EVOLUTION STYLES IN PRACTICE
Refactoring Revisited as Evolution Style

Olivier Le Goaer, Mourad Oussalah, Dalila Tamzalit
LINA, University of Nantes, My Street, Nantes, France

Djamel Serai
Department of Computing, Ecole des Mines de Douai, Douai, France

Keywords: Software Evolution, Modeling, Components, Design, Languages.

Abstract: The evolution of pure software systems remains a time-consuming and error-prone activity. But whatever the considered domain, recurring practices can be captured and reused to alleviate the subsequent amounts of effort. In this paper we propose to treat domain-specific problems-solutions pairs as first-class entities called "evolution styles". As such, an evolution style is endowed with an instantiation mechanism and can be considered at different conceptual levels. Applied on arbitrary domains, an evolution style is intended to evolve a family of applications whereas its instances evolve given applications. The evolution style's format is a component triple where each component is highly reusable. In this way, evolution styles are scalable knowledge fragments able to support large and complex evolutions, readily available to be played and replayed.

1 INTRODUCTION

To face changing desiderata, the evolution of software intensive systems is an inevitable activity. Indeed, software models must be changed very often to remain useful and meaningful. Nowadays highly reactive and interactive environments (e.g ubiquitous computing (Greenfield, 2006) and Ambient Intelligence) calls for a new approach to facilitate and replay these evolutions. This reactivity is crucial to cope with the contemporaries open systems where other ones can be plugged or unplugged, but also with the fashionable agile software development (Cockburn, 2002) which encourages many feedbacks and hence countless requirements modifications. For all these reasons, an evolution engineering emerged over the last few years to address this challenge. This promising engineering field encompass popular models transformations field, various refactoring techniques and modernization scenarios for good practices, or more generally any principle for structural or behavioral alteration of an existent software system. In this paper we argue in favor of analysis and design steps dedicated to software evolution. The key idea is to extract the evolution features from traditional software design, for strong reuse purposes. Indeed, we think that the reuse of recurring evolution scenarios as patterns is a profitable way to alleviate the amount of effort required by software maintenance activities. For that, we advocate that software evolution has to be identified, encapsulated, reused and managed. This calls for the treatment of evolution as a first-class entity, called an evolution style. Externalization and modularization of features variations and commonalities have been studied with *Traits* (Ducasse et al., 2006), which are composable units of behavior, and *Classboxes* (Bergel, 2005), which are minimal module models supporting local rebinding. Despite their added-value, these concepts are not dedicated to evolution, lack of semantics and are confined to OO systems while evolution should be challenged in a wider scope. Nevertheless, a recent and promising work is the *Changebox* concept (Nierstrasz et al., 2006), a component to scope changes over time, even if no modeling conjecture is given. On our side, an evolution style is a general-purpose concept, applied to arbitrary domains, to tame and model the evolution at a wide scope and in a uniform way. It characterizes a family of evolutions by factorizing commonalities of evolution practices and expresses the semantics of changes. A style is a fragment of knowledge specified as a component-

Le Goaer O., Oussalah M., Tamzalit D. and Serai D. (2007).
EVOLUTION STYLES IN PRACTICE - Refactoring Revisited as Evolution Style.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 138-143
DOI: 10.5220/0001341401380143

triple which can be hierarchically organized. The proposed specification has a favorable influence on other issues like evolution management, including evolution styles matching and classification, but this is not discussed in this paper.

The remainder of this paper is organized as follows: in the next section (sec. 2) we discuss the keys ideas of our proposal. Section 3 describes the modeling environment we adopted to support the evolution style approach. In section 5 we show how a well-known refactoring can be revisited as an evolution style, before we conclude in section 6.

2 STYLES FOR SOFTWARE EVOLUTION

In this section we introduce the foundation of our work. We mainly describe the benefits of the reification of evolution as a first-class entity, the modular specification format chosen and the possible organizations and capabilities of evolution styles.

