Chapter 7 Mixed-Strategy approach – an overview

Summary of this chapter:

Last chapter motivated the quest for effective solutions to software evolution beyond conventional techniques. Among many problems complicating evolution of programs, two are particularly acute, namely poor visibility of changes, and explosion of look-a-like component versions caused by weak generic design capabilities of conventional techniques. If similarities and differences among past releases remain to much extent implicit, it is difficult to reuse features already implemented in past releases when building new release of an evolving software. Any changes affecting past releases must then be done case-by-case on each system release. In this Chapter, we present the concepts of mixed-strategy approach which relies on clear exposition of similarities and differences among the software releases, and leads to more effective reuse-based evolution. The purpose of this Chapter is to build an intuitive understanding of the approach, which will help the reader easier absorb technical details presented in subsequent chapters.

7.1 The concepts of reuse-based evolution and mixed-strategy approach

The idea behind reuse-based evolution is to *reuse the knowledge of past changes* to effectively implement future changes. The essence of reuse-based evolution is clear visibility of changes, clear understanding of software similarities and differences at all granularity levels, and non-redundancy achieved by unifying *similarity patterns* of evolutionary changes with generic structures.

Reuse-based software evolution is a concept, or a principle of how to address changes so that evolution is easier, faster and cheaper. Different techniques can be used to realize reuse-based evolution. However, we already hinted at some problems that haunt approaches that attempt to tackle evolutionary changes at the level of concrete programs. For example, if a Software Configuration Management (SCM) [2] tool is used, we work with component versions and component configurations that occurred in various system releases, with documented differences among them. We face the problems of explosion of look-alike components [4] and poor visibility of how changes affect software during evolution. The first problem is inherent in evolution of concrete components as we often need a component versions before and after changes. The second problem can be eased with tools that automatically analyze the SCM repository and visualize the change history [5][6][8].

While difficult at the level of concrete components, we can attempt to manage changes at an *extra plane* beyond a program itself. We explain by an analogy to a well-known technique of Aspect-Oriented Programming (AOP) [10]. Computational aspects that crosscut modules (classes) of the primary program decomposition cannot be defined as self-contained modules using conventional OO design techniques. AOP introduces a meta-level *extra plane* to create unconventional modules localizing crosscutting aspects, easing maintenance of both crosscutting aspects and program modules of primary decomposition.

In mixed-strategy approach, we extend a conventional program with a meta-level *extra plane* to manage changes in evolving software. We represent programs in generic (non-redundant), highly adaptable form, capable of accommodating changes arising during evolution. Mixed-strategy representation also encompasses the knowledge of how to derive custom components of any system release from these generic forms. The main idea is to keep change specifications separately from code, but fully and formally integrated with code structures.

Consider the following evolution scenario: Rather than burying changes into the code, we keep *change specifications* visibly separated from the base code, for as long as it is useful to do so[1]. We integrate change into the base code permanently when its visibility is not required anymore. Change specifications contain knowledge of a complete chain of modifications – at architecture and code levels – linked to a particular source of change.

Change specifications are in both human- and machine-readable form, so that we can easily see and automatically produce a program before or after any given change. This book introduces such a notation, XML-based variant Configuration Language, XVCL for short. XVCL allows us to express program modifications of any conceivable type and granularity, at both architecture (components, interfaces, subsystems) and detail code levels. A tool, called XVCL Processor, interprets change specifications and modifies a program accordingly.

> The idea behind mixed-strategy approach is to use a programming language to express the syntax and semantics of a program – it's component architecture and runtime behavior in terms of user interfaces, business logic, and a database. Then, to impose XVCL on conventional program structures to express the syntax and semantics of change. Term *mixed-strategy approach* reflects this synergistic merger of conventional programming techniques and XVCL, to achieve separation of change, and other concerns that matter at program design time, from the definition of runtime program structures (such as functions, classes or components).

XVCL change mechanisms have a capability to manage multiple changes implemented over time. This includes reconciling change specifications that interact with each other, and building generic design solutions unifying similar threads of changes that emerge during evolution. XVCL Processor can selectively inject into the system changes implemented at different times during evolution, or take them out of the system.

Groups of similar changes are handled in a common way, by means of generic, customizable mixed-strategy program representation. This generic representation is very flexible: XVCL Processor can instantiate it, based on change specifications, to create many variant forms that appear in our system (or even in many similar systems). System views at all abstraction levels become fully integrated with the knowledge of changes affecting them. The knowledge of similarities and differences at all abstraction levels remains explicit throughout evolution.

