The Past, Present, and Future of Software Evolution

Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo, CANADA
email: migod@uwaterloo.ca

Daniel M. German
Software Engineering Group
Department of Computer Science
University of Victoria, CANADA
email: dmg@uvic.ca

Abstract

Change is an essential characteristic of software development, as software systems must respond to evolving requirements, platforms, and other environmental pressures. In this paper, we discuss the concept of software evolution from several perspectives. We examine how it relates to and differs from software maintenance. We discuss insights about software evolution arising from Lehman's laws of software evolution and the staged lifecycle model of Bennett and Rajlich. We compare software evolution to other kinds of evolution, from science and social sciences, and we examine the forces that shape change. Finally, we discuss the changing nature of software in general as it relates to evolution, and we propose open challenges and future directions for software evolution research.

1. Introduction: The inevitability of change

As Lehman famously observed, software systems must be able to evolve or they risk an early death [27]. Software systems, like economic theories and Galápagos finches, are embedded in an environment that is also continually changing [9, 17, 42]. For software, the environment has both technical and social characteristics: the deployed system operates within a technical run-time infrastructure, while at the same time users form opinions of the system based on their direct and indirect experiences with it.

As in economics and biology, it is the interactions of the software system with its environment that determine its ultimate success: Does the system provide enough functionality to do its intended job? Is it easy to use? Is the data secure? Can I use it on my cell phone? Is it open source? Is the price reasonable? In turn, the lessons of these interactions comprise feedback into the development of subsequent versions of the system: Which new features were enthusiastically adopted? What absent features are users asking for? In what surprising ways was the product used? How does our system compare to that of our competitors? How easy would it be to port to MacOS? Are users still angry about the spyware incident? As new features are devised and deployed, as new runtime platforms are envisaged, as new constraints on quality attributes are requested, so must software systems continually be adapted to their changing environment.

This paper explores the notion of software evolution. We start by comparing software evolution to the related idea of software maintenance and briefly explore the history of both terms. We discuss two well known research results of software evolution: Lehman's laws of software evolution and the staged lifecycle model of Bennett and Rajlich. We also relate software evolution to biological evolution, and discuss their commonalities and differences. Finally, we survey the evolving road ahead for research into software evolution; in particular, we find that the very nature of a software system has begun to change, and we discuss the open questions and research challenges that lie ahead.

2. Evolution versus maintenance

Historically, both "software evolution" and "software maintenance" date from the 1960s but neither term was adopted into common use until years later.[1] "Maintenance" was first to achieve widespread acceptance, due in part to Canning's well known paper of 1972, "That Maintenance Iceberg" [4], and also to Swanson's typology of maintenance activities introduced in 1976 [41]. We now discuss both terms in more detail.

2.1. Software maintenance

It is Swanson who is responsible for the categories of maintenance kinds — corrective, adaptive, and perfective — that are still in common use today [41]. Swanson's original descriptions of the categories differ somewhat from

[1] Chapin *et al.* have provided a detailed discussion of this topic [7].

most modern formulations,[2] but a reasonably faithful updating of the terms might be given as:

Corrective maintenance are changes that fix bugs in the codebase.

Adaptive maintenance are changes that allow a system to run within a new technical infrastructure.

Perfective maintenance are any other enhancements intended to make the system better, such as adding new features, boosting performance, or improving system documentation.

According to this categorization, corrective and adaptive maintenance tasks do not alter the (intended) outward semantics of the system, while perfective maintenance includes a wide variety of possible changes to the system. Some later taxonomies (e.g., [15]) add a fourth category:

Preventive maintenance are changes made to ease future maintenance and evolution of the system, such as reorganizing internal dependencies to improve cohesion and coupling.

In this view, preventive maintenance tasks should leave the external semantics of the system unchanged, as the main goal is to improve the *internal* design of the system. In practice, however, preventive maintenance activities often lead to improved design ideas that, in turn, result in outward changes.

This last category is somewhat controversial; some consider the term to be ill-defined, and others note that it can be seen as fitting within the domain of perfective maintenance [6]. For example, the 1990 *IEEE Standard on Software Engineering Terminology* defines preventive maintenance as "maintenance performed for the purpose of preventing problems before they occur"[3] [15], while the subsequent 1998 *IEEE Standard on Software Maintenance Terminology* omits the term from the main body of definitions, referring to it only in the appendix [16].

Swanson's taxonomy is based on the intent of the developer toward the system rather than the nature of the changes themselves; others have presented alternative taxonomies of software change. Chapin *et al.* described an expanded classification scheme of 12 categories based on how artifacts and activities were observed to have changed [7]. Mens *et al.* proposed a taxonomy of software evolution based on the characterizing mechanisms of change and the factors that influence these mechanisms [33]. Their classification scheme is organized into the several logical groupings: temporal properties, objects of change, system properties, and change support. Kitchenham *et al.* described an ontology of software maintenance terms in the form of a UML model [22]; their particular goal was to identify factors that might affect the results of empirical studies on software maintenance and evolution.

