Received: 13 January 2018 | Accepted: 29 January 2018
DOI: 10.1002/smr.1949

PERSPECTIVE PAPER

WILEY Journal of Software: Evolution and Process

Five recommendations for software evolvability

Václav Rajlich

Department of Computer Science, Wayne State University, Detroit, MI, U.S.A.

Correspondence
Václav Rajlich, Department of Computer Science, Wayne State University, Detroit, MI 48202, U.S.A.
Email: rajlich@wayne.edu

Abstract

Evolvability of software lies in intersection of 3 factors: evolving system properties, human factors present in the developer team, and evolution demands. The paper presents 5 recommendations that enhance software evolvability: defined processes of software change, distinction between evolving and stabilized part of the code, analyzable code segments, significant concept encapsulation, and avoidance of wrapping.

KEYWORDS

analyzability, concept encapsulation, evolvability, evolving and stabilized code, software change process, wrapping

1 | INTRODUCTION

Evolutionary software development (ESD) is a family of processes where developers add new features or properties to already existing—albeit incomplete—software. The overwhelming majority of software is developed by ESD, and several ESD processes have been identified, including iterative, agile, directed, solo, exploratory, safeguarded, open-source, and inner source processes.[1]

Evolutionary software development processes however can be used only when software is evolvable, ie, when it allows the addition of new features or new properties. Evolvability is therefore an important software quality. If it is lost, software cannot be further evolved and the project faces a major challenge or even failure. Because of its importance, evolvability motivates a large part of software engineering research and practice.

We observe that evolvability of a software system is the interplay of 3 fundamental factors: *system properties*, *human factors* present in the developer team, and *evolution demands*. Evolution is possible only when these 3 fundamental factors are properly aligned, ie, specific system properties allow the specific developer team to satisfy the specific demands of evolution. Evolution may be impossible whenever that alignment is missing, for example, when the developer team is unable to deal with the idiosyncrasies of the software system or the evolution demands are excessive. On the other hand, some highly capable teams with large resources are capable to evolve software of any quality, as large software companies have repeatedly demonstrated.

The problem of software evolvability requires study of all 3 factors and their interplay and is outside of the scope of this paper. Instead, this paper focuses on 5 select recommendations about how to preserve or improve software evolvability.

These recommendations are directed towards software written in object-oriented languages. Examples of such languages are C++, Java, C#, Python, and so forth. Object-oriented software system written in these languages consist of *units* (files, classes, functions, and so forth) that interact through *dependencies*. Architecture patterns are recognizable and repeatable configurations of units and their dependencies. They occupy a middle granularity between software architectures on one hand and coding practices that are used within the units on the other hand. *Good patterns (patterns)* facilitate evolvability, while *antipatterns (bad smells and technical debt)* hinder it.

From the point of view of human factors, software has to be comprehensible to software developers. Also, the developers should use *good development practices (practices)* that improve evolvability and avoid *malpractices* that harm evolvability. All 5 recommendations of this paper deal with patterns and/or practices.

Evolution demands are expressed as a product backlog consisting of change requests to be satisfied. The nature of this product backlog also impacts what are acceptable patterns and practices.

As a word of caution, evolvability should not be confused with maintainability. Both evolution and maintenance consist of repeated software changes, and classical taxonomy characterizes software changes as perfective, corrective, and adaptive.[2] Evolution involves perfective software changes that add new functionalities or properties and are absent during maintenance.[1] Therefore, evolvability is a more demanding criterion than

Copyright © 2018 John Wiley & Sons, Ltd.

maintainability. While evolvable software is always maintainable, the opposite may not be true. Also, software that is not maintainable is also not evolvable, while nonevolvable software still may be maintainable.

Section 2 summarizes previous relevant work. Sections 3 to 7 deal with the 5 specific recommendations that preserve or enhance evolvability and are the core of the paper. Section 8 contains conclusions.

