Software Similarity Patterns and Clones: A Curse or Blessing?

Stan Jarzabek
Bialystok University of Technology, Faculty of Computer Science, Bialystok, Poland

Keywords: Software Clones, Generic Design, Software Maintainability and Reusability, Software Complexity.

Abstract: Similarities are inherent in software. They show as *software clones* – similar code fragments, functions, classes, source files, and bigger program structures spreading through software systems in multiple variant forms. Often, these recurring program structures represent important concepts from software requirements or design spaces. Interestingly, despite potential benefits, avoiding many of such redundancies is often either impossible or would require developers to compromise important design goals. In this paper, I discuss software similarity phenomenon, its sources, the many roles clones play in programs, the software productivity benefits that can be gained by avoiding clones, and difficulties to realize these benefits with conventional programming languages and design techniques. I point to generative techniques as a promising approach to address software redundancy problems.

1 INTRODUCTION[1]

Much similarity within and across programs creates potential for program simplification and reuse. The extent to which similar program structures deliberately spread through programs indicates that this potential may not be fully exploited. The main theme of this paper is software similarity phenomenon and its manifestation in programs as software clones, in relation to program simplification, understanding, changeability and reuse.

Similarity patterns arise in both problem domain and program solution spaces. If not tackled, similarities show as program structures repeated many times within a program or across programs. We observe similar program structures of various types and granularity such as architectural patterns of components, patterns of collaborating classes, similar classes, source files, or code fragments. Recurring program structures are termed as *software clones*. Clone detection, analysis and visualization has been an active area of research in last two decades, e.g., see survey (Muhammad, et al., 2020).

In a number of studies, my team at the National University of Singapore (NUS) observed that extensive cloning sometimes occurred due to the lack of strong enough generic design mechanisms that would allow programmers to avoid repetitions without compromising other engineering goals that mattered to them. Much redundancy is common in old, heavily maintained programs. However, we found much cloning also in newly developed programs that, in our judgment, were well designed in view of their design goals and technology used.

As many of these recurring program structures represented important concepts from software requirements or design spaces, these observations seemed to point to some interesting and may be fundamental issue, worth investigation. In follow up research, my team developed a clone detection tool called Clone Miner (Basit and Jarzabek, 2009) that allowed us to find and study large-granular clones (Kumar et al., 2016) in addition to similar code fragments. We also developed an Adaptive Reuse Technology, ART[2] for representing groups of cloned code structures with parameterized, generic, adaptable, therefore, reusable meta-components. We applied these tools in projects across a wide range of application domains and programming platforms, observing consistently 50%-90% levels of redundancies. In this paper, I summarize our

[1] This study was supported by a grant S/WI/2/2018 from Bialystok University of Technology and founded from the resources for research by Ministry of Science and Higher Education

[2] Adaptive Reuse Technology, http://art-processor.org

Jarzabek, S.
Software Similarity Patterns and Clones: A Curse or Blessing?.
DOI: 10.5220/0009820000050017
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 2*, pages 5-17
ISBN: 978-989-758-423-7

findings, with references to relevant publications. A detailed discussion of our earlier projects can be also found in a monograph (Jarzabek, 2007).

Generic design can help avoid redundancies, reducing conceptual complexity, as well as the physical size of programs. STL (Musser and Saini, 1996) is a premier example of engineering benefits of generic design in the domain of data structures. However, in many other domains the potential of similarity patterns for program simplification and reuse remains often untapped. Software Product Line SPL approach (Clements and Northrop, 2002) attempts to address the problem at the architecture-component level. As it is often the case, "the devil is hidden in detail", and we need much finer level variability management mechanism to tackle redundancies and fully reap their potential for program simplification and reuse.

In this paper, I share experiences from my research on clones. In the remaining sections, I discuss the multi-faceted nature of software redundancy, software clone definition, the reasons why clones occur in programs, their impact on software development and maintenance, productivity benefits that can be gained by avoiding clones, and difficulties to realize these benefits with conventional programming languages and design techniques.

2 DEFINING CLONES

Software similarity is a multi-faceted phenomenon that escapes precise definition. The notion of similarity changes depending on the context: Whether or not we consider two code structures as similar depends on what we want to do with them.

We can characterize clones that are likely to meet our goals by metrics such as the minimum size of clones, the percentage of common code among clones (Kamiya et al., 2002), or the editing distance (Levenhstein, 1966) among clones, measured in terms of editing operations required to convert one text fragment to another.

We introduce clones informally as follows: Two program structures of considerable size are clones of each other if they meet a certain user-defined threshold of similarity measure. The required size and similarity threshold are subjective, vary with context, and therefore must be set by a programmer to meet goals of a specific clone analysis exercise.

Most of the interesting clones are similar but not identical. Differences among clones result from changes in their intended behaviour, and from dependencies on the specific program context in which clones are embedded (such as different names of referenced variables, methods called, or platform dependencies).

Clones may or may not represent program structures that perform well-defined functions. Considering their form and size, we distinguish two types of clones, namely:

- *simple clones:* similar segments of contiguous code such as program functions or any code fragments (Ducasse, 1999)(Kamiya et al., 2002)
- *structural clones*: patterns of inter-related components/classes emerging from design and analysis spaces, or from design solutions repeatedly applied by programmers (Basit and Jarzabek, 2005)(Basit and Jarzabek, 2009). Examples include design patterns (Gamma et al., 1995), analysis patterns (Fowler, 1997) enterprise patterns using .NET™, core J2EE™ patterns, and so-called "mental templates" (Baxter et al., 1998).

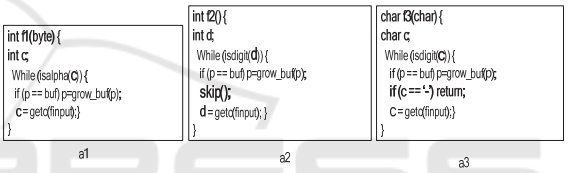

Figure 1: Simple clones.

