

## Chapter 14 Other techniques related to XVCL

---

### *Summary of this Chapter:*

In this Chapter, we discuss other techniques, both conventional and unconventional, comparing them to XVCL. We primarily focus on engineering goals that are typically achieved by various techniques, but also comment on the actual mechanisms underlying discussed techniques. Generic design, componentization, reuse, separation of concerns and enhancing the visibility of changes are the among the main leading themes of reuse-based evolution, and also among main forces that motivated XVCL. Therefore, we discuss other techniques around those themes, relating them to XVCL. Conventional techniques include type parameterization [17], other forms of parameterization such as higher order functions [39], modular decomposition with information hiding [29], inheritance with dynamic binding, design patterns [16] and mechanisms supported by modern component platforms (such as J2EE™ or .NET™). Unconventional techniques include generative approaches [12].

### **14.1 Frame Technology**

XVCL has its roots in Frame Technology™, by Netron, Inc [3]. A number of frame-based systems have been implemented in both industrial and academic institutions [15]. We believe any system based on frame principles can achieve similar engineering benefits in areas of maintenance, evolution and reuse as demonstrated with XVCL in this book, independently of a specific syntactic representation that different systems may use. Whether or not similar engineering benefits can be achieved by other technical approaches - remains an open question. We are not aware of reports demonstrating capabilities of other techniques to tackle evolution challenges discussed in Chapter 1, in particular in the area of unifying software similarity patterns.

Frame Technology™ has been extensively applied to maintain multi-million-line COBOL-based information systems and to build reuse frameworks in companies [3]. An independent assessment by QSM Associates, Inc. showed that frames could achieve up to 90% reuse, reduce project costs by over 84% and their time-to-market by 70%, when compared to industry norms [3]. We are in the lucky situation that the basic principles of XVCL have been already tested in practice, tough in a different setting than ours. Our contribution is that we refined frame concepts into a general-purpose technique of XVCL [41]. We also demonstrated that XVCL can enhance modern programming paradigms in areas of maintainability and reusability, and formulated a mixed-strategy approach to guide developers in exploiting frame benefits in their projects.

### **14.2 Conventional techniques**

Search for effective generic programming and design techniques to unify software similarities has a long history. Most often, we try to achieve genericity through parameterization, whose general forms were proposed by Goguen in 1983 [18]. Programming languages such as Ada [1], Eiffel [27], C++ , Java JDK 1.5 and C# support some form of generics. Generics use type parameters to unify groups of similar classes. In C++, type parameterization is called

templates and can be also applied to unify similar unattached functions. Garcia et al. [17] compare generics in six programming languages. Generics show limitations when we need unify similar classes or functions that differ in non-type parameters. We discussed difficulties in applying Java Generics to unify differences among buffer classes in Chapter 10. C++ templates are more powerful than other generics due to light integration of templates with the C++ language core. Unlike Java generics, C++ templates also allow constants and primitive types to be passed as parameters. STL [28][36], a hallmark of powerful generic programming, achieves much genericity via type parameterization. STL implements type-safe polymorphic containers. Still, repetitions exist in areas of associative containers [2] which, despite many similarities, cannot be easily unified with templates.

Static C++ template meta-programming technique can deal with complicated forms of parameterization, achieving remarkable results in building adaptable software [12]. As such unconventional usage of templates may lead to overly complex software representations, the engineering qualities of the solution should be consciously evaluated when the technique is applied.

Some limitations of type parameterization can be alleviated with function parameters (e.g., higher-order functions [39] or pointers to functions in C++).

Modularization with information hiding [29] is also a simple form of generic design. Here, a similarity pattern is reflected by module's API. A hidden implementation plays a role of a parameter that makes a module generic. Instantiation of such a "generic module" is done by defining a specific data representation, and implementing API operations in terms of the chose data representation.

XVCL provides parameterization that is not restricted by the rules of programming languages used together with XVCL to form mixed-strategy solutions. The type and granularity of similar program structures unified by generic representations built with XVCL is unrestricted, too. They may range from small granularity similar code fragments (such as class methods), to classes, to components, to patterns of collaborating components, and to large granularity architecture-level code formations (e.g., subsystems). Domain analysis [31] is essential in identifying such high-level, large granularity patterns of similarity. Mixed-strategy generic solutions unifying such similarity patterns are most beneficial for programmer's productivity as they can significantly reduce the size and complexity of the solution.

