Beyond Templates: a Study of Clones in the STL and Some General Implications

Hamid Abdul Basit
Department of Computer Science
School of Computing
National University of Singapore
+65 6874 2834
hamid@nus.edu.sg

Damith C. Rajapakse
Department of Computer Science
School of Computing
National University of Singapore
+65 6874 2834
damith@nus.edu.sg

Stan Jarzabek
Department of Computer Science
School of Computing
National University of Singapore
+65 6874 2863
stan@comp.nus.edu.sg

ABSTRACT

Templates (or generics) help us write compact, generic code, which aids both reuse and maintenance. The STL is a powerful example of how templates help achieve these goals. Still, our study of the STL revealed substantial, and in our opinion, counter-productive repetitions (so-called clones) across groups of similar class or function templates. Clones occurred, as variations across these similar program structures were irregular and could not be unified by suitable template parameters in a natural way. We encountered similar problems in other class libraries as well as in application programs, written in a range of programming languages. In the paper, we present quantitative and qualitative results from our study. We argue that the difficulties we encountered affect programs in general. We present a solution that can treat such template-unfriendly cases of redundancies at the meta-level, complementing and extending the power of language features, such as templates, in areas of generic programming.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques – Software libraries; D.2.10 [Software Engineering]: Design – Representations; D.2.13 [Software Engineering]: Reusable Software - Reusable libraries, Domain engineering

General Terms
Design, Experimentation, Languages.

Keywords
Software Maintenance, Clones, Meta-programming

1. INTRODUCTION AND MOTIVATION

Code clones are identical or near identical fragments of source code. The main problem with clones is their tendency to create inconsistencies in updating, which hinders maintenance and contributes towards "software ageing" [26]. Code clones often hint at the existence of design-level similarity patterns in a program (or across programs). Such large granularity similarity patterns – we call them “structural clones” - signify reuse opportunities that, if properly exploited, could lead to simpler, easier to maintain, and more reusable program solutions [19]. The reasons why clones appear in source code have been analyzed [6][14][20] and clone detection tools have been proposed [1][6][14][21]. Methods for clone resolution include refactoring [1][2][3][15], macros [6] and meta-level techniques [19].

In class libraries, clones often stem from the well-known “feature combinatorics” problem [5][7][19]. A proper parameterization mechanism can combat this emergence of clones, increasing software reuse and easing software maintenance. At the language level, generics (in Ada, Eiffel, and recently proposed additions to Java [8] and C# [23]) and templates (in C++) are the main parameterization techniques.

In our previous case study [11], we experimented with the proposed generics in Java. We tried to unify classes in the Java Buffer Library that differed in the type of a buffer element. We observed that type variation also triggered many other non-type parametric differences among similar classes, hindering application of generics. As the result, despite striking similarities across library classes, only a small part of the library could be transformed into generic classes.

Careful examination revealed that most of the issues that hindered a complete generic solution for the library were specific to Java generics. However, some other issues were of more fundamental nature. We thought further work was needed to draw the fine line between the two.

The Standard Template Library (STL) [18] provides a perfect example to strengthen the observations made in the Buffer Library case study. Firstly, parameterization mechanism of C++ templates is more powerful than that of Java generics. Due to light integration of templates with the C++ language core, template parameters are less restrictive than parameters of Java generics. Unlike Java generics, C++ templates also allow constants and primitive types to be passed as parameters.
Secondly, the STL not only uses the most advanced template features and design solutions (e.g., iterators), but it is also widely accepted in the research and industrial communities as a prime example of the generic programming methodology.

The STL needs genericty for simple and pragmatic reasons: There are plenty of algorithms that need to work with many different data structures. Without generic containers and algorithms, the STL's size and complexity would be enormous. Such simple-minded solution would otherwise ignore similarity among data structures, and also among algorithms applied to different data structures, which offers endless reuse opportunities. Redundant code sparking from unexploited similarities would contribute much to the STL's size and complexity, hindering its evolution. The object of the STL was to avoid these complications, without compromising efficiency [18].

Still, we found much cloning in the STL. Our study confirmed that these clones varied in certain ways that could not be easily unified by template parameters. To demonstrate that such unification was feasible and beneficial, we built a clone-free representation with a meta-level parameterization supported by XVCL1. With meta-level unification of clones, we can avoid template-unfriendly clones, while still retaining the simple design and the efficiency of the source code, as is the hallmark of the STL. In the paper, we discuss trade-offs among template-based and meta-level parameterization mechanisms.

