Software maintenance and evolution

Introduction to organizational and process challenges

Yazid Hamdi
ISR Software Engineering Master Programs
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
yazid@cmu.edu

***Abstract*— Why are maintenance and evolution an important subject in Software Engineering? As Pressman [3, p797] puts it, "it is not unusual for a software organization to expand as much as 60 to 70 percent of all resources on software maintenance". April and Abran also state that 50% to 90% of software lifecycle costs come from software maintenance [2, p3]. Barry Boehm is also quoted for stating that for every dollar spent on development, two will need to be spent on maintenance [2, p2]. In an effort to explain the origin of these high figures, Pressman explains that (1) traditionally, older code has more focus on optimizing performance due to the limitations of technology at the time it was written (CPU, memory, storage) which results in minimalistic code not written to be readable by humans, and (2) the mobility of software engineers makes it so that the original developers of a software will never be available to maintain the code when problems start to be reported or changes need to be made. Although the first reason seems to have been made obsolete by the advances in technology and software engineering practice since the 1980s, the second reason is still valid and weighs heavily on software organizations' ability to handle defect fixing and changes to software.**

It is therefore clear that there is a consensus on how costly (and as a consequence important) software maintenance and evolution are. The fact that most of the cost goes to maintenance and evolution raises questions about the reasons behind it, the approaches taken to minimize the cost, and the business and organizational models that are used to manage software maintenance and evolution effectively.

In this paper, we present a review of research and practice results about software maintenance and evolution. The scope is limited to management methodology throughout activities ranging from first customer support to software product reengineering.

We start by providing definitions and context of software maintenance and evolution. We then cover some of the major challenges of software maintenance and evolution, before moving on to some elements of existing knowledge on how to deal with them. To finish, we explore the influence of design and architecture on software maintenance and evolution, then we present some reflections for modern software engineering teams on how to best manage that activity based on the knowledge reported in this paper.

Keywords—software engineering; maintenance; evolution; process; cost; management; services; support

I. Definitions and context

A. Assmuptions

The first assumption in this paper is that the primary software being analyzed here is enterprise software, since most of the literature deals with maintenance and evolution in that precise context for historical reasons: reflections on software maintenance and evolution date back to times when software was exclusively being used by businesses. There is a lack of literature about more modern types of mainstream software such as app-store mobile apps development, probably since process discipline is not a popular engineering approach for that type of software. The level of complexity and the types of challenges differ considerably between enterprise software and mobile apps, as an example.

We are also assuming that the "software" is commercial software, and excluding Free and Open Source Software from our analysis since the dynamics and processes of the engineering of FOSS software have fundamental differences with those of classic commercial software.

The final assumption is that by "software" we designate mostly purpose-built software rather than Commercial Off The Shelf (COTS) combinations and integrations. This is an assumption made in most of the literature because of subtle process and operational differences between software developed from scratch and COTS.

B. High level perceptions of software maintenance and evolution

Pressman [3, p795] cites Manny Lehman and his colleagues' "unified theory for software evolution", which takes form in 8 largely cited "laws" [1, Table 2.5][3, p796][6] that characterize different aspects of software maintenance and evolution. We think that they are a good starting point for any reflection about software maintenance and evolution since they were developed over a long period of time (the earliest was established in 1974 and the latest was established in 1996), which provides an overview of almost 40 year's lessons learned about this subject. These laws are established for Lehman's E-type systems as per the SPE taxonomy described in detail by Tripathy et al. [1, p 46-50], which means stands for "Evolving" software, as opposed to "Specified" and "Problem" software. We tried to

classify these laws into three categories depending on the information they carry about software maintenance and evolution [1, Table 2.5][3, p796][6]:

- Laws describing the challenges:
    - The law of increasing complexity: evolving software systems become more complex over time unless effective work is done to limit that
    - The law of declining quality: the quality of an evolving software system will appear to be declining unless effective adaptation to operational environment changes is performed
- Laws presenting recommended practices or stating practical goals:
    - The law of continuing change: software must evolve continually otherwise it will become unsatisfactory
    - The law of conservation of familiarity: an evolving software system must strive to maintain stakeholders' familiarity with its different aspects (code, UI, documentation, etc.)
    - The law of continuing growth: The functionality of an evolving system must be continually increased to maintain user satisfaction
    - The law of feedback system: feedback from the different stakeholders of an evolving software system must be the driver of any changes it goes through
- Laws stating facts:
    - The law of self-regulation: the process of software systems evolution is self-regulating with a temporal distribution close to a bell curve. This is confirmed by Tsui and Kram's Rayleigh curve (Figure 1) and essentially means that maintenance and evolution activities start off at low rate then reach a maximum level of activity to end up fading out as the product is transitioned to termination.
    - The law of conservation of organizational stability: the activity rate (or rate of software usage by the client organization's users) is on average invariant over the product's lifetime.

These laws formalize fairly known information about software maintenance and evolution. Increasing complexity and declining quality are perceptible challenges in practice and most Software Development LifeCycles (SDLCs) include ways or phases to deal with them. The general recommendation of embracing software change as an intrinsic activity to software engineering instead of leaving it as an unplanned risk, and basing that change on stakeholders' feedback tend to provide a perception that initial software development should extended into maintenance and evolution with similar mechanisms.

