Genericity - a “Missing in Action” Key to Software Simplification and Reuse

Implications and Treatment

Stan Jarzabek
Department of Computer Science, School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 117543
stan@comp.nus.edu.sg

ABSTRACT
We hypothesize that certain program complexities and difficulties to realize reuse potentials have their roots in weak mechanisms for generic design of today’s programming techniques. We back this hypothesis with empirical studies of program similarity patterns. In three lab projects and two industrial projects, we found 50%-90% rates of repetitions that deliberately recurred in newly developed, well-designed programs. Not only did those repetitions increase conceptual complexity and size of programs, but also signified unexploited reuse opportunities. With suitable generic design solutions unifying similarities, we could avoid repetitions, raising reuse levels proportionally to the rates of repetitions that we found. Despite this potential benefits, avoiding repetitions with conventional generic design solutions was either impossible or would require developers to compromise other important design goals. We believe these problems are common to many programs, and in the second part of the paper, where we interpret empirical studies, we discuss some of the fundamental phenomena that trigger the problems. As a constructive solution to the observed difficulties, we propose to apply a generative technique based on the industrial applications of our approach. We discuss trade-offs involved in employing meta-level techniques, based on the industrial applications of our approach and our own studies.

Categories and Subject Descriptors
D.1.5 [Programming Techniques]: Object-Oriented Programming; D.2.2 [Software Engineering]: Design Tools and Techniques – Object-oriented design methods, Software libraries; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.10 [Software Engineering]: Design – Representations; D.2.13 [Software Engineering]: Reusable Software - domain engineering;

General Terms
Design, Languages, Experimentation

Keywords
generic design, generics, reuse, maintenance, component-based development, static meta-programming

1. Introduction
Much similarity within and across programs creates potential for program simplification and reuse via generic design solutions. The main themes of this paper are software similarity patterns and generic design to unify them, in relation to program understanding, changeability and reuse.

By similarity patterns, we mean similar program structures, of significant importance, repeated many times within a program or across programs. We consider similarities of any kind or granularity, and at all the abstraction levels, from problem domain, to architecture, to detailed design, and down to the code level. The term covers recurring simple code fragments, as well as patterns of collaborating classes and architectural patterns of components.

Generic design, as understood in this paper, aims at achieving non-redundancy for simplification reasons, by unifying similarity patterns at design and code levels, whenever there is an engineering benefit of doing so (e.g., for easier changeability or reuse).

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, repeated similar program structures cause update anomalies complicating maintenance – vide research on re-engineering [7], refactoring [17] and clone detection [7][21][35]. 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” [11]. Common sense suggests that developers should be able to express their design and code without unwanted repetitions, whenever they wish to do so.

Poor design and ad hoc maintenance are the two often mentioned reasons why programmers multiply similar code structures (often termed as clones in literature). It must be noted that, at times, cloning is done in a good cause, for example, to speed up development, increase program’s performance or improve its reliability. Such repetitions cannot be eliminated from a program even if a suitable refactoring [17] could do the job. In our studies, however, we did not focus on repetitions that played a constructive role in the above sense, or occurred because of mere convenience of “copy-paste-modify” practice, to achieve quick productivity gains.

Identifying similarities and variability in software is the prime goal of domain analysis for reuse. Architectural approaches,
component-based, OO design techniques (inheritance, design patterns and OO frameworks), mechanisms provided by modern platforms such as J2EE™ or .NET™, and language features such as generics (or C++ templates) are main means to exploit similarity patterns observed across software products in a domain.

In the paper, we argue that the above mechanisms alone are not sufficient to unify many similarity patterns that commonly occur in programs. As a result, we fail to exploit reuse opportunities in the areas where reuse has much potential benefits to offer.

We back the above observation by empirical studies in which we tried to understand the nature of software similarities, their causes, the ways they manifest in well-designed programs as recurring similar program structures, and the difficulties to deal with them in the frame of conventional programming techniques. In the paper, we present empirical results from three lab studies and two industrial projects that showed 50%-90% rates of repetitions that deliberately recurred in newly developed, well-designed programs. Our studies of similarity patterns covered a range of application domains (business systems, Web Portals, command and control, class libraries), programming languages (Java, C++, C#, ASP, JSP, PHP) and platforms (J2EE, .NET, Unix, Windows) (3][4][20][29][37]. Similarity patterns ranged from similar class methods, to similar classes and to patterns of collaborating classes/components representing relatively large parts of a program. For example, the extent of similarities in Java Buffer library was 68% [20], in parts of STL, a well-known example of powerful generic solution - over 50% [3], in Web Portal (J2EE) – 68% [37], and in certain ASP Web Portal modules – up to 90% [29] (contributed by our industry partner ST Electronics). A survey of 17 Web Applications revealed 17-60% of code contained in clones [31].

