In Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE) 2005, pages 13-22, IEEE Computer Society, 2005. DOI:https://doi.org/10.1109/IWPSE.2005.7

Challenges in Software Evolution

Tom Mens
Service de Génie Logiciel
Université de Mons-Hainaut
Belgium

Michel Wermelinger
Computing Department
The Open University
United Kingdom

Stéphane Ducasse
Université de Savoie
France

Serge Demeyer
Lab on Reengineering
University of Antwerp
Belgium

Robert Hirschfeld
Future Networking Lab
DoCoMo Euro-Labs
Germany

Mehdi Jazayeri
Faculty of Informatics
University of Lugano
Switzerland

Abstract

Today's information technology society increasingly relies on software at all levels. Nevertheless, software quality generally continues to fall short of expectations, and software systems continue to suffer from symptoms of aging as they are adapted to changing requirements and environments. The only way to overcome or avoid the negative effects of software aging is by placing change and evolution in the center of the software development process. In this article we describe what we believe to be some of the most important research challenges in software evolution. The goal of this document is to provide novel research directions in the software evolution domain.

1 Introduction

Today's information technology society increasingly relies on software at all levels. This dependence on software takes place in all sectors of society, including government, industry, transportation, commerce, manufacturing and the private sector. Productivity of software organisations and software quality generally continue to fall short of expectations, and software systems continue to suffer from symptoms of aging as they are adapted to changing requirements. One major reason for this problem is that software maintenance and adaptation is still undervalued in traditional software development processes.

The only way to overcome or avoid the negative effects of software aging is by placing change in the center of the software development process. Without explicit and immediate support for change and evolution, software systems become unnecessarily complex and unreliable. The negative influence of this situation is rapidly increasing due to technological and business innovations, changes in legislation and continuing internationalisation. One must therefore advance beyond a restricted focus on software development, and provide better and more support for software adaptation and evolution.

Such support must be addressed at multiple levels of research and development. It requires: (i) basic research on formalisms and theories to analyse, understand, manage and control software change; (ii) the development of models, languages, tools, methods, techniques and heuristics to provide explicit support for software change; (iii) more real-world validation and case studies on large, long-lived, and highly complex industrial software systems.

2 Classification of challenges

In April 2005, a workshop on *Challenges on Software Evolution (ChaSE 2005)* was jointly organised by the ESF Research Network *RELEASE (Research Links to Explore and Advance Software Evolution)* and the ERCIM Working Group on Software Evolution. The workshop attracted 37 participants originating from 10 different European countries and Canada. Its goal was

to identify the most important challenges on software evolution, and to come up with a list of important future research topics in the domain of software evolution. This paper lists and explains the most important challenges in software evolution that were identified during the workshop.[1]

To increase the readability of this paper, the challenges are classified according to a number of more or less orthogonal dimensions[2]. Table 1 gives a comprehensive overview of all these challenges using this classification. A letter of the alphabet is used to identify each challenge.

Time horizon. Is a *short*, *medium* or *long*-term effort required in order to achieve results?

Research target. Is the challenge related to the *management*, *control*, *support*, *understanding* or *analysis* of software evolution?

Stakeholders. Who is interested in, involved in, or affected by the challenge? Given the diversity of challenges, many different people can be involved: *managers, developers, designers, end-users, teachers, students, researchers*, and so on.

Type of artifact under study. Which type of artifact(s) does the challenge address? Artifacts should be interpreted in the broad sense here since they can refer to *formalisms*, *tools*, *techniques*, *models*, *metamodels*, *languages*, *programs*, *processes*, *systems*, and many more.

Type of support needed. Which type of support is needed in order to address the challenge? Although this question is completely different from the previous one, the list of possible answers is the same. One can provide *formal support*, *tool support*, *language support*, *process support* and so on in order to address software evolution.

[1] Because the *ChaSE 2005* participants and the *RELEASE* network members only reflect a subset of the entire research community on software evolution, we may have missed some important challenges.

[2] This classification works well for the purpose of this paper, but other classifications may be used instead to structure the list of challenges.

3 Enumeration of challenges

A. Preserving and improving software quality

The phenomenon of *software aging*, coined by Dave Parnas [28], and the *laws of software evolution* postulated by Manny Lehman [22] agree that, without active countermeasures, the quality of a software system gradually degrades as the system evolves. In practice, the reason for this gradual decrease of quality (such as reliability, availability and performance of software systems) is for a large part caused by external factors such as economic pressure.[3] The negative effects of software aging can and will have a significant economic and social impact in all sectors of industry. Therefore it is crucial to develop tools and techniques to reverse or avoid the intrinsic problems of software aging. Hence, the challenge is *to provide tools and techniques that preserve or even improve the quality characteristics of a software system, whatever its size and complexity*.