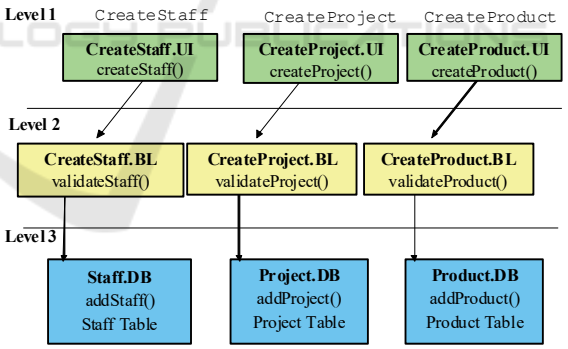

Figure 2: Structural clones.

Figure 1 and Figure 2 show intuitive examples of simple and structural clones. Three code fragments (a1,a2,a3) in Figure 1 differ in code details highlighted in **bold**. We can consider them as simple clones of each other, provided they meet a user-defined similarity threshold.

Figure 2 shows three structures implementing features `CreateStaff`, `CreateProject`, `CreateProduct` in a web portal for project management. Boxes are PHP files implementing user interface (at the top), business logic (in the

middle), and database aspects of respective features. Each of the files consists of PHP functions. Functions across files at each level (UI, BL and DB) are similar to each other forming simple clones such as shown in Figure 1. As PHP files are densely covered by clones, we consider files at each level as similar (abstraction step). As there is a calling relation among functions in respective PHP files, the three structures form a collaborative structural clone class. We conclude that features `CreateStaff`, `CreateProject`, and `CreateProduct` are similar to each other (abstraction step).

Clone detection techniques (Basit and Jarzabek, 2009)(Baxter et al., 1998)(Ducasse et al., 1999)(Kamyia et al., 2002) can automate finding clones in programs, and refactorings (Fowler, 1999) can help us to free programs from some clones. At times, clone elimination may be hindered by risks involved in changing programs (Cordy, 2003), or by other design goals that conflict with refactorings (Jarzabek and Li, 2003)(Kim et al., 2004).

3 SOFTWARE SIMILARITY PHENOMENON

3.1 Reasons for Cloning

Whether clones are good or bad it all depends on the motivation for cloning, the role clones play in a program, and the perspective from which we judge the impact of clones.

Similarities stem from different sources, depending on the nature of an application domain and design techniques used. Therefore, the form of clones, as well as reasons why they are there, vary across different program situations. Poor design and ad hoc maintenance are the two often-mentioned reasons for cloning. Such repetitions can often be avoided with good design or refactored.

Despite the benefits of non-redundancy, at times, cloning is done in a good cause. With *copy-paste-modify* practice, we can speed up development, achieving quick productivity gains. Developers also duplicate code to improve program performance or reliability. Such repetitions are intentional and should not be eliminated from a program even if a suitable refactoring could do the job. In maintenance of legacy software, changes involved in refactoring clones may create risks that are unacceptable for business reasons (Cordy, 2003) – it is safer to maintain own piece of code rather than a generic solution shared with other developers who may also be changing the same functionality. (Kapser and Godfrey, 2006) discuss a number of situations that justify cloning. Developers may choose to live with repetitions, as the lesser of the two evils, for variety of such reasons.

Modern component platforms (such as JEE™ or .NET™) encourage architecture-centric, pattern-driven design that naturally induces much redundancy to programs. Patterns lead to beneficial standardization of program solutions and are basic means to achieve reuse of common service components. Standardization of program solutions has many benefits, and creates an interesting case for our discussion of clones. At times IDEs support application of major patterns, or programmers use manual *copy-paste-modify* to apply yet other patterns. Representing patterns in generic form and enhancing their visibility can be beneficial, as it reveals a simpler view of a program. With generic pattern representation, we can provide better support for pattern instantiation and injection of pattern instances into a software system under construction. Generic design can help us avoid explosion of look-alike program structures, pattern instances. The knowledge of the location of pattern instances and the exact differences among them is helpful in understanding, maintenance and reuse.

Yet other repetitions occur because avoiding them with conventional approaches is either impossible or would require developers to compromise other important design goals. Kim estimates that 34% of clones cannot be refactored (Kim, 2004). This type of unavoidable repetitions is of our primary interest in this paper. We pay special attention to large-granularity program structures (Kumar et al., 2016), signifying important design concepts, recurring many times in variant forms, whose noticing may bring significant engineering benefits.

Summarizing the above discussion, we classify clones into the following categories:

1. *Desirable*. Such clones are useful at runtime (e.g., for performance or reliability) and cannot be eliminated from programs. Intentional clones induced by an implementation technique (e.g., by J2EE or .NET architecture and patterns) also belong to this category.

2. *Avoidable*. These clones are caused by the programmer's carelessness. For example, similar code fragments introduced by poor design or ad hoc *copy-paste-modify* practice during maintenance often fall into this category.

3. *Problematic*. These are all the clones that are not *desirable* but are difficult to avoid using conventional design techniques, without compromising important design goals. As the name suggests, nothing definite can be said about problematic clones. They are relative to design techniques and design goals. Despite their enigmatic nature, we find the concept useful in discussing cloning problems. Most of the clones discussed in the Buffer library case study belong to this category.

Katsuro Inoue, one of the precursors of software clone research, is an author of CCFinder (Kamiya et al., 2002), a clone detection tool used by thousands of software companies in Japan and world-wide for software quality assessment (Yamanaka et al., 2012). They consider the extent of cloning as one of the important indicators to estimate the expected maintenance cost in outsourcing software maintenance. The relation between cloning and the cost of changing programs can be explained as follows: Even if clones are created with good intentions, most of the clones increase the risk of update anomalies, and hinder program understanding during the maintenance in at least, two ways: (1) a programmer must maintain more code than he/she would have to maintain should the clones be removed, and (2) when one logical source of change affects many instances of a replicated program structure scattered throughout a program, to implement a change, a programmer must find and update all the instances of the replicated structure. The situation is further complicated if instances of an affected program structure must be changed in slightly different ways, depending on the context.