The paper is organized as follows: Section 2 describes the parts of the STL relevant to the experiment. Section 3 gives an overview of experiment methodology. Section 4 discusses interesting examples of cloning in the STL. Section 5 shows a meta-level parameterization solution to STL clones, and in Section 6 we discuss the results. Related work and conclusions end the paper.

2. THE STRUCTURE OF THE STL
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. Algorithms and data-structures commonly used in 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 is beyond the scope of this paper and can be found in [18]. We provide enough description here to facilitate the understanding of the experiment that is described next.

Generic containers form the root of the STL. These are either sequence containers or associative containers. In sequence containers, all members are arranged in some order. In associative containers, the elements are accessed by some key and are not necessarily arranged in any order. All the STL containers are parameterized by type so that a single implementation of the container template can be used for all types of contained elements.

The second major component in the STL is the algorithms that work on the generic containers. Algorithms in the STL are decoupled from the containers, and are implemented as global functions rather than member functions. Further generalization of algorithms is achieved by implementing them to work on a range of elements rather than knowing the container that holds those elements.

Iterators are used in the STL to achieve the decoupling of algorithms from containers. Iterators are generalization of pointers in C++. This ensures that all algorithms that take in an iterator as a parameter also work with normal pointers. Iterators provide an abstraction of the containers free of their storage details. For example, the operator ++ of an iterator for a linear container will simply increment a pointer, while the same operator will perform a tree walk on a tree container.

3. EXPERIMENT OVERVIEW
We analyzed the STL code from the SGI website [18]. Our analysis went through two stages, namely (1) automatic detection of similar code fragments such as class methods or parts of them (so-called simple clones), and (2) manual domain analysis with the objective of discovering design-level similarities. Group of similar associative container templates, as discussed in Section 4, is an example of this design-level similarity found in the STL.

For clone detection we used CCFinder [21]. CCFinder can find simple clones – code fragments that differ in parameteric ways. Since container classes form the backbone of the STL, they were the first to be analyzed for clone detection. CCFinder revealed a lot of clones when the minimum clone size was set at 30 tokens. When it was set at 50 tokens, the smaller clones were filtered out. Examination of clones revealed that cloning in container classes was not an ad-hoc phenomenon. We found extensive cloning in the associative containers and in the container adaptors - stack and queue.

We did not find significant cloning in the algorithms (in file 'stl alg h'). Some clones were observed in the set functions, e.g., set union, set intersection, set difference and set symmetric difference, but they were restricted to the checking of pre-conditions rather than the actual implementation of the algorithm. Iterators were also relatively clone-free, but the supporting files ‘type_traits.h’ and ‘valarray’ exhibited excessive cloning.

Having identified clones, we studied the nature of variations among them, and tried to understand the reasons why cloning occurred. Heavily cloned areas led us to identifying groups of templates that exposed enough similarity to become candidates for generic design solutions.

We also analyzed the impact of both simple and structural clones on understanding and evolution of the STL.

Finally, we built a clone-free representation for the STL templates under study. For this, we applied a meta-level parameterization technique of XVCL (section 5).

4. ANALYSIS OF CLONES IN THE STL
In this section, we give examples of clones we found, and possible causes for their presence in the STL. Then, we comment on the problems such clones may cause.

1 XVCL: XML-based Variant Configuration Language, is a public domain meta-language, method and tool for enhanced reusability and maintainability, available at: fxvcl.sourceforge.net
4.1 Cloning in Containers

CCFinder detected a substantial amount of cloning in the container classes as shown in Table 1.

<table>
<thead>
<tr>
<th>File Group</th>
<th>No of Files</th>
<th>No of Clone Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Clone size &gt;= 30 Tokens</td>
</tr>
<tr>
<td>Associative Containers</td>
<td>6</td>
<td>616</td>
</tr>
<tr>
<td>All Containers</td>
<td>21</td>
<td>1051</td>
</tr>
<tr>
<td>All Analyzed Files</td>
<td>48</td>
<td>1273</td>
</tr>
</tbody>
</table>

Table 1 shows that of the total number of clone pairs detected by CCFinder, majority are present in the container classes. Given next are some interesting simple clones that were detected in the containers.

