Electronic Communications of the EASST
Volume X (2007)

Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007)

Evolution styles: change patterns for Software Evolution

Olivier Le Goaer and Peter Ebraert

10 pages

Guest Editors: Tom Mens, Maja D'Hondt, Kim Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Evolution styles: change patterns for Software Evolution

Olivier Le Goaer[1] **and Peter Ebraert**[2]

[1] olivier.le-goaer@univ-nantes.fr, http://lina.atlanstic.net/
Laboratoire Informatique de Nantes Atlantique
University of Nantes
2 rue de la houssiniere
F-44000 Nantes, France

[2] Peter.Ebraert@vub.ac.be, http://prog.vub.ac.be/
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium

Abstract: Patterns have been proved useful in many problem domains. In the domain of software evolution, only behaviour-preserving patterns (e.g. refactorings) have ever been proposed. This paper proposes to broaden the scope of change patterns by means of a reification of any evolution efforts into styles. We define an evolution style as a first-class entity which is specified once and can be applied many times. Evolution styles allow the specification of (non) behaviour-preserving change patterns. We exemplify the use of the evolution style concept by means of two applications which evolve in a style-based way.

Keywords: Evolution styles, Change patterns

1 Introduction

It was Christopher Alexander, who first introduced the idea of capturing design ideas as patterns [AIS77]. Being an architect, he was constantly confronted with similar problems concerning the design of buildings and cities. For not having to re-solve similar problems over and over again, he came up with patterns, which recorded the design decisions taken by *many* builders in *many* places over *many* years in order to resolve a particular problem. In the late eighties, patterns were introduced in the field of software design [BC87]. Design patterns for example, gained popularity in computer science after the book Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994 by Gamma et al [GHJV94]. They defined a design pattern as a general repeatable solution to a commonly occurring problem in software design.

Two decades ago, Opdyke introduced patterns in the domain of software evolution [Opd92]. A few years later, Fowler took up on that track and defined a "pattern" as an idea that has been useful in one practical context and will probably be useful in others [Fow97]. Two years later, Fowler presented a catalog of refactorings, which are patterns of change to a computer program which improve its readability or simplifies its structure without changing its results [Fow99].

Changing requirements (e.g. the need to introduce secure transactions) make the evolution of

computer programs inevitable. It has early been stressed that programs continuously need to change to remain useful [LB85]. Some types of changes often show up in software engineering: introducing privacy, transactions, security, logging, etc. [CHK+01]. Just like Alexander, Gamma and Fowler, we think that those recurring problems can be solved by a dedicated solution. We define a *change pattern* as a general repeatable solution to a commonly occurring problem in software evolution. Change patterns are more general than refactorings, as they also allow expressing changes which alter the outcome of an application. As such, refactorings *are* change patterns, while change patterns *are not* refactorings.

The rest of this paper is structured as follows. Section 2 illustrates how two very different application architectures are facing changing requirements which impose similar changes on the architectures. Section 3 presents the evolution style concept as a mean to specify those changes as change patterns. In Section 4 we show that those evolution styles can be applied on different applications to cope with their changing requirements. Finally, we stipulate the concluding remarks and the tracks of future work in Section 6.

2 Problem statement

This section first introduces a banking application in Java with changing requirements. We specify an evolution scenario which is applied on the banking application to cope with its new requirements. We then introduce another application – a chat room – which is developed in Smalltalk, and which is also a subject of changing requirements. We show that, while both applications are very different, the changes they are undergoing are very related. This illustrates the need for change patterns, which are specified only once and can be applied many times.

2.1 Banking application and its evolution

Assume the information system which manages the services of a bank depicted in Figure 1. This application is implemented in Java. The `Cashier` is an employee of a `Bank` who performs a `Cashier Transaction`. He can transfer, deposit and withdraw money from the `Account` of the `Customer`.

