STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

RECURSIVE AND DYNAMIC STRUCTURES IN GENERIC PROGRAMMING

ÁBEL SINKOVICS AND ZOLTÁN PORKOLÁB

Abstract. Generic programming – an emerging new programming paradigm – best known from Standard Template Library as an essential part of C++ provides an opportunity to extend libraries in an efficient way. Both containers (abstract data structures) and algorithms working on them could be independently designed and implemented at O(1) cost. Unfortunately, smoothless cooperation of generic programming and object orientation is still an open problem. In this paper we will focus on reducing development and maintenance costs of systems using generative programming with recursive data structures to avoid multiple implementations of the components. For cases when separate implementation of algorithms can't be avoided we provide a system protecting us against changing existing code during extension. Providing such a design is not a trivial problem using currently available tools. We will show a possible solution using a graphic library to demonstrate the problem and our solution with.

1. Introduction

In software development, working with recursive data structures is an ubiquitous problem. Graphic editors, web browsers, office software, etc. have to work with complex systems with sets of different (including recursive) component types. These software have to deal with algorithms operating on the components. The longest and most time and money consuming part of a software system's life is maintenance, and with poor design it is hard to maintain and extension is always expensive. Extending the system with new components, all algorithms have to be implemented for them. Extending the system

Received by the editors: September 14, 2008.

2000 *Mathematics Subject Classification.* 68N15, 68N19.

1998 *CR Categories and Descriptors.* D.2 [**Software Engineering**]: D.2.3 Coding tools and techniques – *Object-oriented and generative programming* D.3 [**Programming Languages**]: D.3.2 Language Classification – *C++*;

Key words and phrases. Generic programming, Software engineering, Expression problem, C++.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

with new algorithms, they have to be implemented for every component. Independent extension could make development and maintenance faster, more flexible, and cheaper. In this paper we will examine commonly used design patterns and will introduce a new one supporting independent development and extension in several cases.

The rest of the paper is organized as follows: In this section we present a practical example of the problem after which we analyse currently existing solutions in section 2. We present our solution in section 3 and use it to solve the practical example in section 3.3. We analyse runtime performance of our solution in section 3.4 and finally we summarize our results in section 4.

As a motivating example we chose a graphic application since it has every feature required to demonstrate the problem. Our sample graphic application supports different shapes and transformations. It is not uncommon to define such a system with at least 20 different shapes and 50 transformations. One of the suggested design methods [3] [10] [12] is using the Interpreter design pattern [15], which indicates to create an abstract base class called `Shape`, inherit all shapes from it and implement the transformations using virtual functions. The other suggested method [3] [10] [12] is using the Visitor design pattern [15], which indicates to create an abstract base class called `Visitor` for transformations with a virtual function for every shape type. For example if the system has `Oval` shape, `Visitor` should have a virtual function called `visitOval` accepting `Oval` shapes. Every transformation should be implemented in a different class inherited from `Visitor` and implement the virtual functions.

The example above refers to a well-known scaling problem of object-oriented library design. Philip Wadler called it the expression problem on the Java-Genericity mailing list for the first time in 1998 [13]. Given a set of recursive data structures and a set of operations operating on the data structures and a design is required which supports extension of both data types and operations independently. Extension (or modification) of one set should not affect the other and should not imply changes in existing code.

Zenger and Odersky presented a list of requirements [3] for a solution which we have extended with an extra item. Our main goal is to find a design which **(1)** is extensible with algorithms (transformations in the example) and data types (shapes in the example). **(2)** is independently extensible. Extensions shouldn't imply changes to previously written code. None of the data types should be changed because of writing a new operation and none of the operations should be changed because of creating a new data type. **(3)** is type safe. Validity of the operations are check at compile time eliminating runtime errors which can remain untested in practice causing embarrassing and commonly expensive issues in production. **(4)** is effective. When types

are available at compile-time, the efficiency of the compiled code should be similar to hand-written code allowing automatic optimalisation of the code. **(5)** supports separate compilation. Different components of the system (different data types and algorithms) can be compiled independently from each other. **(6)** supports the reduction of the number of implementations where possible by using generic algorithms to describe similar implementations. This is our extension to Zenger's and Odersky's list. Generic algorithms – first introduced in Ada – can support data types which are written independently of the algorithm, maybe later after the creation of the algorithm.