[2] Swanson also subsequently revised his own definitions [29, 28].

[3] One wonders if the IEEE considers clairvoyance to be a job requirement for software maintainers.

2.2. Software evolution

While the terms "evolution" and "maintenance" are often used interchangeably with respect to software, there are important semantic differences between them. As Parnas and others have pointed out, "maintenance" connotes the idea of keeping an existing system running without changing the design; thus, for physical systems, maintenance often entails replacing worn out parts. However, software does not wear out in the same sense; rather, the impetus for change comes from dissatisfaction with the current system [27, 36]. Thus, the practice of software maintenance requires changing the *design* of the system: by fixing bugs so that the design is correct, by adapting the system for use in new environments, by adding new features, or by improving the internal design so that the system is easier to manage and change. It is innovation, not preservation, that drives software change: a new system, better adapted to its environment, *evolves* from the old one.

2.2.1. Evolution as a perspective on change

Since software maintenance is by nature a design activity, it tends to be more intellectually demanding and thus also riskier than the maintenance of physical systems. Consequently, many have started to use the term "software evolution" as an alternative to describe the various phenomena associated with modifying existing software systems.

This term has several advantages: First, evolution subsumes the idea of *essential* change that maintenance simply does not connote. Maintenance suggests preservation and fixing, whereas evolution suggests new designs evolving from old ones.

Second, maintenance is usually considered to be a set of planned activities; one performs maintenance *on* a system. Evolution, on the other hand, concerns whatever happens *to* a system over time; in addition to the planned activities, unplanned phenomena often manifest themselves also. For example, no one ever plans for interfaces to become bloated, for architectural drift to set in, or for fundamentally new uses of the system to emerge over time, but these are all evolutionary phenomena.

Finally, it can be argued that maintenance and evolution offer different perspectives on the nature of change. Research on software maintenance can be seen as addressing practical, engineering goals: What should we do next? How risky is it? How should we validate our work? Research on

software evolution, on the other hand, can be seen as asking questions of a broader, more scientific nature: How fast can a system grow before it becomes resistant to change? How do internal module boundaries emerge over time? Does open source development differ from industrial software development in measurable ways?

While others may previously have used the term "software evolution", it is Lehman and his collaborators who are usually credited with being the first to consider how the study of software evolution differs fundamentally from that of software maintenance [24]. We now briefly review some of their research contributions.

2.2.2. Lehman's Laws of Evolution

While the term had been used previously, Lehman and his collaborators are generally credited with pioneering the research field of software evolution. Building on empirical studies of the evolution of IBM's OS360 and other large-scale industrial systems [24, 26], Lehman formulated a set of observations that he called his Laws of Evolution. Initially, three laws were postulated, but five more were subsequently added [27, 25]. The laws concern what Lehman called E-type systems: monolithic systems produced by a team within a company that solve a real world problem and have human users.[4] Lehman's Laws may be summarized as:

1. *Continuing change* — A system will become progressively less satisfying to its users over time, unless it is continually adapted to meet new needs.
2. *Increasing complexity* — A system will become progressively more complex, unless work is done to explicitly reduce the complexity.
3. *Self-regulation* — The process of software evolution is self regulating with respect to the distributions of the products and process artifacts that are produced.
4. *Conservation of organizational stability* — The average effective global activity rate on an evolving system does not change over time; that is, the average amount of work that goes into each release is about the same.
5. *Conservation of familiarity* — The amount of new content in each successive release of a system tends to stay constant or decrease over time.
6. *Continuing growth* — The amount of functionality in a system will increase over time, in order to please its users.
7. *Declining quality* — A system will be perceived as losing quality over time, unless its design is carefully maintained and adapted to new operational constraints.
8. *Feedback system* — Successfully evolving a software system requires recognition that the development process is a multi-loop, multi-agent, multi-level feedback system; thus, for example, as a software system ages, it tends to become increasingly difficult to change due to the complexity of both the artifacts as well as the processes involved in effecting change.

Laws 1, 2, 6, 7, and 8 have immediate appeal and offer strong intuition into the nature of evolving software systems. Of these, Law 8 is the most subtle and complex, and is perhaps deserving further elaboration and study; for example, it also implicitly recognizes the role of user feedback in providing impetus for change. Laws 3, 4, and 5 propose hypotheses that are more easily testable by empirical study[5], and perhaps warrant periodic re-examination and re-evaluation as the nature of software and software development also changes.