Differences in operator symbols were a common variation. Figure 1 shows a generic form of such clones. @op marks the two variation points of this set of 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; 
}
```

Figure 1. A clone that varies by operators

Figure 2 shows two clone examples where @op is ‘==’ and ‘<’ respectively. Clones of this type are difficult to unify using C++ templates.

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

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

Figure 2. Two clone instances that differ by operator

Some variations were caused by keywords. Figure 3 provides an example of such a variation between iterators for Map and Set.

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

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

Figure 3. Keyword variation example

Other types of variations included the following cases (we omit specific examples to save space):

- Extra typedefs in cloned fragments
- Extra functions in cloned classes
- Fine grained algorithmic variations in cloned functions
- Extra parameters in cloned template definitions or template instantiations
- Different class and function names in cloned classes and functions
- Type variations in cloned typedefs

We selected associative containers for further detailed manual analysis because of its high level of cloning. An associative container is a variable-sized container that supports efficient retrieval of its elements based on keys. Feature diagram of Figure 4 depicts features of associative containers in the STL. 'Ordering', 'Key Type' and 'Uniqueness' are the feature dimensions. Relevant features are shown below the respective feature dimension boxes. In this diagram, we omit feature dimensions already successfully parameterized in the STL (e.g., 'Element type').

![Feature diagram for associative containers](image)

Figure 4. Feature diagram for associative containers

There are several variations of the associative containers in the STL. The elements of a 'hashed' associative container are not guaranteed to be in any meaningful order. 'Sorted' associative containers use an ordering relation on their keys. 'Key Type' dimension describes the nature of the keys used. In a 'Simple' associative container, elements are their own keys. A 'Pair'

2 Note that a total of 43 files were excluded from the analysis as they were merely present for backward compatibility and consist of a few "include" statements only.
associative container associates a key with some other object. In a 'Unique' associative container each key in the container is unique, which need not be the case in a 'Multiple' associative container.

Any legal combination of these features yields a unique class template (Table 2). For example, the container 'set' represents an associative container where Storage=sorted, Uniqueness=unique, and Key type=simple. There is much similarity across associative containers independently of the specific features they implement which leads to clones.

<table>
<thead>
<tr>
<th>Class template</th>
<th>Feature dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Storage</td>
</tr>
<tr>
<td>Set</td>
<td>sorted</td>
</tr>
<tr>
<td>Map</td>
<td>sorted</td>
</tr>
<tr>
<td>Multimap</td>
<td>sorted</td>
</tr>
<tr>
<td>Multiset</td>
<td>sorted</td>
</tr>
<tr>
<td>Hash set</td>
<td>hashed</td>
</tr>
<tr>
<td>Hash multiset</td>
<td>hashed</td>
</tr>
<tr>
<td>Hash map</td>
<td>hashed</td>
</tr>
<tr>
<td>Hash Multimap</td>
<td>hashed</td>
</tr>
</tbody>
</table>

Our analysis showed that all four 'sorted' associative containers and all four 'hashed' associative containers could be unified into two generic containers, reducing the size of related code by 57%, from 827 LOC to 358 LOC.

4.2 Other Examples of Cloning

Container adaptors also had a high level of cloning. A container adaptor is implemented on top of some underlying container type to provide a restricted subset of container functionality. Three classes—stack, queue, and priority queue—are considered container adaptors. They vary along the feature dimension 'Retrieval method'. Stack uses last-in-first-out (LIFO) strategy, queue uses first-in-first-out (FIFO) strategy, and priority queue returns the element with the highest priority. We found that 37% (LOC: 194 → 123) of the related code could be eliminated through clone unification. Given next are some variations we found between stack and queue:

- Retrieval functions in stack and queue are called top() and front() respectively, to reflect the different retrieval methods, with minor implementation difference between the two
- The difference in retrieval method causes small algorithmic variations in some functions
- Certain overloaded operator definitions appear with inline keyword in queue, but without it in stack. This could probably be an oversight by the programmers resulting from inconsistent updating of the cloned fragments

```
Self& operator++() {
  --current;
  return *this;
}
Self operator++(int) {
  Self__tmp = *this;
  --current;
  return __tmp;
}
Self& operator--() {
  ++current;
  return *this;
}
Self operator--(int) {
  Self__tmp = *this;
  ++current;
  return __tmp;
}
```

In algorithms, 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. Generic form of this clone is shown in Figure 5. @opType represents the varying method name. Break points/regions show other variation points where variation may occur in some instances, but not in all. Unifying these eight segments shrinks the related code by 52% (LOC: 196 → 95).

Among the iterators (in file 'stl_iterator.h'), we found code segments which were exact clones, like the two copies of the code given in Figure 6.

