An Empirical Study on Limits of Clone Unification Using Generics

Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek
Department of Computer Science, School of Computing, National University of Singapore
{hamidabd, damithch, stan}@comp.nus.edu.sg

Abstract

Generics (templates) attempt to unify similar program structures to avoid redundancy. How well do generics serve this purpose in practice? We try to answer this question through empirical analysis from two case studies. First, we analyzed the Java Buffer library in which 68% of the code was redundant due to cloning. We were able to remove only 40% of the redundant code using the Java generics. Unification failed because the variations between cloned classes were either non-type parametric or non-parametric. To analyze whether this problem is specific to Java generics, we investigated the C++ Standard Template Library (STL), an exemplary application of C++ templates, as our second case study. Even though C++ templates are more powerful, we still found substantial cloning. We believe that we are dealing with a fundamental phenomenon that will cause many other class libraries and application programs to suffer from the code redundancy problem.

1. Introduction

Many modern programming languages support some form of generics (C++, Eiffel, Haskell, Ada, Modula-3 and recently Java). Generics are used to unify similar program structures (so called clones) to avoid explosion of redundant code. The main problem with clones is their tendency to create inconsistencies in updating, hindering maintainability. Clones signify reuse opportunities that, if properly exploited, could lead to simpler, easier to maintain, and more reusable program solutions [6]. In class libraries, clones often stem from the well-known "feature combinatorics" problem [2][3][6]. Generics can combat this emergence of clones, increasing software reuse and easing software maintenance. How well do generics serve this purpose in practice? We try to answer this question through empirical analysis from two case studies.

In our earlier paper [6], we analyzed redundancies in the Buffer library and identified that 68% of the code is redundant. The Buffer library was built without generics. An interesting question is how much of the redundant code could be eliminated by applying generics? In our first case study we looked into this problem. We observed that type variation triggered many other non-type parametric differences among similar classes, hindering application of generics.

For the second case study, we chose the Standard Template Library (STL) [5] as it provides a perfect case to strengthen the observations made in the first case study. Firstly, parameterization mechanism of C++ templates is more powerful than that of Java generics. Secondly, the STL is widely accepted in the research and industrial communities as a prime example of the generic programming methodology. Still, we found much cloning in the STL.

Our overall observations show that while generics provide an elegant mechanism to unify a group of similar classes through parameterization, in practice, there are many other situations that also call for generic solutions, but cannot be tackled with generics.

The remainder of this paper is organized as follows. After the related work section given next, Sections 3 and 4 briefly describe Buffer library case study and STL case study, respectively. Section 5 analyzes the observations made in the two case studies and illustrates them with examples. Concluding Remarks section ends the paper by summarizing findings and outlining the directions for future work.

2. Related work

A comparison of generics in six programming languages is presented in [4]. The languages considered are C++, standard ML, Haskell, Eiffel, Java and Generic C# (proposed). A considerable part of the Boost Graph Library has been implemented in all these languages using their respective generic capabilities. The authors identified eight language features that are useful to enhance the generics capabilities beyond

implementing simple type-safe polymorphic containers, namely multi-type concepts, multiple constraints, associated type access, retroactive modeling, type aliases, separate compilation, implicit instantiation and concise syntax. These features are essential to implement reusable libraries of software components, a fast emerging and a promising area where the generics can be effectively utilized. For the exact nature of these features, refer to [4]. However, the presence of all these features does not solve the problem discussed in this paper; rather it is only of help in avoiding "awkward designs, poor maintainability, unnecessary run-time checks, and painfully verbose code" [4].

3. Case study 1: Java Buffer library

The Buffer library in our case study is part of the java.nio.* package in JDK since version 1.4.1. The concept 'buffer' refers to a container for data to be read/written in a linear sequence. The (partial) class diagram of the Buffer library given in Fig. 1 shows the explosion of the many variant buffers that populates the library, a classic incarnation of the feature combinatorics problem. Even though all the buffer classes play essentially the same role, there are 74 classes in the Buffer library (In this analysis, we only consider the 66 classes that contribute to code duplication, leaving out helper classes, exception classes etc.).