In recent years, research has started to emerge that challenges some of the laws and their assumptions. For example, studies of open source systems such as the Linux kernel found that they may be able to continue to grow at a geometric rate even after many years of active development; Linux, in particular, seems to violate laws 3, 4, and 5 [13, 38].[6] These studies serve to highlight the changing nature of software and software development: Lehman's studies examined mostly proprietary, monolithic systems developed by identifiable teams of developers within an industrial environment, whereas open source systems and their development appear beholden to very different evolutionary pressures. Furthermore, we note that the very nature of software system has begun to evolve; as the use of services, components, frameworks, and powerful libraries becomes increasingly commonplace, the notion of how systems are designed and composed, what dependence means, and the complexity of change and its management will demand that many of our models will have to be re-examined and rebuilt.

2.2.3. Software evolution and process

One of the side effects of the confusion between the terms "software maintenance" and "software evolution" has been a lack of attention given to process models for continually evolving long-lived systems. The classic waterfall model, for example, is usually depicted as treating maintenance as merely the final step in development [40, 32]. While this may model the *logical* progression of the stages of initial development (first requirements, then design, testing, deployment, and finally maintenance), it does not accurately model the *practice* of development on a long-lived

[4] According to Lehman, E-type systems are *embedded* in the real world. They are to be distinguished from, for example, software that serves purely as infrastructure and implements a standardized interface. Such systems may not endure the same kinds of evolutionary pressures as E-type systems.

[5] Lehman detailed his own studies of several large industrial systems to support these claims [26, 27].

[6] In particular, these studies appear to violate the 1997 version of the laws; Lehman and Ramil have since revised them to take these results into account [25].

system. More recent models consider software development to be essentially *iterative* and *incremental* [3, 18, 30]. That is, development is composed of a series of mini-cycles of requirements, design and implementation, and testing; evolution and maintenance do not exist as explicit stages *per se*, rather the whole iterative and incremental lifecycle models the evolution of the system.

Bennett and Rajlich have presented a descriptive model of software evolution — called the *Staged Model* of maintenance and evolution — that summarizes many of these ideas [2]. Their model divides the lifespan of a typical system into four stages:

1. *Initial development* — The first delivered version is produced. Knowledge about the system is fresh and constantly changing; in fact, change is the rule rather than the exception. Eventually, an architecture emerges and stabilizes, ideally with a view to likely future development.
2. *Active evolution* — Simple changes are easily performed, and more major changes are possible too, although the cost and risk are now greater than in the previous stage. Knowledge about the system is still good, although many of the original developers will have moved on. For many systems, most of its lifespan is spent in this phase.
3. *Servicing* — The system is no longer a key asset for the developers, who concentrate mainly on maintenance tasks ("keep it running") rather than architectural or functional change. Knowledge about the system has lessened and the effects of change have become harder to predict; the costs and risks of change have increased significantly.
4. *Phase out* — It has been decided to replace or eliminate the system entirely, either because the costs of maintaining the old system have become prohibitive or because there is a newer solution waiting in the wings. An exit strategy is devised and implemented, often involving techniques such as legacy wrapping and data migration. Ultimately, the system is shut down.

We note that the Staged Model is primarily descriptive rather than prescriptive; that is, it describes the typical activities that one would observe at the various stages of a software system's life but does not attempt to prescribe which activities ought to be performed. Its primary contribution is thus aimed at improving understanding of how long-lived software evolves, rather than aiding in its management.

3. On the nature of evolution

Having examined the nature and history of software evolution and its research, we now turn our attention to other ideas of evolution. Within the realm of science, "evolution" usually connotes the study of biology, as this is where the modern use of the term has come to have strong meaning. But what is evolution, exactly? Are there useful lessons that the evolution of biological species can teach us about the evolution of software systems? To answer these questions, let us briefly review the basics of biological evolution.

3.1. Biological evolution in a nutshell

Darwin's memorable phrase to describe biological evolution was "descent with modification", by which he meant that — over many generations — new species with new structural traits can and do come into existence from older ones [19]. A more modern definition may be given as "inheritable change in a population of individuals over time" (see e.g., [12]). This definition bears some careful discussion. First, we note that biological evolution pertains only to *inheritable* change, that is, modifications to genetic encoding that can be passed on to offspring. Second, biological evolution concerns change over time in the traits of *groups* of individuals such as species and subspecies[7]; a population is considered to have evolved if the set of relative frequencies of inheritable traits has changed, or if new traits have emerged.

At this point, it is useful to introduce two more terms that we will revisit later [9]:

- The *genotype* of an individual is its specific genetic encoding; in the case of species that reproduce sexually, the genotype of an individual is entirely determined by that of its parents, each of whom contribute half of their own genotype to the offspring.[8]
- The *phenotype* of an individual is the sum of all of its observable characteristics; roughly speaking, the phenotype is determined by the genotype plus interactions of the individual with its environment over time.

For example, a person's eye colour (a phenotypic trait) is due entirely to their genotype, while their height (also a phenotypic trait) is due to a combination of genotype and interactions with their environment as they were growing up (e.g., malnutrition can stunt your growth). And a tattoo is a phenotypic trait that has little, if any, genotypic origin except insofar as your brain may be favourably disposed to the idea that tattoos are desirable.