3.2 How Much Cloning?

In controlled lab experiments and industrial projects, we typically observed 50%-90% rates of repetitions in newly developed, well-designed programs. Our studies covered a range of application domains (business systems, Web Portals, command and control, mobile device applications, class libraries), programming languages (Java, C++, C#, JSP, PHP) and platforms (J2EE, .NET, Unix, Windows). For example, the extent of similarities in Java Buffer library was 68% (Jarzabek and Li, 2003), in parts of STL (C++) - over 50% (Basit et al., 2005), in Web Portal (J2EE) – 68% (Yang and Jarzabek, 2006), and in certain .NET Web Portal modules – up to 90% (Pettersson and Jarzabek, 2005). A survey of 17 Web Applications revealed 17-60% of code contained in clones (Rajapakse and Jarzabek, 2005). We measured the percentage of redundancies by comparing the subject program against a non-redundant representation for the subject program built with ART outlined later in this paper. Not always does size reduction lead to program simplification. However, we focused only on repetitions that created reuse opportunities, induced extra conceptual complexity into a program, and/or were counter-productive for maintenance.

Other studies revealed lower, but still substantial rates of repetitions, 20%-30% (Kim et al., 2004), and indicated that 49%-64% of clones “were not easily refactorable due to programming language limitations”. It is important to note that these other studies focus only on cloned code fragments, while our notion of similarity and studies cover large-granularity program structures, that may involve, for example, patterns of collaborating components recurring in variant forms.

4 GENERIC DESIGN

Generic design aims at achieving non-redundancy by unifying differences among similar program structures. The importance of generic design in managing software complexity have been recognized for long. Macros were one of the early attempts to parameterize programs and make them more generic. (Gougen, 1984) popularized ideas of parameterized programming. Among programming language constructs, type parameterization (called generics in Ada, Eiffel, Java and C#, and templates in C++), higher-order functions, iterators, and inheritance can help avoid repetitions in certain situations (Garcia et al., 2003). Design techniques such as design patterns (Gamma et al., 1995), table-driven design, and information hiding are supportive to building generic programs.

There are three engineering benefits of generic design (and three reasons to avoid unnecessary repetitions): Firstly, genericity is an important theme of software reuse where the goal is to recognize similarities to avoid repetitions across projects, processes and products. Indeed, many repetitions merely indicate unexploited reuse opportunities. Secondly, repetitions hinder program understanding. Repeated similar program structures cause update anomalies complicating maintenance. Thirdly, by revealing design-level similarities, 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” (Brooks, 1986). Common sense suggests that developers should be able to express their design and code without unwanted repetitions, whenever they wish to do so.

5 REDUNDANCIES IN STL

STL (Musser, 1996) is a classical and powerful example of what skilful generic design can do for complexity reduction. STL implements commonly used algorithms, such as sort or search, for a variety of container data structures. Without generic containers and algorithms, the STL’s size and complexity would be enormous, hindering its evolution. Such simple-minded solution would unwisely ignore similarity among containers, and among algorithms applied to different containers, which offers endless reuse opportunities. Generic design with templates and iterators helped STL designers to avoid these complications, without compromising efficiency.

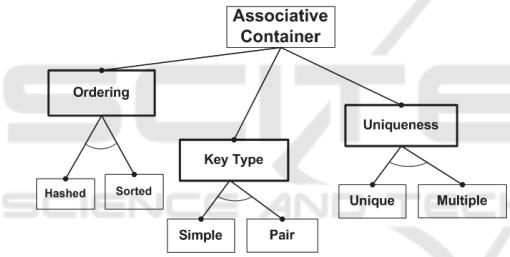

Figure 3: Associative container features (STL).

In STL, generic solutions are mainly facilitated by templates and iterators. We analysed associative containers - variable-sized containers that support efficient retrieval of elements based on keys. Figure 3 shows variant features of associative containers. There are eight STL templates, one for each of the eight legal combinations of features.

We focused on the STL regions that showed high cloning rates. We ran a clone detector to identify these regions. We found that container classes displayed a remarkable amount of similarity and code repetition. Four ‘sorted’ associative containers and four ‘hashed’ associative containers could be unified into two generic ART containers, achieving 57% reduction in the related code. Stack and queue classes contained 37% of cloned code. Algorithms set union, set intersection, set difference, and set symmetric difference (along with their overloaded versions) formed a clone class with eight instances. On overall, non-redundant representation of these parts of STL in ART contained 48% of code found in the original STL (Basit et al., 2005).

There were many non-type-parametric differences among associative container templates. For example, certain otherwise similar methods, differed in operators or algorithmic details. While it is possible to treat many types of non-parametric differences using sophisticated forms of C++ template meta-programming, often the resulting code becomes “cluttered and messy” (Czarnecki and Eisenecker, 2000). We did not spot such solutions in STL, and believe their practical value needs to be further investigated.

The reader may find full details of the STL case study in (Basit et al., 2005).

6 REDUNDANCIES IN THE JAVA BUFFER LIBRARY

A buffer contains data in a linear sequence for reading and writing. Buffer classes differ in features such as a memory scheme: Heap or Direct; element type: byte, char, int, double, float, long, or short; access mode: writable or read-only; byte ordering: S – non-native or U – native; B – BigEndian or L – LittleEndian.

Each legal combination of features yields a unique buffer class, with much similarity among classes. As we combine features, buffer classes grow in number, as observed in (Batory et al., 1993). Some of the buffer classes are shown in Figure 4. A class name, such as DirectIntBufferRS, reflects combination of features implemented into a given class. Class names are derived from a template: [MS][T]Buffer[AM][BO], where MS – memory scheme: Heap or Direct; T – type: int, short, float, long double, char, or byte; AM – access mode: W – writable (default) or R - read-only; BO – byte ordering: S – non-native or U – native; B – BigEndian or L – LittleEndian. All the classes whose names do not include ‘R’, by default are ‘W’ – writable. VB – View Buffer is yet another feature that allows us to interpret byte buffer as Char, Int, Double, Float, Long, or Short. Combining VB with other features, yields 24 classes ByteBufferAs[T]Buffer[R][B|L]. The last parameter [B|L] means “B or L”.

