A Software Project Perspective on the Fitness and Evolvability of Personal Learning Environments

Christian R. Prause
Fraunhofer FIT
Schloss Birlinghoven, Sankt Augustin, Germany
christian.prause@fit.fraunhofer.de

ABSTRACT

This position paper deals with the exploration of fitness and evolvability of personal learning environments (PLEs). Taking a software engineer's perspective, PLE evolution is a software project. Software quality characteristics like Functionality and Usability map to the PLE's *fitness*, while Maintainability is important for *evolvability*. Only adaptation can secure future fitness. But for this, the software project has to be a good PLE for its developers in its on right.

1. INTRODUCTION

Common wisdom of software development — going back to Edward V. Berard — says: *"Walking on water and developing software from a specification are easy if both are frozen."* The success of Personal Learning Environments (PLEs) not only depends on their fitness for a certain purpose or environment, but no less than this depends on their ability to evolve, i.e. to adapt to changes. In the world of software, the continuous change of requirements is as sure as death and taxes. A PLE that fails to catch up with new requirements, ages and eventually becomes useless.

Bear with me, while I relate to the workshop's natural evolution metaphor: The extinction of dinosaurs is attributed to their failure to adapt to a changing environment. Their races showed only few diversification and innovativeness in behavioral strategies. When their world changed, only two species attempted an adaptation to new foods [6]. The dinosaurs' seemingly unbreakable predominance abruptly ended, making room for mammals that had waited in a niche. Mammals instantly filled the gap, and diversified into a plethora of species. Today, they emboss the planet's face as successful predators. If dinosaurs had not failed to adapt, they would have remained invincible competitors for any other species.

Predominance and wide spread were limited predictors of fitness and evolvability. Predominance can suppress competitors, but for how long? It is no disgrace to wait for a chance like the early mammals. To avoid extinction and eventually prevail, PLEs must evolve. Different from nature, where mutation of organisms occurs by accident and without the intent to optimize a creature's fitness, adaptation happens through conscious decision and human developers.

I take on a software engineer's view in the discussion on *fitness* and *evolvability* of PLEs. In this view, evolvability E is understood as a PLE's ability to embrace natural change, i.e. evolution E'. Fitness F does not imply evolvability, nor does evolvability imply fitness. Yet both are prerequisites of successful evolution $F \wedge E \Leftarrow E'$. New clades of PLEs often start from research. While fitness is usually tested thoroughly there, evolvability is often neglected.

The easier developers perform changes, the higher the chance that a PLE will cope with emerging requirements. Only this can make a PLE remain fit. Section 2 addresses the *ease of change* in software projects. This leads to the finding that learning is essential, and to the dualism that evolution is a PLE itself (Section 3). As long as a PLE's fitness suffices to safe it from extinction, evolvability is most important. A more evolvable PLE will adapt to changing environmental demands faster and easier. In conclusion, this is not least a matter of how easy PLE developers can obtain the necessary knowledge to make change happen.

2. EMBRACING THE CHANGE

Evolvability means to be prepared for changing environments and the unknown. It cannot be said in an across-the-board fashion what that practically means. It would imply to summarize the achievements of software engineering in a few sentences. In the Iron Triangle, the prime resource is *people* supported by *processes* and *technology* [5]. A full discussion of all three factors would be way out of scope of this paper. Instead, here are some fundamental considerations:

Whenever a software system grows larger, its complexity increases to a level that is no longer easily handable. Any successful software will eventually grow to that size. Abstraction and structuring that organize it into an understandable *architecture* become necessary. A good architecture means that developers can change parts without having to understand everything. But for the individual developer, having to adhere to architecture rules can be cumbersome. In a multi-tier Web-Service project, developers of front-end components bypassed the middle layer, and directly accessed back-end layers. This sped up development at first, but degraded architecture to a costly mess. Evolvability assessment should take into account how an architecture is protected, and how technical debt (see also [1]) is dealt with.

The term *architecture* should not be confused with integration platform. An integration platform can be something like UNIX's toolbox concept with its many small programs. It can be Web-Services, or a single program based on OSGi. The different platforms have different strengths and weaknesses that influence PLE fitness. Yet from an evolvability point of view, they are similar, all allowing fast adaptation through reuse of components. Do not think that a technology has reuse built in; instead, reuse is a discipline [12]. Here, it is more important to look at the processes.

Even with the best architecture, building a software architect's knowledge costs a hundred million. The combination

of deep domain knowledge and system engineering capabilities is invaluable [2]. Will the architect stay with the PLE project? What endeavors are made to train new architects?

Is the business model associated with the PLE project sustainable? While a potent company may be able to handle closed-source evolution on its own, also the openness of open source — mind the license — has advantages for evolvability: open standards, interoperability, cost effectiveness, attractiveness for users, possibly unlimited branching and experimentation, and a higher number of potential developers. However, a major road block to becoming a productive executor of PLE evolution, is knowledge about the software.