The important point to note here is that non-genotypic change is not inheritable. Instead, the term "culture" is used to denote phenotypic traits — such as ideas, traditions, and fashions — that are passed on through the generations but are not "in the genes" [9].

[7] The term "development" connotes change in an individual over time, such as a human embryo growing into a child, and later into an adult [5].

[8] When mistakes are made in copying genetic information the genotype of offspring may differ from both parents; this is called *mutation*.

3.1.1. The mechanics of biological evolution

There are two key ideas to the working process of biological evolution:

1. Various mechanisms of change act to increase and decrease the relative frequencies of inheritable traits.
2. Evolution takes places within an environment, which also changes over time.

The mechanisms of change include *mutation*, which introduces new genotypic values, which in turn may introduce new phenotypic traits; *natural selection*, in which individuals having genotypic values that provide phenotypic advantage tend to reproduce more often than individuals that do not; *genetic drift*, which occurs when some genotypic values come to dominate others by probabilistic chance alone; and *gene flow* where individuals from one population may provide new genotypic values to a neighbouring population, resulting in new genotypic combinations [31].

The notion of evolution being embedded in an environment is also important. Since individuals change (and are changed by) their environment what succeeds today may not work a few generations from now; the environment may be very different then [31].

3.2. The evolution of software

While biology has the best studied and most concrete body of knowledge concerning evolution, it is a simple fact that evolution is a pervasive phenomenon. While the ontologies, pressures, and mechanics may differ, there can be no doubt that, at least in some sense, culture evolves as well. Dawkins, in his landmark book on evolutionary biology *The Selfish Gene*, coined the term "meme" to connote a unit of culture that can evolve over time, analogous to a gene in biology [9]. While the study of memes is somewhat controversial — due mainly to subsequent scholarship that is perceived as lacking rigour — it is also beyond the scope of this paper. Instead, we shall now proceed to sketch out how software evolution is similar to — and different from — biological evolution.

3.2.1. Definitions

We will frame our model around the idea of a version of a software system being an individual, with the totality of all deployed versions of a given system comprising a population. The source code of a system version seems a good fit for the label *software genotype*, as the phenotypic software system that users interact with is entirely derived from processing the source code though a compiler and related tools. We can then examine the notion of a *software phenotype*: an executable program deployed within both a technical run-time infrastructure and a social user environment.

Now we come to our first important difference: In biology, non-genotypic change typically has no influence on evolution.[9] The cause and effect relationship between genotype and phenotype is remote and indirect; it is hard to feed information back into the design process since the design process — the creation of new genetic ideas — is imprecise and due mostly to chance i.e., mutation.

Software, on the other hand, is designed by humans, directly and concretely. We can easily observe the effects of our design efforts, both within a technical sense — How fast is it now? Do the new features work properly? — and a social sense — Do users like it any better than last time? — and use this information as explicit input into the design of the next version. That is software, unlike biology, is dependent on intelligent design for change.

3.2.2. The pressures on software to evolve

In the software world, we can also observe external mechanisms that encourage change resulting in new "individuals": requests for new features, the existence of new platforms, and the desire to improve quality attributes such as performance. There are also forces that tend to limit change: market saturation, political and legal concerns, and the complexity of the software system itself. Finally, we repeat the observations that software systems are also embedded within an environment; as Lehman noted in his eighth law, that environment forms a complex feedback loop that affects the system's ability to further evolve.

4. The road ahead: Challenges + opportunities

Having surveyed the past of software evolution research, let us now turn our attention to the state of the world, and survey the challenges — and opportunities — that lie in the road ahead.[10] We see six main areas of interest, each with its own set of challenges: model building and empirical studies, open source development, evolutionary pressure and emergent design, improving the collective memory of software developers, the emergence of software "ecospheres", and improved understanding of economic tradeoffs and risks.

4.1. Model building and empirical studies

Challenge: *How do we proceed?*

[9] To be careful, phenotypic change can influence the environment, which can in turn influence which genotypic values are selected for success. For example, in a human society that values tattoos, individuals who are more favourable disposed towards getting a tattoo may have more reproductive success than those who are not. Obviously, tattoos themselves are not inherited by offspring, but any genes that encourage open mindedness with respect to tattoos may be.

[10] Mens and Demeyer have recently published a survey of current research trends in software evolution [34].

One of the major goals of software evolution research is to devise models that can be used to describe the past, present, and future evolution of a software system. Ideally, we would like such models to address both qualitative and quantitative properties of systems, and also to have some predictive power both in the short and long term.