2.1 Evolution as First-class Entity

According to us, the evolution features of a software system being built must be analyzed and designed separately from business features, in order to promote reuse following two complementary axis. First, our objective is to extract and modularize evolution, that is, to capture the structure, behavior and the semantics of evolution. As such, an evolution style encapsulates important design decision about evolution, as promoted by the principle of information hiding. Evolution styles factor commonalities between evolutions, just like classes do for objects in the object-oriented development. More precisely, they capture the parameters, assertions and implementation shared by several evolutions. They express semantics of refactoring and other changes. This last point is important because we are convinced that stakeholders and tools must be aware of meaning of evolution to work effectively. Thus, grouping evolutions into styles helps avoid the specification and storage of much redundant information and hence constitutes the first axis of reuse. The second axis of reuse extends the information hiding capability one step further. The key idea is to support fragmentation and extension of evolution. Practically, the composition and inheritance mechanisms encourage this kind of reuse and lead to treat evolution on a hierarchical basis.

2.2 Evolution Style Specification

The evolution style specification is motivated by the "componentization" of patterns and influenced by Knowledge-based systems (KBS) (Oussalah, 2002). The proposed three-parts specification to address evolution question is the convergence of the Context-Problem-Solution triple defined by patterns (Erich GAMMA and VLISSIDES, 1995) and the Domain-Task-PSM[1] triple defined by KBS.

Let us make a closer examination of the three components constituting an evolution style:

- Domain: evolution depends on an area of knowledge specified by a set of concepts and links among them. It corresponds to a domain model and hence defines a vocabulary through a set of types. The instantiation of a domain engenders an application able to be evolved.
- Header: evolution can be defined by a contract which stipulates the starting situation and the resulting one. It is specified by inputs/outputs parameters typed with elements of the domain, plus a set of assertions (post/pre-conditions, invariants) as constraints for the evolution achievement.
- Competence: evolution contains a body of knowledge to fulfill the evolution contract, that is, to ensure the morphing from the starting situation to the resulting one. It can be specified by a block of declarative or imperative instructions. The choice is driven by the desired abstraction level.

The three points mentioned above can be described formally and be clearly modularized as three independent but complementary components. An evolution style has a unique name, has a goal described in natural language, and is viewed as the aggregation (with the object meaning) of a domain component, a header component and optionally a competence component. This partitioning promotes (a) description reuse because the components are quite interchangeable and shareable by several styles at the same time, and (b) flexibility because the competence component can be omitted when not enough information is available. In this last case, we deal with abstract evolution styles.

2.3 Evolution Style Topologies

The evolution styles are arranged and linked to form a graph or topology. The nodes are evolution styles and the edges between two nodes represent specialization

[1] *Problem Solving Method*

or composition. As a result, the graph can be studied following a composition viewpoint or a specialization viewpoint.

Composition viewpoint. A composition hierarchy is a hierarchy of evolution styles in which an edge between a pair of nodes represents the IS-PART-OF relationship, that is, the higher node is composed of the higher level node. For a pair of evolution styles, the level style is called a *composite style* and the lower level style is a *component style*. A composite evolution style references multiple children styles. Each child can in turn reference their own children styles. A parent evolution style is dependent of its children styles. This underlines that an evolution style provides a service but may requires other ones. This property makes a distinction between independent styles called *Basic evolution styles* and dependent styles called *Complex evolution styles*. A methodology is given further in this paper to extract a basis of complex and basic evolution styles, named the evolution core.

Specialization viewpoint. A specialization hierarchy is a rooted hierarchy of evolution styles in which an edge between a pair of nodes represents the IS-KIND-OF relationship, that is, the lower node is a specialization of the higher level node. For a pair of evolution styles, the level style is called a *superstyle* of the lower level style, and the lower level style is *substyle* of the higher level class. The three components (Domain, Header, Competence) specified for a style are inherited by all its substyles. Hence, a substyle may have:

- a more specific domain: using the concepts and concept-links provided by a more specific domain.
- a more specific header: adding or redefining parameters and constraints.
- a more specific competence: redefining the implementation block in case of a code-level representation, or adding new rules or states in case of a rule-based or state-based representation.

