Design of Symbol Table by using Design Patterns

Miloš Stamenović[1] and Suzana Stojković[2]

***Abstract* – This paper presents an object-oriented approach in symbol table design. Symbol tables are data structures storing data about symbolic names in the program. Symbolic names are names of different kinds of items in the programs. Because of that, symbol tables contain very heterogeneous structures of data. In this solution symbol table is divided on: type table, symbol table (storing only variable names) and function table.**

Design patterns are used often in object-oriented design. Composite, factory method and singleton patterns are used in symbol table design presented in this paper.

***Keywords* – Symbol tables, compiler design, object-oriented design, design patterns**

I. Introduction

Symbol tables are data structures that are used for identifiers storing in compile time [1],[2]. Symbol tables are used for identifying syntax and semantics errors and warnings within a program. Identifier in the program can be the name of: variable, constant, type, function, etc. For different types of tokens (that are named by identifiers), different data must be stored in the symbol table. For example:

- for a variable - name, type, dimension (for array variable), last value definition, last using, etc;
- for a function - name, returned type, number of formal arguments, types of arguments, types of argument passing, etc.

There are three basic methods manipulating with a symbol table: method for inserting new symbol in the symbol table, method for deleting symbol from the symbol table and symbol lookup method (method for searching the symbol within the symbol table). The symbol lookup method is very often used and because of that, it must be very fast. Therefore, symbol tables are obligatory realized as a hash tables.

It is obvious that structure of the symbol table is very complex. Because of that, symbol table construction is one of the most significant problem in the compiler construction.

Object-oriented approach in software design and implementation enables reusing of the existing source and reusing of existing the design details (known as the design patterns [3],[4],[5]). In the last years, design patterns are often applied in compiler construction and in the symbol table design, too [6],[7]. Design patterns usage in symbol table design for a simple script language interpreter will be shown in this paper. The script language is very specialized, but proposed design is general and applicable for the other interpreters and compilers.

II. Design Patterns

"Design pattern systematically names, explains and evaluates an important and recurring problem in object-oriented systems" [3]. Each pattern is defined by 4 elements:

- name (identifies the pattern),
- problem (describes when pattern is applied),
- solution (describe design elements making up the pattern and relationships between them),
- consequences (describe results and trade-off of using the pattern).

In the next section we will give a brief sketch of design patterns which are used in our symbol table design.

A. Composite pattern

The Composite pattern [3], [5] models composition of objects into tree structure. This pattern is used for representation of part-whole hierarchies of objects when clients should not know about difference between composite and leaf objects. Structure of the Composite pattern is shown on Fig. 1.

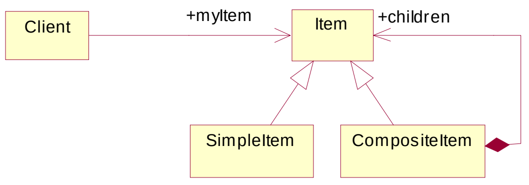


Fig. 1 Class diagram of the Composite pattern

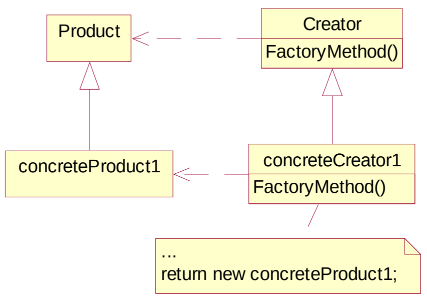


Fig. 2 Structure of the Factory method

[1] Miloš Stamenović emploied in the Troxo d.o.o, Dusanova 55, 18000 Niš, Serbia and Montenegro, e-mail: milostam@eunet.yu.

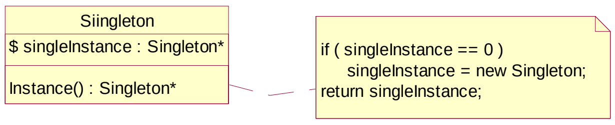
[2] Suzana Stojković emploied in the Faculty of Electronic Engeneering, Beogradska 14, 18000 Niš, Serbia and Montenegro, e-mail: suza@elfak.ni.ac.yu

B. *Factory method*

The Factory method [3] (known as Virtual Constructor) models an interface for creating an object. Derived classes "know" which class to instantiate. Structure of the Factory method pattern is shown on Fig. 2.