We gain the visibility of change patterns and the impact of changes on software architecture and code. From a mixed-strategy generic representation of an evolving program, we can derive any concrete component configuration required in a specific system release, by injecting changes relevant to that release.

---

[1] A typical situation when it is beneficial to keep changes separately from code components is when different changes are needed in component versions used in systems released to different customers.

During evolution, changes accumulate. We continuously refine a mixed-strategy representation so that similarity patterns of both changes and an evolving system itself are exposed and controlled, rather than implicit and multiplied in a large number of ad hoc instances.

7.2 Change-design vs. design for change

There is a difference between *change-design* concept and *design for change*. *Change specifications* discussed in the last Section are related to *change-design*.

Design for change is a design goal that we achieve by means of programming language constructs. Parnas [11] proposed information hiding technique to realize that goal. Object-Oriented languages introduced classes with public interfaces and hidden private parts of class implementation to directly support design for change with information hiding. Component-based techniques pushed further Parnas' ideas whereby we can access components via APIs without knowing the components' source code or even identity. Other conventional techniques that help achieve design for change are generics, inheritance, dynamic binding, and design patterns [7]. Model-based design described in Chapter 4 (Part I) is a powerful design for change technique in some cases. AOP [10] and MSDOC [12] simplify changes in certain types of crosscutting concerns.

The idea of *change-design* is to create a plane at which all changes become the first class citizens, can be designed and understood in synergy, but without conflicts, with a program structure. A programming language defines runtime component structure and takes care that all runtime quality goals are met, while the change-design mechanism takes care of maintenance and evolution concerns, such as changeability or genericity. The concept of reuse-based evolution motivates such separation, but does not prescribe any particular approach to achieving the separation.

7.3 Concepts of the mixed-strategy approach

Mixed-strategy solution is built be separating program *construction-time concerns,* such as maintenance, evolution or reuse concerns, from the actual program runtime structures, such as functions, classes or components, implemented in one of the programming languages. Program construction-time concerns refer to knowledge of how various features have been implemented in system releases, visibility of past changes, changeability, traceability of information from requirements to code, genericity, crosscutting concerns – anything that helps during development, evolution, or reuse. Program runtime concerns refer to program behavior (user interface, computation logic, persistence storage of data, etc.), componentization, underlying component platforms, and quality attributes such as performance, reliability or availability.

The essence of mixed-strategy is that we use a conventional programming languages (e.g., Java, C++, C, Perl or PHP) and platforms (e.g., J2EE or .NET) to deal with runtime concerns, and XVCL to deal with construction-time concerns. Conventional program technology defines the syntax and semantics of a program, while XVCL defines the syntax and semantics of change and other construction-time concerns. The overall representation that integrates conventional program with XVCL forms a mixed-strategy solution (Figure 1).

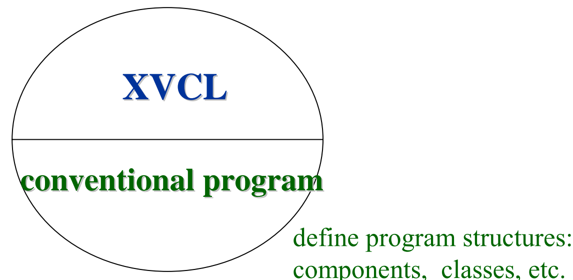

Figure 1. Mixed-strategy solution

We consider all kinds of changes that arise during day-to-day maintenance such as fixing bugs or enhancements of program's functions, as well as accumulative results of many changes over years of software evolution. Here is an intuitive description of the mixed-strategy approach. As program evolves, change specifications, recorded in XVCL, are imposed on a base program. A base program plus change specifications form a mixed-strategy representation that developers work with. This representation contains the evolution history – architecture and code for each of the releases, along with complete trace of changes applied to *generic components* to obtain a given release. Generic components are built by applying XVCL to the base program. They are continuously refined during evolution, to accommodate changes relevant to various system releases. Therefore, the same mechanisms of XVCL are used for representing both changes specifications and generic components that together form a mixed-strategy solution.

A unit of mixed-strategy representation decomposition is called an *x-frame*. Change specifications, generic components and all the base code are contained in x-fames. Being unconstrained by the rules of the underlying programming language, x-frames can be small or big, coinciding with the natural granularity of change.