Macro systems are one of the oldest attempts to generalize programs. Macros are a simple form of a generative technique [12], not strictly a conventional technique in the sense we use term 'conventional' in this book. We comment on macros in this section because of their common use in main-stream programming. Macro systems work on the principle of code expansion according to a certain set of rules. A macro-processor replaces macros found in a program with text that defines a given macro. This text may contain code and also invocations of other macros, allowing one macro to invoke other macros. Macros may have parameters that are instantiated in the process of macro expansion. Macros handle variant features only at the implementation level, which causes well-known problems when trying to tackle more complex change situations with macros [22]. We believe in the context of software evolution challenges discussed in Chapter 1, macros can play an important supporting role, but it may be difficult to attack the core of the problems with macro-based solutions.

XVCL also works on the principle of code expansion according to a certain set of rules. Some of the XVCL commands (such as `<set>`, `<adapt>`, `<ifdef>` or `<select>`) have close counterparts in macro systems: `<adapt>` corresponds to macro invocation, `<set>` corresponds to setting macro variables, and `<ifdef>` and `<select>` play similar role to macros defining conditional compilation. Other XVCL features do not have a macro counterpart or are less attentively defined in macro systems. They include XVCL's ability to define customizations for generic x-frames at their adaptation points, or to propagate customizations across an x-framework whereby globally defined customizations override customization defined locally. XVCL allows us to place logical groupings of change specification commands in dedicated, easy to identify x-frames, and propagate them from there – in variant forms - to any program component that needs be changed, without affecting components that do not need that change. We refer the reader to [3] pp. 116-120 which explains the details of technical differences between frame principles and macros. It is the right collection of mechanisms and rules (described in Chapter 8), rather than the fundamental principle of operation, that gives XVCL its unique power to solve engineering problems posed by software evolution and reuse.

Software Configuration Management (SCM) systems have been applied to handle variant features in software [10][38]. For each legal combination of features, an SCM tool maintains a separate component version – the situation analogical to multiplying similar classes in the Buffer library. SCM tools are strong in handling variants at the file level, but weak in handling small, inter-dependent variants, spreading over many components. File-level variations may not neatly map into design concepts. In XVCL, we capture component variability specifications separately from the components themselves, producing custom components with a required combination of variant features on demand. We discussed CVS in context of evolution in Chapter 5. We then contrasted CVS approach with XVCL in Chapter 11.

Software architectures [33], architectural styles and patterns [9] help developers avoid repeatedly designing the same solution by providing component plug-in plug-out capability. Reuse via software product line approach is often supported by stabilizing software component architecture [9]. Component-based reuse is most effective when combined with architecture-centric, pattern-driven development which is now supported by the major platforms such as .NET™ and J2EE™. Component platforms provide an infrastructure for reuse of pre-defined common services. Patterns are basic means to achieve reuse of common service components. Standardized architectures together with patterns lead to beneficial uniformity of program solutions. Interactive Development Environments support application of major patterns, or programmers use manual copy-paste-modify to apply yet other patterns.

XVCL can enhance the benefits of modern platforms by automating pattern application, emphasizing the visibility of patterns in code, and helping to avoid explosion of similar components and component patterns by unifying them with generic representations. Pattern-driven design facilitates reuse of middleware service components, but tends to scatter application domain-specific code. With XVCL, we can package and isolate otherwise scattered domain-specific code into reusable generic components. Such extensions improve development and maintenance productivity, and allow reuse to penetrate application business logic and user interface system areas, not only middleware service component layers.

It is interesting to notice that component platforms hide implementation of some of the functionality that would normally be scattered across program components, so-called

crosscutting concerns, discussed in more detail in the next section. Component platforms provide transparent access to them via APIs. In J2EE™, containers provide general mechanism to access via APIs services whose implementation would otherwise lead to crosscutting concerns. Examples of such services include transaction management, persistence, security, authentication/authorization and session management, depending on a container used [26]. Without J2EE infrastructure or support of unconventional technique such as Aspect-oriented Programming (AOP) [23], these concerns have crosscutting effect.

### **14.3 Generation techniques**

Generators [34] produce custom programs from problem specifications in domain-specific languages. Domain-specific languages can be developed in well-understood and stable domains. As problem specifications can be very compact, generators can yield high productivity gains in domains formalized by generation solutions. Most of generators easily accommodate changes that can be expressed in the scope of a domain-specific language, but do not provide explicit support for changes beyond that. A common pitfall of generators in context of maintenance and evolution is that any modifications of the generated code cause disconnection of the code from abstract specifications from which code has been generated. From that point onwards, the generated code has to be maintained by hand.