2. Existing solutions

Matthias Zenger and Martin Odersky collected a set of (partial) solutions for the problem in [3]. We will go through them and see their benefits and drawbacks. Structural and functional decomposition are the two most important ones since the rest of the approaches are extensions or improvements of them.

2.1. **Structural decomposition.** Structural decomposition uses the Interpreter design pattern [15]. It requires a base class from which every data type is inherited, and the base class has a pure virtual function for every algorithm. Every data type implements it's own version of the algorithm.

Extension with a new data type is easy, a new class implementing the data type need to be created. One of the disadvantages is that every algorithm has to be implemented for it, but the main problem with this solution is that every class has to be changed during extension with a new algorithm: a new virtual function has to be created in the base class, and it has to be implemented in every class inherited from the base class.

2.2. **Functional decomposition.** Functional decomposition uses the Visitor desing pattern [15]. There are no restrictions for data types. Each algorithm is implemented by a class with multiple member functions – one for every data type. The objects of these classes are called visitors, and the classes are inherited from a base class which has a pure virtual function for every data type. These pure virtual functions are overridden in the visitor classes to implement the algorithm for the data types. To run an algorithm for a data object a visitor object needs to be created and it's member function for the data object needs to be called.

Extension with a new algorithm is easy, a new class has to be created for the new algorithm. It has the same problem as structural decomposition: every algorithm has to be implemented for every data type. The main problem with this solution is that extension with a new data type is difficult: every visitor has to be extended with a new member function.

(1) Extensible visitors Krishnamurti, Felleisen and Friedman [2] extended the Visitor pattern [15] to be more flexible. They refined the way to introduce new data types: visitors don't need to be changed, the set of member functions can be extended by subclassing. The main problem with this solution is that it is still not type safe – it requires casting. Zenger and Odersky advanced the approach [17] by adding default cases to types and visitors to handle future extensions, but was still not satisfactory because of allowing application of visitors to data variants they were not designed for.
(2) External extension of classes Some programming languages [16] support external extensions of classes making possible extension of a class without changing it's code. Defining functions externally requires default implementations making separate compilation impossible.
(3) Reflection based approach Palsberg and Jay advanced the Visitor pattern [15] to Walkabouts [11] which use reflection to iterate over attributes of unsupported objects, but their solution has no static type safety because of using reflection.
(4) Self types Kim B. Bruce presented a way [10] of using a type construct called `ThisType` to advance the Interpreter design pattern [15]. `ThisType` changes in subclasses and the signature of inherited methods using `ThisType` ensure better static type safety, but the dynamic type of an object has to be available at compile time when calling a method expecting an object with `ThisType` as it's static type.
(5) Generics based approaches Mads Torgersen used generic classes [12]. He presented an approach based on structural decomposition (which he called data-centered) and one on functional decomposition (he called operation-centered). They were both difficult to use for programmers and did not support independent extensibility.

3. Our solution

We approached the problem with generic programming but in a different way Mads Torgersen did [12]: we rely on the term concept defined in [4]. A concept is a set of requirements for a class, and a class models the concept if it meets the requirements. These requirements can be syntactic, semantic, etc. Syntactic requirements will be supported in the upcoming standard of C++ called C++0x [19] [20] [21]. We assume the existence of a concept every data type (including recursive ones) models. A data type can be any class model the concept and an operation can be any function relying only on the concept.

An example for this in the C++ Standard Template Library [9]: a class models the forward iterator concept if it can read through a sequence of objects in one direction. A container is forward iterable if it provides a subclass modelling the forward iterator concept and reading through the elements of

the container. Algorithms use these iterators to access the containers (which they know nothing more about), they rely only on the concept.