However, the creation of such models requires that significant background work be done first. Building scientific understanding for a domain typically follows a multi-step process: a) observation of phenomena, b) building of models and hypotheses, c) testing of hypotheses, and then d) optimizing models and retesting. In the domain of software evolution, the first stage has been (and continues to be) performed through empirical studies, such as exploratory case studies of single system versions, longitudinal studies of single systems, and comparative studies of multiple systems and multiple versions.

However, there are a variety of problems endemic to trying to generalize from these kinds of studies. First, and most vexing, is the fact that software systems — even from the same domain — tend to be much more different than alike in their details. For example, while it is likely that most relational database systems have a similar broad architecture (a client interface, a query engine, a storage manager, *etc.*), it is unlikely that will be much commonality at the level of source code. Systems from a similar application domain at a similar point in time may respond to similar evolutionary pressures, but the measurable impact of these pressures is recorded on the unique codebases of the individual systems.

The second problem (which is discussed in more detail in the next section) is of accessibility and generalizability. While there are now a very large number of open source systems that can be studied freely, the vast majority of commercial software is still created as proprietary and closed source. Developing sound models and hypotheses requires the study of many different software systems, and the trade secret nature of software makes this difficult. While organization do sometimes permit researchers to study their systems, such work is typically covered by a non-disclosure agreement (NDA), which limits what information can be made public. There does exist a significant body of work of studies of such systems (see e.g., [11, 26]), but many of the details are hidden and typically the software artifacts are not easily available to others to perform follow-up studies.

A third problem (discussed in more detail in Section 4.5) is that the very nature of what a system is — and how it is designed, developed, and deployed — is evolving; consequently, it is hard to know how to model (and therefore measure) what a system is.[11] Previously, it was enough to study monolithic systems that depended only on standard libraries, but today development artifacts comprise more than just source code: XML customization files, web interfaces, user scripts, multimedia images, bug tracking logs, version control logs, email messages, and user forums are commonly tracked and stored in repositories. Web services, components, frameworks, toolkits, and libraries make it easier to build powerful applications by adding a relatively small amount of code. And some systems, such as Apache and GIMP, have a plug-in architecture where the system provides a basic infrastructure, but is mostly unaware of the myriad of small utilities that users populate the deployed system with. It is hard to know just what to measure and how. Even within a monolithic system, there are basic framing questions that must be addressed. For example, more than 60% of the source code in the Linux kernel consists of drivers, which interact with the rest of the system through a simple, well defined interface which in turn effectively isolates the complexities of the system and the drivers from each other [13].

Empirical study is the cornerstone of software evolution research. To build our models and hypotheses we need to analyze many different types of systems from different domains produced using different processes, while at the same time we must realize that each system will have its own unique social history and set of design peculiarities that will strongly influence any kind of systematic measurement.

Any interesting empirical study will be prone to objections: Why didn't you study more systems? more versions? more domains? more closed source systems? *etc.* It is unrealistic, of course, to comprehensively study all possible software systems to determine if a certain property always holds; we should not expect to be able to do so. Instead, the challenge of empirical study is to investigate *both* the general — to see if well known properties hold — and the particular — to seek to better understand unexpected phenomena.

4.2. Open source vs. proprietary software

Challenge: *Can we create general laws of software evolution from studying mainly open source systems?*

As discussed above, studies of proprietary software have usually been subjected to the restrictions such as NDAs that limit the details that can be published. This poses two fundamental problems: first, many empirical studies cannot easily be replicated or extended; and second, so much detail is typically left out of the study that we are asked to take a lot on faith. These studies ask us to trust that the researchers have done the study properly, and also to accept the results without fully understanding the design or even domain of the target system.

Some open source systems have been around for more than a decade, and have preserved their histories at the same time. The availability of such open source systems initiated a golden age of research in software evolution, where any

[11] Bennett and Rajlich have made a similar observation [2].

researcher could suddenly have access to rich histories of software systems to study. Furthermore, empirical studies of open source systems can now be replicated to either confirm or revise previous results, or present discoveries that might have been previously overlooked.

While the recent explosion in the amount of empirical work on open source systems is good news to the research field, the question must be asked: Is the evolution of open source software a good indicator of the evolution of software in general? If not, what are the differences?

We *do* know that the goals, processes, economics, and even politics of open source software development are strikingly different from that of most proprietary systems [14, 23]; for these reasons, the study of open source software evolution is a worthy topic on its own. What we must ask is: How different are these goals, processes, *etc.*, and how do they affect the resulting systems? The preliminary results are mixed: some studies have found significant differences [13, 38], while others have not [35, 37].

Finally, we note that just as there are a wide variety of industrial software development processes, there are also many development models that fit under the umbrella of "open source": some projects are driven mainly by part-time enthusiasts, and decisions are made by informal consensus; some are initiated by companies but then donated (i.e., abandoned) to the user community; and some are supported by organizations that spearhead and control official development efforts. This last category — which includes such well known systems as Mozilla, MySQL, Eclipse, and Mono — is particularly interesting as the organization usually hires and manages most of the key developers and so functions within a kind of hybrid process model: the system is designed, developed, and managed by a formal organization, but development is "out in the open". Perhaps these kinds of systems hold the key to an improved understanding of the differences and similarities between open source and proprietary software.