XVCL is an application domain- and programming language-independent technique. It does not rely on any form of abstract specifications that would not be transparently linked to code. Every detail of an x-framework can be exposed to a developer, who also has full control over every detail of transformations performed by the XVCL Processor. XVCL Processor does not inject any hidden code into a custom program produced from the x-framework. Specifications of custom changes are visible, and their impact on a custom program produced, as well as on the processing can be traced to every detail. Custom code produced from an x-framework never gets disconnected from the x-framework.

XVCL shows most benefits in domains where frequent, unexpected changes occur at both large and small granularity levels. In such domains, it is usually difficult to formalize enough abstractions to build generators. In case generation solutions exist for certain sub-domains, subsystems serviced by generators can be integrated with other subsystems controlled by XVCL. XVCL can be also used to implement generators. A simple example of that was demonstrated in Chapter 10, where we produced many class variants from a small number of x-frames [19][20]. In other projects described in Chapter 13, we produced modules of Web Portals from module templates [30][42], and patterns of collaborating components implementing various combinations of domain entities, such as User, Task or Resource, and operations applied to those entities, such as Create or Update, from small number of pattern templates.

Generators for domain-specific languages can be made to emit x-frameworks, permitting the automatic re-customization of regenerated code. In this way, the system's high-level abstractions remain connected to the executable code throughout its maintenance and evolution. Such solutions have been developed at Netron for Frame Technology™.

### **14.4 Separation of concerns and its relation to generic design**

An important class of generative techniques [12] focus on separation of concerns, a principle introduced by Dijkstra's [14] to the software domain in early 1980's. These techniques

attempt to bring separation of concerns from the concept level down to the design and implementation levels. Aspect-Oriented Programming (AOP) [23][24], MDSOC from IBM [37] and AHEAD [5] are among most widely published among such techniques. Separation of concerns helps in maintainability, long-term evolution, and is also supportive to building more generic, reusable software. Though we did not come across applications of techniques based on separation of concern to solve evolution problems such as we studied in this book, both the principle and techniques that help in its realization are most relevant to the theme of this book.

A *concern* can be any area of interest in a program solution, pertinent to functional features, quality requirements, software architecture, detail design or implementation. The idea of separation of concerns is to break a program into distinct concerns in order to deal with them separately. As we do so, we try to limit interactions between concerns as much as it is possible. The motivation for separation of concerns is to better cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability.

One modular program decomposition cannot give equal importance to localization of all the computational concerns. Therefore, any modular decomposition of a given program can be nicely aligned with only some of the concerns. Such concerns can be localized to a single module (a component, class or function) or a group of modules (e.g., component layer), exposing an abstract program interface (API) to its clients. The details of a concern implementation become hidden behind the API [29].

However, other concerns, that are not aligned with a given modular decomposition, necessarily crosscuts modules of primary decomposition (in AOP) and other concerns. Separation of crosscutting concerns, if possible at all, requires unconventional solutions. Generative techniques attempt to separate concerns at a *meta-level extra plane*. Techniques differ in the nature of that extra plane and in mechanisms used to transform a meta-level representation into concrete components.

As we discussed in Chapter 6, problem domain concerns become intimately and unavoidably interwoven with problem solution concerns in program components. We also commented on limitations of software componentization induced by that fact. “Configuration knowledge” technique [12] uses unconventional means to achieve separation of these two important concerns, by explicating mappings between the problem domain and the solution spaces.

Other techniques focus on any kind of concerns, trying to combat their crosscutting effect, to allow for their independent treatment (e.g., maintenance and reuse), and to facilitate composition of concerns.

In AOP, various computational aspects are programmed separately and weaved into the base of conventional program modules of primary decomposition (e.g., classes). Aspect code is weaved into program modules at join points that are specified in a descriptive way. AOP can simply and elegantly separate a range of programming aspects such as synchronization, persistence, security transaction management, or authentication/authorization. Due to such separation, aspects can be easily modified and also added or deleted to/from program modules, which automatically become more generic and reusable in different contexts. AOP can be extended to allow for parametric differences among aspect code weaved at different join points [25].

