# Alternative Generic Libraries

**Zoltán Porkoláb, Róbert Kisteleki**

Department of Computer Science, Eötvös Loránd University of Sciences
`zoltan.porkolab@elte.hu, robert.kisteleki@elte.hu`

**Abstract**

Multi-Paradigm design is one of the emerging paradigm in software technology. Multi-Paradigm helps the designer to choose the rigth paradigm for each problem domain. In one specific case of variable problem scenarious where the objects we use has little or no common structure but there is similar behaviour. In this cases we successfully use the paradigm of *generic programming*. The goal of this style is to reveal the foundations and programming methods of generic and therefore reusable components and libraries.

Until quite recently generic programming has only one well-known existing implementation: the *C++ Standard Template Library (STL)*. STL has a certain goal: there are data structures and specific algorithms working on them. This is a very specific problem-domain which can affect the implementation. We think this one implementation of the generic programming is not enough to examine this new paradigm. Therefore we try to define a set of demonstrative generic libraries, implement them as working software, and making comparison on the internal structure: select common patterns and leave problem or implementation-specific parts.

We present in this article the *Graphical Template Library (GTL)*. GTL works on graphical shapes like circle or polygon, implementing common functions like move, rotate, etc. The idea of GTL comes from real-world applications. Graphical drawing programs can benefit from this kind of design.

One of the main advantages of the use of generic programming is the significant reduce of *code complexity*. The average library has n objects on k basic types with m algorithms, has O(n*k*m) methods developed by object-oriented way, but this can be redused to O(n*m) with generic paradigm. *Iterators* are in central role of generic programming. Iterators are the fundamental program elements connecting data structures and algorithms. STL has a certain hierarchy of iterators. In GTL we extend the set of iterators, but we insert them into STL's hierarchy. The new iterators have practical usage in generic libraries and parallel environments. These iterators need further researches.

## 1 Introduction

Nowadays, new programming paradigms came into view in software technology. Multi-Paradigm Design [Coplien99] by James O. Coplien and others is focusing to help the designer to create abstraction for arbitrary domains. Object-oriented design arms the designer with tools that produce modules of a certain shape; as long as the problem domains lends itself well to object-shaped abstractions, the object paradigm works well. However, some problems have little to do with objects. Multi-Paradigm steps above any single paradigm to help the designer choose the right paradigm for each project domain.

Among of different programming paradigms the most interesting new paradigm is *generic programming.* The goal of this style is to reveal the foundations and programming methods of generic and therefore reusable components and libraries. In the terms of Multi-Paradigm theory we use generic programming in those cases when the *objects* has no or little common structure but the *behaviour* (the methods used on objects) are similar. Using this approach we can greatly reduce the complexity of a software library. For example, if we have a library with $n$ data structures, each with $k$ base type and $m$ algorithms, then using the traditional object-oriented way the comlexity of the library is *O(k*n*m).* Using generic programming this reduced to *O(n+m).*

The software library designs that have resulted from this generic programming approach are markedly different from other software libraries: the precisely-organized, interchangeable building blocks that result from the approach permit many more useful combinations that are possible with more traditional component designs. The design is also a suitable basis for further development of components for specialized areas such as databases, user interfaces, and so on. By employing compile-time mechanisms and paying due regard to algorithms issues, component generality can be achieved without sacrificing efficiency. This is in sharp contrast to the inefficiences often introduced by other C++ library structures involving complex inheritance hierarchies and extensive use of virtual functions. The bottom-line results of these differences is that generic components are far more useful to programmers, and therefore far more likely to be used, in preference to programming every algorithmic or data structure operation from scratch.

On the other hand, generic programming has only one well-known existing implementation: the *C++ Standard Template Library (STL).* Because in any implementation, the elements of the general principles and the particular programming language inevitably blend, one implementation is not enough to examine the structure of generic libraries and programming.