X-frames in a mixed-strategy representation are analogous to functional program abstractions such as components, classes, methods and functions in programming languages. However, the goals of mixed-strategy decomposition and the goals of conventional program decomposition are different:

- The goal of mixed-strategy decomposition is to take care of construction-timeconcerns: To explicate changes, facilitate future changes via adaptation of generic components capable of accommodating variant features arising during evolution, unify similarity patterns observed during evolution in changes and the structure of the base program.
- The goal of conventional program decomposition is to ensure that runtime program behavior meets its functional and quality requirements. Here, the concerns are software architecture in terms of components and interfaces, software behavior that meets all its functional requirements and quality requirements such as of usability, performance, reliability, availability and many others.

The goals of mixed-strategy and conventional decompositions are intimately interwoven, and complementary to each other. In conventional programming, these goals are often conflicting.

Based on analysis of trade-offs, developers must compromise less important goals so that they can ensure that more important goals are met.

The benefit of mixed-strategy is that we can give full attention to construction-time concerns, optimizing them to required levels, without compromising any of the runtime program properties. Both construction-time and runtime properties may be optimized separately and independently of each other, without constraining each other. Mixed-strategy achieves separation of construction-time concerns from the actual program structures and their required qualities.

It is interesting to note that such separation of construction-time concerns from the issues concerning the structure and properties of the actual product is common in traditional engineering. It marked maturity of engineering disciplines. Most often, each of the two areas of concern are even backed by scientific theories and well-established processes, standards and tools. For example, the design of a car or a bridge involves much math calculations based on laws of physics and material sciences. On the other hand, car production or bridge building is based on management science, theories of manufacturing processes and tools supporting the construction process.

The developer's mental model of software under evolution must encompass both x-frames and functional program abstractions (components, methods and functions). Therefore, it is beneficial to have natural and transparent mappings between x-frames and program abstractions. For example, it is common that an x-frame corresponds to a subsystem, group of similar components, classes, methods or interface definitions (e.g., in IDL).

The following properties of mixed-strategy are important for practical reasons:

- *Traceability: Full transparency of mappings between a mixed-strategy generic representation and a program itself.* This ensures that virtual generic views get never disconnected from the base code. Failing to do so is a common pitfall of generators, and an important reason why abstractions disconnected from code are not trusted by programmers [3].
- *The ability to organize generic structures in a hierarchical way*. For example, generic classes from generic methods, generic components from generic classes; generic component configurations from generic components and patterns defining composition rules; generic subsystems from generic component configurations. This ensures that similarities at all levels and of any granularity can by unified, inter-linked, and the whole mixed-strategy solution space can be normalized for non-redundancy.
- *The ability to express easily and naturally differences among similar program structures of arbitrary type and granularity*. This ensures that any group of similar program structures can be unified with a generic representation, whenever this is required for simplification reasons, interpedently of differences among structures, members of a group. It helps to achieve "small change – small impact" effect, which is so difficult to achieve with conventional program abstractions such as procedures, classes or components [1].

The overall mixed-strategy representation of a software system under evolution is a hierarchy of x-frames called an *x-framework*. An x-framework is a mixed-strategy counterpart of an evolution history stored in an SCM repository. It contains full details of software architecture and code for each release, generic structures representing similar, recurring patterns of design

(e.g., group of similar components), changes that lead to various releases, and other information that helps in systematic maintenance and evolution.

7.4 A preview of the mixed-strategy approach: an example

We illustrate main concepts with FRS evolution example, discussed in Chapter 5 and recalled below in Figure 2.

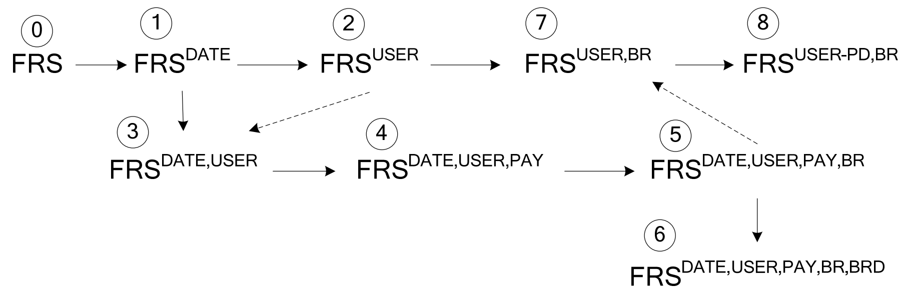

Figure 2. Stages in FRS evolution

FRS components are organized into three tiers, namely the user interface, business logic and database tiers. User interface components allow FRS administrators and reservation requestors to interact with the system. Business logic components accomplish various actions related to managing users, facilities and reservations. The database stores data related to users, facilities and reservations.