In all the above studies, with the exception of [31], we gave attention only to repetitions of significant engineering importance, meaning that they created reuse opportunities, induced extra conceptual complexity into a program, and/or were counter-productive for maintenance. Avoiding them with conventional approaches was either impossible or would require developers to compromise other important design goals.

In the latter case, developers often chose to live with repetitions as the lesser of the two evils (as also observed in [13]). An independent study [25] makes an even stronger statement about the reasons why clones remain in code saying that 49% to 64% of clones “are not easily refactorable due to programming language limitations”.

The extent that similar structures deliberately recurred in programs under study encouraged us to have a deeper look into the issue, which revealed a general nature of observed symptoms and their causes, as discussed in Section 5.

We believe that to unify many types of similarity patterns commonly found in programs, we need generic structures that are highly flexible, easily adaptable to variety of the specific forms that are required in multiple program contexts where their concrete instances occur. We hypothesize that conventional programming approaches mostly lack strong enough mechanisms to define such generic structures, and in many cases we need accept these limitations or look beyond current paradigms for effective solutions.

In previous papers [3][4][20][29][37], we described how complex similarity patterns can be unified with generic design formed by meta-structures imposed on top of conventional programs. Such a synergistic application of conventional and meta-level generative techniques [14] we call a “mixed strategy” approach. We discuss one such technique, called XVCL [39], in this paper. We show how the “mixed strategy” approach can raise reuse levels, proportionally to the rates of repetitions we found in programs under study. The approach is practical as it complements, rather than competes with other programming paradigms. It has been successfully applied in industrial projects [29].

The paper is organized as follows: In Section 2, we discuss related work. In Section 3, we briefly review a meta-level technique of XVCL [39]. In Section 4, we present results from empirical studies of similarities and generic design solutions. In Section 5, we postulate the general nature of observed symptoms and their causes. As none of the solutions is without pitfalls, in Section 6, we discuss trade-offs involved in employing meta-level techniques. Concluding remarks end the paper.

2. Related work

Modular decomposition with information hiding [28], macros, generics in Ada or Java [19], templates in C++, other forms of parameterization such as higher order functions [35], inheritance with dynamic binding, and design patterns [18] are some of the conventional design techniques to achieve genericity. Aspect-Oriented Programming (AOP) [24] and MDSOC [34] support genericity by separating cross-cutting concerns. In AHEAD [6] (based on the earlier Batory’s work on GenVoca), genericity is supported by feature composition and refinement. Many techniques described under the umbrella of generative techniques [14] achieve some forms of separation of concerns that is supportive to genericity (although avoiding repetition is not the prime goal of these techniques). Domain analysis [30] is essential in identifying high-level, large granularity patterns of similarity. Generic solutions unifying such patterns are most beneficial for programmer’s productivity as they can significantly reduce the size and complexity of the solution. Software architectures [9][12], architectural styles [32] and patterns [12] help developers avoid repeatedly designing the same solution by providing component plug-in plug-out capability. Component platforms such as J2EE™ or .NET™, provide also an infrastructure for reuse of pre-defined common services.

Code cloning has received much attention in research. As clones are closely related to the notions of similarity patterns and genericity, we discuss them in this section. Cloning has been studied in the context of re-engineering [7], refactoring [17] and clone detection [7][21][2]. In an empirical study of cloning practices Kim et al. [25] observed that “Limitations of particular programming languages produce unavoidable duplicates in a code base”.

We introduce clones informally as follows: Two code structures of considerable size are clones of each other if there is significant similarity between them. The actual size and similarity (which can be measured, for example, in terms of percentage of repeated code) varies depending on the context, and is left to human judgment. Clones may or may not represent program structures that perform well-defined functions. The above notions involve human judgment and, therefore, are subjective. The very concept of similarity escapes precise definition.

We distinguish two types of clones, namely:

- **simple clones**: contiguous segments of similar code such as class methods or fragments of method implementation, and
> *structural clones*: patterns of inter-related classes emerging from design and analysis spaces; patterns of components at the architecture level; design solutions repeatedly applied by programmers to solve similar problems (so-called “mental templates” [7]).

Most of the interesting clones, particularly structural clones, are similar but not identical. Changes among clones result from differences in intended behavior, 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).

Programmers clone code to speed up development and maintenance. Sometimes, cloning is done to increase the robustness of life-critical systems, for better performance, or to minimize dependencies among developers in large projects. Poor design and ad hoc maintenance may induce clones. Clones arise when developers face difficulties of avoiding them using conventional techniques. Clones may be generated by tools. Recurring problems of a similar structure in analysis and design spaces lead to structural clones. Analysis patterns [16] and design patterns [18] exemplify these situations.

While there are good reasons for creating certain clones, most of them, independently of the reasons why they occur, are counterproductive for future maintenance, as they increase the risk of update anomalies: When change affects a replicated fragment, a programmer must find and update all the instances of it. The situation is further complicated if an affected fragment must be changed in slightly different ways, depending on the context.