B. A common software evolution platform

A major difficulty when trying to address the previous challenge, has to do with *scalability*. The need is to develop solutions that are applicable to long-lived, industrial-size software systems. Many of the tools that must be built to manage the complexity intrinsic to software evolution are too complex to be built by single research groups or individuals. Therefore, a closely related challenge, raised by Michele Lanza, is *to develop and support a common application framework for doing joint software evolution research*. This challenge raises issues such as tool integration and interoperability, common exchange formats and standards, and so on. As an example of such a shared framework that served as a common software evolution research vehicle within the *RELEASE* network is the *Moose* reverse engineering environment [8]. A concrete goal could be to try and extend this framework with tools to analyse, manage and control software evolution activities.

Another candidate that may serve as a common platform is *Eclipse*. It has the advantage of visibility and

[3] Identifying these pressures, and determing measures to evaluate, respond to and control them represents, in itself, a major challenge.

industrial acceptance and also permits reuse of certain components (e.g., Java parsing). An important disadvantage is its lack of control over releases. One researcher mentioned that he had to keep several versions of the platform because not all plug-ins work on all versions. There is also the issue of exploratory prototyping, which is better supported by environments like Smalltalk. Both options should probably co-exist, although this of course implies duplication of effort.

C. Supporting model evolution

Although support for software evolution in development tools can still be advanced in many ways, there are already a number of success stories. One of them is *program refactoring*, introduced by John Opdyke in the early 1990s as a way to improve the structure of object-oriented programs without affecting their desired external behaviour [27]. Since the publication of Martin Fowler's book on refactoring [9], this program transformation technique has gained widespread attention [26]. Today, refactoring support has been integrated in many of the popular software development environments.

Unfortunately, it is observed that almost all existing tool support for software evolution is primarily targeted to programs (i.e., source code). Design and modelling phases (supported by UML CASE tools, for example) typically provide much less support for software evolution. Taking the example of refactoring, we didn't find any modelling tool providing adequate means for refactoring design models. Research in model refactoring is just starting to emerge [31, 35].

This can be generalised into the following challenge: *Software evolution techniques should be raised to a higher level of abstraction, in order to accomodate not only evolution of programs, but also evolution of higher-level artifacts such as analysis and design models, software architectures, requirement specifications, and so on.*

Since the advent of model-driven software engineering [24], this challenge becomes increasingly more relevant, and techniques and tools for dealing with model evolution are urgently needed.

D. Supporting co-evolution

A challenge that is related to the previous one is *the necessity to achieve co-evolution between different types of software artifacts or different representations of them.* Modification in one representation should always be reflected by corresponding changes in other related ones to ensure consistency of all involved software artifacts.

To give but a few examples, support for co-evolution is needed between:

- programs (source code) and design models [6, 34] or software architectures
- structural and behavioural design models. This is for example the case with UML, where different models are used to express structure (e.g., class diagrams) and behaviour (e.g., sequence diagrams and state-transition diagrams)
- software (at whatever level of abstraction) and the languages in which it is developed. Whenever a new version of the programming, modeling or specification language is provided, it is quite possible that programs that worked perfectly in a previous version of the language fail to function in the new version.
- software and its business, organisational, operational and development environment. Changes in each of these environments will impact the software and conversely. This feedback loop is well-known in software evolution research [20, 32].
- software and its developer or end-user documentation

To provide better support for co-evolution, it is worthwhile to take a look at other domains of science that can hopefully provide better insights in the matter. Linguistic theory and the history of natural language evolution may increase understanding in how software languages evolve. In order to better understand software co-evolution, it could be interesting to look at co-evolution in biology. In fact, the term co-evolution originated in biology, and is borrowed by computer scientists to describe a similar situation in software development.

E. Formal support for evolution

According to Wordsworth [33], "a formal method of software development is a process for developing software that exploits the power of mathematical notation and mathematical proofs." For several decades, formal methods have been advocated as a means to improve software development, with an emphasis on software specification, verification and validation. Nevertheless, as Robert Glass observes in his Practical Programmer column [11]: "Formal methods have not, during that extended period of time (well over 30 years by now), had any significant impact on the practice of software engineering." He points out a major cause of this problem: "What in fact most practicioners tell me about specifications is that the needs of the customers evolve over time, as the customer comes to learn more about solution possibilities, and that what is really needed is not a rigorous/rigid specification, but one that encompasses the problem evolution that inevitably occurs."

