MARKUP LANGUAGE FOR DESIGNING DIAGRAMS

Maria AFTENI[1], Sevastian BAJENOV[1], Mihaela CUȘNIR[1*], Alexandru FURDUI[1], Gabriel GÂTLAN[1]

[1]*Department of Software Engineering and Automatics, FAF 213, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Moldova*

*Corresponding author: Mihaela CUȘNIR, mihaela.cusnir@isa.utm.md

Tutor/coordinator: Vasile DRUMEA, univ. asis., *Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Moldova*

Abstract. *This article represents a contemplation and analysis of domain-specific language in developing a markup language for designing diagrams. Because it is already a well-known fact that UML diagrams have a considerable purpose in visualizing the future of a project (not only in the software engineering domain), this paper aims to present an efficient approach, explaining the grammar and functionality of such language in creating and editing diagrams.*

Keywords: *diagram, Domain-Specific Language, grammar, Unified Modeling Language.*

Introduction

According to Girish Managoli in his article “What developers need to know about domain-specific languages”, Domain-Specific Language (also known as DSL) is a basic language created to be used in specific, but vast areas [1]. Furthermore, even if it has limited influence on users’ domains, it is excellent when used in its spectrum of applicability.

Not too far, the Markup Languages is one of the DSL’s spheres and noted when it comes to “annotating an electronic document” [2]. In other words, a Markup Language will “rule” how the document, or any kind of information, will be displayed. The most popular examples are HTML, XML, SGML and MathML.

Unified Modeling Language is a standardized language used in software engineering for creating visual models of software systems [3]. Beyond the idea of being used in software development, UML diagrams are used in system engineering, business analysis, database design, network engineering, and also electrical engineering. So there is no surprise that having a great Markup Language for designing diagrams is more than just a tool, it is a must have for a successful project.

Language Overview:

Generally speaking, there is no concrete concept of a data structure in a markup language. The entities existing in our DSL are being created with the help of tags, not variables, which are actually defining only the behavior of an object. Because the DSL is being created in order to simplify designing UML diagrams, then the main object of the language will be <diagram>. It will describe each entity representing a diagram and its parameters will be listed using attributes (type=state machine, for instance). The next data structures will be constructed in a smaller scale and dependent on the type of the diagram they belong to. For instance, use-case diagrams will have substructures <actor> or <use case> which on their own will be described by more attributes and interconnected using various types of relations. Relations also will be dependent on the diagram category and have their own parameters (for example, <relation type=dependency...>).A markup language for creating UML diagrams can take different inputs, including textual, graphical, programmatic, and mixed.

Our markup language relies on tags, which are a form of programmatic input. These tags will allow us to define a wide range of elements, including packages, systems, diagrams, and the relationships between them. By using tags to define these elements, can be created a highly structured and flexible language that can be used to represent a wide range of systems, from simple diagrams to complex software architectures. As for the output, a program in this markup language for UML would produce diagrams that represent the UML elements and their relationships defined by the input tags. The output could be in a variety of formats, such as image files or vector graphics, depending on the needs of the user. Additionally, the program could also produce textual representations of the UML diagrams, which could be useful for documentation or further analysis.

Grammar

When referring to formal language theory, a grammar is a set of production rules or principles used in creating valid strings from an alphabet, except for their meaning or how they can be used in any context.

Table 1

Meta notation

Notation	Meaning
<symbol>	A nonterminal symbol
symbol	A terminal symbol
*	Repeated 0 or more times
+	Repeated 1 or more times
?	Means 0 or 1 times
|	Separates alternatives

Since UML has more than one type of diagram, the grammar for each one can differ, as presented below:

<uml> ::= <use-case-diagram>| <class-diagram> | <component-diagram> |
<deployment-diagram> | <activity-diagram> | <sequence-diagram> |state-diagram|

Aspects of the grammar:

<type> ::= **int** | **float** | **double** | **string** | **boolean** | **<class-name>**
<identifier> ::= <letter> (<letter> | <digit> | "_")
<letter> ::= **a** | **...** | **z** | **A** | **...** | **Z**
<digit> ::= **0** | ... | **9**
<text> ::= (<letter> | <digit> | <punctuation> | " ")
<punctuation> ::= **.** | **,** | **;** | **:** | **!** | **?** | **-** | **/** | \ | **"** | **"** | **(** | **)** | **[** | **]** | **{** | **}**

1. Use Case Diagram:

<use-case-diagram>::=<use-case-diagram><use-case>|<use-case-actors></use-case-diagram> <use-case> ::= <use-case> <use-case-name> </use-case>
<use-case-name> ::= " <identifier>"
<use-case-actors> ::= <actors> λ|<actor>| <use-case-actors> </actors>
<actor> ::= "<identifier>"