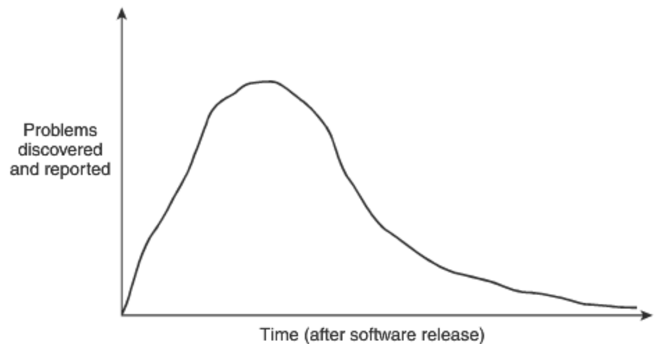

Fig. 1. A Rayleigh curve illustrating problem arrival [4, Figure 12.1]

Finally, the laws of self-regulation and conservation of organizational stability confirm that software engineering teams must adapt their software to the often-evolving business processes since its usage rate is going to be stable, and expect a recurrent pattern of maintenance and evolution activities over the time following the release.

C. Software support and supportability

In their book "Essentials of Software Engineering", Tsui and Karam [4, p250] have an interesting take on maintenance and evolution, which places these activities, as well as customer support under the umbrella of "Software Support and Maintenance". Their main point joins the law of feedback system (explained in section I.A of this paper) in that it places customer "product defect support" not only as a service provided to the customer but also as the main source for feedback driving maintenance and evolution.

Tsui and Karam also state that many software organizations charge as much as 10 to 20% of the original product price for one year of postrelease product support and customer service [4, p250], confirming the important weight of software support and maintenance. They later go on to present a view of customer service and support organization (Figure 2) which has an external layer dealing with customer feedback, and an internal one extracting required software fixes/evolutions from that feedback and routing them to the software maintenance and evolution teams (which are assumed to be a separate organizational entity, and will sometimes be an external service provider as we will see).

Pressman also describes the concept of "supportability", which qualifies "the capability of supporting a software system over its whole product life" [3, p798]. Supportability is described as an important element to consider in the early phases of every project in order to make supporting the software easier in postrelease. This would imply planning for user support, problem reporting and resolution, and update distribution.

Acknowledging the importance of the customer support tier as the main entry point for user feedback resulting in fixes and new functionality, in our paper we will focus only on the software maintenance and evolution tier which covers the

organizational entity and the activities which change the software (code, documentation, processes).

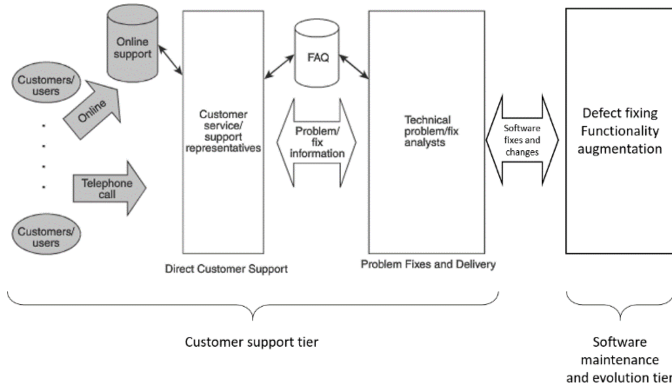

Fig. 2. Software support and maintenance organizational structure [4, Figure 12.2 - edited]

This choice is motivated by two reasons:

- Most of the literature we reviewed abstracts away the customer support tier and considers it outside the maintenance and evolution activities.
- The problems software engineers deal with differ in nature and scope with those that customer support deal with. Although some communication is important (creating FAQs, providing fixes and updates, creating customer support walkthroughs etc.), the core business activities for both groups are very different. We are obviously concerned with the software maintenance and evolution activities rather than those of customer support, although Tsui and Karam present a comprehensive overview of it in their book [4, pages 252-255].

D. *Definitions of software maintenance and evolution*

Stepping back, the terms "maintenance" and "evolution" have different meanings for different people. Sometimes, maintenance carries the meaning of both maintenance (as in "keep the software running and remove defects") and evolution (as in "add new functionality"). Different definitions of this all-encompassing term have been suggested, and looking at the "often quoted definitions of software maintenance" listed by April et. al [2, fig 1.3], we can discern these common elements in the definition:

- Modification of software
- Between the end of development/delivery and the retirement of software
- The goal is to keep the software running and "useful" (i.e. providing stable functionality required by the users).

The terms maintenance and evolution are also sometimes used interchangeably [1, p1], but there are elements of their definitions that make the difference between them, from different points of view [1, p2][2, Figure1.3]:

- Semantics:
  - Maintenance: Preventing the software from failing.
  - Evolution: Taking the software from a lesser to a higher state.
- Goals:
  - Maintenance: Providing updates and fixes to remove any defects and patch any security vulnerabilities.
  - Evolution: By adding new functionality, extending existing functionality or improving quality attributes.
- Lifecycle occurrence:
  - Maintenance: Takes place in the support after delivery (bugs, etc.)
  - Evolution: A change in requirements and any subsequent activities up to the release of new deliverables (software, update, documentation).
- Activities performed:
  - Maintenance:
    - Fix bugs, mostly planned and sometimes unplanned.
    - No major changes in architecture.
  - Evolution: Create new designs from existing ones, to variable extents depending on the goal behind that evolution.