Clone detection techniques [7][21] can automate finding clones in existing programs, and refactorings [17] help us to free programs from clones. At times, clone elimination may be hindered by risks involved in changing programs [13], or by other design goals that conflict with refactorings [20][25].

3. “Mixed strategy” approach to generic design with XVCL

Parameterization is a prime concept for generic design. In XVCL [39], we use unrestricted meta-level parameterization which, unlike conventional generics [19], is decoupled from the language core. The overall generic design solution is decomposed into a hierarchy of parameterized meta-components (denoted by capital letters in Figure 1). Each meta-component in the hierarchy (e.g., G) defines a generic design solution in terms of lower-level meta-components (L and N), and also contributes to generic design solutions at the higher level (B and C), as their building block. Instructions of how to instantiate the generic design are contained in the SPC. XVCL Processor interprets SPC, traverses meta-component hierarchy accordingly, adapts visited meta-components and emits the custom program. By varying specifications contained in the SPC, we can instantiate the same generic design in different ways, obtaining different custom programs.

We can build a meta-level generic design to simplify a single program or to form a generic representation from which we can derive many similar programs (so-called product line architecture [9][12]). In the first case, instances of meta-components populate a single program. By building the meta-level generic design representation we gain changeability, as non-redundancy reduces the risk of update anomalies. In the second case, instances of meta-components populate many similar programs that we can build from the generic design. We gain reusability across similar programs. Two projects described in Section 4.1 exemplify the first case, while projects described in Sections 4.2 and 4.3 exemplify the second case.

XVCL is not yet another programming language. We still use one of the programming languages to define program behavior. The usual strategy is to develop a program solution using conventional design techniques, as long as they yield a simple generic design solution that does not conflict with other design goals. Then, we apply XVCL to deal with cases that can benefit from genericity, but for which conventional techniques fail to provide a workable generic solution. XVCL approach complements, rather than competes with contemporary programming paradigms. An overall solution formed by a conventional technique and XVCL we call “mixed strategy” solution.

![Figure 1. Generic design with XVCL](image)

Meta-components may represent program elements of arbitrary kind, structure or complexity, such as functions, classes, or architectural elements (interfaces and components). Meta-components can be parameterized in fairly unrestrictive ways. Parameters range from simple values (such as strings), to types and to other meta-components. Parameters mark variation points in meta-components and their placement is not restricted in any way.

Meta-level partitioning is parallel to partitioning of a program along component boundaries (at architecture level) and class, function/method and statement boundaries (at the program level). The boundaries of meta-level partitioning are not restricted by rules of a programming language or the semantics of a programming problem being solved. The boundaries are solely dictated by the concerns of generic design.

XVCL achieves generic design by flexible “composition with adaptation” mechanism which works as follows: At the meta-level, differences among similar program structures are unified by a unique generic structure. Variations among similar program structures are specified as deltas from the generic structure and automatically propagated to the respective clones in a program. This feature is critical for genericity as it allows us to adapt meta-components representing a generic solution to form specific program structures in a variety of required variant forms. From the XVCL window, a designer has a precise picture of program similarities (a generic structure) and differences among instances of a generic structure. Any future changes are also done via generic structures. XVCL provides simple yet powerful mechanisms to exercise a full control over the cases when certain instances of a generic structure are to be treated differently from others. Unavoidably redundant code may be emitted as a result, but that code is no longer the canonical specification of the solution. Non-redundant generic structures together with their instantiating deltas now play that role.
XVCL is based on Frame Technology [1]. It has been described in more detail in [3][4][29][20][36][37][38], and its specifications and implementation are available at a public Web site [39].

4. Empirical studies of software similarities

We summarize results from three lab projects and two projects by our industry partner ST Electronics. Studies covered third party class libraries in Java and C++, and application programs written by our industry partner. Applications included information systems (Java), command and control system (C#), and Web Portals (ASP and J2EE). All the programs were new, not affected by maintenance changes. While the experiment method was similar in all the projects, some details varied from project to project, depending on the type of a program (a class library or an application), and the goal of the similarity analysis and generic design (either reuse in the scope of one program or across programs).

In programs under our study, similarity rate was 50%–68%, with the exception of some of the ASP Web Portal modules where similarity rate reached 90% [29]. (We believe the higher rate of similarity in the Web Portal modules was due to the lack of inheritance in ASP.) We considered as similar only repeated code that, in our opinion, had reuse potential or was counter-productive for maintenance.

In all the studies, we applied XVCL [39] on top of a conventional program to form a “mixed strategy” representation of the subject program. We addressed both simple clones (found by manual inspection or with help of an automated clone detector [21]) and structural clones (identified by top-down domain analysis fed with inputs from bottom-up simple clone detection). We measured the percentage of redundancies by comparing the subject program against the XVCL-enabled “mixed strategy” solution from which the subject program could be obtained in the original form. Full details of XVCL solutions can be found in other papers [4][20][29][37]. We interpret all the results in Section 5.