2 | PREVIOUS WORK

Most of the evolvability literature deals with software properties and touches only indirectly on human factors and evolution demands. For example, Breivold et al[3] characterize evolvability as a compound software property, consisting of analyzability, integrity, changeability, extensibility, portability, testability, and additional domain-specific attributes.

Some earlier literature and standards conflate evolvability and maintainability. An international standard characterizes evolvability, called mistakenly "maintainability," as a compound property that consists of modularity, reusability, analyzability, modifiability, and testability.[4]

There is a sizeable literature that addresses *software craftsmanship* that deals with the practical evolvability issues. An example is the book by Martin that recommends numerous patterns and practices that enhance evolvability and warns against certain antipatterns and malpractices.[5]

One of the evolvability issues that was addressed extensively by both researchers and practitioners is *comprehensibility* of software. The developers need to comprehend software so that they can correctly modify it. Several surveys of comprehension field are available.[6-8] The most desirable comprehensibility is accomplished when the code units correspond to the domain concepts.[9]

A frequent question that developers ask during evolution is as follows: What does this specific unit do, or in other words, what is its responsibility? The responsibility of unit X can be expressed as a *contract* between a unit X and its clients. The contract specifies the bounds within which the clients can make the requests to X; these bounds are described by the *precondition*. If the precondition is satisfied, unit X takes the expected action and produces results that are described by the *postcondition*. The couple <precondition, postcondition> is the contract.[10]

Mouchawrab et al[11] and many other publications explore software *testability*. One of the testing strategies, *unit testing*, verifies correctness of unit contracts, and many processes of software evolution rely on unit testing for verification of the correctness of the code.[12] Feathers[13] lists numerous antipatterns that limit unit testability and introduces practices that mitigate these limits.

Refactorings are modifications of the code that improve code structure and comprehensibility while preserving the functionality. They are an indispensable part of all evolution processes as accumulated impact of code modifications often leads to deterioration of that structure. Several surveys of refactoring field are available.[14-16] Refactorings are often presented as remedies for *code smells*; code smells are antipatterns that make software less evolvable.[17]

One of the widely explored antipatterns that complicate software evolution is code *clones*, ie, duplicates of the code. Several surveys of this field are available.[18,19]

Analyzability is considered by many as an indispensable component of evolvability.[3,4] The most common analysis tasks related to software evolution are variants of call graph constructions[20] and slicing.[21]

3 | DEFINED PROCESSES OF SOFTWARE CHANGE

This section deals with the first of the 5 recommendations. We note that adoption of formally defined process models in other contexts has demonstrably led to improvements in software productivity and quality.[22] Based on that, we speculate that an adoption of a defined process of software change can lead to similar results and recommend it as a good practice. In contrast, undefined processes, which differ from developer to developer and from task to task, do not offer predictable productivity and quality and therefore are a malpractice.

Phased model of software change (PMSC) is a process model that is geared towards software changes that localize in the code.[23] It consists of several subtasks or phases: *Initiation* involves requirements elicitation, analysis, and prioritization. *Concept location* identifies the units where the bulk of the change is to be implemented. *Impact analysis* determines the strategy and full impact of the change in the old code. *Actualization* modifies the code to satisfy the change request, and it includes *change propagation* that updates all units that must be changed. *Refactoring* modifies the structure of software without changing its behavior. When it is done before actualization, it is called *prefactoring* and it gives the old code a structure that will make the actualization easier. *Postfactoring* is a removal of technical debt that was accumulated by previous code modifications. *Verification* certifies correctness of the code, and it is a part of all phases that modify the code, ie, actualization and both refactoring phases. *Conclusion* includes commit of the updated code to the code repository and may also include system testing and update of other work products like documentation. *Enactment of PMSC* can contain all or just some of the phases. Several examples of PMSC enactment are available in the literature.[24-26] Senin[27] demonstrated the improvement in developer productivity when using PMSC.