The single root of the specialization hierarchy is a system-defined abstract evolution style named EVOLUTION whose wide semantics depicts all evolution practices supported on the considered domain. There is also two system-defined abstract evolution substyles named COMPLEX and BASIC, as a standardized way to explicitly separate dependent and independent styles.

2.4 Evolution Core

To obtain the most fine-grained and useful evolution styles, we propose a methodology which says to project generic operators on relevant entities of the considered domain. The resulting set of evolution styles, called the evolution core, is enough to sustain small but relevant evolution practices. In table 1 we have identified recurring operators encompassing change activities during the software maintenance phase, reconsidering a work on class characteristic migrations (Castellani et al., 2001). Two kinds of operators are mentioned: simple operators and advanced operators. The latters need simple operators to be functional. Consequently, the styles built from projection of simple operators are basic evolution styles while the styles built from projection of advanced operators are complex evolution styles. Therefore, after an exhaustive analysis, for each domain, a set of basic and complex evolution styles has to be released or extracted following this methodology.

Table 1: Simple and Advanced operators for evolution.

NAME	DESCRIPTION
Simple	
`Add`	Connect an entity to another one
`Remove`	Disconnect an entity from another one
`Modify`	Alter an entity's property
Advanced	
`Transfer`	Cut/Paste some entities from a location to another
`Clone`	Copy/Paste some entities from a location to another
`Merge`	Merge two entities into a single one
`Split`	Split a monolithic entity into several ones

Let us consider the Java Language for a quick exemplification. Thus, starting from a Java domain metamodel and the previous table, we can construct basic evolution styles such as `AddClass(Class super, Class sub)`, to insert a new Java Class in a hierarchy, or `ModifyClassName(Class super, String newName)` to rename an existing Java Class.

3 EVOLUTION STYLE MODELING ENVIRONMENT

A strong abstraction purpose emerged in the last ten years to help people to apprehend the variety, the complexity and the synergies of software artifacts. The same demand subsists to characterize the evolution in a global way. Therefore, the obscure or tacit

evolution activities should be reinterpreted as a clear and coherent set of models.

4 LAYERED ARCHITECTURE

The modeling environment we work with is a layered architecture of models, separated by an instantiation relationship. This architecture exhibits three distinct layers (or levels) and distinct concerns. We respect the modeling attitude envisioned by the OMG's modeling stack.

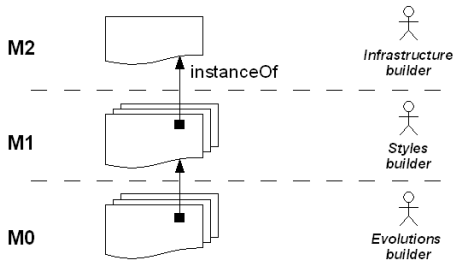

Figure 1: Evolution style modeling stack.

The $M2$, $M1$ and $M0$ levels are separated by an ontological instantiation (`instanceOf`). In this way, an element of a M_i level is created from its definition of the M_{i+1} level. According to this architecture, runtime evolutions (i.e evolution styles instances) are located at $M0$, evolution styles are located at $M1$ and the evolution style language is located at $M2$. In addition, the layers reveal three distinct stakeholders with their own duties. The infrastructure builder intervenes at the $M2$ level, the styles builder intervenes at the $M1$ level and lastly the evolutions builder intervenes at the $M0$ level. This cutting of responsibilities tends to share the global evolution effort by separating the concerns and skills.

4.1 Modeling Levels

Each level introduces particular conceptual abstractions and is intended to particular modeling purpose. Les us make a closer but summarized examination of the $M2$, $M1$ and $M0$ levels.

Model at level 2. Model at the level 2 is called a meta-model, that is, a model defining other models. The infrastructure builder provides minimal but necessary syntactic and semantic constructions allowing experts to define evolution styles. The meta-model is not given in this paper but can be summarized as follows:

- an evolution is an aggregation of a domain component, a header component and a optionally a competence component. The proposed theoretical meta-model explicits various kind of relationship between components, usually implicit.
- the meta-model is self-defined, that is, is a domain and an application at the same time. An evolution core can be extracted from the meta-model and the resulting evolution styles can be instantiated to evolve the meta-model itself.