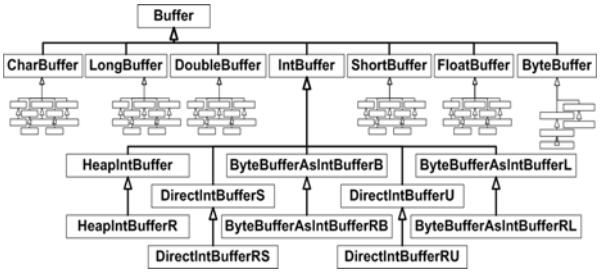

Fig. 1. Partial class hierarchy of Buffer library

Feature diagrams [8] are a common approach used in domain analysis to illustrate the variability of a concept. Feature diagram for the Buffer library is given in Fig. 2. It shows four mandatory feature dimensions and one optional feature dimension. 'Element Type (T)' is a mandatory feature dimension that represents the type of elements held in the buffer. It has seven alternative features corresponding to seven valid element types: int, short, double, long, float, char and byte.

To describe the feature diagram, we use the concept of 'peer classes':

Definition 1. Peer classes - A set of classes that differ along a given feature dimension only. For example, classes **HeapIntBuffer** and **HeapDoubleBuffer** are peers along the Element type dimension because the only variation between the two buffers is element type.

Feature dimension 'Access Mode (AM)' has two alternative features corresponding to read-only buffers and writable buffers respectively. Writable **HeapByteBuffer** and read-only **HeapByteBufferR** are peers along this dimension.

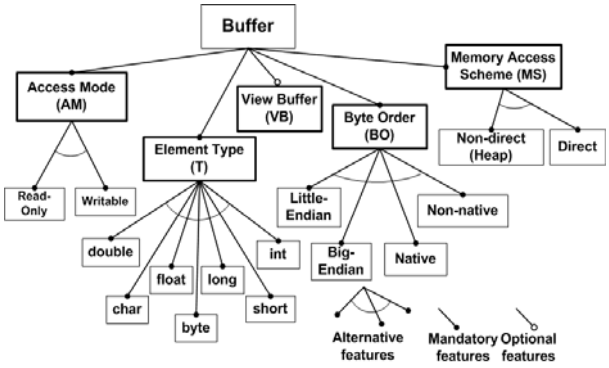

Fig. 2. Feature diagram for Buffer library

Other feature dimensions with alternate features are 'Memory Access Scheme (MS)' and 'Byte Order (BO)'. Feature dimension 'View Buffer' is optional. Each legal combination of these feature dimensions yields a unique buffer class. (e.g., **DirectIntBufferRS**, represents the combination T = int, AM = read-only, MS = direct, BO = non-native, and VB = false)

3.1. Analysis method

For this case study, we manually analyzed the Buffer library to identify groups of similar buffer classes. Then, we studied differences among classes in each group, and attempted to unify groups of similar buffers with suitable Java generics. Upon careful observation of the Buffer library, we found that only 15 buffer classes fall neatly into the generics-friendly layout and could be replaced by 3 generic classes **Buffer<T>**, **HeapBuffer<T>,** and **HeapBufferR<T>** (The solution can be viewed at [12]). According to this result, generics can reduce only 40% (calculated in terms of physical lines of code (LOC), excluding comments, blank lines and trivially short lines) of redundant code from the Buffer library. This solution still relies on wrapper classes for primitive types (as Java generics do not allow parameterization with primitive types).

A detailed analysis of the different types of generic-unfriendly situations that we encountered in the Buffer

library will be given in Section 5. More information on this case study can be found at [12].

4. Case study 2: Standard Template Library

The Standard Template Library (STL) is a general-purpose library of algorithms and data-structures. It consists of containers, algorithms, iterators, function objects and adaptors. Most of the basic algorithms and structures of computer science are provided in the STL. All the components of the library are heavily parameterized to make them as generic as possible. A major part of the STL is also incorporated in the C++ Standard Library. A full description of the STL can be found at [5].

Generic containers form the core of the STL. These are either sequence containers or associative containers. Among the containers, we selected the associative container slice for detailed analysis because of its high level of cloning. Feature diagram of Fig. 3 depicts features of associative containers in the STL. 'Ordering', 'Key Type' and 'Uniqueness' are the feature dimensions. Any legal combination of these features yields a unique class template (eight in total). For example, the container 'set' represents an associative container where Storage=sorted, Uniqueness=unique, and Key type=simple.

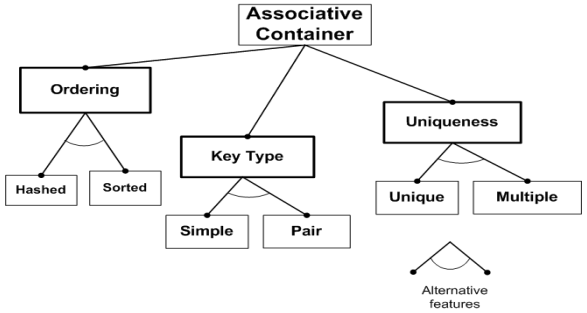

Fig. 3. Feature diagram for associative containers

4.1. Analysis method

We analyzed the STL code from the SGI website [5]. For clone detection we used CCFinder [7]. Having identified clones, we studied the nature of variations among them, and tried to understand the reasons why cloning occurred.