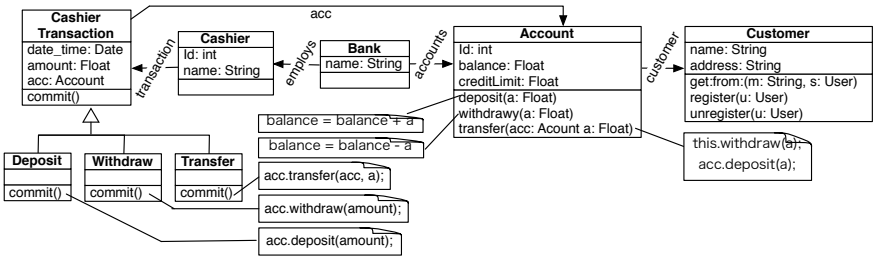

Figure 1: A Java banking architecture

The banking application of Figure 1 only includes one kind of account. Changing demands on the market, however, impose the banking system to model different kinds of accounts. Saving

accounts get a higher interest rate, but are limited in the transactions they allow. Deposit accounts allow all kinds of transactions, but provide less interest. Next to that, money transfers can be sent over a network while the privacy of the users must still be guaranteed.

To cope with the first requirement, we introduce different kinds of accounts by subclassing the `Account` class with a `SavingAccount` class and a `DepositAccount` class. Next, the `transferMoney` method is pushed down to the `DepositAccounts` class, so that it cannot be called from a `SavingAccount`. These changes are annotated in red in Figure 2.

To ensure the customer's privacy, the amounts sent over the network are encrypted. This is done by adding an `encrypt` method to the `CashierTransaction` class and by invoking it when committing the transaction (in the `commit` method). When the message is received by the `transfer` method from the `Account` class, it needs to be decrypted by calling the `decrypt` method, which was added to that class. These changes are annotated in green in Figure 2.

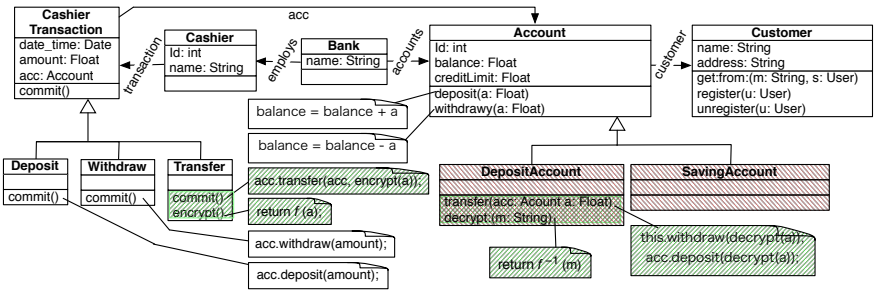

Figure 2: The banking architecture after evolution

2.2 Chat application and its evolution

Consider the chat application depicted in Figure 3. This application is designed conform to the class-based objected-oriented meta-architecture of Smalltalk. It originally consists of two classes, `User` and `Chatroom`, which respectively maintain a reference `cr` and `users` to one another. A user can subscribe to a chatroom using the `register` method and exchange text messages with the rest of users of the chatroom using the `send` and `receive` methods. Text messages sent to the chatroom are propagated to all the registered users.

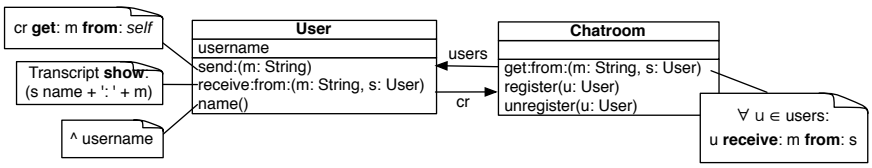

Figure 3: A Smalltalk chat architecture

Assume now we need to differentiate between registered and guest users. For doing so, we

add two subclasses of `User` to the application program, `RegisteredUser` and `Guest`.The difference between the two types of users is that the registered users can be identified by their name in the chat room whereas the guests cannot: Accordingly, the `username` attribute of `User` class is pushed down to the `RegisteredUser` class. Figure 4 shows this first feature added to the application (in red).