4.3. Environmental pressure and emergent design

Challenge: *Can we anticipate how a system will respond to environmental pressures?*

Is it possible to accurately predict how a system will evolve? To some degree, the answer is often yes. For example, a system might be designed in such a way that some classes of new features can be implemented in a repeatable and well defined way e.g., frameworks, device drivers that implement a standard interface, systems with a plug-in architecture. In such cases, the organization has successfully designed for change [36]; it is clearly planning the future evolution of the system and had prepared its resources — human and technical — to cope with it.

Sometimes future evolution is obvious even at a distance, because the organization understand the technical and social environment in which the system is embedded. For example, operating system developers needs to track the technical space — e.g., the USB standard is upgraded from one version to another, hence the operating system is expected to implement support for the new version — as well as the social one — e.g., USB webcams are dropping in price, and users like them, pressuring the operating system to include drivers for them.

How far ahead can an organization *successfully* plan the evolution of its software system? The key is how is "successfully" defined. It is always possible to plan ahead, and without regard to the changing environment under which the system runs, continue to evolve in a preplanned manner. The likelihood of the software system evolving as planned will depend on how resilient it is to external pressures that lead in a different direction.

Perhaps one of the most decisive factors that affect planned evolution is the economic conditions in which the software systems exists. If it has a tight monopoly on its market, such as in-house developed applications that are expected to satisfy the needs of the same organization's users, then it can probably get away with poorly adapting to the evolving changes in its environment as long as it satisfies certain fundamental needs of its users. Most of us have used a software system that is antiquated, hardly satisfies our needs, and could be improved, but it is *the* application that our organization provides and we are forced to use it if we want to do our work.

However, it is often the case that the software system lives in a highly competitive market and the user will choose a better one if his or her needs are not satisfied and if the costs of migration are not high. The development team needs to understand its current and future deployment environment, both technical and social, and try to plan ahead. Sometimes "better" is a highly subjective adjective and has nothing to do with the technical prowess of a system.

Evolution is opportunistic: an organization may decide that its system can be extended to satisfy new user needs, and in the process, to grow its market. However, opportunistic evolution is hard to plan; the challenge is to create a system that can be adaptable to initially unforeseen circumstances without crippling it through over-engineering. Extensibility has a cost.

Evolution is strongly influenced by negative properties of the system, such as architectural drift, feature creep, and progressive hardware dependence [36, 27]. These phenomena are particularly pernicious, as they tend to go unnoticed at first but accrue over time leading to a deterioration in the system's design and its ability to respond to desired changes. And if they are detected after much time has passed, they may be risky and expensive to fix.

Thus, another challenge for software evolution researchers is to develop techniques to detect such problems as quickly (and unobtrusively) as possible. Software metrics have traditionally been suggested for this purpose; however, in practice developers often consider them uninformative and prone to misuse by managers. Evolution patterns and anti-patterns seem more promising (see e.g., [10]); each pattern describes a set of preconditions — technical, social, economic — and the potential outcomes of its application. It is then up to the development team to determine if and when a pattern should be applied.

4.4. Creating a collective memory

Challenge: *How can we unify a sea of disparate development artifacts to improve our understanding of how systems evolve?*

Mining software repositories (MSR) is an emerging research field that aims to synthesize knowledge about development from a myriad of artifacts that may appear, at first, to be only loosely related.[12] For example, while version control systems offer easy access to the source code of a system, it is often hard to relate a change that implements a particular bug fix back to the original bug report. While tools — or, more often, developer conventions — do exist to aid in recording these kinds of relationships, often the information is simply not there. The field of MSR aims to recover latent relationships between many kinds of software artifacts: CVS check-ins, bug reports, test suites, email messages, user forum postings, and various kinds of documentation. This is done using a variety of techniques, such as parsing, data mining, various kinds of analysis, and the use of heuristics. MSR researchers can be likened to software archaeologists who sift through remnants of the past trying to piece together a coherent history of a software project.

Developers are starting to realize the value of these historical artifacts and are taking steps to preserve them. For example, a version control system is a necessity for collaborative work, and many projects have begun to switch from the CVS system to that of Subversion. The simplest way to do this is to establish a new Subversion repository with a current version of the system source code. However, many projects have gone to the trouble of migrating the entire version control history to the Subversion; clearly, they see value in having easy access to history.

Developers are also now recognizing the value of correlating information. For example, one can explicitly link a commit that fixes a particular bug back to the original bug report by adding the defect number in the log of the commit. Such acts create traceability — it is possible to know why a certain change was performed — and facilitate the automatic recovery of relationships between these two different types of information — it is possible to scan logs of commits to find those that fix particular defects. Unfortunately, this practice is not yet the norm.