Note that PMSC applies to changes that localize in few units. Most of the perfective changes are of that nature and some corrective and adaptive changes also fit that mold. In contrast, some corrective changes start with a symptom of a problem and require diagnostic phase that finds the causes of the problem; hence, they follow a different process than PMSC.[21,28]

Also, changes in the fundamental assumptions on which software is built sometimes require *massive changes*. They usually belong into the category of adaptive changes and are triggered by changes in technology, supporting hardware, operating systems, programming languages,

libraries, and so forth. They impact large portions of software and require different process than PMSC. They often overwhelm ordinary developers, and only some teams can marshal resources necessary to do these changes correctly. They are rare in software evolution, but they sometimes appear in prolonged software maintenance. Their process is currently being researched.[29]

4 | EVOLVING AND STABILIZED PARTS OF THE CODE

The demands of evolution often originate from the outside world, and the developers react to these demands. To some extent, the developers can predict these demands and use these predictions in their work. For example, in student registration system, it is unlikely that the format of student address will change, while the registration process is likely to experience a significant evolution as the university policies frequently change.

Stabilized part of the code is unlikely to be evolved in the future, while *evolving part of the code* will undergo evolutionary modifications. The evolvability patterns are needed only in the evolving part, and stabilized part can contain evolvability antipatterns without jeopardizing overall software evolvability. Since the stringent evolvability patterns lead to higher software costs, such a division of the code may lower the cost of software development.

It should be noted that agile development assumes that future evolution is completely unpredictable.[30] By implication, it assumes the entire software should be kept in an evolvable state. However, most software systems have stabilized parts. For example, there are libraries that cannot be modified by the developer team and hence they are stabilized. A good and realistic practice tries to assess what is the best split between evolvable and stabilized parts of the code.

Experience has shown that evolution demands often surprised developers in the past and therefore anticipation of future evolution demands is always risky. In this context, it is a malpractice to leave this anticipation to an individual developer. A good practice is to have an explicit and transparent process that deals with the distinction between evolvable and stabilized parts and where several stakeholders participate.

A specific instance of this anticipation is the deliberate transfer of software from evolution to maintenance. Perfective changes constitute the bulk of software evolution, but they are absent during maintenance; therefore, the transition means that the stakeholders anticipate that all future changes will just be corrective and adaptive.[1] Like with all anticipations, if there is a misjudgment and a return to evolution becomes necessary, the deteriorated evolvability may make that return to evolution difficult or even impossible.

5 | ANALYZABLE SEGMENTS

Analyzability of software is frequently mentioned as a part of evolvability.[3,4] However, additional detail is necessary to clarify this quality.

Determination of unit responsibility is one of the frequent evolution tasks. However, a unit often cannot handle all expected responsibility and delegates' parts of it to other units, the *suppliers*. Hence, to answer the question "What does this unit do?" we need to understand not only the unit itself but also its suppliers.

In object-oriented languages, suppliers of class X may be implemented as components of X, as base classes of X, or as other constructs. For example, class Item in the Point of Sale program is responsible for items sold in the store. Its supplier, class Price, assumes the responsibility for the price of an item, including price changes and sale prices. It is implemented as a component of class Item, ie, a variable of type Price is a member of class Item.

There is a *dependency* between units A and B, denoted <A,B>, if and only if B is a supplier of A. A *dependency graph* D is a directed graph where nodes are units of the program and edges are dependencies. Let G = (C,D) be a dependency graph, ie, a directed graph where C is the set of classes and D a set of dependencies. Then *supplier segment of unit X* is denoted *seg*(*X*), and it is a set containing *X*, its suppliers, suppliers of suppliers, and so forth. Formally, $seg(X) = \{A \mid A = X \text{ or there is a dependency } \langle Y, A\rangle \in D \text{ such that } Y \in seg(X)\}$. Supplier segments are the true implementers of the unit responsibility, rather than the unit alone.

The first step towards comprehension of the responsibility of unit X is the determination of its supplier segment *seg*(*X*). The segments may consist of units that are dispersed throughout the code base, and therefore, developers need help of a tool. The tool should determine supplier segment with acceptable precision, recall, and scaling.