Model at level 1. The need for domain-specific evolution libraries is the same than for API's in the programming language field or components in large-scale development field: providing readily available software units in order to alleviate the amount of efforts. Styles builders (experts) describe evolution styles by instantiating the elements of the $M2$ level provided by the infrastructure builder. For each domain, a topology of styles is built incrementally, packaged and released as an evolution library. According to the $M2$ level, a topology is also a domain. The immediate result is that (a) evolution libraries are viewed as components and hence are more easily distributable and subsequently integrable, and (b) evolution libraries can be evolved by instances of evolution styles expressed at the $M2$ level. In other words, evolution libraries are built and maintained by evolution styles.

Model at level 0. The $M0$ level depicts the instance of evolution style to evolve applications. The instantiation of a style is performed by the instantiation of its three components. Indeed, the instantiation of a domain component engenders an application, the instantiation of a header component engenders valued parameters and constraints checking while the instantiation of a competence component engenders the behavior execution. Note that for this reason, an abstract evolution style cannot be instantiated.

5 REFACTORING SCENARIO

To illustrate our approach, we consider an object-oriented system in which we attempt to restructure some portion of code. This section reinterprets a well-known refactoring example as an evolution style. Although we present a small evolution style in this section, the perspective still remains to build large evolution styles by incremental composition of existing ones.

5.1 What is Refactoring?

According to Martin Fowler (Fowler, 1999), refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.

5.2 Extract Method Refactoring

The Extract Method Refactoring consists into taking a fragment of code inside a subroutine and turning it into its own routine. The Object-oriented systems are particularly prone to such a refactoring activity. The Java source code below, inspired from Martin Fowler's Web site's examples[2], clarifies the before&after treatment.

```
void printOwing() {
        printBanner();

        //print details
        _amount = _amount * 1.5;
        System.out.println ("name:  " + _name);
        System.out.println ("amount " + _amount);
    }

----------------------------------------
  AFTER THE EXTRACT METHOD REFACTORING
----------------------------------------

    void printOwing() {
        printBanner();
        printDetails();
    }

    void printDetails () {
        _amount = _amount * 1.5;
        System.out.println ("name:  " + _name);
        System.out.println ("amount " + _amount);
}
```

This simple modification improves the understandability and the reusability of code to make it easier for human maintenance in the future. Likewise, the Extract Method refactoring provides a more natural pointcut that can be used with AOP (Binkley et al., 2005).

[2]http://www.martinfowler.com/

5.3 Evolution's Abstraction

The purpose is to provide a style-based model for extract method refactoring on object-oriented systems. Thus, the considered domain is the object-oriented paradigm here. As an example, the domain is assumed to be FAMIX (Demeyer et al., 1999), a common exchange model which provides a language-independent representation of object-oriented source code. The core presented in figure 2 is a simplistic version of FAMIX but adequate here.

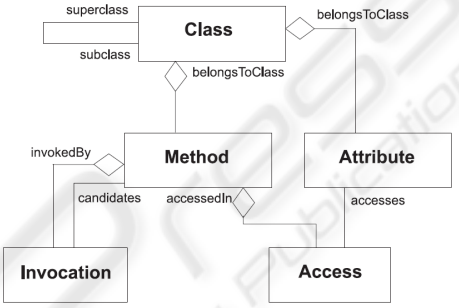

Figure 2: Core of FAMIX model.

At first glance, the Extract Method pattern can be summarized with a small set of changes:

1. Addition of a new method in the considered class;
2. Transfer of the considered code (here 1 write access plus 2 invocations) from the initial method to the new one;
3. Addition of an invocation to the new method in the initial method;

The aforesaid changes are matching with some basic evolution styles obtained following the methodology introduced in section 2.4. More precisely, they are excerpts of projections of the simple operators `Add` and `Transfer` on the FAMIX model. Next, the complex evolution style named "ExtractMethod" is composed from the four distinct basic evolution styles presented in the table 2. The topology represented in figure 3 is a model for style-based Extract Method refactoring, using the UML and OCL notations. The four basic evolution styles discussed above and the complex ExtractMethod evolution style are added into the specialization hierarchy, starting from the three standardized abstract evolution styles. In this model we also decided to add a supplementary abstract style to factorize the common parameters for the transfer from a method to another one. We have also extended the initial ExtractMethod style to build the LazyExtractMethod style. In this latter, a new precondition is

added to avoid to execute the extract method if there is less than four instructions (accesses or invocations).

