

## **Chapter 6 Conventional method support for managing changes during maintenance and evolution**

---

### *Summary of this chapter:*

In Chapters 2-5, we discussed static program analysis, reverse engineering, model-based design and Software Configuration Management techniques. We commented on their strengths and limitations in addressing the challenges of software maintenance and long-term evolution.

In this Chapter, we further highlight the difficulties to combat the problems of change with conventional techniques, in view of maintenance and evolution challenges discussed in Chapter 1. We also briefly preview some of the remedies to fight these problems using unconventional approaches. With emphasis on complexity inherent in software, we focus on the following problems: Poor visibility of past changes triggers much re-work that could be avoided should the knowledge of the past changes be available to developers. Each new change is re-invented afresh and implemented in its own way, despite possible similarities to past changes. By ignoring similarity of evolution patterns, we reinvent the way changes must be implemented, redoing the same kind of costly change impact analysis all over again. We blow up code size and complexity. The number of similar component versions explodes, degrading conceptual integrity of the program design. The overall complexity of an evolving program grows, making any future changes even harder.

This Chapter motivates the mixed-strategy approach discussed in Part II which applies a generative technique to address the above problems. The premise of mixed-strategy is that we can ease the above problems by preserving the knowledge of past changes in the form that enables us to effectively reuse this knowledge when implementing new changes. The concept of reuse-based evolution is about that.

### **6.1 Software complexity factor**

Software complexity is the main factor affecting productivity of maintenance programmers. The three prime approaches to manage software complexity are componentization (“divide and conquer”), separation of concerns and abstraction. In his famous paper “No Silver Bullet” [4], Brooks argues that whether we use a machine language or a high-level programming language, we can’t rid of programs’ essential complexities. None of the technological advances of last decades proved Brooks wrong, and we think none ever will.

Indeed, it is difficult to find a general way to reduce program complexity beyond certain threshold. Three main factors determine the essential complexity threshold, namely (1) the complexity of a problem and its solution at the concept level, (2) the complexity of a problem solution by means of a computer program, and (3) the complexity of mappings and interactions between the conceptual and program level solutions.

The fact that conceptual and programmatic concerns cannot be cleanly separated one from another in program components is one of the important forces that constrains our efforts in using componentization in combating software complexity. Most often, we can’t make

software to play by the rules of a lego model which works so fine for hardware construction [2][23]. This difficulty to achieve clean decomposability marks an important difference between hardware and software.

Software components are less stable and interact in more complex ways than hardware components. They are often changed, rather than replaced, therefore must be much more flexible than hardware components. Another important difference is that we never try to change hardware as often as we change software, and in such drastic ways. Consequently, software architectures are softer - less stable and more intangible - than architectures in traditional engineering disciplines, and must be flexible to be practical. This fragile nature of software components and architectures to much extent determines how well various techniques can effectively help in maintenance and evolution.

Programs that are not subject to frequent and unpredictable changes, can be effectively componentized (e.g., some of the middleware service components). As the technology changes, such components are most often replaced by new ones (like an obsolete hardware device) rather than evolved. But stable or exchangeable components is not the source of our main software problems. What we find really difficult to work with is software in unstable, poorly understood domains [2]. Any successful program of that kind is bound to change often. Information hiding [21] and separation of concerns [9] are the two main mechanisms for containing changes. But again – if we could always localize and hide the impact of change into small number of components – we would not have a problem with changing programs. Computing concerns (or aspects) are not easily separated either, although recently there have been a number of attempts to provide a suitable mechanism for separating concerns [20][27].

Modern middleware component platforms (such as .NET™ or J2EE™) use standardized architectures, patterns and APIs to enable reuse of common service components pre-implemented in the platform infrastructure. Model-Driven Engineering can further hide the complexity induced by platforms providing domain-specific solutions available directly on a platform of our choice [25].

Still, componentization in system areas above the middleware remains a problem. Software components have not become a commodity to the extent as optimistic forecasts of late 1990s were predicting (see, for example, Gartner and Cutter Consortium reports).

However, the most serious limitation of componentization as means to fight software complexity is that as long as we compose, test and maintain concrete components, we are bound to face the complexity proportional to the system size. In a system of thousands of interacting components, comprising millions lines of code, the complexity of validating, testing and controlling maintenance changes must eventually explode beyond control. It is a matter of speculation to what extent we can reduce this complexity by composing already implemented, concrete components. In author's view, this complexity is part of the Brook's essential complexity [4]. It can be helped, but cannot be substantially reduced.