Feature-Oriented Programming AHEAD technique [5], based on GenVoca [3], attempts to elevate separation of concerns to the level of user-level requirements. Features represent software requirements, but can also refer to any other computational concerns. Features are defined separately, as mathematical functions. Programs are produced by refining features and composing them in different combinations. Legal combinations of features are described by a GenVoca grammar. AHEAD models software as a mathematical structure of nested equations, making it possible to study formal properties of refinements and resulting programs.

The trust of the MDSOC approach [37] is separation of concerns to overcome a “tyranny of a dominant decomposition” of programs into functional modules. Among other goals, MDSOC attempts to achieve better alignment of requirements, design and code [8]. Hyperslices are meta-level abstractions that encapsulate specific concerns and can be composed in various configurations to form custom programs. Hyperslices are written in the underlying programming language and can be composed by merging or overriding program units by name and in many other ways. Compositions yield programs with modified or extended behavior. Unlike in AOP, it is typical for hyperslices to represent functional units.

Situations where concerns occur in programs in many different combinations lead to explosion of look-alike program components. Termed as “feature combinatorics” problem, the phenomenon was first observed in class libraries [4][6]. Both class libraries and application programs are affected by the symptoms of “feature combinatorics” problem (as illustrated in Chapters 9 and 13). Thousands of component versions may be produced in industrial projects as a result of growing number of features and their mutual dependencies [13]. Program representations that successfully achieve separation of some of the concerns also become more generic, helping to combat explosion of look-alike component versions.

Like AOP, MDSOC or AHEAD, XVCL offers a mechanism to define alternative program decompositions using unconventional means, at the meta-level extra plane. This extra plane is a mixed-strategy representation formed by XVCL applied together with programming languages. While groups of inter-related x-frames often correspond to concerns, analysis of similarity patterns and design of generic mixed-strategy representations unifying similarity patterns plays a driving role in the process of developing a mixed-strategy solution.

XVCL’s mechanisms cater for both generic design and separation of concerns. There is an interesting relation between these two principles that we explain below.

As we already noticed, program representations that achieve separation of concerns also become more generic, parameterized by other concerns. However, we often observe that concerns become so tightly coupled one with another (or with modules of primary decomposition) that their physical separation becomes unthinkable. For example, performance concern in some real-time systems has pervasive impact on many design decisions. While we can express and conceive performance concern conceptually (e.g., by documenting design decisions that have to do with performance), “physical” separation of performance concern from functional modules or yet other concerns that interact with performance may not be feasible. In other systems, where performance strategies are simpler, it may be possible to localize the performance concern in certain modules, or separate it by means of AOP.

We believe many concerns in application domain-specific areas, often referred to as features [5][21], are inseparable just as performance concern is inseparable in time-critical systems. This was the case of features in the Buffer library (Chapters 9 and 10), STL, Web Portals (ASP and J2EE), DEMS (Chapter 13) and other projects. Situations where concerns become difficult to separate are important as they shed light on requirements for techniques to achieve separation of concerns, and possible limits of what we can practically expect from such techniques.

We found that generic design can be a natural extension of separation of concern into the areas where separation of concerns tends to show its limits.

For example, it is difficult to observe the exact impact of features in the Buffer library on class implementation (Chapters 9 and 10). Here, features are buffer element type, memory allocation scheme, byte ordering and access mode. Separation of concerns only succeeds in small number of classes that differ in numeric buffer element types, and then is achieved by type parameterization which is also a generic design solution. If we continue with similarity analysis from the point where separation of concerns becomes difficult, we can achieve further simplifications. In our analysis described in Chapter 9, we came up with the following seven groups, each containing similar classes:

1. **[T]Buffer**: 7 classes at Level 1 that differ in buffer element type, **T**: Byte, Char, Int, Double, Float, Long, Short
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.

We can unify classes in each group with generic mixed-strategy Java/XVCL representation (Chapter 10). The above seven groups of similar classes are clearly organized around concerns: each group is characterized by concerns that vary across classes in a group, and yet other concerns that are fixed. Mixed-strategy representation improves the visibility of concerns, due to groupings of similar classes into groups, but here the separation of concerns is less systematic, only as much as it is practically possible. Therefore, focusing on unifying similarity patterns with generic design representations, we also do selective and imperfect separation of concerns.

Concept of similarity is less formal than a concept of cleanly separated concern. It is easier to find similarities than to spot the exact impact of concerns. Focusing on similarities, is more pragmatic as we do not even have to fully understand the exact nature of a given concern or complex interactions among the concerns. Instead, we stay at the level of observing the symptoms of net effect of concern interactions. We can use a clone detector [2] together with top-down domain analysis to zoom into similarity areas that are significant.