The experiment covered 74 buffer classes that contained 6,719 LOC (physical lines of code, without blanks or comments). We identified seven groups of similar classes where each group comprised 7-13 classes:

1. [**T**]Buffer: 7 classes at Level 1 that differ in buffer element type, **T**: int, short, float, long double, char, or byte
2. Heap[**T**]Buffer: 7 classes at Level 2, that differ in buffer element type, **T**
3. Heap[**T**]BufferR: 7 read-only classes at Level 3
4. Direct[**T**]Buffer[**S**|**U**]: 13 classes at Level 2 for combinations of buffer element type, **T,** with byte orderings**: S** – non-native or **U** – native byte ordering (notice that byte ordering is not relevant to buffer element type 'byte')
5. Direct[**T**]BufferR[**S**|**U**]: 13 read-only classes at Level 3 for combinations of parameters **T**, **S** and **U**, as above
6. ByteBufferAs[**T**]Buffer[**B**|**L**]: 12 classes at Level 2 for combinations of buffer element type, **T,** with byte orderings: **B** – Big_Endian or **L** – Little_Endian
7. ByteBufferAs[**T**]BufferR[**B**|**L**]: 12 read-only classes at Level 3 for combinations of parameters **T**, **B** and **L**, as above.

Classes in each of the above seven groups differed in details of method signatures, data types, keywords, operators, and editing changes. We paid attention only to similarities whose noticing could simplify class understanding and help in maintenance. Some of the classes had extra methods and/or attributes as compared to other classes in the same group. Many similar classes or methods occurred due to the inability to unify small variations in otherwise the same classes or methods. Generics could unify 15 among 74 classes under study, reducing the code size by 27%. The solution with generics was subject to certain restrictions that we discussed in (Jarzabek and Li, 2006).

So why did Buffer library designers chose to keep redundancies?

Any solutions to unifying similarities must be considered in the context of other design goals developers must meet. Usability, conceptual clarity and good performance are important design goals for the Buffer library. To simplify the use of the Buffer library, the designers decided to reveal to programmers only the top eight classes (Figure 4). For conceptual clarity, designers of the Buffer library decided not to multiply classes beyond what was absolutely needed. We see almost one-to-one mapping between legal feature combinations and buffer classes.

In many situations, designers could introduce a new abstract class or a suitable design pattern to avoid repetitions. However, such a solution would compromise the above design goals, and therefore was not implemented. Many similar classes or methods were replicated because of that.

Many similarities in buffer classes sparked from feature combinations. As buffer features (such as element type, memory scheme, etc.) could not be implemented independently of each other in separate implementation units (e.g., class methods), code fragments related to specific features appeared with many variants in different classes, depending on the context. Whenever such code could not be parameterized to unify the variant forms, and placed in some upper-level class for reuse via inheritance, similar code structures spread through classes.

Method **hasArray()** shown in Figure 5 illustrates a simple yet interesting case. This method is repeated in each of the seven classes at Level 1. Although method **hasArray()** recurs in all seven classes, it cannot be implemented in the parent class Buffer, as variable **hb** must be declared with a different type in each of the seven classes. For

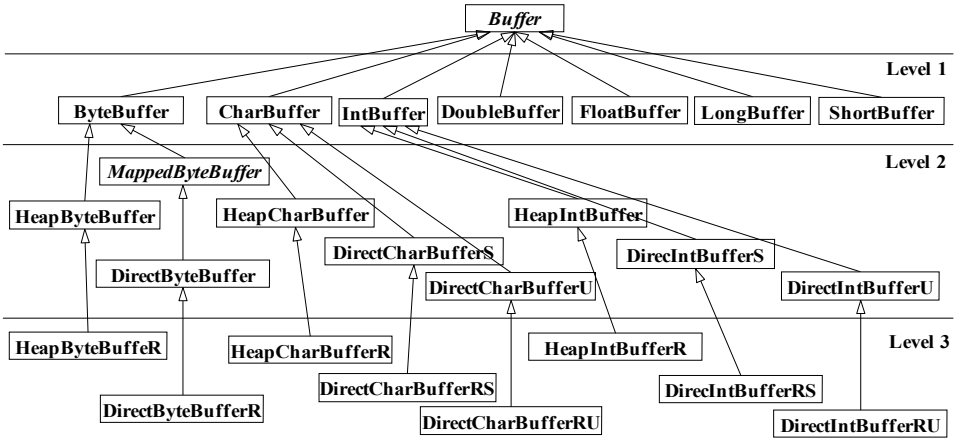

Figure 4: A fragment of the Buffer library.

example, in class **ByteBuffer** the type of variable **hb** is **byte** and in class **IntBuffer**, it is **int**.

```
/* Tells whether or not this buffer is backed by
      an accessible byte array. */
     public final boolean hasArray() {
     return (hb != null) && !isReadOnly; }
```

Figure 5: Recurring method hasArray().

One could presume that type parameterization, JDK 1.5 supports generics, should have a role to play in unifying parametric differences among similar classes. However, generics have not been applied to unify similarity patterns described in our study. Groups of classes that differ only in data type are obvious candidates for generics. There are three such groups comprising 21 classes, namely **[T]Buffer**, **Heap[T]Buffer** and **Heap[T]BufferR**. In each of these groups, classes corresponding to Byte and Char types differ in non-type parameters and are not generics-friendly. This leaves us with 15 generics-friendly classes whose unification with three generics eliminates 27% of code. There is, however, one problem with this solution. In Java, generic types cannot be primitive types such as int or char. This is a serious limitation, as one has to create corresponding wrapper classes just for the purpose of parameterization. Wrapper classes introduce extra complexity and hamper performance. Application of generics to 15 buffer classes is subject to this limitation.

```
/*Creates a new byte buffer containing a shared
  subsequence of this buffer's content. */
public ByteBuffer slice() {
      int pos = this.position();
      int lim = this.limit();
      assert (pos <= lim);
      int rem = (pos <= lim ? lim - pos : 0);
      int off = (pos << 0);
      return new DirectByteBuffer(this, -1, 0, rem,
        rem, off); }
```

Figure 6: Method **slice().**

Repetitions often arise due to the inability to specify small variations in otherwise identical code fragments. Many similar classes and methods differ in parameters representing constants, keywords or algorithmic elements rather than data types. This happens when the impact of various features affects the same class or method. For example, method **slice()** (Figure 6) recurs 13 times in all the Direct[T]Buffer[S|U] classes with small changes highlighted in bold in. Generics are not meant to unify this kind of differences in classes.