To help this situation we created an (experimental) template library: the *Graphical Template Library.* GTL manipulates *graphical objects* (circle, polygone, text, etc.) using *generic algorithms* (move, rotate, mirroring, magnificate, etc.) similar way as STL does it with containers and algorithms. Like in STL, graphical objects has no one common base class, and we intentionally implemented the objects internally in different ways. With the help of GTL we can examine the internal structure of generic programming.

## 2 The intention of creating alternative generic libraries

Our goal is to examine the deep structure of generic programming. This can be done in two elemental way: the theoretical one and the practical one. With the respect of the first way [Hermann98] [Kistel99] in this article we present the second case, choosing existing software products and try to analyze them. This includes to collect common features and leave the customized, problem- or implementation specific elements.

In computer science, practice often anticipate theory. A good example is Simula67 programming language. Simula67 language had become popular before major theoretical work has been done on object-orientation. Hovewer, studying Simula67 we can recognize that next to object-oriented elements (class, inheritance, etc.) there are co-rutines and other similar stuff. Those have nothing to do with object-orientation, they are the problem-specific part of the language created to solve simulation tasks. Studying *only* Simula67 does not help to classify the features.

We feel similar situation with generic programming. Until quite recently generic programming has only one well-known existing implementation: the *C++ Standard Template Library (STL)* [Musser98] (and it's derivates: Java Collection Framework, ObjectSpace's JGL [JGL], etc.). This library has a certain goal: there are *data structures* and specific *algorithms* working on them. We think this one implementation of the generic programming is not enough to examine paradigm. Therefore we try to define a set of demonstrative generic libraries, implement them as working software, and making comparison on the internal structure: select common patterns and leave problem or implementation-specific elements.

## 3 The GTL Library

The experimental library we will discussed in this article is the GTL, the *Graphical Template Library.* GTL manipulates *g*raphical objects (circle, polygone, text, etc.) using *generic algorithms* (move, rotate, mirroring, magnificate, etc.) similar way as STL does it with containers and algorithms. Like in STL, graphical objects has no one common base class, and we intentionally implemented the objects internally in different ways.

This is based on the Multi-Paradigm theory, where we can distinguish between different problem-domains. In those projects where acting objects have a *common structure* and have (slightly) different behaviour, we choose object-oriented method to stress the commonality in objects. In other cases we have no such commonality in data structures – so choosing a common base class is not practical. However *common behaviour* can be expressed using common (generic) algorithms.

The back-door intent in the definition of GTL was:

- define an easy-to-understand but not trivial library, where the benefit of using the idea of generic programming is clear. As we shown previously, this depends on the common behaviour of objects with different internal structure.
- keep the problem within limits to minimize the implementation-specific elements. Implementing large libraries neccessarry deform the original idea.
- specify a real-world problem. This is essential if we want to avoid *l'art pour l'art* design, and want to concentrate on practical solutions. The library we defined can be act as the engine behind a real graphical drawing tool (like *xfig* in UNIX/Linux).

About the points above: the whole library is about 800 C++ lines. A set of example client programs demonstrate the usage of the library. Future plans include creating graphical representation in the style of the *xfig* utility.

## 4 Complexity

One of the major advertized advantages of generic programming is the capability to *reduce* the amount and complexity of *c*ode. let examine this question with the help of the Graphical Template Library. In object-oriented textbook examples graphical "shapes" are derivated from one common root basic class. Algorithms are mostly *virtual* methods on this base class. The root class is often *abstract* as some methods are specific to the certain graphical shapes and impossible to implement generally (they are *pure virtual* functions). In this pattern we write code for (most of) the methods at the derived class level. This produces complexity of *O(n*m)* level. This sceme is shown in the next code example:

```
class graphical_base
{
public:
    virtual void move( /* ... */ ) = 0;
    // other virtual functions as algorithms
private:
    // some common structure
};


class a_certain_shape : public graphical_base
{
public:
    void move( /* ... */ )
    {
        // the concrete implementation of i
        // an algorithm on a certain shape
    }
private:
    // the implementation of the certain_shape
};
```

In GTL hovewer, there is no common base class. Each graphical class defines *iterator* classes (at least input and output iterators), and iterators carries data to algorithms. This reduces complexity to *O(n+m)*.