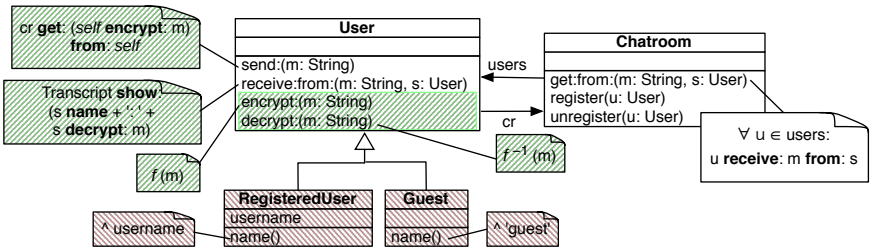

Figure 4: The chat architecture after evolution

The second change incorporates to ensure the privacy of the users, which in this case corresponds to encrypting and decrypting the messages when they are sent and received respectively. In order to match these requirements, the architect needs to adds two methods to the `User` class, `encrypt` and `decrypt`, which are called from within the `send` and `receive` methods respectively. Figure 4 shows this second feature added to the application (in green).

2.3 The need for evolution reuse

Both applications are different: They are used in different areas (banking and communication) and are even implemented in different programming languages (Java and Smalltalk). Despite of their different nature, both the banking and the chat applications have been evolving in the same way. First, two subclasses were introduced. Second, a method is pushed down to one subclass. Third, a complex pattern for introducing encryption and decryption is applied to each application.

The empirical observation of recurring change patterns shows the possibility to capture those patterns as evolution practices. Refactorings were already proposed as best practices for improving the structure of applications [Fow99]. Refactorings, however, are not general enough, as they only cover change patterns which do not change the observable behaviour of the software system [Opd92]. Behaviour-changing change patterns (e.g. introducing encryption) are not covered by refactorings. That is why we propose to generalize the idea of change patterns for Software Evolution in *evolution styles*, which is elaborated on in the following section.

3 Evolution styles

The evolution style concept is a domain-specific specification of a solution for a recurring problem in software evolution. Styles specify a solution which can be applied over and over again

and enhance the understandability of the software evolution by raising the level of abstraction of undertaken changes. This section briefly describes the specification format of an evolution style. For a more detailed explanation, we refer the reader to [SOTG06].

3.1 Specification

An evolution style is a first-class entity that can be referenced, queried and passed along. The core of the meta-model of the evolution-styles is shown in Figure 5. Every `evolution style` class possesses a `name` (allowing a communication about the different styles at a more abstract level) and a `goal` (providing a textual explanation of the purpose of the evolution style). The `Domain` of an evolution style corresponds to the meta-architecture on which that evolution style is expressed. Its `Header` is a set of input/output parameters which are expressed on the building blocks of the domain or the application. They can be specified as pre- and postconditions of the evolution style. A `Competence` represents the sequence of changes that must be applied when the evolution style is invoked. These changes are expressed on the building blocks of the domain or the application.

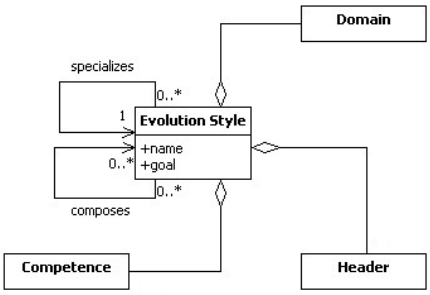

Figure 5: Core meta-model of the Evolution Style

Developers can `specialize` and `compose` evolution styles. Specialization supports a white-box reuse of existing specification, allowing a *sub-style* to supplement or redefine informations of a *super-style*. Composition supports a black-box reuse, allowing a *complex style* to delegate changes to other styles, until *basic styles*. The next section provides three examples of evolution styles which are specified in the a domain based on the [DTS99] meta-model.

3.2 Examples

Every evolution style is specified in a certain domain. We use Famix as the common domain for the three evolution style examples which are about to follow. Famix provides a language-independent model for class-based object-oriented source code. Both Smalltalk and Java programming languages adhere to Famix, whose core is depicted in Figure 6.