Experiences from other projects confirm the above observations. We believe the principles of separation of concerns and generic design are intimately interrelated and can be applied in a synergistic way.

At the level of actual mechanisms, unlike in other approaches, XVCL's compositions are defined in operational way and take place at designated program points marked with `<adapt>`, `<break>` and other XVCL commands. Concerns encapsulated in x-frames, in areas where separation of concerns with XVCL is feasible, are unconstrained in the sense that they may overlap one with another or form concern hierarchies, as one concern may contain other concerns. XVCL's concerns can be parameterized (e.g., via XVCL expressions or `<select>` commands), which further enhances programmer's ability to define variations in code at any level of granularity that is required, from a single program statement to a component or subsystem. It is the right collection of mechanisms and rules (described in Chapter 8) that gives XVCL its unique power to solve engineering problems posed by software evolution and reuse.

XVCL is a lower level language than AOP, hyperspaces or AHEAD. It is an assembly language of program manipulation and synthesis. XVCL's explicit and direct articulation is the source of its expressive power, but also the source of its weakness, as x-frames may become overly verbose. Currently, we address this problem with tools that reveal simplified, abstract views of x-frames. In the future, we hope to discover mixed-strategy abstractions that will allow us to define higher-level forms of XVCL, equally expressive but free of current pitfalls.

### **14.5 Conclusions of this Chapter**

Each technique has its own merits, and allows developers to meet specific engineering goals, in specific situations. For different types of problems, either AOP, MDSOC, AHEAD or XVCL may yield the simplest, most elegant and useful solution. It is essential to understand strengths and limitations of a technique, trade-offs involved in its application, and possibly ways of using techniques in synergistic combination to best meet engineering goals in hand.

## **References**

- [1] ANSI and AJPO Military Standard: Ada Programming Language, ANSI/MIL-STD-1815A-1983, February 17
- [2] Basit, H.A., Rajapakse, D.C., and Jarzabek, S. "Beyond Templates: a Study of Clones in the STL and Some General Implications," Int. Conf. Software Engineering, ICSE'05, St. Louis, USA, May 2005, pp. 451-459
- [3] Bassett, P. Framing software reuse - lessons from real world, Yourdon Press, Prentice Hall, 1997
- [4] Batory, D., Singhai, V., Sirkin, M. and Thomas, J. "Scalable software libraries," ACM SIGSOFT'93: Symp. on the Foundations of Software Engineering, Los Angeles, California, Dec. 1993, pp. 191-199
- [5] Batory, D., Sarvela, J.N. and Rauschmayer, A. "Scaling Step-Wise Refinement," Proc. Int. Conf. on Software Engineering, ICSE'03, May 2003, Portland, Oregon, pp. 187-197
- [6] Biggerstaff, T. "The library scaling problem and the limits of concrete component reuse," 3rd Int. Conf. on Software Reuse, ICSR'94, 1994, pp. 102-109
- [7] Brooks, P.B The Mythical Man-Month, Addison Wesley, 1995
- [8] Clarke, S., Harrison, W., Ossher, H. and Tarr, P. "Subject-oriented design: toward improved alignment of requirements, design, and code," Proc. 14th Conf. on Object-oriented Programming, Systems, Languages, and Applications OOPSLA'99, Denver, November 1999, pp. 325-339
- [9] Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, 2002