The problem with this model is the lack of the common base class and inheritance hierarchy. Sometimes the common base class is essential. That is the foundation to collect data structure in a type-safe way. Therefore it is important to mix the two model, to win the reduced complexity from the generic programming model and keep the inheritance hierarchy from object-oriented model.

Therefore we propose a new model, a kind of object-oriented wrapping of the generic programming model. Let's define a common graphical base class with *non-virtual* methods as the algorithms. Methods are realized with the help of virtual *iterator-functions* defined on the base class, but implemented on the derived classes (so the base class remains *abstract*). This model exploits the pattern of generic programming, but in the same time keeps the conventional object-oriented model.

```
class graphical_base
{
public:
    void move( vect distance )
    {
        // implementation of the concrete algorithm
        outp_iter() = inp_iter() + distance;
    }
    // other concrete functions as algorithms
protected:
    virtual vect  inp_iter() const = 0;
    virtual vect& outp_iter() = 0;
private:
    // some common structure
};


class a_certain_shape : public graphical_base
{
public:
    vect inp_iter() const
    {
        // each call produces next call
    }
    vect& outp_iter()
    {
        // each call produces reference to next node
    }
private:
    // the implementation of the certain_shape
};
```

The complexity of code is the same as the case of GTL. However, there still remains a practical drawback against GTL: if we want to implement a new algorithm, we should modify the base class, which is not necessary in GTL. (Interestingly, adding a new graphical class does not involve the modification of existing classes. Just create a derived class, and implement the iterator functions.)

## 5 Common base class

In the previous section we discussed the role of the common base class in a certain aspect. There could be other arguments too. In most of the graphical drawing tools there is a possibility to grouping objects, somehow glue them together and then handle the group as a single object. How can one implement this feature in GTL?

There is an elegant way. A new graphical class: *composit* can collect other graphical objects. The iterator of the composite can be class implemented in the following way: each iteration supplies a participant object's iterator. Here an algorithm, like *move()* called on a composit object will roam all the group members, calling move on the certain elementary graphical class. The "recursion" (this is not really recursion, rather recursive resolvance of template function parameter) will stop when it arrives to the specialized *move()* function on *node*. These resolvings happen mostly compile-time, so there is no run-time cost of this design.

Hovewer there is an important point here. Inside the composite object we should aggregate the iterators of the (elementary) member objects. To enable to store the iterators of different elementary graphical objects in type-safe way requires a *common base class of the iterators.* This sounds correct, since there is a common behaviour of these iterators, they are e.g. *forward* iterators, they can fulfill a common interface.

## 6 Iterators

Iterators has a very strict classification in STL [Musser98]. There are input or output iterators, forward iterators, bidirectional iterators and random access iterators. Each iterator class has the functionality of the previous one - namely bidirectional iterators are also forward iterators etc. This is very similar to an inheritance relationship. Why STL iterators has not been derived from a hierarchy of abstract classes (interfaces)? The example in the previous section shows the advantage of that construction.

GTL has different iterators. Since we work with both open and closed shapes, we have forward iterators, bidirectional iterators, cyclic iterators and random iterators. Cyclic iterators use the fact that for closed shapes (like a polygon) the *begin()* and the *e*nd() iterators supply the same values. As we can calculate a regular polygon's node coordinates based on two arbitrary node coordinates, regular polygons implement random iterators.

Based on the experiences we have got during the design of GTL we think the collection of possible iterator classes are much wider than we have seen it in STL. We can imagine iterators, which randomly choose data from the container (never choose the same item twice) until the container is "exhausted". Such iterators could be used in parallel way in certain algorithms (*for_each()*, or GTL *move()*).

## 7 Algorithms

Algorithms are another basic element of generic programming. Algorithms are working with iterators, independent from the concrete data structures. Our examination is that the generic algorithms have common architecture: evaluate the iterator from a starting point until the iterator exhausted. Sometimes this later event detected as we read an extream iterator value. This is the way as STL works. However other solutions can work also perfectly: in GTL we detect the end of iteration reading the starting iterator element in secound times.