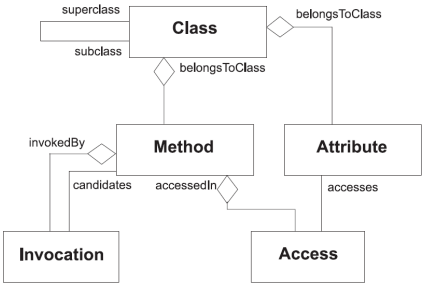

Figure 6: The FAMIX core [Based on [DTS99]]

Introducing a subclass – The basic `AddSubClass` evolution style captures the simple change pattern for introducing a subclass of a class. It is specified as follows:

```
Name          AddSubClass
Goal          Add a new class as a specialization of an existing superclass.
Domain        Famix
Header        Parameters {Class super, String name}
              Constraints {
                  pre: not existsClass(c, name, ?superClass)
                  pre: existsClass(super, ?name, ?superClass)
                  post: existsClass(c, name, super)
              }
Competence    c := Class new: name
              c superclass: super
```

The parameters of this style are typed with building-blocks of the Famix meta-model. The constraints are specified in a logic programming style. The precondition `not existsClass(c, name, ?superClass)` is a clause which verifies that there does not exist any class `c` with the name `name` with any superclass in the system. The postcondition `existsClass(c, name, super)` states that after applying the evolution style, the system contains a class `c` named `name` with a superclass `super`. A logic engine can be used to unify those variables to the values of the styles' parameters. Note that the competence of the pattern is specified in Smalltalk, but only uses concepts of the Famix domain. It sends a `new:` message to the `class` class in order to create a new class with `name` as its name. Afterwards, it sets the superclass of the new class to super.

Shifting down a method – In order to specify a reusable evolution style, we propose a three-step specification for this change pattern. First, we specify general two basic evolution styles intended to add (or delete) a method `m` to a class `target`. The following code snippet show the style for adding a method to a class. The equivalent style `RemoveMethod` which removes a method from a class is omitted for the sake of brevity.

```
Name          AddMethod
Goal          Add a method to a class
Domain        Famix
Header        Parameters {Method m, Class target}
              Constraints {
                  pre: existsClass(target, ?name, ?superClass)
                  pre: not existsMethod(m, target)
                  post: existsMethod(m, target)
              }
Competence    target addMethod: m
```

Secondly, we specify a `TransferMethod` style as a composition of the `RemoveMethod` and the `AddMethod` styles. A `TransferMethod` verifies that all classes exist and that `m` is a method of class `source`. The competence of this style applies the `RemoveMethod` style to remove the method from `source` and applies the `AddMethod` style for every class in the target enumeration to add `m` to each one of them. The following snippet shows the specification of the style:

```
Name          TransferMethod
Goal          Cut/paste a method from a class to some other classes
Domain        Famix
Header        Parameters {Method m, Class source, Class[] targets}
              Constraints {
                  pre: existsClass(source, ?name, source)
                  pre: existsClass(tar, ?name, source), member(tar, targets)
                  pre: existsMethod(m, source)
              }
Competence    RemoveMethod(m, source) apply
              target do[:c ! AddMethod(m, c) apply]
```

Thirdly, we specialize the `TransferMethod` style to capture the *push down method* refactoring[1]. The `PushDownMethod` style supplements the inherited specification of its super-style with an extra precondition to ensure that the target classes are subclasses of the source class, and redefines the goal for a more precise semantics. Its competence consists of just invoking the competence of its super-style.

```
Name          PushDownMethod
Goal          Push down a method from a superclass to some of its subclasses
Header         Constraints {
                  pre: forall elementOf(c, enumeration), existsClass(c, ?name, source)
              }
Competence    super apply
```