In the rest of this paper, we will use maintenance and evolution as separate terms representing separate concepts, as differentiated by the information mentioned in this section.

1) *Software maintenance*

Software maintenance is defined by the ISO/IEC 14764 standard [1, p25] as "… the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage". In this definition, we can see three elements we already covered: (1) supporting a software system as the purpose of maintenance, (2) planning for maintainability as the pre-delivery activity and (3) performing maintenance activities as the post-delivery activity.

Maintenance, operations and development are sometimes difficult to differentiate. According to April et al.:

- The blurring between maintenance and operations can be elucidated by looking at which types of activities are performed and which teams perform them: maintenance is usually performed by a development team which codes, tests and delivers deployment packages, while operations are performed by an IT team which manages databases, infrastructure, application configuration, backups, recovery etc.
- The blurring between maintenance and development originates in the similarity of some activities that are performed for example to fix bugs and to develop functionality: in both cases, requirements are established, code is written, test are run, and validation

is provided. Maintenance is different from development in one major aspect [2, p11-12], which is the organizational structure of the entity performing each of them:

- For software development, a project is created with a time frame, a budget, defined deliverables and a plan. This structure disappears after delivery.
- For software maintenance, things take a different form: the team deals with randomly occurring events and requests from users, and the only pre-defined parameter is the support duration for the software (number of years after which support will be discontinued). Workload is managed using queue management rather than project management techniques, since requirements (expressed by user requests for fixes) are prioritized by criticality, and that priority can shift at any given moment with the arrival of more critical requests. Maintenance requests are also usually smaller in scope, they are typically manageable by two developers.

Tripathy et al. present an interesting taxonomy (Figure 3) of software maintenance [1, p29] which clusters types of maintenance according to the artifacts being modified in the maintenance process (business rules, software properties, documentation or support interface). Some types overlap with the definition of evolution that we present in the next section (Enhancive, Performance, Groomative), and others overlap with the user support activities we excluded from our definition of maintenance and evolution (especially support interface activities), it is worth noting that the highest impact in this taxonomy is attributed to activities changing business rules: enhancive, corrective and reductive.

2) Software evolution

It may seem like evolution is less of a priority than maintenance, but some figures indicate the opposite: April et al. quote Lientz and Swanson on the fact that "as far back as 1980, 55% of requests routed to maintenance organization concerned new functions rather than failure correction" [2, p4]. This is explained by several of Lehman's laws [6], most importantly the laws of continuing change and the law of continuing growth.

Tripathy et al. [1, p44-46] establish the fact that software evolution has not historically had a single widely-adopted definition, but quote what they present as a reasonable definition: "continuously changing software from a worse state to a better state" [1, p45].

They also quote an input-output illustration of evolution (Figure 4), which shows that many elements (people, existing software, machine and resources) are employed by the software evolution process to generate revised software.

Another important aspect which defines evolution is that it is triggered by changes in requirements [1, p45].

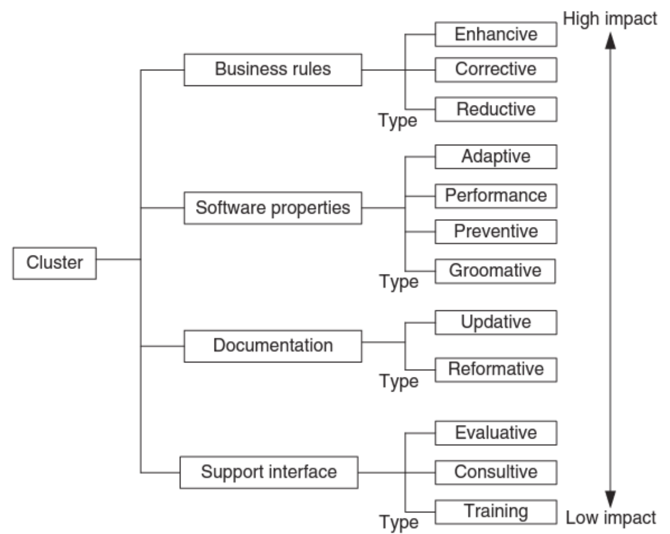

Fig. 3. Software maintenance taxonomy [1, 29]

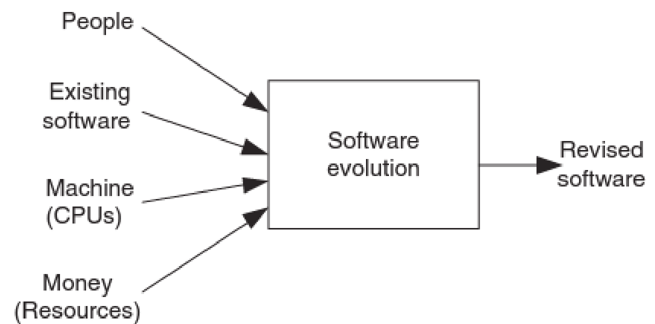

Fig. 4. Inputs and outputs of software evolution [1, p45]

3) Maintainability

Maintainability can intuitively be defined as the ability of a software to support maintenance activities, most importantly changes and extensions in code. Pressman defines maintainability through attributes that highly maintainable software must exhibit [3, p797-798]:

- Effective modularity through the partitioning of the software into modules that are easy to understand, modify and extend.
- Use of design pattern, which are standard and familiar code patterns frequently used by developers, which helps improve code readability.
- Use of coding standards and conventions which help maintainers understand code faster.
- Passing of Quality Assurance (QA) tests
- Engineering with knowledge transfer in mind, including informative comments in the code, availability of design and architecture documentation, etc.