C. *Singleton*

The Singleton pattern [3] models a class that has exactly one instance. The Singleton class is responsible for creating its unique instance and contains an *Instance()* operation. Clients access Singleton instance through the *Instance()* operation. Structure of Singleton pattern is shown on Fig. 3.



```
if ( singleInstance == 0 )
     singleInstance = new Singleton;
return singleInstance;
```

Fig. 3 Structure of the Singleton pattern

III. Script language characteristics

We have requirement to create a C++ documentation generator. Documentation generation is based on usage of documentation templates. Documentation templates describe how the generated software documentation will look like. Documentation templates are Word documents containing certain scripts. The base problem in documentation generator development is to realize interpreter of the scripts. The symbol table, presented in this paper, is a part of that interpreter. Characteristics of the script language important for symbol table design are:

- The script language has limited number of types build-in the language definition. It is not possible to define any new type. There are three kinds of types: simple types (*integer, string, image, float*), structures (sset of data of different types) and collections (set of data of the same type).
- The script language does not contain declarations of script variables. Thus, types of variables are determined by the context of their using.
- The script language has limited number of functions build-in the language definition, too. Those functions generate data about C++ project that can be integrated in generated documentation.
- The script language has *for each* iteration statement,. Therefore we have to involve a special type – collection type.

In the following text, we will emphasize some main components of the specific script interpreter and recommend our approach in symbol table design.

IV. Symbol table design

Symbol table in general, has to store type names, function names and variable names. As it was mentioned before, our implementation of script language has limited number of types and functions included in definition of the language. Type names never appear in the scripts. New functions cannot be defined within the scripts. Symbol table is divided into three tables: type table, symbol table and function table because set of data describing the functions, types and variables are very different. Since, all types and functions are well known in design phase, type table and function table are initialized and filled before interpreting starts.

Architecture of our script language interpreter is shown on Fig. 4. Every table is represented as class package. Every package has one main class that is designed like singleton class. Thus, type table has TypeTable singleton class, symbol table has SymbolTable singleton class and function table has FunctionTable singleton class.

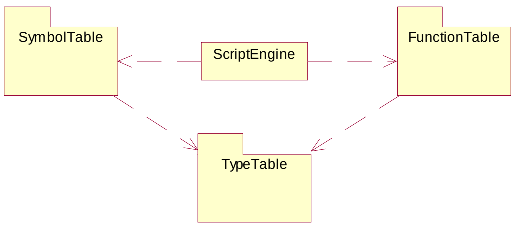


Fig. 4 Architecture Overview

A. *Type table*

A limited number of types allow us to define data structures for all types that will be used in the script language. Data structure including all script language types definitions is called Type table. Basic demand for type table is to allow maintainability and extensibility. We will use some combination of design patterns to accomplish these requirements. Class diagram of the Type table is shown on Fig. 5.

We define generic *type* with the Type class and all other kinds of types as subclasses of the Type class. The following classes are derived from Type:

- SimpleType
- StructureType
- CollectionType

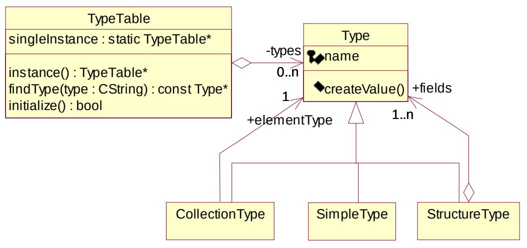


Fig. 5 Type table class diagram

The SimpleType class encapsulates simple types like *integer, string, imagee, float, etc.* StructureType has various fields, and every field has its own *type*. For example, some instance of StructureType can have one field of an *integer* type, one field of any StructureType and one field of any CollectionType. The CollectionType class represents collection of elements where all elements have the same type.

We want to describe this types hierarchy and treat all objects uniformly. Also this types structure could be very complex. This consideration leads us to *composite design pattern* as obvious choice. The types objects are used by one very complex object, called ScriptEngine. Composite design pattern separates ScriptEngine class from types hierarchy, which is liable for changes.

TypeTable is a singleton class and its findType method is used to find type in hash table where the key is a function of type name. Certain symbol can be created when its type is determinated. First, we have to get *type* object by type name, then we create *value* for this *type* and, finally, create *symbol* with this *value*

B. Symbol table

The Symbol table, in our interpreter, contains only data about variables in the scripts. The Symbol table includes the following classes:

- SymbolTable class. The SymbolTable class is implemented as hash table of symbol objects. As it was mentioned before, this class is designed like a singleton class.
- Symbol class. The Symbol class represents a variable in the scripts. Symbol has the following attributes describing variable in a program: *name*, *lastUsed*, *defined*, *DTName* (name of documentation template in which variable is defined) and *value*. Atribut *value* is necessary because Symbol table is used in interpreter, i. e. Symbol table exists in a run-time.
- Values class hierarchy. This hierarchy includes Value class and Value subclasses (SimpleValue, StructureValue and CollectionValue). It is designed by using composite pattern, too. When we expand this composite object structure we will get tree structure with SimpleValue-s as leafs. Only SimpleValue contains data in the *value* attribute. The *value* attribute is implemented as string, because all simple types *integer*, *date*, *float* etc could be represented as strings.