7.4.1 Evolving FRS with mixed-strategy representation: a big picture

Mixed-strategy representation is built around a model of software system structure from subsystem, to component layer (e.g., user interface or business logic), to component, to class/procedure, and to class/procedure implementation details. Such a precise model of structure in a form that can be manipulated by XVCL Processor is necessary to deal with evolutionary changes affecting software at all those levels.

Rather than melding changes into code, we specify the impact of evolutionary changes on software in XVCL. Code structures affected by changes are continuously refined into generic mixed-strategy representation structures that can be easily adapted to changes arriving from various sources (e.g., changes relevant to various system releases). The goal is to reuse the same mixed-strategy structures in all system releases.

Figure 3 shows a sketch of an x-framework that represents five initial stages of FRS evolution. The x-framework contains full knowledge of how to build each FRS release from generic building blocks, shown as x-frames enclosed in the dashed rectangle. In Figure 3, we see top-level view of FRS decomposition, with XVCL <**adapt**> commands showing how to compose FRS from its building blocks. Solid arrows correspond to <**adapt**> commands. Tick dashed arrows at the bottom show five FRS releases produced from the x-framework. Any of the top-most x-frames, called an SPC, contains SPeCifications of how to build a specific FRS release. For example, SPC-$FRS^{DATE,USER}$ sets up parameters and controls, and contains definitions of other change specifications (e.g., unique features) required for $FRS^{DATE,USER}$. XVCL Processor interprets change specifications, propagates changes across the x-framework, producing custom component versions required in a given system release.

Numbers attached to arrows identify stages of evolution and correspondence between SPC and respective FRS releases.