4.1 Class libraries

We studied the Buffer class library from java.nio.* packages JDK 1.4.1 and C++ STL [27]. In evaluation of design solutions, we concentrated on conventional OO techniques such as generics (or templates in C++), inheritance, abstract classes and dynamic binding.

4.1.1 The Buffer library

A buffer contains data in a linear sequence for reading and writing. Buffer classes differ along features dimensions such as buffer element type, memory allocation scheme, byte ordering and access mode. Features of buffer classes are shown in Figure 2, as a feature diagram [22]. We see five features dimensions, with specific variant features listed below a feature dimension box. Each legal combination of features from various dimensions yields a unique buffer class. As we combine features, buffer classes not only grow in number, as observed in [5], but also become polluted with numerous redundancies, as observed in our experiment.

4.1.1.1 The experiment

We analyzed classes manually to identify similar classes and methods. We applied domain analysis [30], which is a commonly used method to identify reuse opportunities across a family of similar programs. We gained a general understanding of buffer classes first. Then, we identified seven groups of classes that we thought would be similar, where each group comprised 7–13 classes. Examination of class details confirmed that indeed there were many similarities among classes in each group. Similar class methods differed in method signatures, data types, keywords, operators, and editing changes. Classes in each of the seven groups of similar classes differed in method details as above. 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.

Having analyzed similarities and reasons why they occurred, we designed generic “mixed strategy” Java/XVCL solution to unify similar classes and methods. We could generate all the classes in each of the seven groups of similar classes from one meta-component playing the role of a template, and a small number of supporting meta-components. XVCL Processor synthesized Buffer classes in their original form from our meta-representation. Familiar domain, small size of the library (~8KLOC ) and clarity of its design, allowed us to complete the experiment in 10 days (though the interpretation of the results took much longer). In the related experiment, we applied JSR-14 [10] on top of the original buffer classes to see which groups of similar classes could be unified by Java generics.

4.1.1.2 Summary of the results

The experiment covered 74 buffer classes that contained 7,663 LOC (physical lines of code, without blanks or comments). Among 74 classes in the Buffer library, only 15 classes could be unified using Java generics JSR-14 [10]. A group of classes could be designed as a generic Java class if all the classes in the group differ at most in type names. Unfortunately, many similar buffer classes differed in constants, keywords or algorithmic elements rather than data types. In yet other cases, application of generics was hindered by couplings among classes across the library. C++ templates are free of some of the above mentioned limitations (e.g., generics parameters may be primitive types, keywords or constants). We discuss the STL experiment in Section 4.1.2.

Usability, conceptual clarity and high performance were important design goals for the Buffer library. To simplify the use of the Buffer library, the designers revealed to programmers only eight classes at the top of the inheritance class hierarchy which constrained the design of the classes below. For conceptual clarity, designers of the Buffer library preserved almost one-to-one mapping between legal feature combinations and buffer classes. While in many situations redundancies could be avoided by introducing a new abstract class or a suitable design pattern, such a solution would compromise the above design goals. A typical situation in which many similar methods occurred was when some classes derived from the same parent, say class A, needed a certain method (or data), and other classes derived from
A did not need that method. We could create a new abstract parent class just to make that method available to the classes that needed it. Creating many such classes would, however, complicate the class hierarchy. We could also place such a method in the parent class. But this solution would either be error-prone or require us to write extra code to disable the method in the classes that do not need it. In yet another situation, a certain method was needed in all the classes derived from class A; but in some of those classes, the method required different parameters, return type or implementation. Furthermore, the implementation of such a method in different classes might refer to non-local attributes defined in the context of different classes. In the above cases, designers often chose to place a method into each class that needed it, thus creating redundant code.

In a “mixed strategy” Java/XVCL solution, we unified each of seven groups of similar classes with unique generic, but adaptable, meta-level structure, along with information necessary to obtain its instances – specific classes or class methods. This unification reduced program complexity as perceived by developers, also reducing the original code size by 68% percent. 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 discuss later in the section.

Non-redundancy achieved in that way also reduced the risk of update anomalies which helps in maintenance. To get a clearer picture of the impact of genericity on maintenance, we extended buffer classes with a new buffer element type, namely complex number (Complex). We addressed three among many new classes that had to be implemented to address the Complex element type. The number of modifications to implement Complex buffer in a generic meta-representation was 11, as compared to 91 modifications required for the same purpose in the original Buffer library. Further details of this study are described in [20].

It is interesting to note that Sun developers used macros and scripts to exploit similarities across buffer classes. We believe this choice of the technique, rather than OO, has to do with the problems we discussed.

4.1.2 The STL

STL implements generic containers (such as stack, set or map) and algorithms (such as sort or search). Generic solutions are mainly facilitated by templates and iterators.

![Associative container features (STL)](image_url)

We analyzed 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 templates, one for each of the eight legal combinations of features.