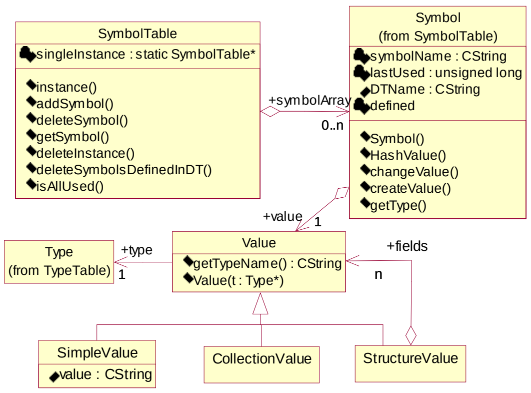


Fig. 6 Symbol table class diagram

SymbolTable structures are shown on the *Symbol table* class diagram (see Fig. 6). It is important to note a connection between Value and Type. Value object always “has knowledge” of its own type by using this connection. Also, the StructureValue knows types of its fields via the StructureType object. Similarly, the CollectionValue knows type of its elements via the CollectionType object.

Imagine that Type subclasses are added on the *Symbol table* class diagram. The result diagram would contain parallel class hierarchies connected with *type* relationship. This relationship is realized with createValue method and we can notice similarity between this Value-Type structures and FactoryMethod (Creator-Product) design pattern. FactoryMethod design pattern is useful when we want to delegate creation responsibility to derived classes. It means that, for example, only StructureType class knows how to create its value (StructureValue).

Fig. 7 shows scenario intending to explain process of creating values. This scenario covers interpreting of a script with a function call containing certain StructureType script variable. First, ScriptEngine gets function from FunctionTable by function name in order to get type of out function argument. Second, new empty symbol has to be added in SymbolTable. Third, StructureValue object has to be created with all its fields. Finally, StructureValue object has to be filled in with real data by using the Function object. Process of creating StructureValue object have to be explained with more details. Creating of StructureValue object implies creating of all its fields. According to that, StructureValue object gets type for each field from its StructureType object and requires from these types to create value by using factory method *createValue*.

Creating value does not mean coping real data to symbol. Symbol is added in symbol table when declaration of some variable is detected, but there is possibility that declared variable would not be used in the following script. For example, CollectionValue object could contain many elements, copying all elements can be an expensive operation, but there is not guaranty that all elements would be used. Therefore, current design could be improved by using virtual proxy technique. It means that only reference to data source is placed in CollectionValue object so CollectionValue object is used like proxy to real data source. Thus, when parsed script requires value of some CollectionValue element, real data will be obtained via proxy.

C. Function table

It was mentioned before, that we had limited number of functions build-in the language definition. Therefore, we can gather definition of all functions in the FunctionTable structures.

FunctionTable class is designed like Singleton class, because there must be only one instance of the FunctionTable. Every function is identified by its name (arguments are not considered like they do many object oriented languages). As it is shown on Fig. 7, when the ScriptEngine detects a function in script it starts creating value based on type of the function out argument. It means that symbol with empty Value object is created. The Value object has to be filled in with information extracted from project for which documentation will be generated. For this purpose, ScriptEngine gets function by its name from FunctionTable and calls the function (*call*