One challenge for software evolution researchers is to persuade developers about the value of cross-referencing information as it is created, when the knowledge is still fresh in their minds. Developers need to be more aware of — and proactive about — how they can help to create a collective memory for their project with a minimum of effort. Such a record can serve as a kind of public diary that can be later perused by others who are interested in piecing together the history and evolution of the system in question.

And this leads to another challenge: software systems need documenters who are also historians of the system. Their role includes documenting not only the system's behaviour, but also the evolution of the system from a more holistic point of view.

Of course, for the roles of developer-diarist and documenter-historian to be filled there must be clear return in value for the effort. And this becomes a chicken-and-egg problem: researchers need to correlate information to be effective, but developers are unlikely to invest heavily in recording such relationships unless they are first proven to be useful. The challenge for research community is to convince developers that, if given better data, the results will be worth the extra effort.[13]

4.5. The emergence of software ecospheres

Challenge: *How can we better understand systems that are deeply embedded in a rich ecology of interdependent applications and services?*

As mentioned in Section 4.1, component- and service-oriented software development is resulting in the creation of applications that rely on other systems — most of them under the control of other organizations — for large chunks of their functionality. The developers of such a system might not know the exact versions of a component, service, library, or framework that will eventually be used by a specific instance of the system they develop. The software phenotype of these software systems has become complex and varies significantly from one deployment to another, to the point that it can no longer be controlled by the development team.

Similarly, the evolution of such a software system is dependent on the evolution of any other system it depends

[12] Kagdi *et al.* have published a survey of recent research in this field [20].

[13] Arkley and Riddle have discussed the related problem of the cost/benefit analysis of traceability with respect to requirements modelling [1].

upon [39]. If the component is no longer available, or is determined to be of unacceptable quality, another will have to be found to replace it and the system adapted to the new part. Of course, one of the key goals of service-oriented computing is to relieve developers from having to worry about such compatibility issues; coupling from application to service is, by design, loose. But this looseness also comes with risks; anyone who has maintained a Linux system is familiar with the problem of "update hell", when the installation of new versions of libraries often breaks the installations of existing applications.

The challenge here for software evolution research is to provide better models for these kinds of systems and their dependent parts, and to model the evolution of this ecology of applications rather than considering each as an isolated and independent system.

4.6. Economic trade-offs and risk

Challenge: *How can we incorporate understanding of economics and risk into the theories of software evolution?*

Sometimes we forget that as software engineers our objective is not to create a perfect software system but rather the best system given a set of resources: How much money can we spend? How skilled are the developers? What is the competition like? How much time is available? What is the projected lifespan of the system? How difficult is the technical task? What existing assets can we make use of? That is, we are often required to make decisions that balance economics and technical risk.

Consider a more concrete example. Researchers sometimes debate how harmful source code cloning (i.e., cut-and-paste coding) is to the design of software systems [21]. They may point out that there is a more elegant design than cloning, that cloning leads to code bloat and inconsistent maintenance, *etc.* yet developers continue to clone. Why? It is because developers judge that they save resources by doing so and that — rightly or wrongly — the trade-off justifies the decision to clone. In so doing, they are making an economic decision that includes risk analysis. One might also ask: Under which circumstances would they choose not to clone? What factors drive their decision? Do they primarily benefit the individual, the software project, or both? Avoiding cloning usually involves refactoring, and we understand the advantages of refactoring. But refactoring is not free; it requires time and effort, and refactoring also risks introducing new defects that will have to be fixed [8].

We believe that there is a cultural gap here. Professional developers are often very skilled at this kind of analysis, especially as it regards their own systems; they also often understand the value of a quick and dirty (and ultimately disposable) prototype. Researchers, on the other hand, sometimes have a hard time understanding why developers would not just design the cleanest, most elegant system from the start and then zealously keep it that way.

A challenge for software evolution research is to be able to model and quantify such questions: "Given the type of system that I am building, its expected lifespan, and the resources available now and in the long term, what is the cost/benefit analysis for each of the potential decisions I must take today?"

5. Conclusions

In this paper, we have argued that software evolution offers a perspective that is distinct from the traditional views of software maintenance: it connotes the idea of essential change within an environment, it naturally encompasses the concepts of both planned and unplanned phenomena, and its study offers insights into questions of both science and engineering.

Software evolution research is still a young field, and it continues to change its focus and even its underlying concepts. We know that software must evolve but we are still learning how to model this evolution, particularly in the increasingly complex environments in which software is designed and deployed. There are many open questions to be explored and several research challenges lie in the road ahead, including: the problems of model building and empirical study, the applicability of studying open source development, the modelling of emergent design, improving the collective memory of developers through better artifact linkage, the emergence of software ecospheres, and the study and modelling of economic trade-offs and risk.