4.1.2.1 The experiment

We applied an automated clone detector CCFinder [21]. CCFinder detects simple clones with parametric differences (such as different type or identifier names). We further analyzed manually parts of STL affected by clones. We analyzed the reasons why similar functions and templates occurred. Then, we built generic XVCL structures to unify differences among similar templates. Our meta-level solution used original template solutions, plus XVCL to unify similar templates displaying non-parametric differences that were not unified by the template mechanism.

4.1.2.2 Summary of the results

In STL, iterators proved very effective in separating algorithmic details from the data structures on which they operated. We did not find significant clones in algorithms. However, containers displayed a remarkable amount of code repetition. Four ‘sorted’ associative containers and four ‘hashed’ associative containers could be unified into two generic XVCL containers, achieving 57% reduction in the related code. Stack and queue contained 37% of cloned code. Algorithms set union, intersection, difference, and symmetric difference (along with their overloaded versions) formed a set of eight clones that could be unified by a generic XVCL set operation, eliminating 52% of code.

There were many non-type-parametric differences among associative container group of 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 metaprogramming, often the resulting code becomes “cluttered and messy” [14]. We did not spot such solutions in STL, and believe their practical value needs to be further investigated.

The “feature combinators” problem [5] is a challenge for many class libraries: As we combine features, classes grow in number and become polluted by redundancies. At times, even the sheer size of a library may become an issue. This would be certainly the case of the STL, and templates helped designers control the situation. Still, as our experiment showed, there was room for improvement. Further details of this study are described in [4].

4.2 C# domain entity management subsystem

This study was the first pilot project with XVCL by our industry partner ST Electronics. The experiment involved a Domain Entity Management Subsystem (DEMS) written in C#. Examples of domain entities were Task, User or Resource. The DEMS provided operations for entity management such as Create, Delete, Update, Find or Save. The DEMS was to serve a range of command-and-control applications, so it was meant to be easily adaptable and extensible. The intention was to make DEMS an integral part of the product line architecture for those applications.

4.2.1 The experiment

The project started by porting an existing DEMS written in Visual Basic into C#. At the same time, new variant features were considered to enhance reusability of the DEMS in a wider range of applications. In the process of re-writing the system, ST Electronics analyzed recurring patterns of program structures to identify potential reuse opportunities. Whenever a suitable solution existed using C# and .NET mechanisms – it was applied. The option of applying generics was taken into account based on the generics proposed for C# [23]. ST Electronics recorded patterns of repetition that could not be unified using available mechanisms, and that were leading either to repetitions or to inefficiencies during customization of the DEMS for reuse in target applications.
In the next step, ST Electronics applied XVCL on top of the C# DEMS to create generic solutions for repetitive situations that could not be unified in C#, but whose reuse was deemed to be beneficial. To facilitate comparison, the same functionality was implemented into both versions of the DEMS.

4.2.2 Summary of the results

The DEMS involved 13 domain entities (such as User or Task) with up to 10 operations per entity (such as Create or Delete). DEMS was implemented with 18,823 LOC of C# code, contained in 117 classes covering GUI, service and database layers. There were many similarities in how the same operation was to be designed and implemented across entities. However, there were also differences caused by different meanings of domain entities: For example, operation Create for a Task required different types of data entry and data validation than Create for a User. Figure 4 depicts sample similarity patterns (structural clones) that occurred in the C# solution induced by the above mentioned situation. The design and implementation of operations for various entities were characterized by a similar pattern of collaborating classes across GUI, service and database layers. Each box in Figure 4 represents a number of classes, with much similarity across classes implementing similar concepts in the same types of operations for different entities. For all the domain entities and their respective operations.

To implement enhancements, the C#/XVCL representation of DEMS required less modifications with smaller impact than its C# counterpart. For example, to add a new domain entity to DEMS with XVCL required writing 133 LOC and took 2 man-hours, while the same enhancement without XVCL required writing 1440 LOC and took 16 man-hours.

The C#/XVCL DEMS could also play a more effective role in the command-and-control product line architecture than the C# DEMS solution.

4.3 Web Portals (WP) on J2EE platform

The main goal of this study was to evaluate J2EE/EJB [33] as a possible platform for Web Portal (WP) product line development. An earlier project by ST Electronics revealed similarity rates in Active Server Pages (ASP) WPs were around 60%, and in certain areas as high as 90% [29]. By unifying similarity patterns with XVCL applied on top of the ASP, ST Electronics designed an ASP/XVCL product line architecture, from which many similar but distinct WPs could be derived at much a lower cost than it was possible with conventional methods. Maintenance productivity figures for the AXP/XVCL solution, as well as for WPs derived from it, were very encouraging, too (see Section 6). The results indicated that, in this project, despite a careful model-based design, ASP and associated techniques were not effective in unifying many types of WP similarity patterns (we refer to detailed account of problems addressed and benefits observed in this project to [29]).

