Towards a Software Metric for Generic Programming Paradigm*

Norbert Pataki
patakino@elte.hu
Dept. of Programming Languages and Compilers,
Fac. of Informatics,
Eötvös Loránd University
Pázmány Péter sétány 1/c.,
H-1117 Budapest, Hungary

Krisztián Pócza
kpocza@kpocza.net
Dept. of Programming Languages and Compilers,
Fac. of Informatics,
Eötvös Loránd University
Pázmány Péter sétány 1/c.,
H-1117 Budapest, Hungary

Zoltán Porkoláb
gsd@elte.hu
Dept. of Programming Languages and Compilers,
Fac. of Informatics,
Eötvös Loránd University
Pázmány Péter sétány 1/c.,
H-1117 Budapest, Hungary

Abstract

Since McCabe's cyclometric measure, structural complexity have been playing an important role measuring the complexity of programs. Complexity metrics are used to achieve more maintainable code with the least bugs possible.

C++ Standard Template Library (STL) is the most popular library based on the generic programming paradigm. This paradigm allows implementation of algorithms and containers in an abstract way to ensure the configurability and collaboration of the abstract components. STL is widely used in industrial softwares because STL's appropriate application decreases the complexity of the code significantly.

Many new potential errors arise by the usage of the generic programming paradigm, including invalid iterators, notation of functors, etc.

In this paper we present many complexity inconsistencies in the application of STL that a precise metric must take into account, but the existing measures ignore the characteristics of STL. We present proposal for a metric which can measure STL-based code too.

1. Introduction

Structural complexity metrics play important role in modern software engineering. However, the software metrics depend on used paradigm [10]. This fact makes hard to create multiparadigm metrics.

Generic programming is one the most untended paradigm from the view of paradigm, because most languages do not support this feature. C++ is multiparadigm language that suppport this paradigm [12]. Most important incarnation is the C++ Standard Template Library.

The C++ Standard Template Library (STL) is the most popular library based on the *generic programming paradigm* [1]. STL is widely-used, because the library is the part of the C++ Standard [12]. It consists of many useful generic data structures and generic algorithms, that work together with containers. STL is based on generalization and generalization results in simplified interface.

C++ STL consists of three main parts: containers, iterators and algorithms. Containers (e.g. vector, list, map, set, etc.) are the generalization of arrays, so they hold elements. Iterators guarantee access to the elements in containers. Iterators are nested types of containers. Iterators are a generalization of pointers, their standard interface originates from pointer-arithmetic. Algorithms are fairly irrespective of the used container, because they work with iterators. For instance, we can use the *for_each* algorithm with all containers. The complexity of the library is greatly reduced because of this layout. As a result of this layout we can extend the library with new containers and algorithms simultaneously. This is a very important feature, because object-oriented libraries do not support this kind of extension. The C++ standard guarantees the complexity of the operations.

STL applies the generic programming paradigm, so we can expect that the common metrics can fail on this library because of the metrics' paradigm-dependence. As we will see, the old metric tools are not precise enough.

2. Positive effects

STL is a popular library, because it greatly reduces the complexity of a program from the view of programmers. The library offers many positive effects to code, but some

*Supported by GVOP-3.2.2.-2004-07-0005/3.0

of these effects cannot be measured by widely-used metrics.

STL makes the code more abstract, more powerful, more expressive, so programmers can avoid many mistakes [5]. STL is a standard library, many books and online references can be found (for example [1, 5, 12]).

3. Trivial inconsistencies

Many inconsistencies can be found between the common metrics and uage of STL. Some of these inconsistencies are quite clear.

One of the most obvious inconsistency is the widely-used object oriented metrics fail on C++ Standard Template Library, because this library is based on generic programming and implementing classes is unnecessary. Of course, we use objects and classes when the STL is applied, but we can write STL-based code without any new classes. Hence, the object-oriented metrics may fail on STL-based programs.

Another important feature is that STL is standardized library, so names of functions and classes in the library are well-known. The names express their behavior, for instance the *copy* algorithm copies elements, the *sort* algorithm sorts a container, etc. No external library can achieve this important feature, and no existing metric can measure this special advantage.

