Journal of Multidisciplinary Engineering Science and Technology (JMEST)
ISSN: 3159-0040
Vol. 2 Issue 4, April - 2015

Mixin-based programming in C++

Lakshmi Navyatha Kasaraneni
Department of Computer Science
University Of Bridgeport
Bridgeport,CT,USA
Email – lakshminavyatha28@gmil.com

Abstract: **This research paper has an attention on the web search tool, how it will work, what are the important steps a search engine will perform like Crawling, Indexing, Retrieving the data from data base and how the Google search engine will work.**

I. INTRODUCTION

Analysts ought to have an enthusiasm for guaranteeing that their articles are listed via web crawlers, for example, Google , Yahoo, Bing and so forth which extraordinarily enhances their capacity to make their articles accessible to the individuals. Like some other sort of positioned query items, pursuits showed in top positions are more inclined to be perused. There are a few distinctions in the ways diverse web files work, in any case they all perform three crucial errands:

They look for the Internet - or select bits of the Internet - in perspective of basic words.

They keep a rundown of the words they find, and where they find them.

The importance of usage of mixin classes in designing the components and obtaining effective implementation is discussed in this article. We are also going to converse about advance features of C++. These added features improve performance but in some situations they turn to be complicating

As we know, most of the successful and prodigious software components or the (Software artifacts) are much intricate to understand. For reducing the complexity, the software component is segregated into units thus making it manageable. Many techniques emerged to for this process reducing complexity. Of those, C++ templates is effective in its work. Mixin-Based programming is nothing but the implementing of the alliance based design using a class pattern which is templatized. This technique is more advantageous than the use of application frameworks which is widely used. When used, these software artifacts turn to be reusable and redundancy is greatly reduced. Dynamic binding is also eliminated which is also a timely factor. But when using this process the question of scalability arises.

Mixin Layers is the other procedure which is used to overcome the deficiencies in this Mixin-Based programming. Mixin Layer means nesting of mixin classes in a pattern in a way that the parameter of outer mixin evaluates the super classes of inner mixins. In this way, it addresses the problems of scalability thus resulting in developing the inventive collaboration based designs.

This paper studies mainly about the problems faced by a new programmer but not for the expertise. Additionally the problems discussed here are never encountered by a compiler. Mainly the concern is all about the design issues and how they can be solved using the mixin implementations. The design issues which we discuss are mainly raised while interacting with the system.

Let us now discuss about the origination of Mixin Programming. This Mixin Programming is first used in LISP language with its object systems like flavors, COLS. Multiple inheritance is the feature where Mixin programming turns to be quite useful. Here in this context, the Mixing Programming helps in extending the state or behavior without defining it formerly. This will be very effective as such a single class is sufficient for extending the state or behavior of many classes. Using parameterized technique these Mixins are implemented. The behavior can be extended dynamically at the time of instantiation. Here is the structure:

```
template <class Super>
class Mixin : public Super {
. /* mixin body */
};
```

For explaining in detail we consider an example of “Operation Counting” in a graph which counts and store the details of visited edges and nodes in a graph during the process of execution. Here in this example, we take operation counting as mixin. This mixin can be implemented all over the classes in the artifact that have same implementation. Here we can consider two different types of mixins for undirected and directed graphs. As previously said, a generalized design can be obtained with these mixins. The functionality will not be changed and will be very much helpful in obtaining a collaboration design. This is an incremental approach.

Mixin layers is a process of encapsulation of multiple classes and refining them incrementally. The basic structure of a mixin layer is as follows:

```
template <class NextLayer>
class ThisLayer : public NextLayer {
public:
class Mixin1 : public NextLayer::Mixin1 { . };
class Mixin2 : public NextLayer::Mixin2 { . };
};
```

The composition of extensions is the main object or goal of the mixin programming. Mixin layer is the mixture of the functionality of an object or the parts of the object. This also may contain the functionality of different objects and also specify clarification to each of them. All such clarifications developed incrementally can be enclosed in a single mixin layer. This mixin layer technique in particular will be very much helpful in designing the collaboration design patterns. By applying this mixin layer, inheritance can be applied at two different level. A new application of inheritance concept is achieved with this mixin layer process. At first, the layer inherits all the classes from its super class and in the second step it inherits all the remaining methods, attributes, variables from the matching inner classes of that layer. Thus implementing a new way of inheritance.

