Prolonging the Aging of Software Systems

Constantinos Constantinides
Concordia University, Canada

Venera Arnaoudova
Concordia University, Canada

INTRODUCTION

The evolution of programming paradigms and languages allows us to manage the increasing complexity of systems. Furthermore, we have introduced (and demanded) increasingly complex requirements because various paradigms provide mechanisms to support their implementation. As a result, complex requirements constitute a driving factor for the evolution of languages which in turn can support system complexity. In this circular relationship, the maintenance phase of the software life cycle becomes increasingly important and factors which affect maintenance become vital.

In this chapter we review the notions of software aging and discuss activities undertaken during maintenance. We also discuss challenges and trends for the development of well-maintained systems as well as for aiding in the maintenance of legacy systems.

BACKGROUND

Aging in Software

In the literature, many authors tend to have drawn analogies between software systems and biological systems (ISO/IEC 12207:1995(E); Jones, 2007; Parnas, 1994). Two such notable examples are the widely used notions of aging and software life cycle, implying that we can view software systems as a category of organisms. This analogy is convenient because it creates certain realizations about software. First, we note that systems exist (by operating as a community of intercommunicating agents) inside a given environment. Furthermore, much like their biological counterparts, they evolve (to adapt to their environment) and they grow old. Finally, when speaking of the life cycle of software, we also imply the unavoidable fact that software systems eventually die.

However, the causes of software aging are very different from those of biological organisms or those that cause aging in other engineering artifacts. Unlike biological organisms (such as humans) software systems are not subjected to fatigue or physical deterioration. Unlike other engineering products (such as machinery and structures), software systems are not subjected to physical wear caused by factors such as friction and climate. Aging in software systems is predominantly (but not always) caused by changes that take place in their surrounding (operating) environment.

In his seminal paper on aging, author David Parnas (1994) describes two causes of software aging: The first factor, referred to as lack of movement, is the failure of owners to provide modifications to the software in order to meet changing needs (requirements) of its environment which results in end-users changing to newer products. The second factor, referred to as ignorant surgery, is the careless introduction of changes in the implementation which can cause the implementation to become inconsistent with the design, or even to introduce new bugs. This latter factor is associated with two significant implications: The first is a bloating of the implementation, resulting in a reduction in performance (memory demands, throughput and response time). This weight gain makes new changes difficult to be introduced quickly enough to meet market demands. The second implication is a phenomenon known as bad fix injection (Jones, 2007), which refers to the introduction of errors during maintenance resulting in a decrease in reliability. As a result, software systems become unable to be competitive in the market, thus loosing customers to newer products.

Measures to Prolong Aging

Certain measures are proposed in the literature (Parnas, 1994) to prolong aging such as:

1. The quality of documentation can be upgraded (retroactive documentation). For example, reverse engineering is a model transformation activity which can read implementation and produce an up-to-date design model.
2. Since we cannot really predict the actual changes, predictions can be made about the types of changes, such as changes to the graphical user interface. Parnas (1994) recommends re-organizing the software in such a way so that elements which are most likely to change, such as the user interface, are confined to small amounts of code (retroactive modularization). A similar view is shared by Fayad and Altman (2001) through an architectural pattern to support software stability where the architecture is built around two notions, conceptualized as two concentric circles. In

the inner circle, we have aspects of the environment that will not change. These aspects will constitute a stable core design (and thus a stable software product). In the outer circle, or periphery, we define a design which will allow changes to be introduced.

3. Eliminate components which are of very low quality (amputation).
4. Eliminate redundant components (major surgery).

These measures take place during the period of operability of a software system and are explicitly treated as a separate phase of the software life cycle which is discussed subsequently.

Software Maintenance

ISO/IEC and IEEE define maintenance as the modification of a software product after delivery to correct faults, improve performance (or other attributes) or to adapt the product to a modified environment (ISO/IEC 14764:2006(E); IEEE Std 14764-2006). The importance of maintenance lies on the following observations: (1) Surveys indicate that it is an activity which tends to consume a significant proportion of the resources utilized in the overall life cycle (consequently consuming a large part of the costs) and (2) Reliable changes to software tend to be time consuming. Prolonged delays during software change may result in a loss of business opportunities.

The objective of maintenance is not to stop the unavoidable effects of aging, but to provide techniques and tools to understand its causes, to limit its effects and to prolong the life of software systems.

Maintenance is not a uniform activity and as the type of required changes may vary, four different types of maintenance can be identified which are also defined in the ISO/IEC; IEEE international standard. Corrective maintenance includes all changes made to a system after deployment to correct problems. Preventive maintenance includes all changes made to a system after deployment to correct faults in order to prevent failures. Adaptive maintenance includes all changes made to a system after deployment to address new requirements. Perfective maintenance includes all changes made to a system after deployment to support operability in a different (software or hardware) environment. The ISO/IEC; IEEE international standard provides a classification scheme by grouping the former two under correction and the latter two under enhancement. Adaptive and perfective types of maintenance are shown in the literature to consume a significantly large proportion of all maintenance effort. Corrective and preventive types of maintenance are reported to consume a relatively small proportion of the overall maintenance effort. It is important to note, that the different types are not mutually exclusive but rather they can be combined concurrently to be mutually supportive.

Also, the four maintenance types do not refer to single activities. Jones (2007) lists 23 discrete topics which involve a modification of an existing system often described under maintenance.

Stages of Maintenance and the Staged Model of the Software Life Cycle

In the literature, Bennett and Rajlich (2000) define a model whereby a software system undergoes distinctive stages during its life: Initial development, evolution, servicing, phase-out, and closedown.

Initial development would produce a deployable system (the first operating version). After deployment, evolution would extend the capabilities of the system, possibly in major ways. Once evolution is no longer viable, the software would enter the servicing stage (often referred to as maturity, or most commonly legacy stage). As the term suggests, only small changes are possible during this stage.

Maintainers often encounter what Bennett (1995) describes as the legacy dilemma: On one hand, a system (or component) is valuable and replacing it may not be a viable (cost effective) solution (e.g., large volumes of data may have to be converted). On the other hand, the cost of maintenance is becoming high and requests for changes cannot be sustained. When faced with legacy systems, organizations have to adopt a strategy which is based on economics (i.e., cost of coping with the current system vs. the cost of investment of improvement) and management (e.g., a replacement system would normally require training of end-users). Table 1 summarizes various options based on two factors, namely business value and quality (adopted from Sommerville, 2007).

Finally, once servicing is no longer viable the system enters a phase-out stage where deficiencies are known but not addressed. At closedown, the system is withdrawn from the market. In an alternative model (versioned staged model), during evolution a version is publicly released and subsequently enters the servicing stage whereas the system continues to evolve in order to produce the next version.

Central to any maintenance activity is the notion of change, discussed in the next subsection.

SOFTWARE CHANGE

Whether new requirements are introduced or existing requirements are refined or dropped, the notion of change is a fundamental activity during evolution and servicing. Bennett and Rajlich (2000) describe a change mini-cycle as one which involves a number of activities: request for change, planning phase, change implementation, verification, and documentation update.