STL has been designed as a generic programming library, so STL has a reduced interface: algorithms can be applied to more container types. The basic usage of the library is easy of attainment because of the reduced interface. This is a good feature, because beginner programmers do not shy away from STL. But this point is also not measured.

4. Complexity inconsistencies

In this section we examine some more sophisticated problems.

4.1. Error diagnostics

Error diagnostics usually do not matter when measuring software complexity. Metrics ignore syntactical and semantical errors in the code and usually examine programs as error-free software.

A simple mistake in STL-based code causes very long and incomprehensible error diagnostics. For example, more thousand character long error messages are not rare and often refer to unknown and unseen types and objects. Sometimes the errormessage points to the implementation of STL.

Some software tools help us to reduce the complexity of the messages, but these tools depend on the compiler and STL implementation.

Modification or maintain of STL-based code can be more difficult because of the complicate error diagnostics, so we should take it into account.

4.2. Functors

C++ functors are special objects that offer an *operator()* to simulate functioncalls. Functors are quite common objects in STL-based code, because functors can avoid the overhead of non-inline functioncalls and some problems about the name of template functions to get the code to compile.

The problems of functors are their special requirements. Functor classes are often inherited from special classes that only support some typedefs. The names of these base classes are *unary_function* and *binary_function*. These base classes do not increase the complexity of a functor from the viewpoint of STL programmer.

Functors are always passed by value. Polymorphism and value passing an object do not work together, because the object would be sliced. So, polymorphic functors are not allowed.

4.3. Sorted ranges

Many problems arise from the inadequate usage of sorted ranges. Some algorithms have a special precondition, e.g. the input range must be sorted (for example, *binary_search*, *equal_range*, *set_union*, etc.). But the compilers do not know what "sorted range" means, so the compiler cannot help us at this point. If we call an algorithm of this kind to an unsorted range, it causes undefined behavior. Unfortunately STLlint [14] cannot discover the improper usage of these algorithms. Using this kind of algorithms increases the complexity of the code.

Using the same sorting predicate to the sort and algorithm is important. If anyone violates this constraint it also leads to undefined behavior.

4.4. Dataflow

Dataflow models measure by the parameter-passing. This means the complexity of a program is based on parameters: how to read or write the arguments.

A basic problem is that we cannot read all parameter-flows from an STL-based code. For example we write a functor and we call an algorithm with this functor as an argument. It is invisible that the code will execute the functor's functioncall operator.

Another problem is that we cannot decide if an algorithm modifies the container. For instance, let us consider the following two declarations. The find algorithm does not modify the container, but the sort algorithm does:

```
template <typename
            InputIter,
          typename T>
InputIter find(InputIter first,
               InputIter last,
               const T& t);

template <typename RanIter>
void sort(RanIter first,
          RanIter last);
```

On the other hand, the parameters are not independent. A container is passed by two iterators that define the range. If we call an algorithm usually call it with special iterators: begin and end iterators. It is so common that the programmers cannot make a mistake. So, iterators as parameters are very closely to count them twice.

4.5. Invalid iterators

Probably the most serious problem is usage of invalidated iterators. The compilers cannot help solve this kind of problem. Many kind of errors arise from usage of invalid iterators.

Different containers have different observance of iterator invalidating. The most trivial example of iterator invalidating is a reallocating vector, because their iterators do not point proper element of the given vector after a reallocating method.

This does not mean that the standard node-based containers are preferrel to contigous-memory containers. Both have advantages and disadvantages. C++ Programmers should know the rules of invaliding iterators.

4.6. Lack of inserter iterators

The problem of lack of inserter iterators also lead to undefined behavior. The compiler does not assist to avoid this problem. This problem arises when an algorithm should add elements to a container without using inserter iterators.