2. Class Diagram:

<class-diagram> ::= <class-diagram> <class></class-diagram>
<class> ::= <class> <class-name> <class-attributes> | <class-methods> </class>
<class-name> ::= "<identifier>"
<class-attributes> ::= <attributes> <attribute> </attributes>
<attribute> ::= <type> <identifier>
<class-methods> ::= "<methods>" <method>* "</methods>"
<method> ::= <type>? <identifier> "(" <parameters>? ")" <method-body>?
<parameters> ::= <parameter> ("," <parameter>)*
<parameter> ::= "<type><identifier>"

3. Component Diagram:

<component-diagram>::=<component-diagram><component><connector></component-diagram>
<component>::=<component><component-name><component-attributes></component>
<component-name> ::=" <identifier>"
<component-attributes> ::= <attributes> <attribute></attributes>
<attribute> ::= "<type> <identifier>"

4. Deployment Diagram:

<deployment-diagram> ::= <deployment-diagram> <node> <artifact> </deployment-diagram>
<node> ::= <node> <node-name> <node-type> <node-attributes></node>
<node-name> ::= "<identifier> <node-type> "::= physical | virtual
<node-attributes> ::= <attributes> <attribute></attributes>
<attribute> ::= "<type> <identifier>"

5. Activity Diagram:

<activity-diagram> ::= <activity-diagram> <activity> <control-flow> </activity-diagram>
<activity> ::= <activity> <activity-name> <activity-nodes></activity>
<activity-name> ::= "<identifier>"
<activity-nodes> ::= <nodes> (<initial-node> | <activity-node> | <final-node> | <decision-node> | <merge-node>)</nodes>
<initial-node> ::= <initial-node> <node-name> </initial-node>
<activity-node> ::= <activity-node> <node-name> <node-attributes><activity-node>
<node-name> ::= "<identifier>"
<node-attributes> ::= <attributes> <attribute> </attributes>
<attribute> ::= "<type> <identifier>"

6. Sequence Diagram:

<sequence-diagram> ::= <sequence-diagram> <lifeline> <message></sequence-diagram>
<lifeline> ::= <lifeline> <lifeline-name> <lifeline-attributes><lifeline>
<lifeline-name> ::= " <identifier>"
<lifeline-attributes> ::= <attributes> <attribute></attributes>
<attribute> ::= "<type> <identifier>"
<message> ::= ("<synchronous-message>" | "<asynchronous-message>" | "<reply-message>") <message-name> <from> <from-lifeline> </from><to> <to-lifeline> </to> <message-attributes><message>
<message-name> ::= "<identifier>"
<message-attributes> ::= <attributes> <attribute></attributes>
<from-lifeline> ::= "<identifier>"
<to-lifeline> ::= "<identifier>"

7. State Diagram:

<state-diagram> ::= <state-diagram><state> <transition></state-diagram>
<state> ::= <state> <state-name> <state-attributes> <state-entry> <state-exit><state>
<state-name> ::= "<identifier>"
<state-attributes> ::= <attributes> <attribute></attributes>
<attribute> ::= "<type> <identifier>"
<state-entry> ::= <entry> <activity</entry>
<state-exit> ::= <exit> <activity></exit>
<transition>::=<transition><transition-name><from-state><to-state><transition-trigger><transition-guard> <transition-action></transition>
<transition-name> ::= "<identifier>"

```
<from-state> ::= <state-name>
<to-state> ::= <state-name>
<transition-trigger> ::= <trigger> <trigger-event></trigger>
<trigger-event> ::= “<identifier>”
<transition-guard> ::= <guard> <guard-condition></guard>
<guard-condition> ::= “<expression>”
<transition-action> ::= <action> <activity></action>
<activity> ::= “<identifier>”
```

Types of relationship in diagrams:

```
<relationship> ::= <relationship> <relationship-type> <relationship-source>
<relationship-target> |<relationship-properties> </relationship>
<relationship-type> ::= "association" | "aggregation" | "composition" | "inheritance" | "realization"
<relationship-source>::=<source><source-name> <source-cardinality>? <source-role></source>
 <relationship-target> ::= <target> <target-name> <target-cardinality> <target-role> </target>
 <relationship-properties> ::= <property> <property-name> = <property-value> </property>
<source-name> ::= “ <identifier>” <source-cardinality> ::= <cardinality> <cardinality-value>
"</cardinality>" <source-role> ::= <role> <role-name> </role>
 <target-name> ::=  “ <identifier> “
<target-cardinality> ::= <cardinality> <cardinality-value> </cardinality>
 <target-role> ::= <role> <role-name> </role>
<property-name> ::= “<identifier>”
 <property-value> ::= “<text>”
 <cardinality-value> ::= <cardinality-range> | <cardinality-exact>
<cardinality-range> ::= <integer> .. <integer>
 <cardinality-exact> ::= <integer> <role-name> ::= “<text>”
```

Code example and its parsing Tree

Since it is known how important visualization is when working on a complex project, the derivation of the grammar is no exception. For presenting it hierarchically, parse tree helps in following the set of production rules for each diagram [4]. Beneath, in the figure 1 and 2, are examples of the parse tree for two different kinds of diagrams.