Unfortunately, existing formal methods provide very poor support (or even none at all) for evolving specifications. Let us take the example of formal verification, which aims to prove mathematically that the implementation of a software system satisfies its specification. Specialists that were consulted in relation to this question agreed that even today there are no truly *incremental verification* approaches available. With current verification tools, even if small localised changes are made to the specification of a program, the entire program needs to be verified again. This makes the cost of verification proportional to the size of the system. What is desired is that it is proportional to the size of the units of change.

This leads to the next challenge in software evolution: *In order to become accepted as practical tools for software developers, formal methods need to embrace change and evolution as an essential fact of life*.

Besides the need for existing formal methods to provide more explicit support for software evolution, there is also a clear need for *new formalisms to support activities specific to software evolution*. As an illustrative example, reconsider the activity of software refactoring. Formal techniques are clearly needed in order to ensure one of the basic properties of refactorings, namely that they preserve certain behavioural properties of the software [25].

F. Evolution as a language construct

As a very interesting research direction, *programming (or even modelling) languages should provide more direct and explicit support for software evolution*. The idea would be to treat the notion of change as a first-class entity in the language. This is likely to cause a programming paradigm shift similar to the one that was encountered with the introduction of object-oriented programming. Indeed, to continue the analogy, one of the reasons why object-oriented programming became so popular is because it integrated the notion of reuse in programming languages as a first-class entity. The mechanisms of inheritance, late binding and polymorphism allow a subclass to reuse and refine parts of its parent classes.

During the workshop it was pointed out that explicit support for software evolution is considerably easier to integrate into dynamically typed languages that offer full reflective capabilities [13]. Classboxes [2] are also a new module system that controls the scope of changes in an application. Changes can be introduced in a system without impacting existing clients, changes are only visible to new clients desiring to see the changes.

G. Support for multi-language systems

Mohammad El-Ramly pointed out that another crucial, and largely neglected, aspect of software evolution research is the need to deal with multiple languages. Indeed, in large industrial software systems it is often the case that multiple programming languages are used. More than three languages is the rule rather than the exception. Therefore, software evolution techniques *must provide more and better support for multi-language systems*. One way to tackle this problem is to provide techniques that are as language-parametric (or language-generic, or language-independent) as possible [15, 30].

Note that this challenge is becoming increasingly more relevant as the number of languages needed or used in software systems is increasing. Programming languages, modelling languages, specification languages, XML-based languages for data in-

terchange, domain-specific languages, business modelling languages, and many more are becoming ever more widely used.

H. Integrating change in the software life-cycle

It is important to investigate how the notion of software change can be integrated into the conventional software development process models. A typical way to include support for change into a more traditional software process models is by resorting to an *iterative* and *incremental* software development process. So-called *agile* software processes (including the well-known *extreme programming* method) already acknowledge and embrace change as an essential fact of life [1]. Other processes, such as the *staged life-cycle model*, have been proposed as an alternative that provides explicit support for software change and software evolution [29].

I. Increasing managerial awareness

Besides better understanding of, and better support for, evolutionary process models, there is also a need to *increase awareness of executives and project managers of the importance and inevitability of software evolution*. Training is needed to convince them of the importance of these issues, and to teach them to plan, organise, implement and control software projects in order to better cope with software changes.

We suggest to explain the importance of software evolution through the SimCity metaphor. This computer game simulates a city and is a typical example of a highly complex dynamic system where continuous corrective actions of the "manager" are needed in order to avoid deteriorating the "quality" of the city and, ultimately, its destruction or desertion.

J. Need for better versioning systems

Although support for software evolution in software development tools can still be improved in many ways, there are already a number of success stories. One of them is *version management*. Version control is a crucial aspect in software evolution, especially in a collaborative and distributed setting, where different software developers can (and will) modify the program, unaware of other changes that are being made in parallel. A wealth of version control tools is available, commercial as well as freeware. The most popular one is probably CVS (`www.cvs.org`).

Nevertheless, for the purpose of analysing the evolution of software systems, these version repositories clearly fall short because they do not store enough information about the evolution. Therefore, the challenge is *to develop new ways of recording the evolution of software that overcome the shortcomings of the current state-of-the-art tools*.