In summary, generics are rather limited in unifying similarity patterns that we find in practical situations, e.g., such as we observed in the Buffer library. It is interesting to note that repetitions occur across classes at the same level of inheritance hierarchy, as well as in classes at different levels of inheritance hierarchy. Programming languages do not have a proper mechanism to handle such variations at an adequate (that is a sufficiently small) granularity level. Therefore, the impact of a small variation on a program may not be proportional to the size of the variation.

Developers of the Buffer library used macros, scripts and makefiles in order to exploit similarities and write/maintain buffer classes with less effort (these macros and scripts can be found in the Community Source Release for the Buffer library). While the reasons why Sun developers escaped to non-OO solution and the solution itself are not explained or documented, its existence hints at difficulties to treat similarity patterns with conventional OO techniques, given the overall design goals the Buffer library had to meet.

7 TOWARDS NON-REDUNDANCY

While practitioners are aware of much repetitions in software, they also know how difficult it is to avoid them. Problems with implementing effective reuse strategies (Deelstra et al., 2000) evidence these difficulties, as well.

It is not clear if and how we could implement buffer classes without redundancies in any of the conventional programming languages. A possible solution calls for flexible parametrization unconstrained by the rules of a programming language. It is as if our need to express program behaviour was in conflict with our need to achieve non-redundancy. To resolve this conflict, generative approaches propose to think about programs at two levels: a meta-level that provides a platform for program construction, and a level of actual program that is compiled and executed. Program generation technologies offer solutions for specific application domains, with abstract notations to specify required program behaviour (a meta-level), and a generator that encodes the semantics of a given application domain, and generates a program ready for execution. Quite often much redundancy can be

avoided in abstract program specifications. We comment further on generation approaches in the following section, and here we outline a general-purpose solution to non-redundancy, based on flexible parameterization at the meta-level, and code manipulation in pre-processing fashion. We explain the solution in a way that ART (Adaptive Reuse Technology) implements these concepts.

On the left-hand-side of Figure 7, we see a non-redundant meta-level representation of buffer classes. Boxes are ART templates that represent building blocks for Buffer classes. As such, they contain relevant Java code instrumented (parameterized) with ART commands. The purpose of parameterization is to enable reuse of ART templates in multiple contexts of the situations when a given functionality is need for building buffer classes. ART Processor interprets ART commands embedded in templates and generates buffer classes on the right-hand-side of Figure 7.

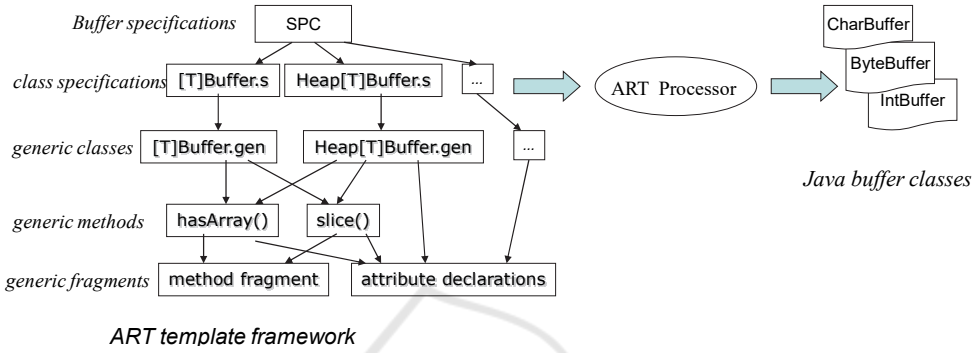

Figure 7: Non-redundant representation of Buffer classes in ART/Java.

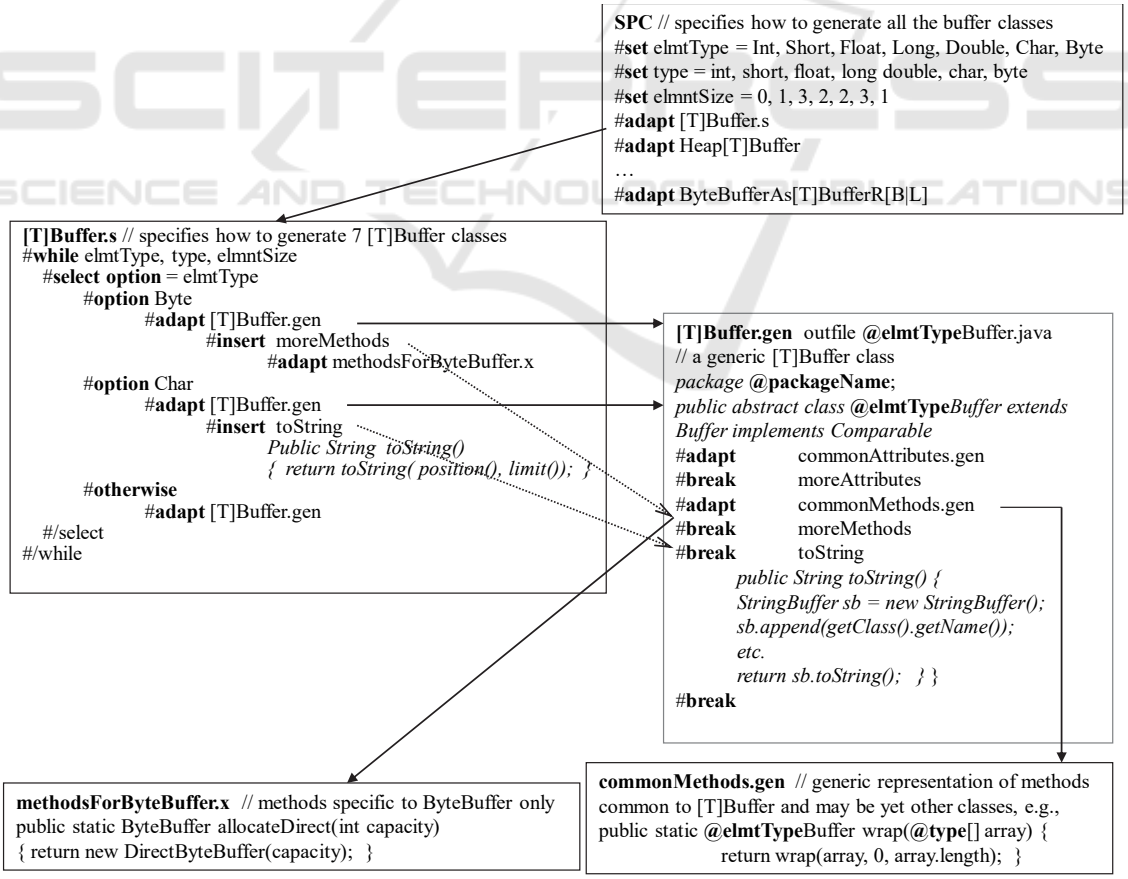

Figure 8: Non-redundant representation for seven [T]Buffer classes in Java/ART (partial).