These systems can be extended in non-intrusive way: new data types can be introduced by creating a new class modelling the concept, new operations can be introduced by writing new generic functions relying only on the concept. This solution is also efficient, since in most cases the compiler knows every type at compile time and can heavily optimise the program.

3.1. **Recursive data types.** We examine creation of recursive data types for this design. Recursive data types contain one or more data objects (modelling the concept) and are data objects themselves, so recursive data types model the concept as well. When the type of the child objects are known at compile time the interface of the children can be used directly: all types are known at compile time. When the types of the child objects are unknown at compile time the only thing the recursive object can assume is that they model the concept. Not only their dynamic but also their static type is unknown at compile time (there is no common base class for data types).

Mat Marcus, Jaakko Järvi and Sean Parent use the Bridge design pattern in [4]. The goal of this pattern is separation of abstraction (in our case the concept) and implementation (classes modelling the concept). They connect static and dynamic polymorphism by creating an abstract base class for every class modelling the concept and a generic wrapper class which is a subclass of the base class and can be instantiated by any class modelling the concept. The abstract base class provides pure virtual functions for every operation required by the concept and wrappers implement these virtual functions by using the wrapped object's interface since every wrapper knows the static type of the wrapped class at compile time.

Inheritance between the base class and wrappers implement dynamic, instantiation of the generic wrapper for each data type implements static polymorphism. Using this idea recursive data types could be implemented when static type of children is unknown at compile time using smart reference objects which could be special objects containing a pointer to wrapper objects and model the concept themselves by calling virtual functions of the wrappers.

Concepts requiring the existence of subtypes modelling another concept (e.g. STL containers need to have an iterator type [9]) make creation of the abstract base class more difficult: since the static type of the wrapped object is not known at compile time, neither does the compiler know the static type of the subtype. The open question is what type should the common base class provide as the subtype. Our answer to this question is repetition of the idea of Marcus, Järvi and Parent [4] for the subtype: the base class could provide the smart reference class to the real subtype as the subtype. For example the base class for STL containers could provide the smart reference class for

iterators as it's iterator type and instances of STL algorithms not knowing the static type of the container at compile time could access the iterators through smart reference objects. For example a container could be created accepting any random access STL container [9] using this solution. First iterators of these containers need to be wrapped, so a base class is required for random access iterators. The codes here are not complete classes, just examples demonstrating the logic of the solution, and to keep examples simple we assume that the container's elements are `int`s.

```
class RandomAccessIteratorBase {
public:
  virtual int operator*() const = 0;
  virtual int& operator*() = 0;
};
```

The wrapper template needs to be implemented for iterators:

```
template <typename T>
class RandomAccessIteratorWrapper :
  public RandomAccessIteratorBase {
public:
  RandomAccessIteratorWrapper(const T& t) : _wrappedObject(t) {}
  virtual int operator*() const { return *_wrappedObject; }
  virtual int& operator*() { return *_wrappedObject; }
private:
  T _wrappedObject;
};
```

Finally a smart reference class needs to be created simulating a random access iterator and calling a wrapper in the background. (We use `shared_ptr` from Boost [18] as an underlying smart pointer implementation). We focus on the core idea here and skip other parts (e.g. copy constructor for `RandomAccessIterator`) which a real implementation has to deal with.

```
class RandomAccessIterator {
public:
  template <typename T> RandomAccessIterator(const T& t) :
    _wrapped(new RandomAccessIteratorWrapper<T>(t)) {}
  int operator*() { return _wrapped->operator*(); }
  int& operator*() const { return _wrapped->operator*(); }
private:
  boost::shared_ptr<RandomAccessIteratorBase> _wrapped;
};
```