When addressing this challenge, it is necessary to communicate and coordinate with the research community on Software Configuration Management, that is trying to address very related issues.

K. Integrating data from various sources

One promising approach to reason about the evolution history of software systems, is the integration of data from a wide variety of sources: bug reports, change requests, source code, configuration information, versioning repositories, execution traces, error logs, documentation, and so on. Besides all of the above information, it is equally important to take into account information about the software process during the analysis of change histories: the software team (size, stability, experience, ...), individual developers (age, experience, ...), the project structure (hierarchical, surgical, flat, ...), the process model (waterfall, spiral, agile, ...), the type of project (e.g., open source), and so on. Indeed, Conway's law [4] postulates that the architecture of a software system mirrors the structure of the team that developed it (and, more generally, the structure of a system tends to mirror the structure of the group producing it).

The main challenge here is *to find out how these different kinds of data can be integrated, and how support for this integration can be provided*.

Having a flexible and open-meta model as the one of the Moose reengineering environment supporting entity annotations [8] and version selection should be regarded as a first step in that direction [7, 10].

L. Analysing huge amounts of data

Given the sheer amount of data that needs to be processed during the above analysis, *new techniques and*

tools are needed to facilitate manipulation of large quantities of data in a timely manner. In order to achieve this, one can probably borrow from related areas of computer science that deal with similar problems. For example, one may consider using data mining techniques as used by the database community, or techniques related to DNA sequence analysis as used in bio-informatics.

These techniques could be implemented as an extension of current tools such as CodeCrawler [16, 17] that already supports the management of large data sets via polymetric views (i.e., views enriched with semantical information). Another attempt that has been made to address this challenge is a technique that suggests to the developer changes to be performed based on the co-occurrence of past changes [36].

M. Empirical research

In the context of software evolution there is *a need for more empirical research* [14]. Among others, comparative studies are urgently needed to measure the impact of

- *process models*: in an industrial setting, which software process is most appropriate for which type of evolution activity?
- *tools*: to which extent does the use of a tool facilitate the execution of a particular evolution activity compared to the manual execution of the same activity; how does one compare the performance of different tools to carry out the same evolution activity?
- *languages*: what is the impact of the programming language on the ease with which certain evolution activities can be performed? For example, dynamically typed languages with reflective capabilities seem to be more suited that other languages to support the task of runtime evolution.
- *people*: to which extent does the experience, background and training of a software developer contribute to his ability to carry out certain software evolution activities?

In order to facilitate such comparative studies, an initial taxonomy for software evolution has been proposed in [3], but further validation and elaboration of this taxonomy is needed.

In order to obtain statistically significant results, a sufficiently large set of representative examples is needed. This is not always easy in an industrial setting, since it is very difficult to obtain data on the evolution of industrial software systems over a long time span (several years).

N. Need for improved predictive models

Predictive models are crucial for managers in order to assess the software evolution process. These models are needed for predicting a variety of things: where the software evolves, how it will evolve, the effort and time that is required to make a change, and so on. Unfortunately, existing predictive models, typically based on software metrics, are far from adequate.

To counter this problem, Miguel Lopez suggested to look at metrology research [23], the science of measurement, which explicitly takes into account the notion of *uncertainty* that is also inherent in software evolution [19]. Girba's "Yesterday's Weather" measurement [10] is another step in the same direction. This measurement characterizes the climate of changes in a system and helps assessing the trust that may be placed in the continuity of changes (based on the assumption that assets that have changed in the recent past are more likely to change in the near future).

O. Evolution benchmark

In order to adequately test, validate, and compare the formalisms, techniques, methods, and tools to be developed for the other challenges, it is useful *to come up with, and reach a consensus on, a common set of evolution benchmarks and case studies which, together, are representative for the kinds of problems needing to be studied.* Given the amount of long-lived, industrial-size, open-source projects available today, it should be feasible to come up with such a benchmark [5].

P. Teaching software evolution

One of the best ways to place change and evolution in the center of the development process is to

educate the future generations of software engineers. However, classroom programming exercises usually are well specified, have a single release, and are small in size. Capstone projects are more amenable to convey the need for dealing with software evolution, but on the one hand they are often supervised in a rather loose mode, and on the other hand it is preferable to prepare students in earlier courses with the concepts and tools they need to handle changes in their project. Therefore, a big challenge for everyone involved in teaching concerns *how to integrate the ideas, formalism, techniques and tools for software evolution into our computer science curriculum in a meaningful way*. As a community, we need to decide upon what we want to teach, how we want to teach it, and provide the necessary course material for it.