An arrow between two templates: X → Y is read as "X adapts Y", meaning that X controls adaptation of Y. We have seven *generic class* templates, one for each of the seven groups of similar classes described in Section 6 (we show only two of them in Figure 7). Each class template defines common part of classes in the respective group. The essence of a generic component (generic class, in our case) is that it can be adapted to produce its instances (specific classes in a group, in our case). Smaller granularity generic building blocks for classes are defined below, namely class methods and fragments of method implementation or attribute declaration sections. Therefore, lower-level templates are composed, after possible adaptations, to construct required instances of higher-level generic components. At the top, we have specification elements – they tell the ART Processor how to generate specific buffer classes, from templates. Top-most SPC, sets up global parameters and exercises the overall control over the generation process.

ART Processor interprets the template framework starting from the SPC, traverses templates below, adapting visited templates and emitting buffer class code. By varying specifications, we can instantiate the same template framework in different ways, deriving different, but similar, program components from it.

We now explain the parameterization and adaptation mechanism, which is the "heart and soul" of how ART achieves goals of non-redundancy:

ART variables and expressions provide a basic parameterization mechanism to make templates generic. **#set** command assigns a value to a variable. Typically, names of program elements manipulated by ART, such as components, source files, classes, methods, data types, operators or algorithmic fragments, are represented by ART expressions. Such expressions are then instantiated by the ART Processor, according to the context. For example, names and other parameters of the seven similar classes **[T]Buffer** are represented by ART expressions in the a template **[T]Buffer.gen**.

ART variables have global scope, so that they can coordinate chains of all the customizations related to the same source of variation or change that spans across multiple templates. During processing of templates, values of variables propagate from an template where the value of a variable is set, down to the lower-level templates. While each template usually sets default values for its variables, values assigned to variables in higher-level templates take precedence over the locally assigned default values. Thanks to this overriding rule, templates become generic and adaptable, with potential for reuse in many contexts.

Other ART commands that help us design generic and adaptable templates include **#select**, #insert into **#break** and **#while**. We use **#select** command to direct processing into one of the many pre-defined branches (called options), based on the value of a variable. With **#insert** command, we can modify templates at designated **#break** points in arbitrary ways. ART expressions, **#select** and **#insert** into **#brea**k are analogous to AOP's mechanism for weaving advices at specified join points (Kiczales et al., 1997). The difference is that ART allows us to modify templates in arbitrary ways, at any explicitly designated variation points.

#while command iterates over template(s), with each iteration generating similar, but also different, program structures. A **#select** command in the #**while** loop allows us to generate classes in each of the seven groups discussed in Section 6.

Figure 8 illustrates how ART mechanisms realize the scheme outlined in Figure 7.

ART template names, ART commands and references to ART variables are shown in bold. References to ART variables parameterize code. For example, a reference to variable **@elmtType** is replaced by the variable's value during processing. Figure 6 shows ART template for method **slice()** from **Direct[T]Buffer[S|U]** classes. Values of variables set in **SPC** reach all their references in adapted ART templates. The value of variable **byteOrder** is set to an empty string, "S" or "U", in a respective #set command placed in one of the ART templates that **#adapt**'s ART template **slice.gen** (not shown in our pictures).

The **#while** loop in **[T]Buffer.s** is controlled by two multi-value variables, namely **elmtType** and **elmtSize**. The i'th iteration of the loop uses i'th value of each of the variables. In each iteration of the loop, the **#select** command uses the current value of **elmtType** to choose a proper #option for processing.

Attribute *outfile* of **[T]Buffer.gen** defines the name of a file where ART Processor will emit the code for a given class.

Having set values for ART variables, **SPC** initiates generation of classes in each of the seven groups of similar classes via suitable #adapt commands. ART template **[T]Buffer.gen** defines common elements found in all seven classes in the group. Five of those classes, namely **DoubleBuffer**, **IntBuffer, FloatBuffer**, **IntBuffer,** and **LongBuffer** differ only in type parameters (as in the

sample method **wrap()** shown in ART template **commonMethods.gen**). These differences are unified by ART variables, and no further customizations are required to generate these five classes from ART template **[T]Buffer.gen**. These five classes are catered for in **#otherwise** clause under **#select**. However, classes **ByteBuffer** and **CharBuffer** have some extra methods and/or attribute declarations. In addition, method **toString()** has different implementation in **CharBuffer** than in the remaining six classes. Customizations specific to classes **ByteBuffer** and **CharBuffer** are listed in the **#adapt** commands, under **#option** s **Byte** and **Char**, respectively.

We refer the reader to (Jarzabek and Li, 2003)(Jarzabek and Li, 2006) for further the details of this study.

A shorter program without redundancies does not automatically mean that such a program is easier to understand and maintain than a longer program with redundant code. For example, compressed code is short but impossible to read and understand. To further support claims of easier maintainability of the ART solution, we extended the Buffer library with a new type of buffer element – Complex. Then, we compared the effort involved in changing each of the two solutions, Java classes and Java/ART representation. Many classes must be implemented to address the Complex element type, but in this experiment we concentrated only on three of them, namely **ComplexBuffer**, **HeapComplexBuffer** and **HeapComplexBufferR**. In Java, class **ComplexBuffer** could be implemented based on the class **IntBuffer**, with 25 modifications that could be automated by an editing tool, and 17 modifications that had to be done manually. On the other hand, in the ART representation, all the changes had to be done manually, but only 5 modifications were required. To implement class **HeapComplexBuffer**, we needed 21 “automatic” and 10 manual modifications in Java, versus 3 manual modifications in ART. To implement class **HeapComplexBufferR**, we needed 16 “automatic” and 5 manual modifications in Java, versus 5 manual modifications in ART.