Now since iterators have been wrapped the wrapper for containers can be created using the iterator wrapper:

```
class RandomAccessContainerBase {
public:
  virtual RandomAccessIterator begin() = 0;
  virtual RandomAccessIterator end() = 0;
};
template <typename T> class RandomAccessContainerWrapper :
  public RandomAccessContainerBase {
public:
  RandomAccessContainerWrapper(const T& t) : _wrapped(t) {}
  virtual RandomAccessIterator begin()
  { return _wrapped.begin(); }
  virtual RandomAccessIterator end() { return _wrapped.end(); }
private:
  T _wrapped;
};
class RandomAccessContainer {
public:
  template <typename T> RandomAccessContainer(const T& t) :
    _wrapped(new RandomAccessContainerWrapper<T>(t)) {}
  RandomAccessIterator begin() { return _wrapped->begin(); }
  RandomAccessIterator end() { return _wrapped->end(); }
private:
  boost::shared_ptr<RandomAccessContainerBase> _wrapped;
};
```

Every STL algorithm [9] for random access containers could work with these wrappers and accept any random access container – without recompiling the algorithm itself. It has a runtime cost but it still acceptable (we have implemented the motivating example using this and measured the runtime cost – see table 1 and table 2).

3.2. **Evaluation of our solution against the requirements.** We have a set of requirements (Zenger's and Odersky's list with an extension in section 3.3): **(1) Extensibility with algorithms and data types.** Algorithms can be added by implementing new generic functions, data types can be added by creating new classes modelling the concept. **(2) Independent extension.** Extension with a new generic function or a new class has no effect on data types or other functions. **(3) Static type safety.** Validity of calling a generic function on a data type is checked when the code calling the function is compiled. Data types have to model the concept and algorithms have to rely only on the concept. In case algorithms rely on a refinement of the original concept data types they are called with have to model that as well. When using unrestricted containers the type of the objects is checked

when the objects are placed in the container, therefore algorithms operating on the elements of the container can assume that every element models the concept. **(4) Efficiency.** When runtime type of data objects is known at compile time the compiler can optimise the code. **(5) Separate compilation.** When using unrestricted containers (or references to objects modelling the concept) algorithms can be compiled separately from data types they use. **(6) Reduction of the number of implementations** Generic algorithms do this.

3.3. **Using this idea for the motivating example.** We are going to use this solution for the motivating example and check how effectively can this approach solve the problem compared to the commonly used design patterns. [3] [10] [12] In the motivating example a set of shapes and a set of transformations operating on the shapes are given. Groups of shapes can be created which groups are shapes themselves, operations need to support them either. Shapes are the data types of the expression problem, groups of shapes make them recursive. Transformations are operations operating on the data types.

Since our solution requires a generic concept for data types, the first step of applying the approach to a practical problem is finding one. This concept needs to be generic enough to avoid restriction of the data types since changes in the concept are likely to indicate changes in every data type and operation. In our example data types are shapes, a concept has to be generic enough to describe all kinds of shapes. There are multiple approaches to find one, we use one we found generic enough here for demonstration.

First we define a concept for vectors: we expect a vector type to have a scalar type associated with it, the scalar values form a field and the vectors form a vector space over this field. For example the vector space of two (or three) dimensional vectors over the field of real numbers satisfy these requirements. A generic concept for shapes can be defined based on the generic concept for vectors. Commonly used shapes can be described by a set of vectors. Here are the shapes of the well-known vector graphics standard the Scalable Vector Graphics (SVG) format [27]: **(1) line** can be represented by it's two endpoints. **(2) triangle** can be represented by it's three vertices. **(3) Rectangle**s can be represented by two or three vectors (depending on if their edges are always parallel to the axis of the coordinate system or they can be rotated by any angle). **(4) Ellipse**s can be represented by their bounding rectangle, which indicates that they can be represented by two or three vectors. **(5) Circle**s can be represented by the origin and one point on the edge. **(6)** Paths and shapes described by them (**polyline**s, **polygon**s, **bezier curve**s, **spline**s) can be represented by their control points. As we can see there are shapes which support extension and reduction of the set of

their points (polylines, curves, etc.) and there are shapes where the number of points are fixed (rectangles, lines, etc.).