- [10] Conradi, R. and Westfechtel, B. "Version Models for Software Configuration Management," *ACM Computing Surveys*, 30(2), 1998, pp. 232-282
- [11] Cordy, J. R., "Comprehending Reality: Practical Challenges to Software Maintenance Automation," *Proc. 11th IEEE Intl. Workshop on Program Comprehension, (IWPC 2003)*, pp. 196-206
- [12] Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000
- [13] Deelstra, S., Sinnema, M. and Bosch, J. "Experiences in Software Product Families: Problems and Issues during Product Derivation," *Proc. Software Product Lines Conference, SPLC3, Boston, Aug. 2004, LNCS 3154, Springer-Verlag*, pp. 165-182
- [14] Dijkstra, E.W. "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 1982, pp. 60-66
- [15] Emrich, M. *Generative Programming Using Frame Technology*, Diploma Thesis, University of Applied Sciences Kaiserslautern, Department of Computer Science, and Micro-System Engineering, 29. July 2003
- [16] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
- [17] Garcia, R. et al., "A Comparative Study of Language Support for Generic Programming," *Proc. 18th Conf. on Object-oriented Programming, Systems, Languages, and Applications, 2003*, pp. 115-134.
- [18] Goguen, J.A. 1983. "Parameterized Programming," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 5, Sept. 1984, pp. 528-543
- [19] Jarzabek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 2003, Helsinki*, pp. 237-246
- [20] Jarzabek, S. and Li, S. "Unifying clones with a generative programming technique: a case study," *Journal of Software Maintenance and Evolution: Research and Practice John Wiley & Sons, Volume 18, Issue 4, July/August 2006*, pp. 267-292
- [21] Kang, K et al. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Pittsburgh, Nov. 1990
- [22] Karhinen, A., Ran, A. and Tallgren, T. 1997. "Configuring designs for reuse," *Proc. Int. Conference on Software Engineering, ICSE'97, Boston, MA., 1997*, pp. 701-710.
- [23] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. "Aspect-Oriented Programming," *Europ. Conf. on Object-Oriented Programming, Finland, Springer-Verlag LNCS 1241, 1997*, pp. 220-242
- [24] Kiczales, G., Hilsdale, E., Hugunin, J. et al. 2001. "An Overview of AspectJ," *Proc. of 15th European Conference on Object-Oriented Programming, Budapest, Hungary, June 18-22, 2001, Lecture Notes in Computer Science, 2072*, pp. 327-353
- [25] Loughran, N., Rashid, A., Zhang, W. and Jarzabek, S. "Supporting Product Line Evolution with Framed Aspects," *3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS'04, March 22-26, 2004, Lancaster UK*
- [26] Mesbah, A. and van Deursen, A. "Crosscutting Concerns in J2EE Applications," *Proc. 7th IEEE International Symposium on Web Site Evolution, WSE'05, Budapest, Hungary, Sept. 2005*, pp. 14-21
- [27] Meyer, B. *Object-Oriented Software Construction*, Prentice-Hall, London, 1988
- [28] Musser, D. and Saini, A., *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley, Reading (MA), USA.
- [29] Parnas, D., *On the Criteria To Be Used in Decomposing Software into Modules*, *Communications of the ACM*, Vol. 15, No. 12, December, 1972, pp.1053-1058
- [30] Pettersson, U., and Jarzabek, S. "Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach," *ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, September 2005, Lisbon*, pp. 326-335
- [31] Prieto-Diaz, R. "Domain analysis for reusability," *Proc. COMPSAC'87, October 1987, Tokyo, Japan*, pp. 23-29
- [32] R Rajapakse, D.C and Jarzabek, S. "An Investigation of Cloning in Web Portals," *Int. Conf. on Web Engineering, ICWE'05, July 2005, Sydney*, pp. 252-262
- [33] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on Emerging Discipline*, Prentice Hall, 1996

- [34] Smaragdakis, Y. and Batory, D. "Application generators," in Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering, J. Webster (ed.), John Wiley and Sons, 2000
- [35] Soe, M.S., Zhang, H. and Jarzabek, S. 2002. "XVCL: A Tutorial," Proc. 14th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE'02, ACM Press, July 2002, Italy, pp. 341-349
- [36] STL, <http://www.sgi.com/tech/stl/>
- [37] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. "N Degrees of Separation: Multi-Dimensional Separation of Concerns", Proc. International Conference on Software Engineering, ICSE'99, Los Angeles, 1999, pp. 107-119
- [38] Tichy, W. "Tools for Software Configuration Management," Proc. Int. Workshop on Software Configuration Management, Grassau, 1988, Teubner Verlag, pp. 1-20
- [39] Thompson, S., "Higher Order + Polymorphic = Reusable", unpublished manuscript available from the Computing Laboratory, University of Kent. <http://www.cs.ukc.ac.uk/pubs/1997>
- [40] Zhang, H.Y., Jarzabek, S. and Soe, M. S. "XVCL Approach to Separating Concerns in Product Family Assets", Proc. Generative and Component-based Software Engineering (GCSE 2001), Erfurt, Germany, September 2001, pp. 36-47
- [41] XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, <http://fxvcl.sourceforge.net>
- [42] Yang, J. and Jarzabek, S. "Applying a Generative Technique for Enhanced Reuse on J2EE Platform," 4th Int. Conf. on Generative Programming and Component Engineering, GPCE'05, Sep 29 - Oct 1, 2005, Tallinn, pp.. 237-255