Introducing Privacy – Finally, we propose a complex evolution style which encapsulates the pattern that introduces privacy in an application adhering to the Famix meta-model. This style adds an encryption method `encM` to a class `senderC` which is able to encrypt a parameter. It then surrounds the invocation of a method `receiverM` in the `senderM` with an invocation of the added `encM` method. On the receiver class `receiverC` it adds a `decM` method which is able to decrypt a parameter and adds an invocation to it inside the `receiverM` method. The `IntroduceMessagePrivacy` style is composed of three styles: the `AddMethod` style, the `EncapsulateInvocation` style and the `EncapsulateParamater` style.

```
Name          IntroduceMessagePrivacy
Goal          Ensure privacy of exchanged messages between a sender and a receiver
Domain        Famix
Header        Parameters {Class senderC, Class receiverC, Method senderM, Method receiverM, FormalParameter par}
              Constraints {
                  pre: existsClass(senderC, ?name, ?superClass)
                  pre: existsClass(receiverC, ?name, ?superClass)
                  pre: existsMethod(senderM, senderC)
                  pre: existsMethod(receiverM, receiverC)
                  pre: existsInvocation(senderM, receiverM, par)
                  post: existsMethod(encM, senderC)
                  post: existsMethod(decM, receiverC)
                  post: existsInvocation(senderM, encM, par)
                  post: existsInvocation(receiverM, decM, par)
              }
Competence
              Method encM := Method new: "encrypt:" parameter: "m" body: "f(m)"
              AddMethod(encM, senderC) apply
              Method decM := Method new: "decrypt:" parameter: "m" body: "f-1(m)";
              AddMethod(decM, receiverC) apply
              EncapsulateInvocation: par of: receiverM with: encM in: senderM
              EncapsulateParamater: par with: decM in: receiverM
```

[1] http://www.refactoring.com/catalog/pushDownMethod.html

4 Validation

In this section, we validate the claim that evolution styles encapsulate change patterns which are specified once and which can be applied many times. In order to do that, we go back to the banking and chat applications which were explained in Section 2 and show that the three evolution styles which were specified in Section 3 can be *applied* on both applications in order to obtain the required application design.

Applying an evolution style consists in parameterizing it with the actual building blocks of the considered architecture, verifying the header's preconditions and executing the competence. In case the preconditions cannot be verified, the style cannot be applied. The following sections respectively describe how the evolution styles are applied on the banking and chat applications.

4.1 Style-based evolution of the banking application

Reconsider the banking application from Section 2.1. We now evolve that application by means of evolution styles. The following code specifies the style-based evolution steps of the evolving banking application. The evolution styles are invoked (like functions) with their parameters being the building blocks of the banking architecture. Applying this change sequence on the application, makes it end up in the architecture which is depicted in Figure 2.

```
AddSubClass(Account, "DepositAccount") apply
AddSubClass(Account, "SavingAccount") apply
PushDownMethod(transfer, Account, {DepositAccount}) apply
IntroduceMessagePrivacy(Transfer, commit, DepositAccount, transfer) apply
```

4.2 Style-based evolution of the chat application

Reconsider the Chat application depicted in Figure 3. The following code snippet shows the evolution of that system, specified in an evolution style way. In comparison with the banking case there is an additional invocation of a `ModifyMethod` style, which alters the definition of the `name` method of the `Guest` class to make it return a 'guest' string whenever invoked.

```
AddSubClass(User, "RegisteredUser") apply
AddSubClass(User, "Guest") apply
PushDownMethod(name, User, {RegisteredUser, Guest}) apply
ModifyMethod(name, Guest, "^ 'guest'") apply
IntroduceMessagePrivacy(User, send, User, receive) apply
```

4.3 Evaluation

Evolution styles specify a pattern of changes which can be applied to solve a recurring problem in software evolution. Defining the styles is the subject of a difficult paradox. On the one hand, the evolution style must be specific enough to encapsulate a solution for a well-defined problem. On the other hand, the style must be as general as possible so it can be applied in many different situations. This puts a burdan on the person who defines evolution styles.

Evolution style specifications include pre- and postconditions. While the preconditions help to verify that the current application is in the right state to apply the evolution style, the post-conditions provide information on what the state of the application will be after applying the style. This information can be used to support the definition of *valid* sequences of evolution style

invocations [KK04]. As such, evolution styles contribute to ensure application and evolution consistency.