Difficulty to cleanly separate concerns by means of decomposition necessarily hinders changeability. What becomes very handy in this context, is the infinite malleability of software, much above any levels that are possible in hardware. Nothing else in the horizon seems more promising than malleability as a potential rescuer in solving the dilemma of easing software changes. Generative approaches [7] apply unconventional means to exploit software malleability to lower the barrier of software complexity. Some of the well-known

techniques attempt to bring Dijkstra's separation of concerns [9] from the concept level down to design and implementation levels. Aspect-Oriented Programming (AOP) [20], Multi-Dimensional Separation of Concerns (MDSOC) [27], and AHEAD [3] modularize concerns at extra, meta-level planes, beyond conventional software components. Program manipulation (such as aspect weaving in AOP) is then used to produce concrete components from a meta-level program representation. "Configuration knowledge" [7] defines mappings that separate problem domain concerns from the solution space.

Separation of concerns also makes a step towards more generic program representations. Concerns that we manage to modularize in unconventional way, as well as conventional modules of primary decomposition (in AOP), become more generic, reusable in multiple contexts, to some extent alleviating the problem of having to deal with the complexity proportional to the size of a physical program.

In Part II, we see how we can exploit the "soft" nature of software with mixed-strategy approach powered by XVCL [28], a generative program manipulation technique based on frame concepts [2]. XVCL's primary goal is to unify similar program structures with generic, meta-level representations, and also to achieve a fair amount of separation of concerns, as much as it is practical. We comment further on generative approaches to addressing separation of concerns, as well as on an interesting relationship between separation of concerns and generic design principles, in Chapter 14.

Other than componentization, we may try to reduce software complexity by raising the level of abstraction of a program representation. This works fine in narrow application domains: 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 essential complexity of the overall program solution, such generation-based domain-specific programming environments shield a programmer from some of the essential complexities which are now contained in the domain-specific code that is manipulated by a generator. DSL may take many different forms, depending on a domain, such as a formal text (e.g., BNF for parser generator), visual interface (e.g., GUI) or models (in Model-Driven Engineering approaches [25]).

Unfortunately, abstraction directly translates into productivity gains only in narrow application domains. Still, problems that can be helped with generators may recur in many programs, even in many domains. Abstraction can reduce complexities induced by programming platforms [25]. Finally, abstraction offers modeling notations, such as UML [24], that play an important descriptive role.

## **6.2 Tackling software evolution challenges**

Evolution challenges, some of which we discussed in Chapter 1, are not new and we believe there have been many attempts to address some of them with conventional techniques. There are, however, technical problems with that, as change capabilities of conventional techniques

are not really geared to tackle evolution challenges. We comment on mature and commonly used techniques, such as parameterization features of programming languages, “design for change” with information hiding, OO and component-based design, and Software Configuration management (SCM). These techniques have been around for long enough to observe their actual impact on our ability to deal with changes.

Each technique provides a sound solution to some aspect of evolution. Still, collectively, these techniques are limited in effective evolution.

Historically, the main role of programming languages has been to provide abstractions to define code structures to be executed on a computer. Typical abstractions are classes, objects, functions/methods, function invocations or messages. Change mechanisms have never become powerful enough to make changes visible and tractable. Change mechanisms continued to remain second class citizens in modeling notations [24], development methodologies, IDEs (Interactive Development Environments), and component platforms, which were necessarily influenced by language concepts.

Generic design aims at achieving non-redundancy by unifying differences among similar program concepts or structures, for simplification purpose. Despite its importance in fighting software complexity, generic design capabilities of conventional programming languages and techniques are often insufficient to effectively unify the variety of software similarity patterns that we encounter during evolution and reuse [18]. Generics [14] (or templates in C++) can effectively unify type-parametric differences among classes/functions. Generics can be hardly considered a change mechanism as very few changes in evolution practice are of type-parametric nature. Still, type parameterization has a role to play in building generic solutions unifying similar program structures, as required in reuse-based evolution. However, even in well-designed class libraries and application programs, we find similar program structure – classes and bigger patterns of collaborating components - with much differences that cannot be unified by generics. Up to 60% of code may be contained in similar classes that differ in generics-unfriendly ways. We refer to detailed studies in Chapter 9 and 10 and also in [1][16][17][18][22][29].