There are *dangerous software constructs* that are parts of the contemporary programming languages that are beyond the current capabilities of program analysis tools.[31] Examples of such dangerous constructs are pointer arithmetic in C++, reflection in Java, etc. Using them haphazardly in the evolvable part of software is malpractice.

A good practice requires preservation of the *analyzability* of the evolvable code. The assessment requires a choice of a specific analysis tool T and a determination of project standard for precision, recall, and scaling of the analysis. As an example, it may be required that all evolvable code must be analyzable by tool T and the results for supplier segments must have recall 100% (safety), precision 80%, and the analysis must not take longer than 30 seconds. All code that does not meet these expectations is an antipattern.

6 | SIGNIFICANT CONCEPT ENCAPSULATION

To satisfy the change request and make a correct modification of the code using PMSC, the developers have to comprehend the impact of the change on the existing code. From that point of view, the code can be seen as an implementation of a collection of concepts and the concept location indicates which of these concepts need to be modified. The concepts whose implementation needs to be modified are called *significant concepts (of the change)*.[23] The developers have to locate these concept implementations in the code.[32] There is at least one significant concept for each change, but there can be more than one.

In a previously published example of software change, the application Drawlets draws figures on a canvas.[24] The change request is "Implement an owner for each figure. An owner is the user who put the figure onto the canvas, and only the owner should be allowed to modify it." There are 2 significant concepts that have to be located to implement this change: Concept "figure properties" needs to be extended by a new property, figure owner, and concept "session properties" also has to be extended by a new property, session owner.

Evolution is facilitated if each significant concept is *encapsulated* in a properly named code unit, because this makes them easier to find, comprehend, and modify. A concept is encapsulated in a unit if that unit contains localized implementation of that concept, without any additional code that would not belong to that concept. Then the developer can easily find that concept and while making modifications does not have to deal with unrelated code. Encapsulation is violated when a concept illogically spans several units. An example of such antipattern is a formula that computes an insurance premium from the customer data that is spread over several units.

The antipattern of *hidden concepts* is an extreme case of concept delocalization. Those are concepts that are not encapsulated as a type, but instead, their representation is being used by the clients. Examples are physical units like kilograms that are represented by plain floating point numbers, currency amounts that are represented by integers, and colors that are encoded as integers. The phenomenon has been also called *hidden dependencies*[33] or *cross-cutting concerns*.[34]

The problem arises when these hidden concepts become significant concepts of a change. For example, when change request demands adding an additional color, or to change kilograms to pounds, the code modification must be made consistently in all places where the hidden type is used. In the worst case that can lead to massive changes and may need a different process and different tools than the typical perfective changes.

The developers sometimes make this choice of hidden concepts to avoid too many classes in the code or to avoid trivial classes that seem to contribute little to the overall architecture of the program. However, this choice is a serious antipattern when made in the evolving part of the code and when the hidden concept becomes a significant concept of a change request.

Another problem is a change request where *the significant concept is a function*, like in "Change student registration process by adding update of instructor class list." The significant concept of this change request is "student registration process," and the support for this process is best represented by a function.

Textbook use of object-oriented programming localizes types, in contrast to imperative programming that localizes functions. In a textbook object-oriented implementation, the implementation of the concept "student registration process" is divided into small pieces that belong to classes Student, Transcript, Course, Section, Instructor, and others, depending on the specifics of the implementation. All these pieces have to be located before the required modification of the registration process can take place, which is a laborious and error-prone task. Hence, the textbook object-oriented structure of the code units is unfriendly towards changes in the processes that the code supports, and it is an antipattern. Contrary to what textbooks say, if likely future significant concepts are processes that are supported by functions, it should be the functions that should be encapsulated, rather than types. This issue was explored in Bhattacherjee et al[35] and Rajlich and Ragunathan.[36]

7 | WRAPPING

Note that patterns may serve different purposes. The original purpose of patterns is reusability, as the title of the original book indicates,[37] but there can be also patterns for evolvability, parallelism, security, and other purposes. Some patterns may serve several of these desirable purposes, but they also may advance one purpose at the expense of the others. Note that reusability increases productivity of the initial development and evolvability increases productivity of evolution and therefore these 2 software qualities are not identical. There are documented cases where reusability patterns hinder evolution, and hence, they are evolvability antipatterns.[38]

Nowhere is this difference clearer than with *Adapter pattern*, also called *wrapper*.[37] It is used in a situation when the client expects a different contract than what the supplier offers. Instead of changing the code of either supplier or client that would fix this gap, a wrapper is an extra unit that is placed between the supplier and client and that translates/augments/changes the supplier responsibility in such a way that the client expectation is satisfied.