```cpp
_Self__tmp = ++current;
return *this;
_Self operator--() {
  _Self__tmp = *this;
  ++current;
  return __tmp;
}
```

Figure 5. Variants of set algorithms

Figure 6. Exact clones found among iterators

---

3 All non-trivial text lines in the code segment under consideration are counted towards LOC.
In iterators, we also found cloned classes having different access modifiers for the same class members, a possible case of inconsistent updating. In the clone given in Figure 7, @access was 'private' in one instance while it was 'protected' in the other.

Intense cloning was also present in the 'type_traits.h' header file that provides a framework for allowing compile time dispatch based on type attributes. The fragment shown in Figure 8 was cloned a remarkable 22 times within the same file, unification of which brings an 83% (LOC 815→144) reduction to the related code. The only variation point @type is a type name (int, float, long, bool, char, short … 22 types in all). This is interesting because templates are supposed to unify type variations. However, these clones are template specializations for 22 types. Therefore, these clones cannot be unified by usual template techniques.

The header file 'valarray' declares types and functions for operating on arrays of numerical values. This file contained eight different code fragments that had been cloned between 10 to 30 times each (137 times in total). The related code is reduced by 83% (LOC 815→144) when we unify these clones.

One interesting type of variation we noticed is due to swapping of code fragments in order to make overloaded operators symmetric. Figure 9 gives an example. Note how the parameter pair (const valarray<_Tp>&, const _Tp& _c) and operand pair (_x[i], _c) are swapped from one clone to the other.

Another example of the cloning is the copyright notices that appear in all files (truncated generic form is shown in Figure 10).

Some files carried two instances of this clone (one where @owner='Silicon Graphics Computer Systems, Inc.' and another where @owner='Hewlett-Packard Company'). Though clones in comments like this may not be regarded as critical as clones in code, comments still have to be maintained.

4.3 Effects of Clones in the STL

There are many advantages of explicating similarities and differences in programs, and problems caused by failing to do so. The main problem with clones is their tendency to create inconsistencies in updating during maintenance, as shown by a few examples discussed above. In the STL, we found clones that
were repeated up to 30 times. This means, there is a need to modify up to 30 locations when one of the clones is modified. To change a clone, the maintainer must find all its instances, analyze them in different contexts to see which clones need to be changed and how. All these steps are error prone and tedious when done manually.

By revealing design-level similarities (i.e., structural clones), we reduce the number of distinct conceptual elements a programmer must deal with. Not only do we reduce an overall software complexity, but also enhance conceptual integrity of a program which Brooks calls "the most important consideration in system design" [9].

5. META-LEVEL PARAMETERIZATION

In this section, we present a meta-level parameterization that can unify all the similarity patterns we found in the STL. Our solution uses XVCL [28], a static meta-programming language, a modern incarnation of Bassett’s frames [4]. We introduce XVCL concepts by example. Other projects with XVCL are described in [19][29][30] and the XVCL Web site [28] contains XVCL specifications, processor and case studies.

Meta-components — that is, generic program structures we build with XVCL — can be parameterized in fairly unrestrictive ways. Parameters range from simple values (such as strings), to types and to other meta-components that can represent program elements of arbitrary kind, structure and complexity. XVCL is language-independent in that it can be applied on top of any programming language. As a result, we can use built-in language features such as templates to unify classes that differ in type parameters, and then use XVCL to unify classes that differ in more complex, template-unfriendly ways. Figure 11 illustrates how XVCL is used.

![Figure 11. Generic solutions in XVCL](image)

Any similar program structures — methods, classes, templates or architectural patterns — can be unified by generic solutions built as meta-component structures. XVCL processor instantiates such generic solutions to generate specific variant instances of program structures. Parameter values and the generation logic are specified in a specification file (SPC for short). By varying the SPC we can use the same generic meta-components to generate different concrete instances of the code. Figure 11 shows how the meta-component 'S_A_Container', together with the appropriate SPC, is used to generate the multiple variants of sorted associative containers given in Table 2.

![Figure 11. Generic solutions in XVCL](image)

All the generic forms of clones we presented so far (e.g., Figure 1) are in fact XVCL meta-components after clone unification. Next, we illustrate how these meta-components are used in the code generation process.

![Figure 12. A sample meta-component](image)

A meta-component of Figure 12 has its variation points marked by variables (e.g., className) or break points (e.g., extraParam). The SPC given in Figure 13 first assigns values to variables using the <set> command. Variable 'className' is a single valued variable while the other two are multi-valued. Multi-valued variables can be used as control variables of the <while> loop. Then, it adapts the meta-component in a <while> loop to generate two variants of the clone. After changing the value of 'className' variable, it uses a second <while> loop to adapt the same meta-component, while inserting an extra parameter to the break. This generates another two variants of the clone.