Let us now consider an implementation of a graph traversal application to explain the collaboration based design using mixin layer. In this application, three different algorithms are implemented on a undirected graph using a depth first traversal. The following are the different components in this graph. For making sure that the graph is cyclic, “Cycle checking” is used. For numbering the nodes, “vertex numbering” is used. The connected graph regions are classified by “Connected Regions”. Describing of graph properties is done in “Graph class”. The “Vertex class” takes care of each node and every node is an instance of this class. Every graph operation is taken place “Work Space”.

We decompose this total application in to five different parts. They are described as follows:

1.Operation of undirected graph

2.Put into code depth first traversals and rest of the three contains the precise of algorithms that are used here in the application.

Each object here does a different functionality and can perform several tasks. For example let us consider the collaboration of “Vertex Numbering”. The generalized way is shown here:

```
template <class Next>
class NUMBER : public Next {
public:
class Workspace : public Next::Workspace {
. // Workspace role members};
class Vertex : public Next::Vertex {
. // Vertex role members};};
```

We can observe that this mixin layer component is satisfying the property of reusability which is very much helpful in reducing the costs of developing the software product. Flexibility is the other property which can be observed.

Let us now discuss about some pragmatic considerations that are considered in this article. Below are the list of pragmatic considerations:

1.Lack of template type checking

2.Synonyms for compositions

3.Designating virtual methods

4.Single mixin for multiple uses

5.Hygienic templates in C++ standard

6.Compiler Support

Templates are typically not the part of C++ programming language. For this reason, type checking is not done till the instantiation time. Methods that are present in this template classes are mark themselves as function templates. But in C++ language, these function templates are instantiated automatically. But all the methods that are present are not type checked as some of them might not be referenced by the objects and so errors may remain in these methods

In C++ programming language, there is a draw back in implementing the concept of inheritance. We cannot inherit the constructor methods that are present in the program. The reason given for this is, for initializing the member functions and variables of a subclass, the constructor method of a super class may not be sufficient. But for, mixin classes there is no necessity for modify the data members. So, there should be a feasibility to inherit but we cannot perform this action in C++ language.

Synonyms for mixin classes can be given using typedef statements. This property will be very much beneficial when developing complex mixin layers and classes. The following is the syntax with which we can give synonyms:

```
typedef A < B < C > > Synonym;
```

In addition to this approach, we can introduce an empty subclass. This is helpful to preserve the properties of the language. Below is the syntax for creating subclass:

```
class Synonym : public A < B < C > > { };
```

Additionally, while using mixin classes in C++, there is a problem while creating virtual methods. Conflicts arise between the super class and sub class while creating these virtual classes. C++ allows super class to declare methods as virtual but it does not allow subclass to approve it and for this reason conflicts may arise between these two. Below is the syntax for creating the virtual methods.

```
template <class Super>
struct MixinA : public Super {
void virtual_or_not(FOO foo) { . }};
struct Base1 {
virtual void virtual_or_not(FOO foo) { . }
. // methods using “virtual_or_not”};
struct Base2 {
void virtual_or_not(FOO foo) { . }};
```

C++ programming language doesn’t provide a proper type checking action. At times this might be beneficial when both the interfaces are very much

similar. For this type of cases, a single mixin may be sufficient as it can take care of both the interfaces as their implementations are closer to each other.

C++ Programming Language has imposed several restrictions for avoiding naming conflicts that may appear in the templates. However, the usage of these rules by the compiler may vary accordingly. But there must be stabilized way of implementing it. This is because the template doesn't have any information about the restrictions. Sometimes, the author may get confused because of the conflicting opinions about these restrictions.

Finally, the compiler role is discussed here in the mixin programming. Nested classes and parameterized inheritance is considered by many of the compilers that use C++. There are some limitations when there are error checking and debugging cases. But this mixin programming is not much complex than the regular template.

Finally, we can say that this mixin programming may bring a lot of changes in implementing the C++ programming language. As said, mixin programming implementation makes the component reusable, less complex thus saving a lot of time and cost. Even this may not require most of the expertise knowledge. Even though there are some sensible aspects with the mixin programming, but the implementation promises feasible and effective implementations.

REFERENCES

{1} J. des Rivieres, and D. G. Bobrow, The Art of the Metaobject

Protocol. MIT Press, 1991.

{2} U. Eisenecker, "Synthesizing Objects", ECOOP 1999, 18-42.

{3}U. Eisenecker, Languages Conference (JMLC'97), LNCS 1204, Springer, 1997, 351-365.

{4} B. Stroustrup, the Annotated C++, 1990

{5} V. Singhal, A Programming Language for Writing Domain-Specific Software System Generators, Ph.D. Dissertation, Dep. of Computer Sciences, University of Texas at Austin, August 1996.