Code:

```
use-case-diagram:
actor "User" as User
rectangle Web_Records {
  usecase "SignUp" as UC1
  usecase "LogIn" as UC2
  usecase "Post" as UC3
  usecase "Edit" as UC3
}
User --> UC1
User --> UC2
User --> UC3
/use-case-diagram
```

The next figure, “Parse tree for a Use Case diagram”, follows the set of production for creating a Use Case diagram.

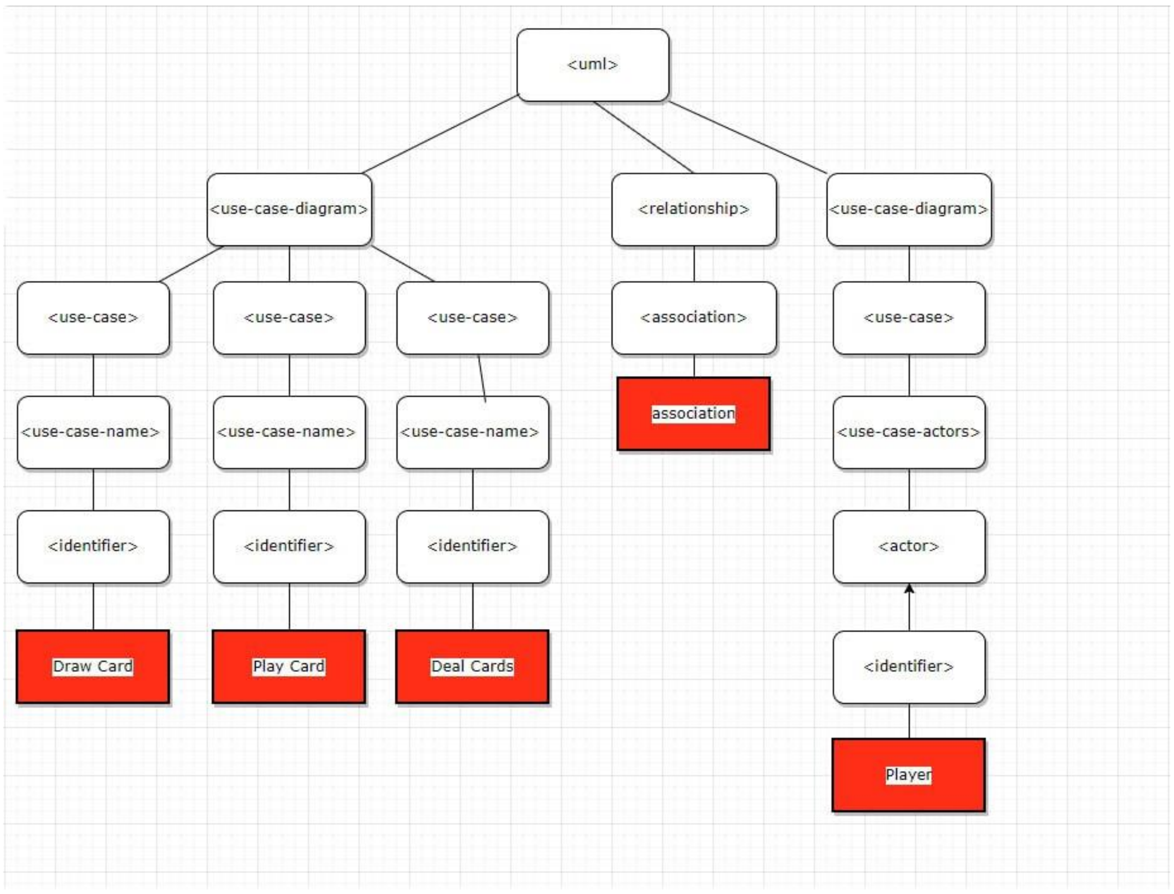

Figure 2. Parse tree for a Use Case diagram

Conclusions

Domain Specific Language is a great instrument for developers and creators around numerous domains, and there is no doubt that its subset, Markup Languages for designing diagrams, make the path of the project towards success ensured. One step, like visualization of the architecture or the project plan, can be a major and decisive stage before the final verdict. Consequently, having a flexible and manageable language for devising corresponding diagrams is indispensable. In other words, having such Markup Language is more than an advantage in efficient working.

References

1. MANAGOLI, G. *What developers need to know about domain-specific languages* [online], 2020. [accessed 06.03.2023]. Disponible: https://opensource.com/article/20/2/domain-specific-languages
2. UNIVERSITY OF LONDON, School of Advanced Study, *An introduction to markup* [online], 2022. [accessed 06.03.2023]. Disponible: https://port.sas.ac.uk/mod/book/view.php?id=568&chapterid=336
3. OBJECT MANAGEMENT GROUP, *Introduction To OMG's Unified Modeling Language™ (UML®)* [online], 2005. [accessed 06.03.2023]. Disponible: http://www.uml.org/what-is-uml.htm
4. DEEPANSHU, R. *Parse Tree in Compiler Design*, [online], 2022. [accessed 06.03.2022]. Disponible: https://www.geeksforgeeks.org/parse-tree-in-compiler-design/