Macros are a simple form of a generative technique, not strictly a conventional technique in the sense we use term ‘conventional’ in this book. We comment on macros here because of their popularity in the main-stream programming. Macros handle changes/variations at the implementation level, by performing local code expansion at designated program points. Macros have been used for a variety of purposes (see [10], for a study of a popular cpp). They have been also used to manage evolutionary changes. Failing to utilize design concepts to manage changes, macros handling code variants related to one source of change become scattered through a system and difficult to trace. Macros related to different sources of changes interact in chaotic ways. As programs with macros become more general, they also become increasingly complex to work with [19].

To support evolution, we need global controls to explicate dependencies among modifications spread across the whole system (or even across system releases) related to a given source of change. Most of the macros lack such global controls. While macros can handle simple, localized changes across component versions, generally they are not geared to unify a variety of similarity patterns that arise in evolution. Frame Technology™ [2] and XVCL [28] (based on frame concepts and described in Part II) are also realized by an expansion operation that

performs “composition with adaptation” of generic, meta-level components. However, unlike macros, frame mechanisms are specifically meant to address evolution challenges discussed in Chapter 1.

Inheritance can explicate certain types of changes via sub-classing. It works fine if the impact of change is local, confined to a small number of classes, and, at the same time, if the impact of change is big enough to justify the need of creating new subclasses. Unfortunately, quite often this is not the case and a small change in requirements may lead to big changes in OO design and code, due to extensive sub-classing. Even small modification of one class method requires us to derive a new class, repeating almost the same code for the modified method. There is no formal way to group all the classes modified for the same purpose or point the exact modification points. While inheritance is critical for usability reasons, we are not aware of inheritance playing a significant role in managing day-to-day changes or long-term evolution. Inheritance is not a technique of choice to build generic design solutions, either. STL [26] – a hallmark of generic programming with C++ templates – has a flat class structure.

Design patterns [13] can ease future changes, and help in refactoring programs into more maintainable and often more generic representation. Design patterns have a role to play in unifying similarity patterns during evolution. Pattern-driven development becomes increasingly important in modern component platforms such as .NET™ and J2EE™. Component-based reuse is most effective when combined with architecture-centric, pattern-driven development. Patterns are basic means to achieve reuse of common service components. Standardized architectures together with patterns lead to beneficial uniformity of program solutions.

However, the visibility of pattern application in a program is somewhat informal and limited. Some patterns can be identified by using standard names for functions, data, classes and files participating in a pattern instance. It is common to comment out pattern application. Similarly, the only way to know how pattern instances differ one from another is to comment out the differences. The benefits of using patterns for maintenance and evolution would be greatly increased if pattern applications, as well as similarities and differences among pattern instances, could be formally represented, and amenable to some automation. Generative techniques open such possibilities by capturing patterns at the extra meta-level plane [29].

In design for change with information hiding [21], we encapsulate changeable elements within program units such as functions, classes, components or component layers. When change occurs, if the impact of change can be localized to a small number of program units, we substitute the encapsulated element with new one, and as long as we do not affect interfaces – changing a program is easy. But again – if we could always localize and hide the impact of change in a small number of components – we would not have problems with changing programs at all. Problems start exactly when the impact of change is not local.

Many unexpected changes have non-local impact on a program as our existing design may not cater for them. Aspects (cross-cutting concerns) are examples of changes that cannot be localized. Already mentioned Aspect-Oriented Programming (AOP) [20] and Multi-Dimensional Separation of Concerns (MDSOC) [27] are motivated by the observation that one modular decomposition cannot cater for all kinds of changes that may happen, and offer unconventional solutions to this problem. AOP caters for changes that can be realized by

weaving aspects into program modules at specified joint points (or taking out aspects from program modules). Not all the changes are of that kind. At times, modifications may occur at arbitrary points in a program; modifications required at different program points may be similar but not the same, or completely different; modifications may affect the design (component interfaces or component configuration). To address this kind of changes, we would need to bring in a new aspect whenever such change occurs. Such aspects could crosscut base code modules in arbitrary, unexpected ways, and crosscut each other. Chains of modifications could occur within certain aspects – we would need parameterized aspects, possibly with other aspects.

During evolution, we often have to keep component (class or function) versions before and after change. Components after change are needed in a new system, while components before change appear in earlier releases of the system, which may be operational and still changing according to their own needs. Even if a change is small, a typical solution is to create a new component version. This may result in a huge number of component versions. A common approach is to use Software Configuration Management (SCM) tools [6] and keep the release history in an SCM's repository (we discuss CVS in Chapter 5). Each such component (or component configuration) accommodates some combination of variant features. When it comes to implementing a new release of a system, we try to select either individual component versions or their configurations “best matching” the new system. Then, we customize them to fully meet requirements of the new system. This selection/customization effort determines programmers' productivity. As it is not easy to distill the overall picture of similarities and differences across system releases from the SCM repository, selecting “best matching” components and customizing them may become expensive [8]. Various approaches have been proposed to extract and visualize information from the SCM repository to aid in evolution [11][12][15].