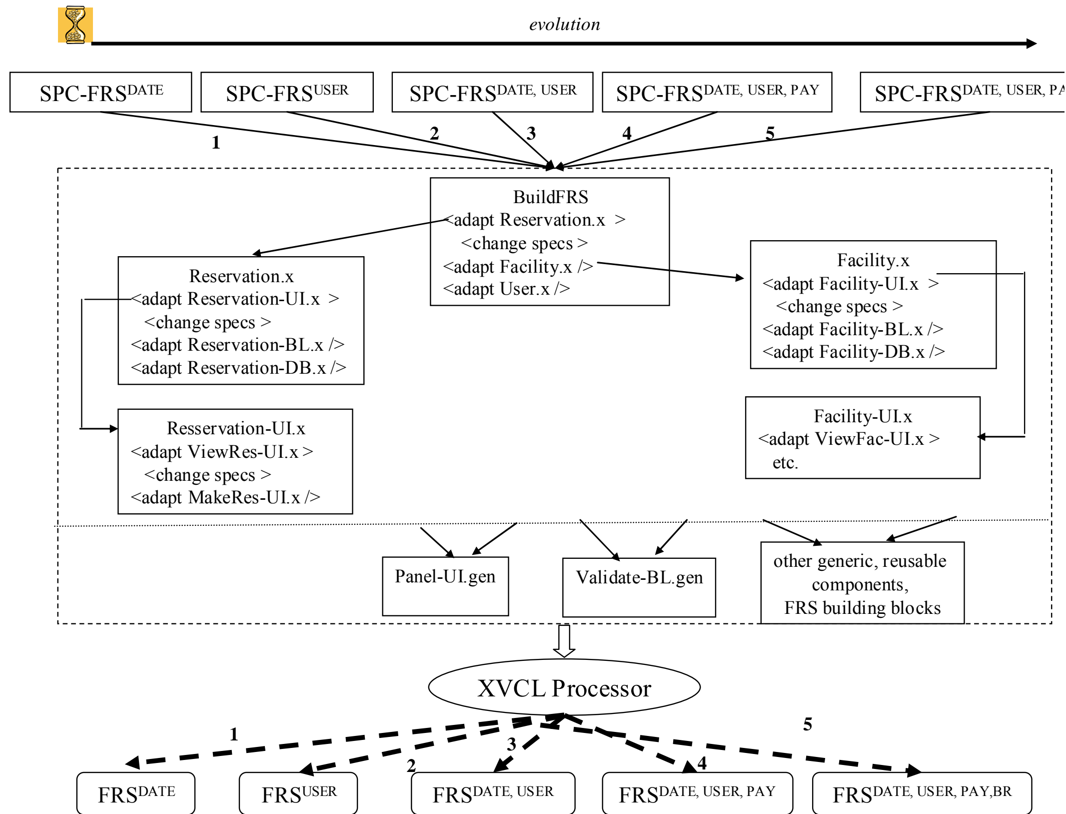

Figure 3. FRS evolution with mixed-strategy

Larger x-frames are composed from smaller x-frames, their building blocks. Composition relationship among x-frames, defined by <**adapt**> command shown as solid arrows in Figure 3. An arrow between two x-frames: X → Y is read as "X adapts Y", meaning that X controls adaptation of Y, while the composition takes place. Composition is the main organizing principle for an x-framework.

The XVCL Processor traverses the x-framework, starting with an SPC, in depth-first order, interpreting any encountered XVCL commands. Whenever XVCL Processor encounters an <**adapt**> command, the interpretation of the current x-frame is suspended and the interpretation of the x-frame designated by the <**adapt**> command begins. XVCL Processor emits "as is" code contained in x-frames visited during processing. Notice also that during processing x-frames in an x-framework are never changed by the XVCL Processor. Only the result of x-frame composition is emitted to the output.

Typically, x-frames at the bottom levels of an x-framework contain implementation of program components (or even classes or class methods). Above them are x-frames representing architecture-level elements (groups of components, interfaces and subsystems). Towards the top of the x-framework, we often see x-frames whose only purpose is to specify customization, configuration and composition rules for lower-level x-frames. Each x-frame specifies changes for the x-frames below, and receives changes from the upper-level x-frames. Therefore, x-frames are both active and passive. X-frames towards the bottom of an

x-framework tend to be more generic, reusable, adaptable to the needs of various FRS releases (e.g., x-frames below a dotted line in Figure 3).

In addition to composition relationship, an x-framework is further organized by a principle of separation of concerns. For example, x-frames with user interface components and x-frames with business logic are kept in separate partitions of an x-framework.

While changes unique to a given release are contained in a respective SPC, change specifications related to features shared by a number of releases often become an integral part of the evolution knowledge. Such changes are embedded in x-frames below the SPC level, as shown in Figure 3.

Change specifications are formally linked to designated variation points in affected x-frames by means of XVCL commands embedded in x-frames. Changes specified in a higher-level x-frame override any changes contained in the adapted x-frames. This change propagation rule helps us achieve reusability of x-frames across system releases during evolution.

As shown in Figure 3, at each adaptation point (marked with <**adapt**> command), we can specify (optional) changes to be applied to the adapted x-frame. These changes are propagated to all the x-frames reached in the adaptation chain. For example, from <**adapt** Reservation.x>, down to x-frames Reservation.x, Reservation-UI.x, ViewRes-UI, and any further adapted x-frames.

By reading change specifications and tracing change propagation across x-frames, we can see the exact similarities and differences among any two FRS releases, from the subsystem level, to component and to component implantation details.

Applying XVCL to build a mixed-strategy representation is an incremental process. Mixed-strategy representation is continuously refined whenever an opportunity arises to simplify it, make more generic, or more adaptable. Such refinements lead to concise, conceptually clear and easy to work with representation of the evolving system, in contrast to the usual decay of software structure in conventional maintenance. Refinements are essential to observe long-term benefits of mixed-strategy.

7.4.2 A preview of detailed change mechanisms

Composition rules defined by <**adapt**> commands form a macro-structure of an x-framework within which more detailed changes are specified. These changes can be specified at each <**adapt**> command and they define customizations to be applied to the chain of x-frames that originates at that <**adapt**> command. XVCL change instruments include variables, expressions and commands such as <**insert**>, <**break**>, <**select**> or <**while**>. XVCL variables and expressions provide the parameterization mechanisms that make x-frames generic and, therefore, applicable to various releases. Typically, class or method names, data types, keywords, operators or algorithmic fragments are represented as expressions that can be then instantiated by the XVCL Processor, according to the context. We use <**select**> to direct processing into one or more of the many pre-defined branches (called options), based on the value of a variable. With <**insert**> we can modify x-frames at designated <**break**> points in arbitrary ways. A <**while**> command allows us to iterate over certain sections of a x-frame, with each iteration generating custom output. A typical use of <**while**> is in generating many similar concrete components needed in a software system, from a generic x-frame playing the role of a template.