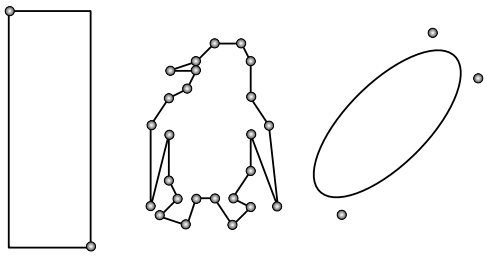

FIGURE 1. Example shapes and vectors describing them

A generic concept for shapes could be defined based on this idea: a shape is represented by a set of vectors completely describing it's shape, location, orientation and size. Each type of shape has a forward iterator type and has `begin()` and `end()` methods similarly to STL containers which can be used to iterate over the vectors representing the shape. Transformations could be implemented similarly to algorithms of the Standard Template Library: they are generic functions using iterators to access the shapes. Here is an example implementation of translation:

```
template <typename Shape>
void translate(Shape& shape, typename Shape::Vector d) {
  for (typename Shape::iterator i = shape.begin();
    i != shape.end(); ++i)
       *i += d;
}
```

Basic shapes like lines, curves, etc. could be implemented by containers of vectors. For example a rectangle or a line could be implemented by an array of vectors to provide O(1) random access to the vectors, a polyline could be implemented by a list of vectors to provide O(1) vector insertion and deletion. Groups of shapes could be implemented by containers of shapes, but they have to be shapes themselves. The union of sets representing the contained shapes could be the set of shapes representing the group itself as a shape since this is the set of shapes which satisfies the expectations of the abstract concept for shapes (completely describes the whole group). This indicates the creation of a special iterator iterating over the elements of the contained shapes. Unrestricted containers could be implemented by creating generic wrappers for shapes. Since the type of iterators is not fixed, generic wrappers have to be created for iterators either.

3.4. **Runtime performance.** We implemented the motivating example to measure the runtime performance of the solution. The test environment was a Linux box with the GNU C++ compiler version 4.1.2, and the code was compiled with level 6 optimalisation. We measured the speed of the translation of two dimensional points and polylines in homogeneous and in unrestricted containers. Using homogeneous containers dynamic type of the shapes are known at compile time making optimalisation possible but restricting flexibility while unrestricted containers accept any type of shapes but have runtime costs because of using dynamic polymorphism.

Table 1. Measurements with point shapes

		Unrestricted container (s)	Homogeneous container (s)
1 000	shapes 1 000 times	0.373	0.056
10 000	shapes 1 000 times	3.497	0.365
1 000	shapes 10 000 times	3.546	0.950

Table 2. Measurements with polylines

		Unrestricted container (s)	Homogeneous container (s)
100	shapes 100 times 100 control points	0.092	0.059
1 000	shapes 100 times 100 control points	0.675	0.367
100	shapes 1 000 times 100 control points	0.940	0.903
100	shapes 100 times 1 000 control points	0.901	0.884

The results are what we expected – runtime polymorphism has a strong impact on runtime speed (unrestricted containers were 3 - 6 times slower than homogeneous ones).

4. Summary

A large class of software is working on recursive data types. Web browsers, office software, graphic editors, etc. have different components containing other components, and perform operations on them. These software need to be designed carefully, since by applying commonly used design patterns, the possibility of independent development and extension of these components and operations could be lost. In this paper we analysed common patterns and found that they supported independent extension of one of data types or operations, but not both of them. We analysed other existing approaches as well to see their benefits and drawbacks. We proposed a new approach using generic programming in C++ and a solution when a concept is available for data types. In our approach data types are required to model the concept and algorithms required to rely only on the concept when accessing data objects.

A drawback of generic programming in C++ is the lack of support for runtime polymorphism which is required to create unrestricted containers for data objects supporting any data type. We used the technique described by Mat Marcus, Jaakko Järvi and Sean Parent in [4] to connect compile time and runtime polymorphism. We extended the idea with support to unrestricted containers. After measuring the runtime cost of unrestricted containers we found that although they were 3-6 times slower than homogeneous ones, but they are more advantegous in means of flexibility, type safety, and quality of source code.