### **6.3 Conclusions of this chapter**

We discussed some of the difficulties in addressing evolution challenges with conventional techniques, main of which are: poor visibility of past changes, the lack of a clear picture of similarities and differences among system releases at all system levels from architecture down to code details, and the lack of strong enough generic design mechanisms. Conventional approaches, based on enumerating component versions, lead to explosion of similar component versions. The more component versions, the more difficult to see what's common and what's different among system releases, and how to reuse already implemented features in building new system releases. Poor visibility of past changes triggers much re-work that could be avoided should the knowledge of the past changes be available to developers. Necessarily ad hoc maintenance degrades the conceptual integrity of the program design. The overall complexity of an evolving program grows, making any future changes even harder.

## **References**

- [1] 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, May 2005, pp. 451-459
- [2] Bassett, P. *Framing software reuse - lessons from real world*, Yourdon Press, Prentice Hall, 1997
- [3] 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

- [4] Brooks, F.P. "No Silver Bullet," *Computer Magazine*, April 1986
- [5] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002
- [6] Conradi, R. and Westfechtel, B. "Version Models for Software Configuration Management," *ACM Computing Surveys*, 30(2), 1998, pp. 232-282
- [7] Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000
- [8] 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
- [9] Dijkstra, E.W. "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 1982, pp. 60-66
- [10] Ernst, M., Badros, G., and Notkin, D. "An Empirical Analysis of C Preprocessor Use," *IEEE Transactions on Software Engineering*, Dec. 2002, pp. 1146-1170
- [11] Fischer, M., Pinzger, M. and Gall, H. "Populating a Release Database from Version Control and Bug Tracking Systems," *Proc. Int. Conf. Soft. Maintenance, ICSM'03*, Sept. 2003, pp. 23-32
- [12] Gall, H., Jazayeri, M. and Krajewski, J. "CVS Release History Data for Detecting Logical Couplings," *Proc. Int. Workshop on Principles of Software Evolution, IWVSE'03*, Sept. 2003, Helsinki, pp. 13-23
- [13] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley
- [14] 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.
- [15] German, D., Hindle, A. and Jordan, N. "Visualizing the evolution of software using softChange," *Proc. 16th Int. Conf. on Software Eng. and Knowledge Eng., SEKE'04*, Banff, Canada, June 2004, pp. 1-6
- [16] Jarzabek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM Press, September 2003, Helsinki, pp. 237-246; paper received ACM Distinguished Paper award
- [17] 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
- [18] Jarzabek, S. "Genericity - a "Missing in Action" Key to Software Simplification and Reuse," to appear in *13<sup>th</sup> Asia-Pacific Software Engineering Conference, APSEC'06*, IEEE Comp. Soc., 6-8 December 2006, Bangalore, India
- [19] Karhinen, A., Ran, A. and Tallgren, T. 'Configuring designs for reuse', *Proc. Int. Conf. Software Engineering, ICSE'97*, Boston, MA., pp. 701-710
- [20] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. Aspect-Oriented Programming," *European Conf. Object-Oriented Programming*, Finland, Springer-Verlag LNCS 1241, 1997, pp. 220-242
- [21] Parnas, D., 1972. On the Criteria To Be Used in Decomposing Software into Modules, *Communications of the ACM*, Vol. 15, No. 12, December, 1972, pp.1053-1058
- [22] 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*, September 2005, Lisbon, pp. 326-335
- [23] Ran, A. "Software Isn't Built From Lego Blocks," *ACM Symposium On Software Reusability*, 1999, 164-169
- [24] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Languages Reference Manual*, Addison-Wesley, 1999
- [25] Schmidt, D. "Model-Driven Engineering," *IEEE Computer*, February 2006, pp. 25-31
- [26] SGI STL, <http://www.sgi.com/tech/stl/>.
- [27] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *Proc. Int. Conf. Software Engineering, ICSE'99*, Los Angeles, 1999, pp. 107-119
- [28] XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, <http://fxvcl.sourceforge.net>
- [29] Yang, J. and Jarzabek, S. "Applying a Generative Technique for Enhanced Reuse on J2EE Platform," *4<sup>th</sup> Int. Conf. on Generative Programming and Component Engineering, GPCE'05*, Sep 29 - Oct 1, 2005, Tallinn, pp.. 237-255