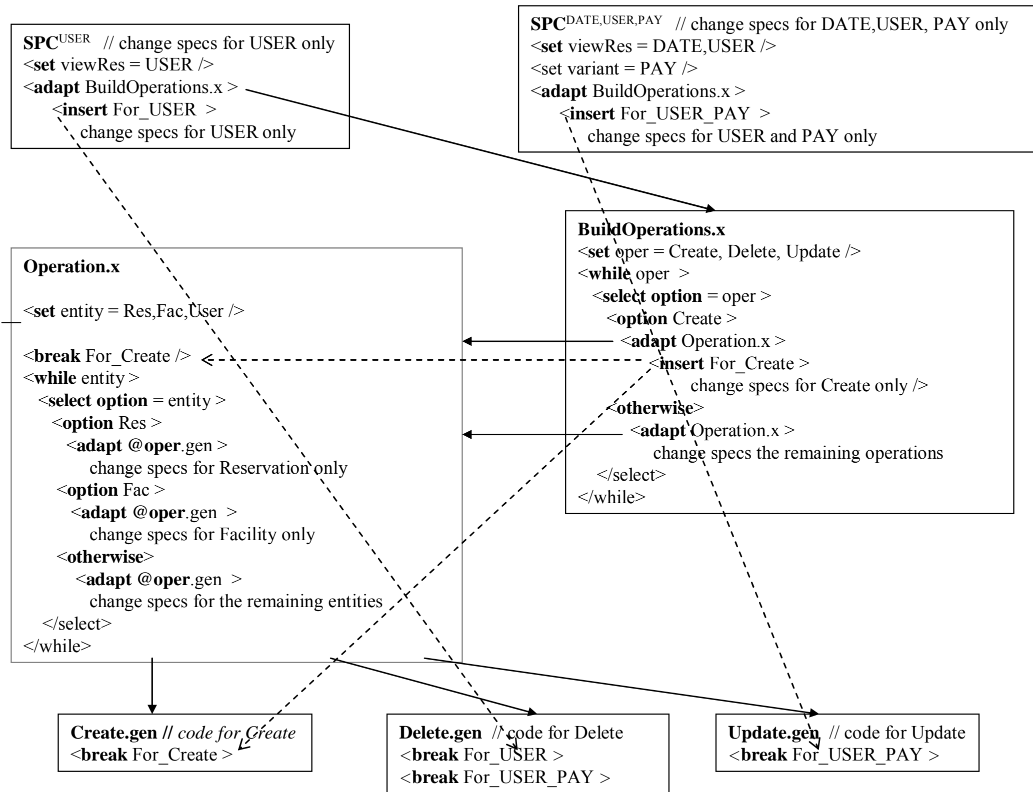

Figure 4. Basic XVCL mechanisms

The usage of the above change mechanisms is informally illustrated in Figure 4. Solid arrows correspond to <**adapt**> commands, while dashed arrows link <**insert**> commands to matching <**break**> points. Suppose we find that there is much similarity among modules in the following groups:

[CreateRes, CreateFac, CreateUser],

[DeleteRes, DeleteFac, DeleteUser], and

[UpdateRes, UpdateFac, UpdateUser].

Despite similarities, there are also differences among modules in each group implied by the semantics of an operation (such as Create, Delete or Update) and the entity (such as Reservation, Facility or User). In addition, in the FRS evolution context, there are differences among modules in each group implied by specific features (such as DATE, USER or PAY) implemented in a given FRS release.

A <**set**> command assigns a list of values to an XVCL variable. XVCL variables **oper** (set in BuildOperations.x) and **entity** (set in Operation.gen) are generic names for operations and entities, respectively. As XVCL variables have global scope, they can coordinate chains of all the customizations related to the same source of variation or change, that spans across multiple x-frames. During processing, values of variables propagate from an x-frame where the value of a variable is set, down to all adapted x-frames. Thanks to this scoping rule, x-frames become generic and adaptable, with potential for reuse in many contexts.

Custom operations for specific entities (e.g., CreateRes, CreateFac, CreateUser) are obtained by adapting respective generic operation x-frames (e.g., Create.gen). X-frames BuildOperations.x and Operation.x navigate the process of building operations.

Changes specific to a certain group of operations (e.g., Create) are specified in x-frame BuildOperations.x. In the example, for operation Create, we insert Create-specific changes into matching <**break**>s named For_Create, placed in x-frames Operation.gen and Create.gen.