5 Related work

We distinguish between related work on patterns *of* evolution, and patterns *for* evolution. The first kind attempts to extract commonalities in software evolution activities in order to improve the understanding of evolution. In [NYN+02], the authors outlined recurring working force schemas and recurring steps that are followed in the development of open-source systems. Another interesting work to that regard is [BKS03], in which the authors consider changes as a phenomenon. They define life-cycle volatility vectors which allow to classify software applications based on the patterns in the changes which they undergo.

The second kind of work tends to help developers by providing recurring practices for various purposes. Refactorings [Fow99] gained popularity and are now embedded in more and more tools. The next step is to integrate patterns for evolution as a natural part in the way software is developed. In order to do that, change classes [Dep07] and change boxes [Zum07] were presented.

6 Conclusion and future work

Patterns are used to specify a solution for recurring problems in their domain and have been proven useful in many different domains such as civil architecture, software design and software evolution. In the software evolution domain, however, only limited use has been made of the pattern principle as only behaviour-preserving patterns (e.g. refactorings) have been asserted. Consequently, developers and architects have been forced to resolve the recurring problems concerning behavioural change.

This paper shows that even very different applications (specified in different programming languages and in different problem domains) can be the subject of similar change scenario's which require a similar adaptation of the application behaviour. We define an evolution style as a domain-specific specification of a solution for a recurring problem in software evolution. An evolution style has a name, a goal and consist of three major parts: (1) the domain – denoting the meta-architecture, (2) the header – describing the style's interface and (3) the competence – specifying the actions which have to be applied in order to reach the style's goal.

A catalog of different evolution styles should be defined in order to grasp the recurring problems in software evolution. Providing styles for all problems, however, is not possible. Consequently, our major track of future work consists in the development of an extensible catalog of evolution styles, which can be queried for patterns.

Acknowledgements: We thank the *Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)* for providing Peter Ebraert with a doctoral scholarship to finance this research. We acknowledge *Dalila Tamzalit*, *Ellen Van Paesschen*, *Pascal Costanza*, *Djamel Seriai*, *Theo D'Hondt* and *Mourad Oussalah* for reviewing, commenting and supporting this research.

Bibliography

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[BC87] K. Beck, W. Cunningham. Using pattern languages for object-oriented programs. Technical report CR-87-43, Tektronix Inc., September 1987.

[BKS03] E. J. Barry, C. F. Kemerer, S. A. Slaughter. On the uniformity of software evolution patterns. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Pp. 106–113. IEEE Computer Society, Washington, DC, USA, 2003.

[CHK+01] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance* 13(1):3–30, 2001.

[Dep07] B. Depoortere. Reasoning about first-class changes for support in software evolution. Master's thesis, Vrije University, 2007.

[DTS99] S. Demeyer, S. Tichelaar, P. Steyaert. FAMIX 2.0 - The FAMOOS Information Exchange Model. Technical report, University of Berne, 1999.

[Fow97] M. Fowler. *Analysis Patterns – Reusable Object Models*. Addison Wesley, 1997.

[Fow99] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Martin Folwer, 1999.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[KK04] G. Kniesel, H. Koch. Static composition of refactorings. *Sci. Comput. Program.* 52(1-3):9–51, 2004.

[LB85] M. M. Lehman, L. A. Belady (eds.). *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[NYN+02] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, Y. Ye. Evolution patterns of open-source software systems and communities. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*. Pp. 76–85. ACM Press, New York, NY, USA, 2002.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Phd thesis, University of Illinois at Urbana Champaign, 1992.

[SOTG06] A. Seriai, M. C. Oussalah, D. Tamzalit, O. L. Goaer. A reuse-driven approach to update component-based software architectures. In *IEEE IRI: Information Reuse and Integration*. Pp. 313–318. 2006.

[Zum07] P. Zumkehr. Changeboxes — Modeling Change as a First-Class Entity. Master's thesis, University of Bern, 2007.