Maintainability is a specific subset of supportability, and can be used as a benchmark to evaluate how easy or difficult maintaining a certain software will be, which helps with maintenance planning.

II. CHALLENGES OF SOFTWARE MAINTENANCE AND EVOLUTION

Software maintenance and evolution face tremendous challenges due to their change-centered nature. An illustration of common issues faced by software maintainers is presented by April et al. (Figure 5) as a result of surveying participants in successive software maintenance conferences. These issues are ranked by importance as perceived by software maintenance engineers.

Rank	Maintenance problem
1	Managing changing priorities (M)
2	Inadequate testing techniques (T)
3	Difficulty in measuring performance (M)
4	Absent or incomplete software documentation (M)
5	Adapting to rapid changes in user organisations (M)
6	A large backlog of requests for change (M)
7	Difficulty in measuring/demonstrating the maintenance team's contribution (M)
8	Low morale due to lack of recognition and respect for maintenance engineer (M)
9	Not many professionals in the domain, especially experienced ones (M)
10	Little methodology, few standards, procedures and tools specific to maintenance (T)
11	Source code in existing software complex and unstructured (T)
12	Integration, overlap and incompatibility of existing systems (T)
13	Little training available to maintenance engineers (M)
14	No strategic plans for maintenance (M)
15	Difficulty in understanding and meeting user expectations (M)
16	Lack of understanding and support from IS/IT managers (M)
17	Maintenance software runs on obsolete systems and technologies (T)
18	Little will or support for reengineering existing software (M)
19	Loss of expertise when a maintenance engineer leaves the team or company (M)
	(M): Management Problem (T): Technical Problem

Fig. 5. Maintenance issues ranked by importance [2, Figure 1.1]

These issues can be classified into:

- People issues: 5, 8, 9, 13, 16, 18, 19
- Requirement management issues: 4, 6, 14, 15
- Process issues: 1, 2, 3, 7, 10, 19
- Risk issues: 11, 12, 17, 19

As you may have noticed, item 19 falls under several categories. Loss of expertise is in fact one of the core issues of maintenance and evolution. In the following sections, we try to look at some of these issues and challenges in depth to get a sense of where the pain points of software maintenance and evolution are.

1) People issues

The most basic people issue influencing maintenance and evolution is the fact that as activities, they should not exist if human error did not exist. But human error exists, and the worse it gets, the more complex it makes maintenance and evolution. However, we would like to shed light on a different type of people issue in this section: stakeholders' perceptions of the process, how they are influenced by the maintenance and evolution process and how their reactions influence that process.

From the end user's point of view, the client organization's point of view by extension, and even the software organization's point of view, maintenance and evolution tasks should ideally be finished as soon as possible. The particular nature of maintenance tasks or requested features, which come unexpected and are often required in the shortest possible time (without a specific deadline, since the software is already in production) creates a tremendous pressure on the maintenance and evolution team. Tripathy et al. [1, p44] conclude that this makes of maintenance a task usually thrown at younger and less experienced developers, which confirms items 9 and 13 in Figure 5. April et al. report similar problems, raising for example the issue of the lack of software maintenance teaching in schools [2, p7]. This influences quality and productivity, since new recruits do not have a solid understanding of the system, and tend to introduce defects in the process of learning while performing maintenance.

More generally, Tripathy et al. qualify the general perception of software maintenance as "less challenging, and, hence, less well rewarded than original work" [1, p44], which plays a big role in the loss of motivation listed in Figure 5 as item 8. The resulting drop in quality and timeliness of fixes and updates will in turn frustrate the users and client organizations and make continuing the maintenance activity a difficult task.

Another perspective on people issues is from the managers' point of view. Depending on the business model (maintenance contracts, or all-inclusive initial price) maintenance will have an initially fixed cost that the activity must meet even if it is unrealistic from a practical point of view. Otherwise, the organization will lose money on this activity, which is the worst-case scenario from a manager's point of view. This pushes managers to issue strong and unrealistic requests to maintenance teams which may end up in conflicts. Adding to that, April et al. report conclude that the organizational culture and manager's perception of the contribution of maintenance has a direct impact on maintenance staff morale [1, p43].

2) Requirements management

Requirement management will differ depending on the activity being performed:

- For maintenance, the main requirement is obviously to deal with the defect backlog and reported security issues as soon as possible [1, p83]. Although this does not look like a well-phrased requirement, it is often the case in reality. Tsui and Karam indicate that throwing some of the weight of this requirement off is possible through providing "work-arounds" to users while the real fix is being developed [4, p253]. Delegating the task of providing work-arounds to first level support teams (Figure 2) is a solution that provides maintenance teams with more time to explore the issues in depth and code fixes that will be shipped in updates.

More immediate and tangible requirements exist for maintenance in the form of problem descriptions and bug reproduction scenarios. It is not always guaranteed that the team analyzing the issue will be able to reproduce the scenario or have all required information in the problem description [1, p103].