![Figure 13. Sample SPC](image)

The resultant code generated is given in Figure 14. In this example, we have reduced the number of generated clones to four. Thus, it appears as if the difference in code size is not significant. But in the real XVCL code for the STL, it is used to generate 24 clones, shrinking the related code by 81% (LOC: 96 → 19).

XVCL solves the inconsistent updating problem mentioned in section 4.3 as follows. First, the maintainer does not need to search for clones, as only one copy of the clone exists at the meta-
level. Second, the maintainer does not have to investigate variations between each clone – XVCL explicates these variations at the meta-level. Third, the maintainer does not have to update multiple copies of clones – updates at the meta-level are propagated to all the clones in program level, in a consistent manner.

6. DISCUSSION OF RESULTS
Cloning is a pervasive problem not confined to the STL or C++ [21]. Our other experiments have uncovered extensive cloning in different applications, implemented in different languages: in a Java library [19] and in a C# CAD application [11]. We found that in many cases, intensive cloning of code fragments signified design-level similarities. An import ant finding from these studies differen 
t applications, implemented in different languages: in a [21]. Our other experiments have uncovered extensive cloning in different languages and design techniques, complementing and enhancing them in areas where conventional techniques fail to provide a satisfactory solution. For example, in this experiment, we applied XVCL on top of the template solution, to unify only those clones that could not be unified with templates.

Removing clones at the language level requires changes to the code. In real systems, clones are often tolerated in spite of their negative effect on maintenance, to avoid the risk of breaking a running system while trying to remove them [12]. When clones are specifically created for performance considerations, it is not advisable to remove them altogether. Similarly, at times the clone resolution may be possible through refactoring [15], but the result may conflict with other design goals that cannot be compromised [19]. Since XVCL works at the meta-level without altering the program, there is no risk involved in terms of breaking a running system, loss of performance or compromising other design goals. For example, XVCL can be applied to unify clones in the STL with no risk of breaking it or any other system that uses the STL.

It is important to notice that a clone-free XVCL representation of the STL is visible only to the maintainers of the STL. Programmers – i.e., users of the STL – need not be aware of XVCL, as the code generated by XVCL is exactly the same as the original STL.

Having to deal with an additional meta-level adds a certain amount of complexity to the problem. However, the feedback from our industry partner indicates that, in practice, the benefit of being able to deal with complexity at two levels outweighs the cost of the added complexity. We are currently collecting empirical evidence to confirm this observation. The syntax of XVCL is easy to learn and in our lab, we are working on "XVCL Workbench"4 that incorporates a number of tools that help in editing, visualizing, debugging and static analysis of XVCL code. Proprietary meta-structures and tools help mitigate this problem of added complexity at the meta-level.

As a final remark, although XVCL pushes the envelope further on unifying clones, one should apply it only when the benefit is worth the effort. For example, we decided to leave some clones intact because similarity level was not worth the effort of unification. On the other hand, we found that unifying design-level similarities with XVCL is almost always beneficial, as it considerably reduces perceived program complexity. So we can always weigh pros and cons of applying XVCL and decide accordingly.

7. RELATED WORK
Other options for clone resolution are refactoring and macros. We discuss the strengths and limitations of these techniques in the context of the problem addressed in this paper.

---

4 to be released soon at [28]
Refactoring is the state-of-the-art technique to improve the internal structure of systems while preserving their external behavior [15][25]. Refactorings that could be applied for the removal of duplicated code are extract method (the duplicated code is extracted into a separate method), remove method (the duplicated methods are merged), pull up method (the duplicated methods are moved up the class hierarchy and inherited by all subclasses), or any combination of the above. The good thing about refactoring is that we stay in the same paradigm but it is not always feasible to resolve all the clones by this technique.

Refactoring based on design techniques (design patterns, inheritance with dynamic binding) is a clone unification option that is fairly independent of the underlying programming language but is closely tied with the design of the program. To eliminate the redundant code in a Java software system, Balazinska et al. [2][3] applied the refactoring based on ‘strategy’ and ‘template’ design patterns, by factoring out the commonalities of methods and parameterizing the differences according to the design patterns. However, the scope of the applicability of this technique is restricted only to specific types of clones.