For instance, the following code leads to runtime-error, because the code does not use inserter iterators:

```
int f(int x)
{
// ...
}
// ...
deque<int> src;
deque<int> dest;
transform(src.begin(),
          src.end()
          dest.end(),
          f);
```

In the code above we wrote values to uninitialized memory: because after the dest.end() there is no necessary initialized space. When the algorithm writea to this space program's behavior would be undefined. We can surmount this problem with inserter iterators:

```
transform(src.begin(), src.end(),
          back_inserter(dest), f);
```

Or we want to add elements to the begin of container:

```
transform(src.begin(), src.end(),
          front_inserter(dest), f);
```

4.7. Containers of auto_ptrs

This problem is related with portability. Because the C++ standard forbid containers that hold auto_ptr (the smart-pointer type in the standard library) because of the auto_ptr's strange method of copy. Unfortunately, some compilers concede the usage of containers of auto_ptrs and programers wants to avoid memory-leaks. But these programes are not portable, because many compilers keep the prohibition.

5. Some proposals

In the paper [9] a multiparadigm metric is described. AV-graph measures three main points of a given program: the structure of the program, the dataflow in the program, and the complexity of the used data structures.

We have seen that the dataflow model is not precise enough. Informally speaking, the control structure also fails on STL-based code, because the usage of STL replaces many loops and if-statements.

It is also a common problem what can we mean by complexity of the STL's data structures. The complexity cannot be an STL implementation-specific value.

Complexity of STL's data structures should be based on some "semantical concepts": for instance, basic behavior of the container (e.g. vector's reallocting strategy), special parameters of a data structure, how copying works, etc.. The previous inconsistencies are should be weighted and taken into account. The weighting should be based on the rate of made mistakes that can be realized from source of industrial application.

6. Conclusion

C++ Standard Template Library is a widely-used library based on the generic programming paradigm. Software metrics are mostly paradigm-dependent, so we can expect that the common metrics fail on C++ STL. In this paper we

present many inconsistencies between STL and the widely used metrics. Our aim is to calibrate an old metric to measure STL-based code.

References

[1] Austern, M. H.: *Generic Programming and the STL.* Addison-Wesley (1999)

[2] Chidamber, S.R., Kemerer, C.F., A metrics suit for object oriented design, *IEEE Trans. Software Engeneering*, vol.20, pp.476-498, (1994).

[3] Howatt, J.W., Baker, A.L.: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting, *The Journal of Systems and Software 10*, pp.139-150, 1989

[4] McCabe, T.J., A Complexity Measure, *IEEE Trans. Software Engineering, SE-2(4)*, pp. 308-320, 1976

[5] Meyers, S.: *Effective STL.* Addison-Wesley (2001)

[6] Pataki, N., Porkoláb, Z., Istenes, Z.: Towards Soundness Examination of the C++ Standard Template Library, *In Proc. Electronic Computers and Informatics*, ECI'06, Herl'any, 2006.

[7] Piwowarski, R.E.: A Nesting Level Complexity Measure, *ACM Sigplan Notices, 17(9)*, pp.44-50, 1982

[8] Porkoláb, Z., Sillye, Á.: Comparison of Object-Oriented and Paradigm Independent Software Complexity Metrics, ICAI'04, Eger, 2004

[9] Porkoláb, Z., Sillye, Á.: Towards a multiparadigm complexity measure, *In. Proc of QAOOSE Workshop, ECOOP*, Glasgow, pp.134-142, 2005

[10] Seront, G., Lopez, M., Paulus, V., Habra, N.: On the Relationship between Cyclomatic Complexity and the Degree of Object Orientation, *In Proc. of QAOOSE Workshop, ECOOP*, Glasgow, pp. 109-117, 2005

[11] Sipos, Á., Pataki, N., Porkoláb, Z.: On Multiparadigm Software Complexity Metrics (extended abstract), *In Proc. of 6th Joint Conference on Mathematics and Computer Science*, Macs' 06, Pcs, 2006

[12] Stroustrup, B.: *The C++ Programming Language*. Special Edition. Addison-Wesley (2000)

[13] Szab, Cs., Samuelis, L.: The A-Shaped Model of Software Life Cycle *In Proceedings of 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics*, Poprad, 2007, pp. 129-135, ISBN 978-963-7154-56-0

[14] Gregor, D.: *STLlint* `http://www.cs.rpi.edu/~gregod/STLlint/`