- For evolutions, requirements take a more familiar shape as feature or quality attribute requests. Often times they come formalized as change requests, which need to be prioritized, tracked and referenced whenever modifications are done to one of the product's artifacts. Bad change request management (missing information, ambiguous descriptions) has similar effects to what badly formed requirements have on initial development: delays, failed tests, failed integration, regressions, etc.

3) Process and activity management

A simpler software maintenance taxonomy than the one presented in Figure 3 is provided by different sources [8, p3] [10, p3] and presents the following types of maintenance:

- Corrective maintenance: consists in fixing any defects reported or found by the team.
- Adaptive maintenance: consists in adapting the software to any change in its environment, like new hardware, new runtime environment, new operating system, etc.
- Perfective maintenance: another term for evolution, implementing new functionality of the system.
- Preventive maintenance: any activity aiming to improve the system's maintainability, such as refactoring.

These types of maintenance and evolution require different processes to be implemented, and different measurements of success. Questions like: how to perform the activity (process)? Who should perform the activity (organization)? How to measure the activity's success (metrics)? One of the major issues of maintenance and evolution is choosing the right process depending on the software type, the SDLC used for development and whether it took into consideration planning for post-delivery support or defined it as a standalone activity. The process complexity will also depend on how deep maintenance and improvement activities are meant to go: for example, it would be ok to use a fairly ad-hoc process dealing with each maintenance or improvement request as a ticket to be assigned to one developer and pushed to the code base in low-complexity non-safety-critical systems. When we move on to more complex or more safety critical software, an equally complex process with enough safeguards must be implemented to make sure to minimize the chances for regressions and test successful resolution of issues more precisely.

4) Risk management

When planning for risk management in the context of software maintenance and evolution come in the form of gaps to be filled [10, p8-10]:

- The gap between the software organization's maintenance capabilities and its maintenance commitments, which need to be managed to fit the client's business needs without absorbing all the software organization's resources in the process.
- The gap between dealing with emergency situations maintenance and dealing with non-emergency situations maintenance. In non-emergency situations, changes are bundled into releases destined for pre-planned dates. In emergency situations, however, reactivity needs to be much higher and the task is much more challenging. The software organization therefore needs to have both capabilities, which is not easy.
- The gap between the terms of the maintenance services performance required in the Service Level Agreement (SLA) with the client (permissible downtime, data backup and loss limits, etc.) and the organization's measurement of maintenance performance. This issue calls for the organization to put in place adequate metrics and monitoring mechanisms that enable it to conform to the terms of the SLA to the best possible extent.
- The gap between the organization's event management system and the goal of keeping all stakeholders informed with all maintenance events and actions taking place and their implications. Events include customer change requests (for evolution), or maintenance-triggering events (incidents, bugs, downtime, etc.). The organization needs to keep a steady information flow with its managers, the client and the maintenance teams to enable better decision making. As an example, the client should be made aware of the forecasted implications of an incident and offered options whenever possible (limit traffic, allocate more resources, take the site down for maintenance, etc.).

The number one risk facing a software organization is failing to meet the contractual terms of the maintenance/warranty agreement with the customer, which would imply penalties, loss of customers, legal accountability and in the worst-case scenario threaten the livelihood of the organization itself if the damages are large enough [17, p2-4]. This can result from any of the gaps mentioned previously not being filled adequately by the software organization.

Another type of important but less immediate risk is when the "knowledge hub employee" for a certain product to leave the organization without leaving behind maintainable software. The results can range from minor inconvenience if the product is not complex and assigning it to other resources can solve the problem with the only negative impact of an added learning delay, up to a total halt in the maintenance and evolution of a complex product since any attempt causes regressions bad enough that the team needs to cancel the changes. This situation

overlaps with issues in the organization's maintenance planning processes which, if they don't plan for knowledge management, can expose the organization to the risk of dependency upon individual developers' presence.

One might consider that opening up communication between product development teams and maintenance team might fill for the gap left by the lack of knowledge transfer between the initial developers and the maintainers, which is the number one issue complicating maintenance and evolution work. However, this is usually a difficult goal to achieve since product development teams view the effort of supporting maintenance team as external noise and as a distraction from their main assignments [1, p44].

In the following section, we will explore many process frameworks/methodologies of dealing with the issues mentioned in this section.

III. Elements of Existing Knowledge

There is abundant literature about how to manage maintenance and evolution efficiently, both in the context of older more process-disciplined software engineering frameworks and in modern less process-disciplined frameworks. The purpose of this section is to explore a generic process and organization framework for software maintenance and evolution, then some specific practices developed to improve how organizations manage maintenance and evolution.

It is worth noting here that most sources we reviewed do not make an explicit distinction in their suggestions between maintenance and evolution activities, considering rather that these activities are two faces of the same coin: improvements can be viewed as fixes to "lack of functionality" type of defects. So whenever unspecified, it should be assumed that the process description is adaptable to both maintenance and evolution.

A. General overview

Tsui and Karam present a general organizational software maintenance framework [4, p254-260] as illustrated in Figure 2. The most interesting part of it for us is the software maintenance and evolution tier, but also the technical problem/fix analysts team, which is responsible of delivering the solutions and updates the maintenance and evolution tier comes up with.