The Maintainability quality characteristic describes a software's capability to be modified and evolve [4]. By being analyzable, easy and predictable to change, and allowing to test changes, software developers can gain a deep understanding of the software through practical experimentation.

All of the aspects in the paragraphs above, help developers to understand the software by being few (complexity-reducing architecture), simple (with reuse in mind), supervised (senior architect guidance), open (open source), and practical (support experimentation) to learn. Knowledge about the software project, i.e. about how to evolve the PLE, is at the center of evolvability. Not only is the process of PLE evolution a software project, but a software project is a PLE itself. This duality is addressed next, when we look at internal documentation, which can be considered as the learning material that supports learning a software system.

3. THE SOFTWARE PROJECT AS A PLE

Modern software systems are too complex to fully understand them. But a certain understanding is necessary for performing changes. Working on a computer system is therefore a continuous learning process. The learning materials are process artifacts like source code, requirements, bug history, etc.; a developer's PLE consists of his individual selection of source code pieces, requirements, searchable bug records and so on that are delivered to him through tools like an IDE or an issue tracker. Developers do not like to create such learning material because it has few value for them [9]. But it is needed to persist collaborative long-term efforts like developing and maintaining a software.

Consider the example of source code (see also [7]): Source code is mostly learning material for us humans. There is an infinite number of ways of writing a same-purpose computer program. Neither does it matter for a computer what programming language one uses, nor does a parser care how functions and methods are named. The instructions that the computer needs are intertwined with the human-readable lines of source code. Functions, data types, objects, comments, macros, etc. and their respective names are just abstractions that make the design appear more clearly from code by masking unneeded implementation details [10]. This way we humans better understand what the computer will do. Programming languages exist so that we can better explain to our fellow developers what the computer will do.

In a small, one-person, throw-away-prototype project it may be sufficient to just code, but any other project will eventually need documentation [11]. The actual way of how code is documented is less important, as long as all the necessary information is conveyed. The difference that matters is that between hacking code quick and dirty, or being nice to fellow developers by making code easier to understand by investing a little more effort. Source code — originally a medium of communication between man and machine — has become a medium of communication among humans [3].

Documentation (as learning material) communicates background, context, and trial-and-error information. This information is extremely valuable [8], but will get lost if not preserved. Motivating developers to create good learning material is a key to evolvability and survival of PLEs.

4. CONCLUSION

Evolvability is important for the success of a PLE, because it allows to adapt it to new environments, and thus stay fit. PLE evolution happens through a software project. Developers, who realize the change of evolution, require a certain knowledge of the software for this. Evolvability then is the availability and ease of obtaining the necessary knowledge.

After all, a PLE's evolution, i.e. its software project, is a PLE in its own right. This duality between software projects and PLEs is the key to evolvability, and future fitness. Does the software project make a good PLE for its developers? If yes, then a big obstacle to survival is cleared out of the way.

Acknowledgment

This paper was invited by the EFEPLE workshop and supported by the CAPLE project.

5. REFERENCES

[1] W. Cunningham. The wycash portfolio management system. In *OOPSLA Addendum*. ACM, 1992.

[2] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Comm. of the ACM*, 31:1268–1287, November 1988.

[3] G. Dubochet. Computer code as a medium for human communication: Are programming languages improving? In *21st Annual PPIG Workshop*, 2009.

[4] ISO/IEC 9126-1: Software engineering – product quality: Part 1: Quality model, 2001.

[5] A. S. Koch. The people premium. online: `http://www.projectsatwork.com/content/articles/227504.cfm`, October 2005. Projects@Work Journal.

[6] G. T. Lloyd, K. E. Davis, D. Pisani, J. E. Tarver, M. Ruta, M. Sakamoto, D. W. E. Hone, R. Jennings, and M. J. Benton. Dinosaurs and the cretaceous terrestrial revolution. *R. Soc. B*, 275:2483–2490, 2008.

[7] C. R. Prause, R. Reiners, S. Dencheva, and A. Zimmermann. Incentives for maintaining high-quality source code. In *PPIG-WIP*, 2010.

[8] J. Raskin. Comments are more important than code. *ACM Queue*, 3(2):64–62 (sic!), 2005.

[9] B. Selic. Agile documentation, anyone? *IEEE Software*, 26(6):11–12, Nov/Dec 2009.

[10] D. Spinellis. Code documentation. *IEEE Software*, 27:18–19, 2010.

[11] S. R. Tilley. Documenting-in-the-large vs. documenting-in-the-small. In *CASCON*. IBM Press, 1993.

[12] M. Wasmund. Reuse facts and myths. In *ICSE*, 1994.