In our analysis of associative containers, we found that if all four 'sorted' associative containers and all four 'hashed' associative containers, were unified into two generic containers, the Reduction in Related Code (RRC) is 57%. RRC is an approximation calculated by comparing the LOC of clones before and after a meta-level unification. A detailed description of this meta-level unification is presented in [1]. In container adaptors – stack, queue and priority queue – we found that 37% of the code in stack and queue could possibly be eliminated through clone unification. Cloning in the algorithms (in file 'stl_algo.h') was localized to the set functions, i.e., we found that set union, intersection, difference, and symmetric difference (along with their overloaded versions) form a set of eight clones that could be unified into one (RRC=52%). Iterators were relatively clone-free, but the supporting files 'type_traits.h' and 'valarray' exhibited excessive cloning. In the 'type_traits.h' header file, a code fragment had been cloned a remarkable 22 times (RRC=83%).The header file 'valarray' contained eight different code fragments that had been cloned between 10 to 30 times each (137 times in total, where RRC=83%).

More information on this case study can be found in [1].

5. Where generics failed

In this section, we illustrate the situations in the two case studies, in which the generics were unable to unify similar program structures.

5.1. Non-parametric variations

It is common to find non-parametric variations in code. Extra or missing code fragments between similar program structures are such variations not addressed by generics. For example, **CharBuffer** of the Buffer library has some additional methods not present in other buffer types. On the other hand, **DirectByteBuffer** is missing a method common to all its peers. 'Extra' or 'missing' code fragments can be of any granularity as shown by the next example.

```
...
public abstract class CharBuffer
   extends Buffer implements Comparable, CharSequence{
...
```

```
...
public abstract class DoubleBuffer
   extends Buffer implements Comparable {
...
```

Fig. 4. Declaration of class CharBuffer and DoubleBuffer

CharBuffer class implements an extra interface none of its peers implement, resulting in the class declaration code shown in first part of Fig. 4. Now compare it with the declaration clause of

DoubleBuffer given second to note the offending extra bit of code in **CharBuffer**. Fig. 5 provides an example of a non-parametric variation of keywords between iterators for Map and Set in STL.

```
iterator begin() const {  return _M_t.begin();  }
iterator begin(){ return _M_t.begin();  }
```

Fig. 5. Keyword variation example

Some algorithmic differences are too extensive to be parameterized. For example, **toString()** method of **CharBuffer** differs semantically from **toString()** method of its peers, as shown in Fig. 6.

```
//In CharBuffer:
public String toString() {
 return toString(position(), limit());}
//In IntBuffer,FloatBuffer,LongBuffer etc.
   public String toString() {
   StringBuffer sb = new StringBuffer();
            sb.append(getClass().getName());
            sb.append("[pos=");
            ...
            sb.append("]");
            return sb.toString(); }
```

Fig. 6. Method toString() of CharBuffer and its peers

Due to this reason, we cannot use generics to unify **CharBuffer** with its peers despite the similarity of the rest of the code. A solution based on inheritance looks feasible, but not without adding another layer to the already complex inheritance hierarchy. Another option is to use template specialization, but Java generics do not support this feature.

```
template <class _Tp>
inline valarray<_Tp> operator+( const valarray<_Tp>& __x,
const _Tp& __c) {
   typedef typename valarray<_Tp>::_NoInit _NoInit;
   valarray<_Tp> __tmp(__x.size(), _NoInit());
   for (size_t __i = 0; __i < __x.size(); ++__i)
      __tmp[__i] = __x[__i]  + __c;
   return __tmp;}
template <class _Tp>
inline valarray<_Tp> operator+( const _Tp& __c, const
valarray<_Tp>& __x) {
   typedef typename valarray<_Tp>::_NoInit _NoInit;
   valarray<_Tp> __tmp(__x.size(), _NoInit());
   for (size_t __i = 0; __i < __x.size(); ++__i)
     __tmp[__i] = __c + __x[__i];
   return __tmp;}
```

Fig. 7. Clones due to swapping

One interesting type of non-parametric variation we spotted in STL is due to swapping of code fragments in order to make overloaded operators symmetric. Fig. 7 gives an example. Note how the parameter pair (**const valarray<_Tp>&**, **const _Tp& __c**) and operand pair (**__x[__i]**, **__c**) are swapped between the two clones.