The results of the above project motivated us to pursue research in similar direction on the J2EE platform. The Java™ 2 Platform, Enterprise Edition (J2EE) provides standardized architecture and supports reuse of common services via portlet technology. Unlike ASP, J2EE supports inheritance, generics and other OO features via Java 1.5. We worked with a portal developed by our industry partner ST Electronics, called CAP-WP. CAP-WP supported collaborative work and included 14 modules such as Staff, Project or Task (Figure 5).

4.3.1 Presentation and business logic layer

Each WP-CAP module (e.g., Staff or Task) implements similar operations (Add, View, Delete, Update in Figure 6). These operations are implemented as functional packages (Action, RequestHandler, ViewHandler), which in turn are implemented as portlet classes. We studied both intra- and intra-module
similarities in CAP-WP design. The nature and degree of intra- and inter-module similarities varied. 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.

Figure 6. Similarities in CAP-WP

In the second part of the experiment, we applied XVCL to unify similarity patterns, taking into account both intra- and inter-module similarities in CAP-WP. Our solution represented each similarity pattern with a unique generic meta-level structure. Not only did such unification reduce the solution size by 61%, but most importantly it increased the clarity of portal’s conceptual structure as perceived by developers. In particular, it reduced the number of conceptual elements a programmer had to deal with and enhanced the visibility of relationships among program elements that mattered during changes: Rather than maintaining multiple variant code structures delocalized across the CAP-WP, in the “mixed strategy” J2EE/XVCL solution a programmer dealt with one generic structure, with full visibility of customizations required to produce instances of variant structures, as well as their exact locations. Non-redundancy of the J2EE/XVCL CAP-WP, achieved by generic meta-level structures unifying similarity patterns, reduced the risk of update anomalies.

4.3.2 The Entity Bean data access layer

We addressed variant features affecting Entity Beans depicted in Figure 7. Domain Entity feature group represents database tables for different portal modules. Entity Beans receive query request from the presentation or business logic layers, connect to the specified database (DBMS feature group in Figure 7), by the specified connectivity mechanism (Connection), receive query results and pass them back to the caller in the required format (Input/Output Parameter).

As any combination of variant features (Figure 7) could be implemented in Entity Beans in some Web Portal, the number of required Entity Beans was huge. While Entity Beans were similar across modules, there were differences among them, too. J2EE/EJB provided little support to abstract similarities among Entity Beans and to deal with the impact of variations in a generic way. Sometimes, we could implement “typical variant combinations” into components, allowing a developer to select variants needed in a WP using EJB mechanisms. However, using this method we covered only a small number of simple cases. An alternative solution was to have a separate component version for each legal combination of features. But then we would end up with huge number of look-alike components. Both solutions were prohibitive to systematic reuse. Consequently, our CAP-EJB was basically formed by components (and component configurations) that already appeared in some modules. These components could be reused in building new modules. As such, they required much manual work during customization. These experiences are in line with problems reported in industrial product lines [15], where many variant features and feature dependencies lead to thousands of component versions, hindering reuse.

Even though we did not create many look-alike components, the CAP-EJB still displayed much similarity that could not be unified by generic solutions. For example, many Entity Beans had similar interfaces and business logic related to handling tables. However, they differed in attributes and some other details. Unfortunately, we could not abstract commonalities among Entity Beans and parameterize them for reuse.

In the second part of the project, we studied similarities in CAP-EJB and applied XVCL to unify them with generic Entity Bean components. We call this solution CAP-EJB/XVCL. We unified multiple similar components for different variant combinations with a small number of generic meta-components. To facilitate fair comparison, we ensured that CAP-EJB/XVCL covered the same functionality and the same variant features as CAP-EJB did. The part of the CAP-EJB under study contained 5,811 LOC, and CAP-EJB/XVCL contained 2,354 LOC.

It is not the size of a generic solution but its conceptual clarity and reuse capability that matter. Unfortunately, the CAP-EJB solution was not without problems, most of which could be linked to the lack of generic solutions in domain-specific areas. To build a specific module, a developer typically starts by defining requirements for it, which includes selecting variant features needed in the module. Then, a developer selects components (or component configurations) from the CAP-EJB that possibly best match components needed in the new CAD application (initial phase in [15]). These components are further customized in a number of iterations until the application meets its requirements (iteration phase and adaptation in [15]). Both selection of the best matching component configuration and component customization was difficult and error-prone in our project. Many variant features had non-local impact on the CAP-EJB, affecting many of its components. We did not find a suitable J2EE mechanism for documenting (not to mention automating) chains of detailed modifications of components in order to address such variants. Consequently, similar modifications had to be applied repeatedly and manually whenever a developer needed a specific variant in
his/her module. Repetitions arising from the lack of suitable
generic solutions further magnified those problems.

To better understand the above problems, we counted the number
of modifications of CAP-EJB components a developer had to do
to address various data access variants. For example, addressing
the input parameter variant required 119 modifications, and the
Entity Bean type – 122. These problems were rectified in the
CAP-EJB/XVCL solution, where the same variants required 18
and 19 modifications, respectively.