For maintenance, from a process point of view, Tsui and Karam state that "the design, code and test cycle of a problem-fix is not very different from the design, code and test activities of the development cycle" [4, p256]. The only difference might be that the discipline required by new product development may be discarded especially when fixes require little change in the code and/or documentation. One recommendation Tsui and Karam highlight is the necessity of keeping all software artifacts updated (requirements, design, code, tests and manuals). A gap between the code of a software and its documentation is dangerous since it will slow down the following fixes/evolutions, by forcing the maintainers to re-discover the changes that were made. This process can introduce defects due to partial understanding of the requirements or code or tests that have been changed.

By contrast, April et al. [2, p21] indicate that although maintenance started off having similar processes to development, by the 80s specific for maintenance process models were developed, and consulting organizations introduced proprietary maintenance standards as a response to the problem of aging large-scale software that required radical solutions by that point. In the 90s, international standards of software maintenance were being released such as ISO 12207 and ISO 14764. An example of that is shown in Figure 6 derived from the ISO 14764 standard, representing the diagram of software maintenance processes (central circle) in the context of other software processes. In this simple example, the maintenance process is situated between implementation (or initial development), migration to another product and retirement of the software. Internally, the process is a loop of problem and modification analysis, implementation, then review and acceptance.

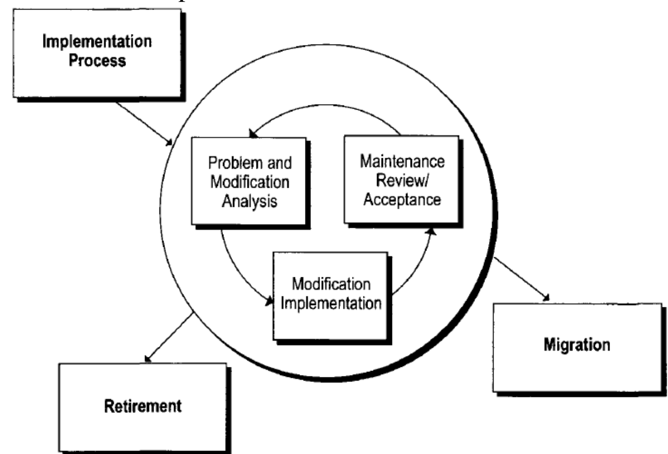

Fig. 6. Software maintenance key processes according to ISO 14764 [2, Figure 1.9]

The rule with these standards is that their processes are all open to customization, they are more guidelines than strict frameworks in order to allow for flexibility imposed by the diverse array of business contexts for software organizations.

To explore one example of software maintenance standard, we will briefly present the IEEE Standard for Software Maintenance, drafted 1994 and released 1988 [8, p5]. This model's process definition is composed of 7 stages: Problem identification, Analysis, Design, Implementation, System test, Acceptance test and Delivery. Each of these stages has 5 associated attributes: Input and output lifecycle products, Activity definition, Control and Metrics. As an example, the analysis phase would have as an input a problem report, resource estimates and any other project documentation necessary. Feasibility analysis, then the establishment of more firm requirements of the modifications to be performed. The output is obviously the fixed code and modified documentation, and some metrics might be the test results (number of passed tests vs initially passed tests), the man-hours this process consumed and the time between receiving the request and delivering the fix.

Evolution is differently handled, but the generally accepted perception is that the process will not differ much from that of

maintenance, except in phases like requirements, planning and tracking. Evolution is generally driven by change requests, which are used to keep track of the initial requirements, their acceptance and any eventual modifications to them [4, p260]. There is more room for a back-and-forth between the software organization when it comes to evolution, which is a fundamental difference with how maintenance is managed.

Whatever the process chosen to manage software maintenance or evolution, it is critical to use metrics to control how successful the process is. An example of maintenance artifacts and characteristics that need to be tracked is presented in Figure 7.

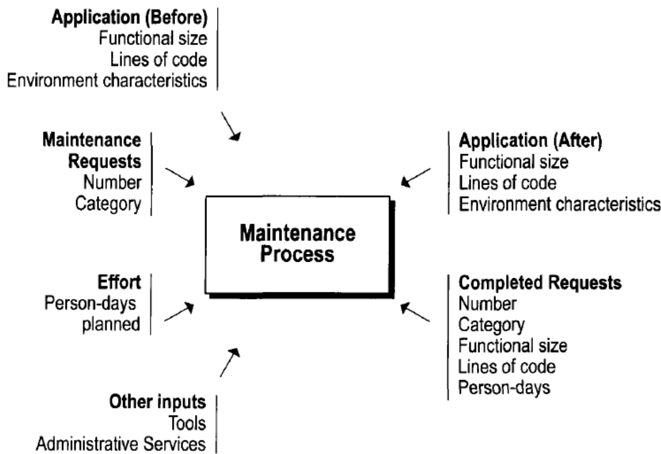

Fig. 7. Selected characteristics of a maintenance process to track [2, Figure 1.13]

The more data is collected, the better the maintenance team can evaluate their performance and be better prepared for the next maintenance iterations.

One can ask the question: what type of maintenance do I need to perform in each case? Tripathy et al. [1, p32] suggest a decision tree (Figure 8) to decide on which type of maintenance to be chosen for each case, which shows a flow of questions and answers that result in the choice of one or more types of maintenance (types listed in Figure 3).