A wrapper provides a fast correction of the mismatch between 2 units that have a dependency between them, and it is an alternative to the PMSC process that requires painstaking and time-consuming change propagation. Effort saving can be substantial, but there is a trade-off: The resulting code is illogical, and wrappers delocalize implementations of supplier concepts because parts of the impacted concepts are now in the supplier and parts are in the wrapper.

The resulting code reflects the history of wrapping rather than the concepts of the code, and it is hard to understand for everybody who does not know this history. Hence, using wrappers in evolvable code is a malpractice, while using it in stabilized code may be a good practice.

Since all maintained code is stabilized, wrapping is an acceptable practice during software maintenance. While it may not significantly impact maintainability, it quickly erodes evolvability due to delocalization of concepts that it causes. Hence, wrappers are sometimes good and sometimes bad patterns, depending on the context of their use.

8 | CONCLUSIONS

Evolution consists of repeated software changes. Defined processes of software change lead to improved productivity and quality of software evolution.

Evolutionary development practices and patterns may increase the cost, but they do not have to be applied to the whole software. It is sufficient to apply them to the evolvable part of software only. In the evolvable part, all unit segments should be analyzable so that developers can identify them with the help of analysis tools. Also, evolvable concepts should be encapsulated, including the concepts that are implemented as functions.

Wrapping (adapter pattern) should be used judiciously and avoided in the evolvable part of software as they lead to delocalization of concepts.

ACKNOWLEDGMENTS

The author discussed program analysis issues with Haipeng Cai and the rest of the paper with Yibin Wang, Chris Dorman, Vasik Rajlich, and Luke Rajlich. Their comments have been incorporated.