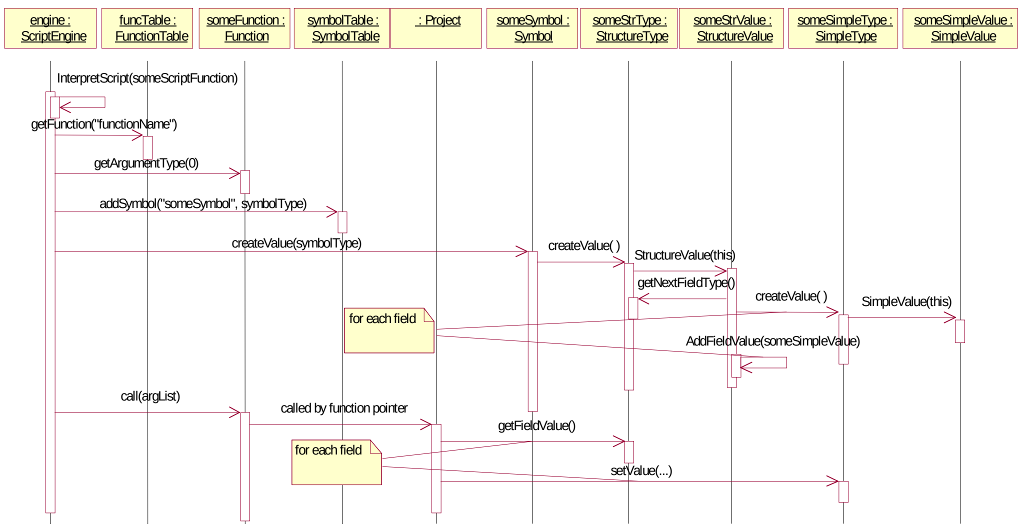

engine : ScriptEngine
funcTable : FunctionTable
someFunction : Function
symbolTable : SymbolTable
: Project
someSymbol : Symbol
someStrType : StructureType
someStrValue : StructureValue
someSimpleType : SimpleType
someSimpleValue : SimpleValue
InterpretScript(someScriptFunction)
getFunction("functionName")
getArgumentType(0)
addSymbol("someSymbol", symbolType)
createValue(symbolType)
createValue( )
StructureValue(this)
getNextFieldType()
createValue( )
SimpleValue(this)
for each field
AddFieldValue(someSimpleValue)
call(argList)
called by function pointer
getFieldValue()
for each field
setValue(...)

Fig.7 Scenario of interpreting certain script

member of Function class) to fill Value object. Method *call* is generic because it is used for every symbol. During the initialization of FunctionTable, *implementation* property of Function class is set. *implementation* property represents function pointer on real function. Thus, ScriptEngine calls real function via Function object, i.e., via *implementation* property.

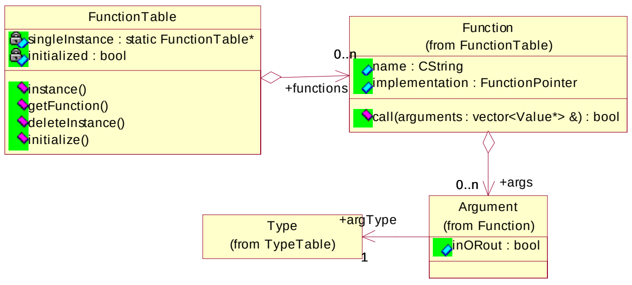


Fig. 8 Function table class diagram

Finally, class Argument defines function parameter or argument. Argument maintains a pointer to Type which is used by ScriptEngine for detecting incorrect parameters.

V. Conclusion

In this paper we have presented an object-oriented approach in development of a script language interpreter. We were focused on symbol table design. The Symbol table is the most important part of language interpreters and compilers. Our basic idea was to create a symbol table independent of script engine implementation and applicable in various interpreters. Therefore, we have used design patterns.

In the presented design, symbol table was divided in three tables: type table, symbol table and function table. Each table has a main class designed as singleton class. For designing of types and variables values, composite pattern was used. It enables that other parts of system do not have to know about differences between types and appropriate values. It provides easy adding new types in the language, or replacing existing type. For creating variable value, the factory method pattern was used.

Usage of UML language and design patterns increased speed and quality of presented design. It enables vary easy extansion of existing script language and application of implemented symbol table in other interpreters or compilers.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addition-Wesely, 1986.

[2] D. Grune, H. E. Bal, C. J. H. Jacobs, K. G. Langendeon, *Modern Compiler Desing*, New York, John Wiley & Sons, 2002.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Element of Reusable Object-Oriented Software*, Addition-Wesely, 1995.

[4] D. Milićev, M. Zarić, N. Piroćanac, *Objektno orijentisano modelovanje na jeziku UML: skripta sa praktikumom*, Beograd, Mikro knjiga, 2001.

[5] R. Riechle, “Composite Design Patterns”, Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications , pp. 218-228, Atlanta, Georgia, United States, 1997.

[6] J. F. Power, B. A. Malloy, “Symbol Table Construction and Name Lookup in ISO C++”, Proceedings of the 37th Conference on Technology of Object-Oriented Languages and Systems, pp. 57-69, Sydney, Australia, 2000.

[7] M. Hind, A. Pioli, “Traveling Through Dakota: Experiences with an Object-Oriented Program Analysis System” , Proceedings of the 34th Conference on Technology of Object-Oriented Languages and Systems, pp. 49-60, Santa Barbara, California, 2000.