Not mention these small problem-oriented differences, algorithms in STL and GTL are the same. In the next example we present the solution of *for_each()* algorithm in STL and in GTL.

```
template <class InputIterator, class Function>
Function for_each( InputIterator first,
                   InputIterator last,
                   Function f)
{
    for ( ; first != last; ++first )
    {
        f( *first );
    }
    return f;
}


template <class InputIterator, class Function>
Function for_each( InputIterator first,
                   Function f)
{
    InputIterator save = first;
    f( *first++ );

    for ( ; first != save; ++first )
    {
        f( *first );
    }
    return f;
}
```

## 8 Results and future plans

GTL now is implemented as fully working library. We learned a lot about generic grogramming and the connection with object-orientation. As it turned out, such light-weight generic libraries are very useful for educational purposes too. We proved (at least for ourselves) that generic programming is not a one-library tool (STL), but a universal paradigm capable to solve problems for a certain class.

Containers in a solution using generic programming paradigm are rather problem specific. There were no similarity between STL and GTL containers. However, general algorithms were surprisingly similar. We identified new types of iterators, and we could extend the iterator-hierarchy of STL. We expect much more type of iterators. We used function-objects, predicates, etc, as technical helpers. We didn't used adapters, but we will likely do in the next versions of GTL.

There are four major directions we plan to continue our examinations.

- Extend GTL, and implement the *composite* class and other features. Create GUI interface and complete the program as a drawing tool.
- Specify new problems to suit the generic programming paradigm. In fact we found this harder then we expected.
- Examine the connection between object-orientedness and generic programming. This includes the implementation of GTL in object-oriented way with virtual iterator functions as described under the Complexity section. The two patterns should be compared in the aspect of complexity, ease of modification and efficiency.
- Examine parallelism in generic programming. Recent implementations known to us lack parallelism. Hovewer some algorithms like *for_each* could gain from massive parallelism. Per-

haps parallelism depends on new type of iterators, which are able to enumerate data in non-seqencial way.

## References

[Coplien99] Coplien, J. Multi-Paradigm Design for C++, 1999, Addison-Wesley

[BreyHugh95] Breymann, U. and Hughes, N. Composite Templates and Inheritance. C++ Report, No.7, Sept 1995, 32-40

[Musser98] Musser, D. R., Saini, A. STL Tutorial and Reference, 1998, Addison-Wesley

[Strou97] Stroustrup, Bjarne. The C++ Language 3rd ed., 1997, Addison-Wesley

[Lippman98] Lippman, S. B., Lajoe, J. C++ Primer 3rd ed., 1998, Addison-Wesley

[Breymann98] Breymann, U. Designing Components with the C++ STL, 1998, Addison-Wesley

[MusStep94] D. Musser, A. Stepanov, Algorithm-Oriented Generic Libraries Software, Practice and Experience, vol. 24(7), July 1994.

[MusStep89] D. Musser, A. Stepanov: Generic Programming, Invited paper, in P. Gianni, Ed. ISSAC'88 Symbolic and Algebraic Computation Proceedings, Lecture Notes in Computer Science, Springer-Verlag, vol. 358, 1989.

[Erlin96] Erlingsson, Kostantinou, Implementing the C++ Standard Template Library in ADA, Rensselaer Polytechnic Institute, Jan 1996.

[JGL] Java Generic Library. http://www.objectspace.com/products/jglOverview.htm

[Breymann95] U. Breymann and N. Hughes, Composite Templates and Inheritance, In C++ Report, No. 7., pages 32 - 40., Sept 1995.

[Hermann98] G. Hermann, Examination of Generic Programming, M.Sc. thesys, 1998. (in Hungarian)

[Kistel99] R. Kisteleki, The Graphical Template Library, M.Sc. thesys, 1999. (in Hungarian)