Q. A theory of software evolution

It seems that often researchers either do empirical investigations into the evolution of a given system over its life-time, or propose tools and techniques to facilitate evolution. But it is not always clear what one gets from all the collected data nor if the tools actually are informed by typical evolution patterns. One needs to study and compare evolution activities before and after the installation of some tool supporting such activities.

To undertake such studies, *it is necessary to develop new theories and mathematical models to increase understanding of software evolution, and to invest in research that tries to bridge the gap between the* what *(i.e., understanding) of software evolution and the* how *(i.e., control and support) of software evolution*. This seems to be a logical continuation of the research that was initiated by Lehman and Ramil in [18], and their theory of software evolution that was proposed in [21]. Nevertheless, this theory still remains to be formalised and enriched. For example, Lehman suggested that many software failures are due to changes that impact on the initial (often implicit) assumptions, and therefore a theory of software evolution must take assumptions into account.

R. Post-deployment runtime evolution

Maintenance and evolution of continuously running systems have become a major topic in many areas, including embedded systems, mobile devices, and service infrastructures. *There is an urgent need for proper support of runtime adaptations of systems while they are running, without the need to pause them, or even to shut them down.* For that, further steps are needed to make the deployment, application, and exploration of dynamic adaptations more comprehensible.

Dynamic Service Adaptation (DSA) is a promising approach trying to address these issues by providing appropriate means to introspect and navigate basic computational structures and to adjust them accordingly [12, 13].

While evolution support at runtime via dynamic adaptation addresses many of the requirements stated above, it does not address program or system comprehension. On the contrary, systems that are changed dynamically are harder to understand using contemporary approaches. Proper tool support is needed for the exploration and manipulation of both basic and enhanced runtime structures as well as change events and their history.

4 Final remarks

This paper proposed, classified and explained 18 essential challenges in the software evolution that need to be addressed in the future. The challenges are not unrelated, and some activities can address more than one challenge simultaneously, making it easier to achieve the goal of proper understanding and support for software evolution. For example, language-independent techniques may help overcome challenges C and G, and studies on evolution-supporting tools contribute to challenges M and Q.

The proposed list of challenges is not the opinion of a single person, but the result of a concentrated effort by the *RELEASE* research network (counting over 20 European researchers active in software evolution) to come up with a list of future research avenues in software evolution. As the output of the ChaSE 2005 workshop, these challenges provide an informed summary of the principle current challenges that face the software engineering community in general and the software evolution community in particular. The suggested list of challenges provides a framework for future work in software evolution research.

Acknowledgements

This research was financially supported by the European Science Foundation through the Research Network *RELEASE*, and by the European Research Consortium on Informatics and Mathematics through the Working Group on Software Evolution. We express our sincere gratitude to everyone who actively participated in the joint ESF-ERCIM *ChaSE 2005* workshop, for contributing so many ideas that were integrated in this paper.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 2005. To appear.

[3] J. Buckley, T. Mens, M. Zenger, A. Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution*, 2005. To appear.

[4] M. Conway. How do committees invent? *Datamation Journal*, pages 28–31, April 1968.

[5] S. Demeyer, T. Mens, and M. Wermelinger. Towards a software evolution benchmark. In *Proc. Int'l Workshop on Principles of Software Evolution*, September 2001.

[6] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented design and implementation. In *Proc. Int'l Symp. Software Architectures and Component Technology*. Kluwer Academic Publishers, January 2000.

[7] S. Ducasse, T. Gîrba, and J.-M. Favre. Modeling software evolution by treating history as a first class entity. In *Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 71–82, 2004.

[8] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005.

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proc. Int'l Conf. Software Maintenance*, pages 40–49. IEEE Computer Society Press, 2004.

[11] R. L. Glass. The mystery of formal methods disuse. *Communications of the ACM*, 47(8):15–17, 2004.

[12] R. Hirschfeld and K. Kawamura. Dynamic service adaptation. In *Proc. Workshop on Distributed Auto-adaptive and Reconfigurable Systems (DARES)*, pages 290–297. IEEE Press, 2004.

[13] R. Hirschfeld, K. Kawamura, and H. Berndt. Dynamic service adaptation for runtime system extensions. In *Lecture Notes in Computer Science*, volume 2928, pages 225–238. Springer, 2004.

