_id=1150355

[48] haskell.org. (2013, October) haskell.org. [Online]. http://www.haskell.org/haskellwiki/IDEs

[49] Sean McGrath. (2008, April) itworld.com. [Online]. http://www.itworld.com/software-development-ide-nlstipsm-080429

[50] wikipedia.org. (2013, October) wikipedia.org. [Online]. https://en.wikipedia.org/wiki/High-level_programming_language

[51] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Programming_language_theory

[52] Wikipedia. [Online]. http://en.wikipedia.org/wiki/Lambda_calculus