References

[1] Thomas Becker, *Type Erasure in C++: The Glue between Object-Oriented and Generic Programming*, ECOOP MPOOL workshop, pp.4-8, Berlin, 2007.

[2] Shriram Krishnamurthi, Matthias Felleisen, Daniel P. Friedman, *Synthesizing Object-Oriented and Functional Design to Promote Re-Use*, LNCS Vol.1445, p..91-111, 1998.

[3] Matthias Zenger, Martin Odersky, *Independently Extensible Solutions to the Expression Problem*, Technical Report IC/2004/33 EPFL, Lausanne, 2004.

[4] Mat Marcus, Jaakko Järvi, Sean Parent, *Runtime Polymorphic Generic Programming - Mixing Objects and Concepts in ConceptC++*, ECOOP MPOOL workshop, Berlin, 2007.

[5] Ronald Gracia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock, *A Comparative Study of Language Support for Generic Programming*, ACM SIGPLAN Notices, Vol.38, Issue 11, OOPSLA conference paper, pp.115-134, Anaheim, 2003.

[6] Scott Meyers, *Effective C++*, Addison-Wesley, 2005, [220], ISBN: 0321334876

[7] Scott Meyers, *More Effective C++*, Addison-Wesley, 1996, [336], ISBN: 020163371X

[8] Scott Meyers, *Effective STL*, Addison-Wesley, 2001, [288], ISBN: 0201749629

[9] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1997, [1040], ISBN: 0201327554

[10] Kim B. Bruce, *Some challenging typing issues in object-oriented languages.* In Proceedings of Workshop on Object-Oriented Development (WOOD'03), volume 82 of Electronic Notes in Theoretical Computer Science, 2003.

[11] Jens Palsberg, C. Barry Jay, *The Essence of the Visitor Pattern*, Proceedings of the 22nd International Computer Software and Applications Conference, p.9-15, August 19-21, 1998
[12] Mads Torgersen, *"The Expression Problem Revisited. Four New Solutions Using Generics."* In: M. Odersky (ed.): ECOOP 2004 - Object-Oriented Programming (18th European Conference; Oslo, Norway, June 2004; Proceedings). Lecture Notes in Computer Science 3086, Springer-Verlag, Berlin, 2004, 123-143.
[13] Philip Wadler, *The expression problem*, Message to Java-genericity electronic mail list, November 12, 1998.
[14] Philip Wadler, *The expression problem: A retraction*, Message to Java-genericity electronic mail list, February 11, 1999.
[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Abstraction and reuse of object-oriented designs*, Addison-Wesley, 1994, [416], ISBN: 0201633612
[16] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, *MultiJava: Modular open classes and symmetric multiple dispatch for Java*, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, pp.130-145, ACM Press, 2000.
[17] Matthias Zenger, Martin Odersky, *Extensible algebraic datatypes with defaults*, Proceedings of the International Conference on Functional Programming, Firenze, 2001.
[18] B. Karlsson, *Beyond the C++ Standard Library, An Introduction to Boost*, Addison-Wesley, 2005.
[19] Bjarne Stroustrup, *The Design of C++0x*, C/C++ Users Journal, May, 2005
[20] Douglas Gregor, Bjarne Stroustrup, *Concept Checking*, Technical Report, N2081, ISO/IEC SC22/STC1/WG21, Sept, 2006
[21] G. Dos Reis, B. Stroustrup, *Specifying C++ concepts*, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.
[22] ANSI/ISO C++ Committee, *Programming Languages – C++*, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.
[23] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.
[24] J. Siek, A. Lumsdaine, *Essential Language Support for Generic Programming*, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.
[25] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994
[26] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.
[27] Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation 14 January 2003.
`http://www.w3.org/TR/2003/REC-SVG11-20030114`

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
E-mail address: `abel@sinkovics.hu, gsd@elte.hu`