In the context of the STL, it would be interesting to see what refactoring techniques could be applied and how much could be achieved. Although the STL is written in an Object-Oriented language, yet it has a flat class structure with no inheritance. There are many stand-alone template functions implementing algorithms separately from the data structures, which itself is not in tune with Object-Oriented concepts. Still, such structure of classes, along with powerful concept of iterators, paved the way for genericity of the STL solution.

Refactoring aims at totally removing clones from the source code. However, this objective is not always achievable nor is it desirable as discussed in Section 6.

Baxter et al. proposed to replace clones with macros [6]. Most of the macro systems are merely implementation level mechanisms for handling variant features (or changes, in general). Failing to address change at analysis and design levels, macros never evolved towards full-fledged “design for change” methods [4][22]. Programs instrumented with macros tend to be difficult to understand and test. Unlike macros, XVCL is a full-fledged method for generic design, in which variant features are directly addressed at both program design and implementation levels. Over time, an XVCL meta-component structure emerges as a well-organized architecture that explicates the impact of variant features on components (or classes) and automates production of custom components. XVCL has unique features to support reuse and evolution such as propagation of meta-variables across meta-components, meta-variable scoping rules that allow us to adapt generic meta-components at inclusion points, meta-expressions to formulate generic names, code selection or insertion at designated breakpoints and a while loop construct to implement generators.

Higher order functions from the functional programming paradigm offer an attractive reuse option [27]. Skeleton objects are introduced by [10] as an object-oriented alternative for the higher order functions and mechanism to build adaptable components using these skeleton objects are discussed. However, the approach may be difficult to implement in languages that do not support function pointers. Even in C++, passing long lists of function pointers as arguments to class constructors severely degrades the readability of the code.

A comparison of generics in six programming languages is presented in [17]. A considerable part of the Boost Graph Library has been implemented in all six languages using their respective generic capabilities. The authors identified several language features that are useful to enhance the generics capabilities beyond implementing simple type-safe polymorphic containers. These features are essential to implement reusable libraries of software components, which is fast emerging as a promising area where the generics can be effectively utilized. However, the presence of all these features does not solve the problems discussed in this paper, rather it is only of help in avoiding “awkward designs, poor maintainability, unnecessary run-time checks, and painfully verbose code” [17].

C++ templates also offer the possibility of meta-programming that is useful in several occasions, for example in generative programming [13]. But this meta-programming technique is an accidental discovery, not a planned language feature. Because of its accidental nature, this powerful mechanism has several drawbacks like complex and hard to understand syntax, lack of debugging facilities, and limited compiler support [13].

8. CONCLUSIONS
Generic design has to do with maintenance and reuse, the two central themes in software engineering research and practice. The goal of generic design is to identify similarity patterns, at the design and code levels, in order to avoid counter-productive repetitions, so called clones. Parameterization is an important paradigm for generic design. In this paper, we presented a study of parameterization via templates in the STL. We found substantial cloning in certain parts of the STL that could not be treated with templates. However, we could create meta-level generic structures unifying those clones with the meta-level parameterization technique of XVCL. While some observations and lessons learned from this experiment are specific to the STL, others are of a more general nature. In particular, we believe that many types of variations among similar program structures are difficult to unify with language-level parameterization techniques such as templates or generics. In the paper, we provided empirical evidence and analytical arguments to support this claim. We also showed how a meta-level parameterization mechanism can deal with template-unfriendly variations, enhancing maintainability and reusability of programs in areas where conventional techniques do not yield a satisfactory solution. Finally, as none of the solutions is without pitfalls, we evaluated pros and cons of various parameterization mechanisms.

In our future work, we plan to extend our studies on the structural clones emerging from analysis and design levels. Structural clones usually represent large parts of programs, therefore their treatment is most beneficial for programmers’ productivity. We also plan to extend our comparative studies of various techniques for clone treatment, to better understand their strengths, weaknesses, and areas where the synergy exists among different techniques. We believe productive technological solutions can be built in that way.
9. ACKNOWLEDGEMENTS
This work was supported by NUS Research Grant R-252-000-178-112. Authors wish to thank Toshihiko Kamiya (PRESTO, Japan), Katsuro Inoue and Shintaro Kusumoto (Osaka University, Japan) for providing us with the tools CCFinder and Gemini (GUI for CCFinder).

10. REFERENCES