5. Interpretation of similarity analysis results

5.1 “Feature combinatorics” problem

As observed in our empirical studies, similar program structures
ranged from small granularity simple clones such as similar class
methods or method implementation fragments, to similar classes
or components, to patterns of collaborating classes, and to large
granularity architectural patterns involving groups of collaborating
components (e.g., Figure 4 or analogical inter-module similarities
in the ASP or J2EE Web Portals, Section 4.3). Instances of those
similar program structures differed one from another in rater
arbitrary, irregular ways.

In many cases, we found that repetitions of similar structures in
variants forms were symptoms of a “feature combinatorics”
phenomenon, first observed in class libraries [5][6]. Authors
showed that as we combine features, the number of look-alike
classes must grow in geometric progression. Many similar
program structures found in our experiments were symptoms of the
same problem. In each of our studies, features meant different
program characteristics. For example, in the Buffer library
features meant data type, memory allocation scheme, access mode
and byte ordering; in the STL – data types, ordering, key type and
uniqueness of elements in a container; in command and control
system and Web Portals – domain entities (e.g., Staff, Project,
Task) and actions that applied to those entities (Create, View,
Delete). Program structures (such as components, classes or
functions/methods) implement the net effect of the required
feature combination that is necessary for correct execution. As we
combine features, variations arise that force us either to find a
unifying generic solution, or to create multiple program structures
in variant forms.

“Feature combinatorics” phenomenon is closely tied to the
semantics of an application domain or a programming domain,
and may show in many different, sometimes hidden ways.
Features may represent functional or non-functional
requirements, design goals, design decisions, crosscutting aspects
[24], or concerns [34]. Not always must feature combinations
necessarily trigger variant forms. At times, a feature can be
encapsulated and separated from other features with techniques
such as macros, modularization, generics, inheritance, design
patterns [18], Aspect-Oriented Programming [24], or AHEAD [6].
Feature combination becomes a problem when conventional
methods fail to encapsulate and localize the features’ impact on
program. In industrial software product line projects [9][12],
feature combinations may lead thousands of similar component
versions [15].

5.2 When conventional methods fail to
address the problem

It was relatively easy for us to identify patterns of similarity in
programs under study. Still, it was not clear to us how to design
double enough generic solutions to unify recurring patterns, using
conventional techniques. What made unification difficult was ad
hoc, irregular nature of variations across similar program
structures arising from “feature combinatorics” problem. There
were many non-type-parametric differences among similar
program structures that could not be unified by generics (or
templates). For example, certain similar classes differed in
keywords, identifier names (or parts of them), operators,
algorithmic details or extra methods. Sometimes, type variation
triggered yet other non-type-parametric differences in the same
classes, as certain code depended on the size associated with a
give type.

Typically, programs are developed with multiple, sometimes
competing, design goals in mind such as development cost,
usability, simplicity, performance, or reliability. Unifying
similarity patterns may not be among most important goals. If
generic solutions to achieve non-redundancy conflict with other
important goals, developers rather choose to multiply similar
program structures. These realities must be taken into account
when evaluating solutions. Most often, some solution could be
invented if we considered each case of recurring similar program
structures in isolation from the whole program. For example, in
programs under study, in some cases we could create an abstract
class or apply a design pattern to avoid repetition. But there were
yet other similarities that also needed treatment, and yet other
design goals to be met. Therefore, the solution could only win if it
was natural and synergistic with all the other factors that mattered.

We believe the above difficulties are quite general. Many
similarities remain unresolved as there is no simple way of
unifying them with conventional generic design solutions, without
compromising other important design goals. “Feature
combinatorics” triggers the process of proliferating similar
program structures, and the need to meet multiple design goals
magnifies its impact.

Others who studied software cloning in a systematic way drew
even stronger conclusions. For example, Kim et al [25] report that
49% to 64% of clones “are not easily refactorable due to
programming language limitations”.

6. A “mixed strategy” solution: trade-offs

We discussed difficulties to achieve genericity with conventional
methods. While generic design may be easier at the meta-level
(Section 3), flexibility that we gain at the meta-level does not
come for free. As we relax the coupling between the
parameterization mechanism and the rules (syntax and semantics)
of the underlying programming language, the power of the
parameterization mechanism increases. For example, with C++
templates we can unify a wider class of variations than with Java
generics. At the end of this spectrum, there are meta-programming
(or generative) techniques that manipulate a program as a text,
with no regard to language rules. By separating genericity issues
from the core constructs typically supported by programming
languages, we can address genericity concerns without
compromising program runtime properties. But as we move
towards less restrictive parameterization mechanisms, we also
decrease type-safety of parameterized program solutions.
Therefore, there are important trade-offs to consider.