5.2. Non-type parametric variations

Some parametric variations cannot be represented by types and hence cannot be unified using Java generics. A prime example of a non-type parametric variation is constants. The clone given in Fig. 8 is repeated several times inside the Buffer library with different constant values (2, 3, and 4) for **@size**.

```
private long ix(int i) {
  return address + (i << @size);
}
```

Fig. 8. Generic form of method ix()

Though parameterization using constants is supported in C++, the question remains whether we should force the user to specify this parameter manually when the value is inferable from another type parameter. One solution is to use traits template idiom [11], at the expense of increased complexity, to encode the type dependent information into the type and pass it as a parameter.

Another parametric variation not supported by generics is keywords. In stl_iterator.h, the clone given in Fig. 9, **@access** was 'private' in one instance while it was 'protected' in the other (a possible case of inconsistent updating).

```
template <class _Tp @moreParams >
class ostream_iterator {
public:
  …
  ostream_iterator<_Tp>& operator*() { return *this; }
  ostream_iterator<_Tp>& operator++() { return *this; }
  ostream_iterator<_Tp>& operator++(int) { return *this; }
@access:
  @streamType* _M_stream;
  const @stringType* _M_string;
};
```

Fig. 9. Access level variation example

```
public ByteOrder order() {
return ((ByteOrder.nativeOrder() @operator
    ByteOrder.BIG_ENDIAN)?ByteOrder.LITTLE_ENDIAN:
       ByteOrder.BIG_ENDIAN);}
```

Fig. 10. Generic form of method order() in direct buffers

At times, code fragments differed in operators, as illustrated in the example from the Buffer library shown in Fig. 10. **@operator** is '==' in **DirectDoubleBufferS** but it is '!=' in **DirectDoubleBufferU**. Such variations also cannot be unified with Java generics.

An indirect solution of the above problem can be through function objects. The different operators can be turned into function objects and passed on to the generic class as a parameter. But this indirect solution may create more clones among the different function objects.

A similar problem is found in STL with operator overloading in the associative containers of STL. Fig. 11 shows a generic form of such clones. @op was replaced by different operators (e.g. '==', '<' etc.) in different instances of the clone. Since these code fragments relate to operator overloading, function objects cannot be used to unify these clones.

```
template <class _Key, class _Compare, class _Alloc>
inline bool operator@op (const
set<_Key,_Compare,_Alloc>& __x,
                const set<_Key,_Compare,_Alloc>& __y) {
 return __x._M_t @op__y._M_t; }
```

Fig. 11. A clone that vary by operators

Also, copyright notices that appear in all STL files exhibit non-type parametric variations.

5.3. Restrictions on type-parametric variations

Type parametric variations between code fragments are the ideal targets for code reuse through generics. Yet idiosyncrasies of generic implementations can sometimes get in the way, even in these ideal situations. For example, parameterization using primitive types (int, short, long, double, etc.) is not allowed in Java.

In STL iterators, we found another case of restrictions on type parameters for templates. In this clone (shown in Fig. 12) the only variation point **@type** is a type (int, float, long, bool, char, short … 22 types in all). These clones are template specializations for 22 types. Therefore, they cannot be unified by usual template techniques.

```
__STL_TEMPLATE_NULL struct __type_traits<@type> {
   typedef __true_type    has_trivial_default_constructor;
   typedef __true_type    has_trivial_copy_constructor;
   typedef __true_type  has_trivial_assignment_operator;
   typedef __true_type    has_trivial_destructor;
   typedef __true_type    is_POD_type;};
```

Fig. 12. Generic form of a clone found in "type_traits.h"

5.4. Coupling

Coupling among classes and modules can also play a role in restricting the use of generics. Given in Fig. 13 is an example of this situation from the Buffer library.

```
public int get(int i) {
 return Bits.swap(
  unsafe.getInt(ix(checkIndex(i))));}
public float get(int i) {
 return Bits.swap(
  unsafe.getFloat(ix(checkIndex(i))));}
```

Fig. 13. Method get(int) of DirectIntBufferS and DirectFloatBufferS

To unify these two methods into a generic method, we need to unify **getInt()** and **getFloat()** methods as well. Sometimes this is not possible: these two methods can be out of scope or they can be generics-unfriendly. Now, we have two ways to proceed. The first is to convert the variant functions into function objects and ask the user to furnish the required function object as a parameter. But this breaks the basic design, since this parameter is not one of the feature dimensions. The second is to find a way (possibly using run-time type information) to infer the proper function to call based on the type parameter. This will introduce further indirections and runtime overheads.