Acknowledgements

We are grateful to Dan Berry, Dan Brown, Ahmed Hassan, Ric Holt, Manny Lehman, Václav Rajlich, Juan Fernandez-Ramil, Gregorio Robles, and Davor Svetinovic for helpful comments and previous conversations.

References

[1] P. Arkley and S. Riddle. Overcoming the traceability benefit problem. In *Proc. of the 13th IEEE Intl. Conference on Requirements Engineering*, Paris, France, September 2005.

[2] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In *Proc. of the Conference on The Future of Software Engineering*, Limerick, Ireland, May 2000.

[3] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5), May 1988.

[4] R. Canning. That maintenance 'iceberg'. *EDP Analyzer*, 10(10), 1972.

[5] S. B. Carroll. *Endless Forms Most Beautiful: The New Science of Evo Devo and the Making of the Animal Kingdom*. WW Norton & Company, 2005.

[6] N. Chapin. Do we know what preventive maintenance is? In *Proc. of 2000 IEEE Intl. Conference on Software Maintenance*, San Jose, CA, October 2000.

[7] N. Chapin, J. Hale, K. Khan, J. Ramil, and W. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1), January/February 2001.

[8] J. R. Cordy. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *Proc. of* 11^{th} *IEEE Intl. Workshop on Program Comprehension*, Portland, OR, May 2003.

[9] R. Dawkins. *The Selfish Gene*. Oxford Univ. Press, 1976.

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003. Freely available at `http://www.iam.unibe.ch/~scg/OORP/`.

[11] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. on Software Engineering*, 27(1), January 2001.

[12] D. Futuyma. *Evolutionary Biology*. Sinauer Associates, third edition, 1998.

[13] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of 2000 IEEE Intl. Conference on Software Maintenance*, October 2000.

[14] G. Hertel, S. Niedner, and S. Hermann. Motivation of software developers in open source projects: An Internet-based survey of contributors to the Linux kernel. *Research Policy*, 32:1159–1177, 2003.

[15] The Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology*, 1990. IEEE Standard 610.12-1990.

[16] The Institute of Electrical and Electronics Engineers. *IEEE Standard for Software Maintenance*, 1998. IEEE Standard 1219-1998.

[17] J. Jacobs. *The Nature of Economies*. Modern Library, New York, 2000.

[18] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[19] S. Jones. *Darwin's Ghost: The Origin of the Species Updated*. Random House, 2000.

[20] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2), March/April 2007.

[21] C. J. Kapser and M. W. Godfrey. 'Cloning considered harmful' considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 2008. To appear.

[22] B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 11(6), December 1999.

[23] K. R. Lakhani and B. Wolf. *Perspectives on Free and Open Source Software*, chapter Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects, pages 3–21. MIT Press, 2005.

[24] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press, 1985. Freely available at `ftp://ftp.umh.ac.be/pub/ftp_infofs/1985/ProgramEvolution.pdf`.

[25] M. M. Lehman and J. Fernandez-Ramil. Software evolution and feedback: Theory and practice. In N. H. Madhavji, J. Fernandez-Ramil, and D. E. Perry, editors, *Software Evolution*. Wiley, 2006.

[26] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *Proc. of the 1998 IEEE Intl. Conference on Software Maintenance*, Bethesda, Maryland, November 1998.

[27] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution — The nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium*, Albuquerque, NM, November 1997.

[28] B. Lientz and E. Swanson. *Software Maintenance Management*. Addison-Wesley, Reading, MA, 1980.

[29] B. Lientz, E. Swanson, and G. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6), June 1978.

[30] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[31] E. Mayr. *What Evolution Is*. Basic Books, 2001.

[32] S. McConnell. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, 1996.

[33] T. Mens, J. Buckley, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), September 2005.

[34] T. Mens and S. Demeyer. *Software Evolution*. Springer-Verlag, 2008.

[35] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[36] D. L. Parnas. Software aging. In *Proc. of the* 16^{th} *Intl. Conference on Software Engineering*, Sorrento, Italy, May 1994.

[37] J. W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30(4):246–256, April 2004.

[38] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Proc. of the Eighth Intl. Workshop on Principles of Software Evolution*, Lisbon, Portugal, September 2005.

[39] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: A case study of a large software compilation. *Empirical Software Engineering*, 2009. To appear.

[40] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proc. of WESCON*, November 1970. Also appears in *Proc. of 1987 Intl. Conference on Software Engineering*.

[41] E. B. Swanson. The dimensions of maintenance. In *Proc. of the* 2^{nd} *Intl. Conference on Software Engineering*, San Francisco, CA, October 1976.

[42] J. Weiner. *The Beak of the Finch: A Story of Evolution in Our Time*. Vintage Books, New York, NY, 1995.