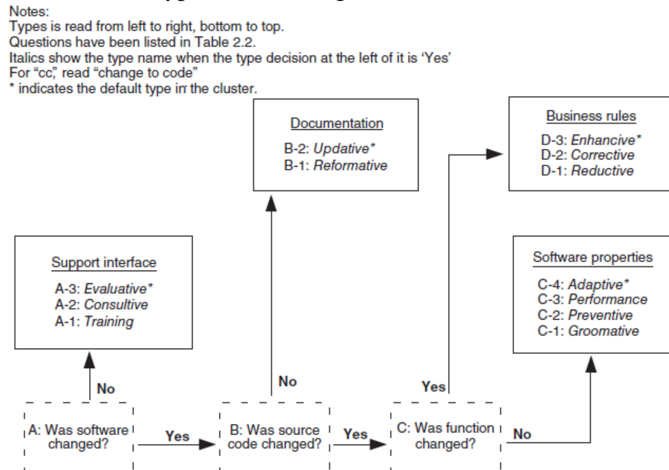

Fig. 8. Deciding on which type of maintenance to perform [1, p32]

With this general overview provided, we will proceed to explore some particular processes and practices in more detail.

B. The simple staged model for closed source software

The simple staged model for Closed Source Software (CSS) ([1, p87-90] and [12]) and established in the early 2000s is represented by a sequence of stages (Figure 9) through which the maintenance and evolution activities occur. The Evolution and Servicing phases in this model map to the evolution and maintenance activities as defined in out paper. They also occur multiple times as needed. The approach of this model derives from the assumption that maintenance should not start unless absolutely necessary, and that up to that point only evolution should occur.

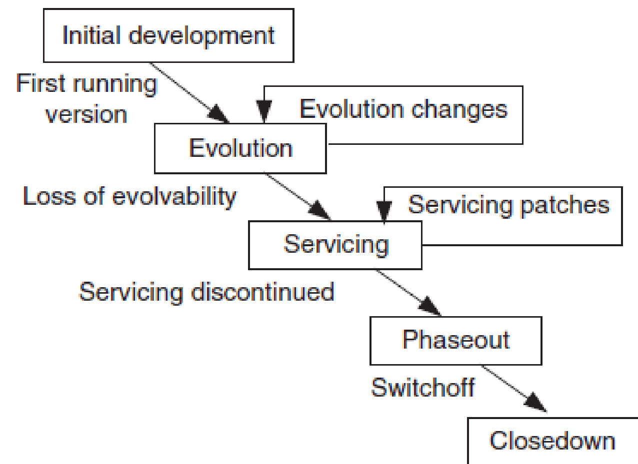

Fig. 9. Simple staged model for CSS [1, Figure 3.5]

The details of the phases are as follows [1, p87-90]:

- Initial development: All the activities up until a first functioning version of the software is released
- Evolution: Immediately after the first release, the product is theoretically in "good shape". In this phase, improvements and changes are performed on the software to meet the customer's expectations. Iteratively, quick patches and new releases are delivered to customers, and customer feedback is collected to prepare the next patches and releases.
- Servicing: The servicing phase is triggered once the strain on the software caused by changes finally breaks architectural integrity, or when the software is perceived to have matured functionally (stabilization of requirements), whichever occurs first. In this stage, changes become expensive so developers try to minimize them using mechanisms such as wrappers. The changes performed represent the bare minimum to keep the software running.
- Phase-out: Servicing of the software stops, and when spending on servicing activities becomes economically unjustified, the software is labelled as a legacy system and prepared for closedown.

- Closedown: The software organization pulls the product from the market and makes recommendations to customers for alternative solutions.

This model offers the advantage of delaying the changes which are expensive until a moment in time where the software is more likely to be profitable, which economically justifies servicing it.

C. Change mini-cycle

The change mini-cycle model embraces software evolution as a necessity and conceptualizes it as "the introduction of new requirements to the existing system" [1, p91-94]. This model was introduced in the late 1970s and later revisited by researchers in the 1980s and 1990s.

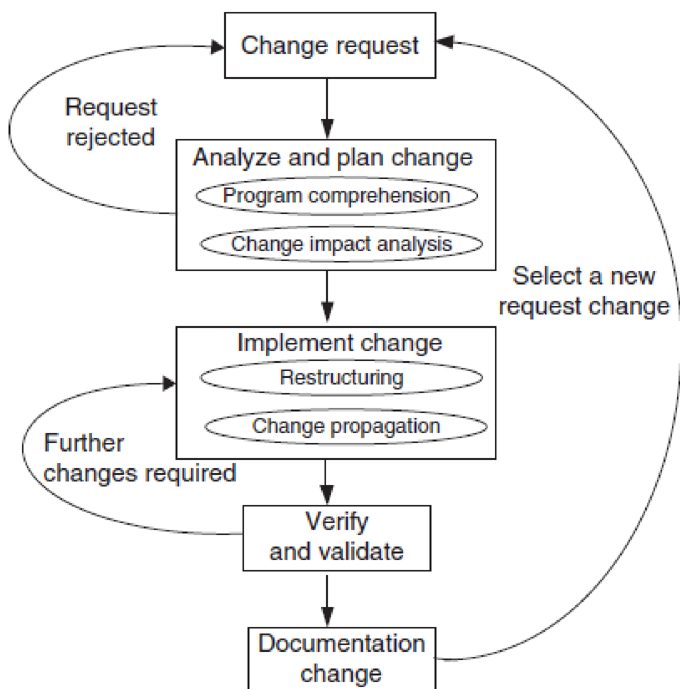

Fig. 10. The change mini cycle [1, Figure 3.8]