6. Concluding remarks

Generic design solutions have to do with both reusability and maintainability, the two economically desirable – but also difficult to achieve - software engineering goals. However, in many cases, genericity is difficult to achieve in the confines of conventional techniques. This is evidenced by high rate of similarity we find in programs. Type parameterization is an important means to achieve genericity. In the paper, we have analyzed the situations in which, despite similarities among program structures, generics fell short of providing suitable clone-free, generic solutions. We have illustrated the problem with examples from the Java Buffer library and the STL in C++.

Cloning is a pervasive problem, not confined to C++ or Java, with much negative impact on maintenance and reuse [7]. Our other experiments have uncovered extensive cloning in command and control applications implemented in C# and J2EE, and Web portals [12]. Effective generics should allow us to unify program structures (such as functions, methods, classes or any patterns of such program elements) resulting from similar domain concepts, to avoid counter-productive redundancy. Generics are a prime language feature for parameterization. Our empirical studies have revealed that generics could unify code portions differing in the type parameters most of the

time, but failed to provide a neat solution in the presence of other variations in code details.

In future, we plan to analyze other class libraries and application programs, written in various programming languages, using a range of design techniques. We hope such work will result in further insights into the nature of problems presented in this paper. We also plan to address so-called structural clones, that is, patterns of repetitions emerging from analysis and design levels. Structural clones usually represent larger parts of programs than the 'simple' clones discussed in this paper, therefore their treatment could be even more beneficial. Design of generic solutions unifying similar program structures at all levels, in particular structural clones, is at the heart of designing software architectures for reuse. Effective parameterization is one of the prime techniques to achieve reuse goals. In our current and future work, we investigate a meta-level parameterization technique such as described in [6]. Meta-level parameterization is less restrictive than generics or templates, and has demonstrated a potential to overcome the limitations we encountered in this study [12]. We plan to conduct comparative studies of various techniques for clone treatment, to better understand their strengths, weaknesses, and areas where the synergy exists among different techniques.

This paper is a first attempt at presenting limitations of language-level parameterization for defining clone-free generic program solutions. We hope our results will encourage others to pursue further studies in the direction of programming language support for effective parameterization.

7. Acknowledgements

Authors thank Pavel Korshunov for participating in the first case study, and Toshihiro Kamiya (PRESTO, Japan) Katsuro Inoue and Shinji Kusumoto (Osaka University, Japan) for providing us with the tools CCFinder and Gemini (GUI for CCFinder)

References

[1] Basit, H. A., Rajapakse, D. C., and Jarzabek, S., "Beyond Templates: a Study of Clones in the STL and Some General Implications," *to appear in proc. 28th Intl. Conf. on Software Engineering (ICSE'05)*, 2005, draft available at http://xvcl.comp.nus.edu.sg/xvcl_cases.php

[2] Batory, D., Singhai, V., Sirkin, M. and Thomas, J. "Scalable software libraries," *ACM SIGSOFT'93: Symp. on the Foundations of Software Engineering*, Los Angeles, Dec. 1993, pp.191-199

[3] Biggerstaff, T. "The library scaling problem and the limits of concrete component reuse," *3rd Int'l. Conf. on Software Reuse*, ICSR'94, 1994, pp. 102-109

[4] Garcia, R. et al., "A Comparative Study of Language Support for Generic Programming," Proc. 18th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, 2003, pp. 115-134

[5] Home page of SGI STL, http://www.sgi.com/tech/stl/

[6] Jarzabek, S. and Shubiao, L., "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," Proc. ESEC-FSE'03, *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2003, Helsinki, pp. 237-246

[7] Kamiya, T., Kusumoto, S, and Inoue, K., "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code," IEEE Trans. Software Engineering, vol. 28 no. 7, July 2002, pp. 654 – 670

[8] Kang, K et al. 1990. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Pittsburgh, Nov. 1990

[9] Kennedy, A. and Syme, D., "Design and implementation of generics for the .Net Common Language Runtime," Proc. ACM SIGPLAN '01 Conf. on Programming Languages Design and Implementation (PLDI -01), New York, June 2001, pp 1-12

[10] Musser, D. and Saini, A., 1996. STL Tutorial and Reference Guide: C++ Programming with Standard Template Library, Addison-Wesley, Reading (MA), USA

[11] Myers N. C., "Traits: a new and useful template technique," C++ Report, June 1995

[12] "XVCL Case Studies" XVCL web site http://xvcl.comp.nus.edu.sg/xvcl_cases.php