8 CLONES IN WEB PORTALS

8.1 ASP.NET Portal

In the ASP Web Portal (WP) Product Line project, our industry partner ST Electronics Pte. Ltd., Singapore, applied state-of-the-art conventional methods to maximize reusability of a Team Collaboration Portal (TCP). Still, a number of problem areas were observed that could be improved by applying ART to reduce redundancies. The benefits of ASP/ART TCP were the following:

- Short time (less than 2 weeks) and small effort (2 persons) to transform the ASP TCP into the first version of a mixed-strategy ASP/ART Product Line architecture.
- High productivity in building new portals from the ASP/ART solution. Based on the ASP/ART solution, ST Electronics could build new portal modules by writing as little as 10% of unique custom code, while the rest of code could be reused. This code reduction translated into an estimated eight-fold reduction of effort required to build new portals.
- Significant reduction of maintenance effort when enhancing individual portals. The overall managed code lines for nine portals under the ASP/ART were 22% less than the original single portal.
- Wide range of portals differing in a large number of inter-dependent features supported by the ASP/ART solution.

The reader may find full details of this project in (Pettersson and Jarzabek, 2005).

8.2 JEE Portal

In the follow up project, we evaluated J2EE™ as a platform for Product Line development. Unlike ASP, J2EE supports inheritance, generics and other OO features via Java.

Component platforms such as J2EE or .NET encourage organizing software around standard architectures. Patterns help programmers solve routine tasks in pre-defined ways in conformance to architectures. Application of patterns further standardizes software at macro and micro levels. Not surprisingly, we find much similarity in software developed in that way. Such uniformity of software structure is beneficial, as similar problems are always solved in a similar way across a system. It also facilitates easy reuse of common services/components provided by a platform. However, not always are pattern instances clearly visible in code. Pattern-driven development could be even more beneficial if we knew the exact location of pattern instances and how instances are similar and different one from each other. This would help in the future maintenance: When the pattern-related code is to be changed, it would be clear which of the pattern’s instances should be changed and how.

Currently, pattern-driven development is mainly limited to the middleware areas such as database communication, coordination between requests, application model and views (e.g., implied by the MVC organization) or reuse of common services. In application domain-specific areas, the benefits of patterns are less. At times, application of patterns may even scatter domain-specific functionality across many components (or classes), which complicates reuse of domain-specific code, and magnifies problems of tracing requirements to code.

In J2EE project, we applied ART to enhance the visibility of patterns and to achieve reuse in application domain-specific areas. We worked with a portal developed by ST Electronics, a variant of TCP. The portal supported collaborative work and included 14 modules such as Staff, Project and Task. We studied similarity patterns in presentation and business logic layers.

Within modules, we found 75% of code contained in exact clones, and 20% of code contained in similar clones (leaving only 5% of code unique). Analysis across modules, revealed design-level similarities, with 40% of code contained in structural clones. Both intra- and inter-module similarities were important for clarity of the design, however they could not be unified with generic design solutions expressed by J2EE mechanisms.

In the second part of the experiment, we applied ART to unify similarity patterns. Unification reduced the solution size by 61%, and enhanced the clarity of portal's conceptual structure as perceived by developers. In a controlled experiment, we found that to implement the same enhancement, J2EE/ART portal representation required 64% less modifications that the original J2EE portal.

The reader may find full details of this project in (Yang and Jarzabek, 2006).

9 GENERATORS

Powerful domain-specific solutions can be built by formalizing the domain knowledge, and using generation techniques to produce custom programs in a domain. Advancements in modelling and generation techniques led to Model-Driven Engineering (MDE) (Schmidt, 2006), where multiple, inter-related models are used to express domain-specific abstractions. Models are used for analysis, validation (via model checking), and code generation. Platforms such as Microsoft Visual Studio™ and Eclipse™ support generation of source code using domain-specific diagrammatic notations.

By constraining ourselves to a specific application domain, we can make assumptions about its semantics. A domain engineer encodes domain-specific knowledge into a generic, parameterized program solution. A developer, rather than working out all the details of the program solution, writes a concise, declarative problem description in a Domain-Specific Language (DSL). A generator uses DSL specifications to instantiate parameters of a generic solution to produce a custom program. Problem specifications in DSL are much smaller and simpler than the instantiated, complete and executable program solution. While we do not reduce the overall program complexity, generation-based solutions shield a programmer from complexities of the domain-specific code that is now manipulated by a generator. DSL may take many different forms, depending on a domain, from a formal text (e.g., BNF for parser generator), to visual interface (e.g., GUI) and to models (in Model-Driven Engineering approaches).

This is in contrast with ART which is an application domain- and programming language-independent technique. There is no concept of DSL in ART. Generators can be built in well-understood and fairly stable application domains. On the other hand ART, performs best in domains where frequent changes occur at both large and small granularity levels.

Generators must overcome a number of challenges to have a greater impact on practice. A common pitfall of generators is that abstract program specifications in DSL can get easily disconnected from the generated code. This happens when the generated code is modified by hand to accommodate changes not catered for by the DSL. As any re-generation of code would override such modifications, future maintenance must be done by hand and developers can't benefit from the generator anymore. Round-trip engineering could overcome this problem, but is difficult to achieve. This problem is particularly acute in the situation when we need to evolve multiple generated programs differing in certain features, as it is often the case of a Product Line. Implementing variant features in the generator will propagate all the variant features to all the programs, which may not be desirable. On the other hand, implementing variant features directly into generated programs that need them, automatically disconnects those programs from the generator.

Another problem faced by generators is that a problem domain served by a generator is often only a part of an overall programming problem

developers need to solve. Strategies for integrating multiple domain-specific generators and embedding them into systems implemented using yet other techniques have yet to be developed. One of the reason for success of compiler generators is that compilation on its own is a self-contained domain.