The high expressive power is the main strength of the meta-level
parameterization such as XVCL. XVCL can deal with arbitrary
types of variations across similar program structures. The current
form of XVCL is an assembly language for generic design via
parameterization. XVCL’s explicit and direct articulation is the
source of its expressive power, but it also adds a certain amount of complexity to the problem. It is easier to understand a concrete program than a meta-program. It is also difficult to validate a meta-program as we can derive many concrete programs from it.

However, we believe that the benefits of being able to deal with issues of genericity at the meta-level plane outweigh the cost of the added complexity. These benefits include ease of reuse with adaptation, the overall reduction of conceptual complexity and size of the solution, improved traceability and changeability. We are currently collecting empirical and analytical evidence to support the above hypothesis. Our lab studies and two projects by our industry partner ST Electronics confirm our expectations. In the ASP Web Portal (WP) Product Line project (mentioned in 4.3 and fully described in [29]), state-of-the-art design methods were used to maximize reusability of a Team Collaboration Portal (TCP) in other contexts. Still, a number of problem areas were observed that could be improved by applying XVCL to increase the genericity of a conventional solution. The benefits of a generic “mixed strategy” TCP-ASP/XVCL solution were the following:

- Short time (less than 2 weeks) and small effort (2 persons) to transform the TCP into the first version of a generic TCP-ASP/XVCL.
- High productivity in building new portals from the “generic TCP”. Based on the TCP-ASP/XVCL, 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 were 22% less than the original single portal.
- Wide range of portals differing in a large number of inter-dependent features supported by the TCP-ASP/XVCL.

At the same time, these industrial applications of XVCL also revealed a number of further problems: With XVCL, developers must know and manage multiple languages (e.g., XVCL, Java, ASP, HTML) and this kind of complexity may have an impact on productivity and maintainability.

Debugging is yet another area into which XVCL induces extra complexities. In the WP project, the runtime debugging was performed on the ASP code generated by the XVCL Processor. As a result, developers had to maintain a ‘mental picture’ of mappings from x-frames to the runtime ASP code. This extra step made debugging more complex.

There is a great opportunity here for XVCL-specific tools to help developers analyze “mixed strategy” solutions. We are working on the XVCL Workbench that helps in editing, visualizing, debugging and static analysis of XVCL code.

Engineering processes play an important role in industrial software development. Currently, we know how XVCL-enabled “mixed strategy” solutions can raise productivity of small teams of highly-skilled expert software developers. We must yet to learn what it takes to inject “mixed strategy” methods into more complex team structures and industrial development processes. Working on those issues is an important direction for our future work, but we realize difficulties involved.

7. Conclusions

We presented empirical and analytical arguments to support a hypothesis that weak support for generic design in today’s programming paradigms induces considerable conceptual complexity into programs and hinders reuse. We analyzed similarity patterns in a number of program situations, and showed why they were difficult to unify with conventional techniques. We traced the roots of some common similarity patterns to the “feature combinatorics” problem. Finally, we proposed a “mixed strategy” approach in which meta-level parameterization and generation technique is applied on top of a conventional program, to strengthen the generic design capabilities. We evaluated the engineering benefits and assessed trade-offs involved in employing meta-level techniques, based on industrial feedback and our own studies. The proposed approach has been successfully applied in industrial projects.

In the future, we plan to extend our empirical studies to yet other application domains, conducting comparative studies involving a range of techniques for generic design. We hope to discover meta-level abstractions that will allow us to define higher-level forms of XVCL, equally expressive but free of current pitfalls. We plan to further work on tools for meta-level techniques and on methodological aspects of “mixed strategy” solutions.

Generic design has to do with both reusability and maintainability, the two economically desirable – but also difficult to achieve - software engineering goals. We believe the “weak generics” dilemma discussed in this paper points to the heart of the many software development problems, and tells us something fundamental about the nature of software. Therefore, it calls for a systematic treatment and effective solutions.

8. Acknowledgments

Thanks are due to students Li Shubiao, Hamid A. Basit, Damith C. Rajapakse and Pavel Kordunov (library experiments) and Deng Ping (J2EE product line study). Ulf Pettersson of ST Electronics shared with us experiences from pilot projects. Collaborations with Paul Bassett and Ulf Pettersson contributed a lot to the results and interpretations described in this paper.

References

Add Generic Types to the Java Programming Language

Adopting and evolving a product-line approach, Addison-Wesley, 2000

Bosch, J. Design and Use of Software Architectures – Adapting and evolving a product-line approach, Addison-Wesley, 2000


Bosch, J. Design and Use of Software Architectures – Adapting and evolving a product-line approach, Addison-Wesley, 2000

Brooks, P.B The Mythical Man-Month, Addison Wesley, 1995


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

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

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


Yang, J. and Jarzabek, S. “Applying a Generative Technique for Enhanced Reuse on J2EE Platform,” accepted for 4th Int. Conf. on Generative Programming and Component Engineering, GPCE’05, Sep 29 - Oct 1, 2005, Tallinn