[14] C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Software Engineering*, 25(4):493–509, July/August 1999.

[15] R. Lämmel and W. Lohmann. Format Evolution. In *Proc. Int'l Conf. Reverse Engineering for Information Systems*, volume 155, pages 113–134. OCG, 2001.

[16] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[17] M. Lanza and S. Ducasse. Codecrawler - an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74 – 94. Franco Angeli, 2005.

[18] M. Lehman, J. F. Ramil, and G. Kahen. Evolution as a noun and evolution as a verb. In *Proc. SOCE 2000 Workshop on Software and Organisation Co-evolution*, July 2000.

[19] M. M. Lehman. Uncertainty in computer application and its control through the engineering of software. *Journal of Software Maintenance*, 1(1):3–27, September 1989.

[20] M. M. Lehman, D. E. Perry, and J. F. Ramil. On evidence supporting the feast hypothesis and the laws of software evolution. In *Proc. Int'l Symp. Software Metrics*. IEEE Computer Society Press, 1998.

[21] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *Proc. 4th Int'l Workshop on Principles of Software Evolution*, pages 70–74. ACM Press, 2001.

[22] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. Int'l Symp. Software Metrics*, pages 20–32. IEEE Computer Society Press, 1997.

[23] M. Lopez, S. Alexandre, V. Paulus, and G. Seront. On the application of some metrology concepts to internal software measurement. In *Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2004)*, 2004.

[24] S. J. Mellor, A. N. Clark, and T. Futagami. Model-driven development: Guest editor's introduction. *IEEE Software*, 20(5), September/October 2003.

[25] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Proc. Int'l Conf. Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.

[26] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–162, February 2004.

[27] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[28] D. L. Parnas. Software aging. In *Proc. Int'l Conf. Software Engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[29] V. T. Rajlich and K. H. Bennett. A staged model for the software lifecycle. *IEEE Computer*, pages 66–71, July 2000.

[30] S. Tichelaar, Stéphane Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proc. Int'l Symp. Principles of Software Evolution*, pages 157–169. IEEE Computer Society Press, 2000.

[31] R. Van Der Straeten, V. Jonckers, and T. Mens. Supporting model refactorings through behaviour inheritance consistencies. In *Proc. Int'l Conf. UML 2004*, volume 3273 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, October 2004.

[32] L. Williams and A. Cockburn. Agile software development: It's about feedback and change. *IEEE Computer*, 36(6):39–43, June 2003.

[33] J. B. Wordsworth. Getting the best from formal methods. *Information and Software Technology*, 41(14):1027–1032, November 1999.

[34] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.

[35] J. Zhang, Y. Lin, and J. Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development - Research and Practice in Software Engineering*. Springer Verlag, 2005.

[36] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. Int' Conf. Software Engineering*, pages 563–572. IEEE Computer Society, 2004.

	Research target	Time horizon	Studied artifact	Support type	Stakeholder
A	preserving and improving quality	long	software system	tools, techniques, formalisms	developer, project manager, end user
B	analysing, managing, controlling	medium	programs	common appl. framework, exchange formats, interoperability standards	researcher
C	controlling, supporting	short	models	tools, techniques, formalisms	software engineer
D	controlling, supporting	medium	any pair of related artifacts	tools	software engineer
E	all types of research	medium-long	formalisms	formalisms	researcher
F	controlling, supporting	short-medium	languages	languages, programs	language designer, tool builder, researcher
G	controlling, supporting	medium-long	languages, software systems	tools, standards	tool builder
H	managing, controlling	medium	software process models	software process models	manager, software engineer
I	motivating	short	managers	metaphors	executives, managers
J	analysing	short	version control tools	tools	tool builder
K	analysing	medium	all information useful to get insight in a software system's evolution	statistical models, empirical studies	researcher
L	analysing	medium-long	release histories of long-lived, large, complex industrial software systems	techniques, tools	researcher
M	analysing	long	every kind of evolving artifact of a software system	empirical studies	researcher
N	analysing, predicting	short-medium	software systems	predictive models, measures, metrics	researcher
O	understanding, comparing	medium	evolving software systems	benchmarks, exemplars	researcher
P	teaching	short	formalisms, techniques, tools, theories	course material	teachers, students
Q	understanding, supporting	medium-long	everything	everything	researcher
R	controlling, supporting	short-medium	languages, execution platforms	languages, execution platforms, programs	tool builder, end user

Table 1. Classification of software evolution challenges