Changes specific to a certain operation for an entity (e.g., CreateRes) are specified in x-frame Operation.gen.

Changes specific to a given FRS release are specified in the respective SPC. For example, in SPC^{USER} we <**insert**> USER-specific changes into matching <**break**> named For_USER, placed x-frame Delete.gen.

In each iteration of <**while**> loop in BuildOperations.x, we generate operations Create for various entities. The i'th iteration of the loop uses the i'th value of a control variable **oper**, as assigned in the respective <set> command. The <while> loop in x-frame Operations.x adapts generic operations defined in x-frames below, according to changes specifications propagated from the upper-level x-frames.

Despite its simplicity, the above example communicates the essence of a mechanism that allows us to handle ad hoc variations related to specific operations, entities and FRS releases, without affecting other operations, entities and FRS releases that should not be affected by these variations. Mechanisms for such selective injection of changes allow us to separate variants from common, generic structures, keeping generic structures reusable and easily adaptable.

With this general introduction, in the next chapter we describe main XVCL commands in more details, explaining their role in handling specific evolution problems.

7.5 The role of genericity in mixed-strategy approach

Genericity of program representation is the heart of mixed-strategy approach to reuse-based evolution. We need generic program components that we can be easily adapt to specific need of various system releases. For example, features DATE and USER (view reservation by DATE and view reservations by USER) affect FRS components ViewRes-UI and ViewRes-BL. Different FRS releases may need either of them, both of them or none. Components ViewRes-UI and ViewRes-BL must be generic to accommodate any required features combination.

Even in case of a single program, by unifying similar, but also different, program structures with generic forms, we can achieve substantial simplifications leading to improved maintainability. The following simple argumentation supports this claim:

Suppose we have 10 user interface forms a_1, …, a_{10} (e.g., data entry forms). Each form, say a_i, interacts with 5 business logic functions $b_{i,1}$,…, $b_{i,5}$ (e.g., data validation rules or actions to be performed upon data entry at certain fields). We further assume that there is considerable similarity among the 10 forms, and business logic functions that each form needs.

If each form and each business logic function is implemented as a separate component, we have to manage 10 + 10*5 = 60 components and 50 interactions (situation shown in Figure 5 (a)).

Suppose we can unify business logic functions across 10 forms, so that all 10 forms use only five generic functions $B_1,\ldots, B_5$. We reduce the solution space to 10 + 5 = 15 components and 50 interactions, with the slightly added complexity of a generic representation. The interactions have become easier to understand as each form interacts with components that have been merged into five groups rather than 50 distinct components (situation shown in Figure 5 (b)).

Suppose we further unify 10 user interface forms with one generic form A. We reduce the solution space to 1 + 5 = 6 components and with five groups of interactions, each group consisting of 10 specific interactions, plus the complexity of a generic representation (situation shown in Figure 5 (c)).

Any form of validation at the level of generic representation depicted in Figure 5 (b) or (c) is bound to be much simpler and effective than validation at the level of concrete components (Figure 5 (a)). Similarly, any changes at the level of generic representation are easier to manage than at the level of concrete components due to clearer visibility of change impact, reduced risk of update anomalies, and smaller number of distinct interfaces that have to be analyzed. Finally, generic representation of components and their configurations form natural units of reuse, potential building blocks of Product Line architectures.