ORCID

Václav Rajlich http://orcid.org/0000-0001-9761-4583

REFERENCES

1. Rajlich, V. Software evolution and maintenance. *Proceedings of the on Future of Software Engineering*. 2014:133-144.
2. Lientz BP, Swanson EB, Tompkins GE. Characteristics of application software maintenance. *Commun ACM*. 1978;21(6):466-471.
3. Breivold, HP, Crnkovic I, Eriksson PJ. Analyzing software evolvability. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*. 2008:327-330.
4. Organización Internacional de Normalización, ISO/IEC 25010:2011 Systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models. 2011: ISO.
5. Martin RC. *Clean Code: A Handbook Of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall; 2008.
6. Robson DJ, Bennett KH, Cornelius BJ, Munro M. Approaches to program comprehension. *Journal of Systems and Software*. 1991;14(2):79-84.
7. Cornelissen B, Cornelissen B, Zaidman A, Van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans Softw Eng*. 2009;35(5):684-702.
8. Rajlich V, Wilde N. The role of concepts in program comprehension. *Proceedings of the 10th International Workshop on Program Comprehension*. 2002: 271-278.
9. Ratiu D, Deissenboeck F. From reality to programs and (not quite) back again. *Proceedings of the IEEE International Conference on Program Comprehension*. 2007:91-102.
10. Jazequel J-M, Meyer B. Design by contract: the lessons of Ariane. *Computer*. 1997;30(1):129-130.
11. Mouchawrab S, Briand LC, Labiche Y. A measurement framework for object-oriented software testability. *Inf softw technol*. 2005;47(15):979-997.
12. Runeson P. A survey of unit testing practices. *IEEE software*. 2006;23(4):22-29.
13. Feathers M. *Working Effectively with Legacy Code*. Upper Saddle River, NJ: Prentice Hall PTR; 2005.
14. Fowler M. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley; 1999.
15. Fowler M. Refactoring. 2010 [cited 2011 June 6, 2011]; Available from: http://refactoring.com/.
16. Mens T, Tourwé T. A survey of software refactoring. *IEEE Trans, Softw Eng*. 2004;30(2):126-139.
17. Khomh F, Di Penta, M, Gueheneuc Y-G. An exploratory study of the impact of code smells on software change-proneness. *Proceedings of the Working Conference on Reverse Engineering*. 2009:75-84.
18. Koschke R. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. 2007. Schloss Dagstuhl-Leibniz-Zentrum für Informatik
19. Koschke R. *Identifying and Removing Software Clones. Software Evolution*. Berlin, Heidelberg: Springer; 2008:15-36.
20. Grove D, Chambers C. A framework for call graph construction algorithms. *ACM Trans Program Lang Syst (TOPLAS)*. 2001;23(6):685-746.
21. Binkley D, Harman M. A survey of empirical results on program slicing. *Advances in Computers*. 2004:105-178.
22. Harter DE, Krishnan MS, Slaughter SA. Effects of process maturity on quality, cycle time, and effort in software product development. *Manage Sci*. 2000;46(4):451-466.
23. Rajlich V. *Software Engineering: The Current Practice*. Boca Raton, FL: CRC Press; 2012.
24. Rajlich V, Gosavi P. Incremental change in object-oriented programming. *IEEE Software*. 2004;21(4):62-69.
25. Febbraro N, Rajlich V. The role of incremental change in agile software processes. In *Agile Conference (AGILE)*. 2007:92-103.
26. Dorman C, Rajlich V. Software change in the solo iterative process: an experience report. In *Agile Conference (AGILE)*. 2012:21-30.

27. Senin Y. Case study of phased model for software change in a multiple-programmer environment. *M.S. thesis, Department of computer Science*. Detroit, MI.: Wayne State University; 2014.

28. Xu S, Rajlich V. Cognitive process during program debugging. in *Proceedings of the 3nd IEEE International Conference on Cognitive Informatics (ICCI'04)*. 2004:176-182.

29. Hughes J, Sparks C, Stoughton A, Parikh R, Reuther A, Jagannathan S. Building resource adaptive software systems (BRASS): objectives and system evaluation. *ACM SIGSOFT Software Engineering Notes*. 2016;41(1):1-2.

30. Beck K. *Extreme Programming Explained*. Reading, MA: Addison Wesley; 2000.

31. Livshits B, Sridharan M, Smaragdakis Y, et al. In defense of soundiness: a manifesto. *Comm ACM*. 2015;58(2):44-46.

32. Rajlich, V. Intension are a key to program comprehension. in *IEEE International Conference on Program Comprehension (ICPC '09)*. 2009:1-9.

33. Yu Z, Rajlich V. Hidden dependencies in program comprehension and change propagation. In: *International Workshop on Program Comprehension*. 2001: 293-299.

34. Eaddy M, Zimmermann T, Sherwood KD, et al. Do crosscutting concerns cause defects? *IEEE transactions on Software Engineering*. 2008;34(4):497-515.

35. Bhattacherjee A, DeShazer K, Gerlach JH, Rierden B. A hybrid approach to OO development: the SUMMITrak project at TCI. *J Syst Softw*. 2001;57(2):91-98.

36. Rajlich V, Ragunathan S. A case study of evolution in object oriented and heterogeneous architectures. *J Syst Softw*. 1998;43(2):85-91.

37. Vlissides J, Helm R, Johnson R, Gamma E. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley; 1995.

38. Vokáč M, Tichy W, Sjøberg DI, Arisholm E, Aldrin M. A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empir Softw Eng*. 2004;9(3):149-195.

How to cite this article: Rajlich V. Five recommendations for software evolvability. *J Softw Evol Proc*. 2018;30:e1949. https://doi.org/10.1002/smr.1949