Table 2: Basic Evolution styles required.

STYLE	OP	ENTITIES
AddMethod	`Add`	`Class x Method`
TransferAccess	`Transfer`	`Method x Method x Access`
TransferInvocation	`Transfer`	`Method x Method x Invocation`
AddInvocation	`Add`	`Method x Invocation`

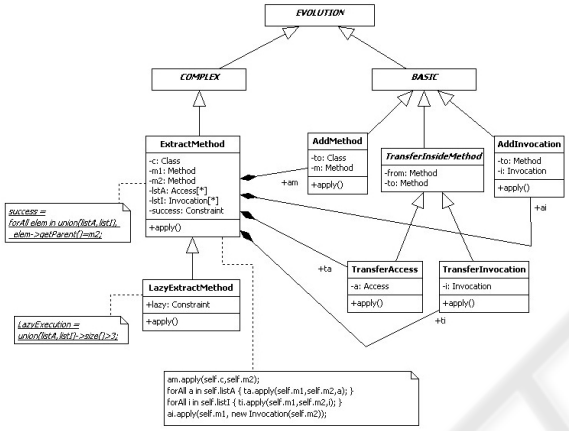

Figure 3: Style-based Extract Method refactoring.

The orchestration of the composition is expressed into the competence component of the ExtractMethod style. In the event, the scheduling of the basic styles' invocations is important to guaranty the scenario to succeed. This behavioral aspect of evolution style can be represented in various way. The schedule can be given in a more abstract way by state models, e.g. finite state machines or Petri nets, just as with rules, or as detailed as on code level like in the given example. Each representation has its own advantages and disadvantages, and its choice is only driven by the style builder needs. To overcome this unreliable thing, we suggest to externalize the behavioral content and hence to locate and use it just like a resource.

6 CONCLUSION

In our opinion, providing readily available models of evolution is a sound way to sustain a mass evolution. Evolution styles offer re-use benefits, guidance benefits and communication benefits for the evolution engineering field. In this paper, we explained that evolution must be exhibited via styles for meaning and reuse purposes, independently of the technology. They are mechanisms to push changes through a system in a controlled way. In addition, we have shown how a well-known refactoring can be revisited as a complex evolution style, built and released by a style builder, and able to be instantiated on-demand by evolution builders. The refactoring scenario illustrated the organizational capabilities of our meta-model and the methodology we provided, using both top-down approach for style specialization and bottom-up approach for style composition.

REFERENCES

Bergel, A. (2005). *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Berne.

Binkley, D., Ceccato, M., Harman, M., and Tonella, P. (2005). Automated pointcut extraction. In *Proceedings of the workshop on Linking Aspect Technology and Evolution workshop*.

Castellani, X., Jiang, H., and Billionnet, A. (2001). Method for the analysis and design of class characteristic migrations during object system evolution. *Inf. Syst.*, 26(4):237–257.

Cockburn, A. (2002). *Agile software development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Demeyer, S., Tichelaar, S., and Steyaert, P. (1999). FAMIX 2.0 - the FAMOOS information exchange model. Technical report.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388.

Erich GAMMA, Richard HELM, R. J. and VLISSIDES, J. (1995). *Design Patterns : Elements of Reusable Object-Oriented Softwares*. Addison Wesley.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Greenfield, A. (2006). *Everyware : The Dawning Age of Ubiquitous Computing*. New Riders Press.

Nierstrasz, O., Denker, M., Gîrba, T., and Lienhard, A. (2006). Analyzing, capturing and taming software change. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*.

Oussalah, M. (2002). Component-oriented kbs. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 73–76, New York, NY, USA. ACM Press.