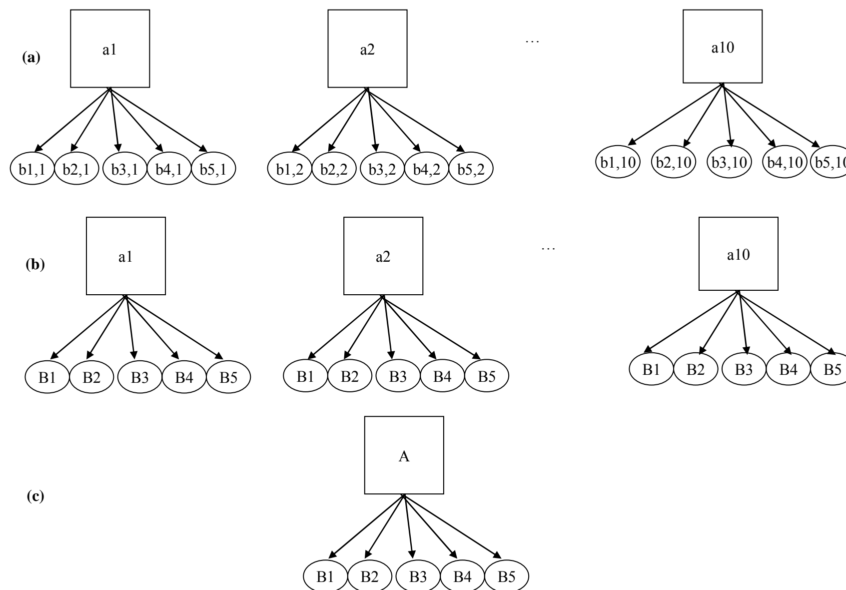

Figure 5. The impact of genericity on complexity

Generic components along with specifications of adaptation changes retain and enhance a clear view of what's similar and what's different among system releases, and help us to avoid the problem of component version explosion. We need generic change specifications to avoid repeating the same or similar change specifications all over again. This further enhances the

visibility of similarity patterns in the evolution history, simplifies change specifications and further boosts reuse of knowledge accumulated during evolution.

7.6 Conclusions of this Chapter

In this Chapter, we motivated the mixed-strategy approach and introduced its main concepts. We use a programming language to express the syntax and semantics of a program. We impose XVCL on program structures to express the syntax and semantics of change. A program along with the knowledge of evolutionary changes forms a mixed-strategy representation. Mixed-strategy achieves separation of construction-time concerns from runtime concern: We can optimize construction-time and runtime properties separately and independently of each other, without constraining each other.

With mixed-strategy, we try to achieve (1) conceptual simplicity, (2) maximum changeability of software, (3) traceability of changes from requirements to system architecture and the base code, and (4) the visibility of similarities and differences among system releases. At the heart of the approach to achieve these engineering qualities is generic design that leads to non-redundant software representation.

An x-framework is created and then evolved by specifying changes related to various releases in XVCL and by packaging components into generic x-frame structures capable of accommodating changes related to various releases. This is the essence of achieving reuse-based software evolution. Software evolution with XVCL is a continuous refinement of the x-framework.

XVCL aims at specifying changes at global and local scale, changes that have impact on system architecture, small changes at the level of program statements. The ability to express a wide range of change specification and generic design issues in a simple, practical, and uniform way was the main philosophy building the design of XVCL mechanisms. We emphasize the uniformity of solutions a lot in mixed-strategy approach. Uniformity is one of the keys to simplicity, so much required in the inherently complex world of software evolution. Therefore, we keep the change mechanisms simple, and favor uniformity of evolution pattern specifications over diversification, unless uniqueness of specifications is justified by the nature of a specific evolution pattern and/or engineering qualities to be met by its specifications.

Mixed-strategy represents commonalties and differences among releases of an evolving system, at any level of granularity, in both human- and machine-readable form. The precise understanding of commonalties and differences allows developers to make use of the knowledge of past changes to effectively implement future changes, which is the core idea behind the reuse-based evolution.

References

[1] Bassett, P. *Framing software reuse - lessons from real world*, Yourdon Press, Prentice Hall, 1997

[2] Conradi, R. and Westfechtel, B. “Version Models for Software Configuration Management,” *ACM Computing Surveys*, 30(2), 1998, pp. 232-282

[3] Cordy, J.R. “Comprehending Reality: Practical Challenges to Software Maintenance Automation”, *Proc. 11th Int. Workshop on Program Comprehension, IWPC’03*, Portland, Oregon, May 2003, pp. 196-206 (keynote)

[4] 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

[5] Fischer, M., Pinzger, M. and Gall, H. "Populating a Release Database from Version Control and Bug Tracking Systsems, " *Proc. Int. Conf. Soft. Maintenance, ICSM'03*, Sept. 2003, pp. 23-32

[6] Gall, H., Jazayeri, M. and Krajewski, J. "CVS Release History Data for Detecting Logical Couplings," *Proc. Int. Workshop on Principles of Software Evolution, IWPSE'03*, Sept. 2003, Helsinki, pp. 13-23

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

[8] 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

[9] Jarzabek, S., Basset, P., Zhang, H. and Zhang, W. "XVCL: XML-based Variant Configuration Language," *Proc. Int. Conf. on Software Engineering, ICSE'03*, IEEE Comp. Soc., May 2003, Portland, pp. 810-811, http://fxvcl.sourceforge.net

[10] 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

[11] 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.

[12] 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