Rich abstractions lead to powerful generators. Without sufficient abstractions, there is not much we can automate. We believe not enough of general-purpose abstractions is the main reason why, despite much research, we have not achieved success in domain-independent, generation-based automatic programming. This also reminds us Brooks' doubts about reducing essential program complexity by means of abstraction (Brooks, 1986).

10 CONCLUSIONS

In the paper, I discussed a multi-faceted phenomenon of software similarities. Starting with software clone definition, I analysed common reasons why clones occur in programs, their impact on software development and maintenance, and productivity benefits that can be gained by avoiding clones. The core of the paper focused on redundancies that, despite potential benefits, are difficult to avoid with conventional programming languages and design techniques. Finally, I demonstrated a possible solution to avoiding such redundancies with meta-level generative techniques.

ACKNOWLEDGEMENTS

Author thanks PhD students and research assistants at the National University of Singapore who developed clone detection tools, implemented ART processor, and participated in studies on software redundancies.

REFERENCES

Basit, A.H., Rajapakse, D. and Jarzabek, S. 2005. Beyond Templates: a Study of Clones in the STL and Some General Implications. In *Int. Conf. Soft. Eng., ICSE'05,* St. Louis, USA, May 2005, pp. 451-459

Basit, A.H. and Jarzabek, S. 2005. Detecting Higher-level Similarity Patterns in Programs. In *ESEC-FSE'05, European Soft. Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Soft. Eng.*, ACM Press, September 2005, Lisbon, pp. 156-165

Basit, A.H. and Jarzabek, S. 2009. A data mining approach for detecting higher-level clones. In *IEEE Trans. on Soft. Eng*., 35(4): 497-514, 2009

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

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. 1998. Clone detection using abstract syntax trees. In *Proc. Int. Conf. Soft. Maintenance 1998*, pp. 368–377

Brooks, F. 1995. *The Mythical Man-Month*, Addison Wesley

Brooks, F.P. 1986. No Silver Bullet. 1986. In *Proc. IFIP 10th World Computing Conference*, H. K. Kugler, ed., Elsevier Science, pp. 1069-1076

Clements, P. and Northrop, L. 2002. *Software Product Lines: Practices and Patterns*, Addison-Wesley

Cordy, J. R., 2003. Comprehending Reality: Practical Challenges to Software Maintenance Automation. In *Proc. 11th IEEE Intl. Workshop on Program Comprehension*, pp. 196-206

Czarnecki, K. and Eisenecker, U., 2000. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley

Deelstra, S., Sinnema, M. and Bosch, J., 2000. Experiences in Software Product Families: Problems and Issues during Product Derivation. *Proc. Software Product Lines Conf.*, *SPLC3*, Boston, pp/ 165-182

Ducasse, S., Rieger, M. and Demeyer, S. 1999. A language independent approach for detecting duplicated code**.** In *Int. Conf. on Soft. Maintenance, ICSM'99,* Oxford, UK pp. 109-118

Fowler, M. 1997. *Analysis Patterns: Reusable Object Models*, Addison-Wesley

Fowler M. 1999. *Refactoring - improving the design of existing code*, Addison-Wesley

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley

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

Goguen, J.A. 1984. Parameterized Programming. *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 5, pp. 528-543

Hammad, M. Basit, H., Jarzabek, S. and Koschke, R. A Mapping Study of Clone Visualization. 2020. *Computer Science Review* (accepted in final form)

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

Jarzabek, S. and Li, S. 2006. Unifying clones with a generative programming technique: a case study. In *Journal of Software Maintenance and Evolution: Research and Practice*, John Wiley & Sons, Vol. 18, Issue 4, pp. 267-292

Jarzabek, S. 2007. *Effective Software Maintenance and Evolution*: *Reused-based Approach*, CRC Press Taylor and Francis

Kamiya, T., Kusumoto, S., and Inoue, K. 2002. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. In *IEEE Trans. Software Engineering,* 28(7): pp. 654-670

Kapser, C. and Godfrey M. 2006. "Cloning considered harmful" considered harmful. In *Working Conf. on Software Reverse Engineering, WCRE*, pp. 19-28 http://dx.doi.org/ 10.1007/s10664-008-9076-6

Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. 1997. Aspect-Oriented Programming. In *Europ. Conf. on Object-Oriented Programming,* Finland, Springer-Verlag LNCS 1241, pp. 220-242

Kim, M., Bergman, L., Lau, T. and Notkin, D. 2004. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. Int. Symposium on Empirical Software Engineering , ISESE'04*, Redondo Beach, California, pp. 83-92

Kumar, K. Jarzabek. S. and Dan, D. 2016. Managing Big Clones to Ease Evolution: Linux Kernel Example. *Federated Conference on Computer Science and Information Systems, FedCSIS, 36th IEEE Soft. Eng. Workshop*, pp. 1767 – 1776

Levenshtein, V.I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Cybernetics & Control Theory* 10-8, pp. 707-710

Musser, D. and Saini, A., 1996. *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley

Pettersson, U., and Jarzabek, S. 2005. Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach. In *ESEC-FSE'05, European Soft. Eng. Conf. and ACM Symp. on the Foundations of Soft. Eng.*, Lisbon, pp. 326-335

Rajapakse, D. and Jarzabek, S. 2005. An Investigation of Cloning in Web Portals. In *Int. Conf. on Web Eng, ICWE'05,* Sydney, pp. 252-262

Schmidt, D. 2006. Model-Driven Engineering. In *IEEE Computer*, pp. 25-31

Zhang, H. and Jarzabek, S. 2004. A Mechanism for Handling Variants in Software Product Lines. In special issue on Software Variability Management, *Science of Computer Programming*, Volume 53, Issue 3, pp. 255-436.

Yang, J. and Jarzabek, S. 2005. Applying a Generative Technique for Enhanced Reuse on J2EE Platform. In *4th Int. Conf. on Generative Programming and Component Engineering*, *GPCE'05*, Tallinn, Estonia, pp. 237-255

Yamanaka, Y., Choi, E., Yoshida, N., Inoue, K. and Sano, T. 2012. Industrial Application of Clone Change Management System. *Proc. 6th Int. Workshop on Software Clones, IWSC*, pp.67-71.