The model organizes the maintenance and evolution process in five major phases (Figure 10): Change Request (CR), analyze and plan change, implement change, verify and validate and documentation change. It is noteworthy that a change request can be either a problem report or an improvement request, which makes this model usable both for evolution and maintenance. Also, the model's phases show awareness that changes impact more than the functionality they are changing, which leads to the modification of all the software artifacts.
The detail of these phases is as follows [1, p92-93]:

- A change request comes from management, users or the customer organization. They describe the desired change or the problem to fix, and they are expected to provide enough context and information to be usable by the maintenance team, otherwise they are rejected. CRs can also be rejected for budget or contractual reasons.
- Change planning and analysis determines (1) which parts of the software are going to be affected by the CR, and (2) the potential impact and risks resulting from the requested change.
- In the implementation phase, the change is performed as well as any refactoring deemed necessary. Changes often need to be propagated due to dependency and coupling, which calls for additional testing and verification.
- The important takeaway from the verification and validation phase should be that the integrity of the software has not been compromised by the modifications, i.e. no regressions have been introduced.
- Updating the documentation has advantages that were previously mentioned, including avoiding the degradation of maintainability.

This process has the advantage of formalizing evolution and maintenance in one streamlined process which is nowadays widely used through different variations all based on change requests as a mechanism to trigger changes.

D. Value-driven software maintenance

Value-driven software maintenance introduced by Sneed and Huang [13] is an example of a more quantitative business-centered approach, which tries to give more weight to the cost/benefit reflection when considering software maintenance and evolution. More precisely, it explores the economic benefit of keeping the software as it is and making incremental changes as necessary. It comes as an alternative to the continuous maintenance models suggested for Lehman's E-type software. This model provides equations to perform the cost/benefit analysis in two steps applicable both for a single change or a maintenance project:

- Predicting the costs of maintenance
- Assessing the benefits of maintenance

We won't go into the details of this approach here. This approach provides a unique perspective on software maintenance and evolution correlating it with cost and benefit predictions before the activities start, which provides the software organization with tools to manage the maintenance risks.

E. DevOps and continuous integration

In the specific context of web and cloud development, practices like DevOps offer exceptional advantages for maintenance and evolution.
DevOps does not enter under the umbrella of software maintenance and evolution activities exclusively. It is rather a practice aiming for more collaboration between the development and operations teams within the software organization, and for more automation of application deployment and change propagation. The underlying assumption here is that the development teams will be the

maintenance teams, especially in the typical context of developing and maintaining web and cloud applications. In this context, DevOps practices offer some unique advantages for maintenance and evolution [5, Chapter 5]. With an automated continuous deployment/integration pipeline in place, as well as more coordination between the development and operations teams, the development teams can automate integration testing, move faster to User Acceptance Tests (UAT), reduce the duration between deployment cycles and give the developers more control over pushing code production, allowing for faster delivery of fixes (which can reach several fixes per day) and better control over evolutions.

IV. REFLECTIONS

Keeping our assumptions and scope in mind (see section I of this paper), our reflections will take the form of recommended maintenance and evolution guidelines for a modern software organization creating enterprise software as a commercial product.

As for any software engineering question, there is no silver-bullet-type solution to better manage software maintenance and evolution. The efficiency of maintenance and evolution depend on many factors in the software organization's processes, human resource management, business model and contractual obligations towards its customers. For a manager tying to implement a better software maintenance and evolution practice in his organization (or any software engineering practice, really), measuring the "pulse" of the organization, knowing its issues and performance in different areas is key.

The first and most determining questions to answer when attempting to improve software maintenance and evolution are: How big and how mature are our software products? How much are our clients willing to spend on the maintenance of products they use? Are there any contractual constraints on that already in place? How do we currently perform maintenance and evolution? How do we deliver updates and fixes? How many developers do we have working on each of them? How is work organized currently, i.e. are separate modules handled by separate small teams or is a large team working on the product at the same time in a waterfall-ish fashion? Do our teams appreciate process-heavy approaches? The answer to these questions will have implications on choosing to create separate maintenance teams or adapt the internal process of each team, on whether or not to adopt a certain maintenance or evolution process model and simply on whether we have enough visibility to make those decisions in the current situation, since most of the time not all the answers are available.

The main recommendation here is to take the time to assess the software organization's goals, processes and capabilities, and how they might influence a certain choice when it comes to maintenance and evolution. This time will prove invaluable, due to the massive share maintenance and evolution take from the resources of the software organization.

Answering these questions will go a long way in orienting a manager's choice of a maintenance and evolution framework built from a combination of process models and practices which can be implemented within the organization. An example framework would be using value-driven software maintenance to perform estimates and CR validation within a change mini-cycle model, all while augmenting the maintenance team's ability to deliver quick fixes and updates by creating a continuous integration pipeline and encouraging DevOps adoption.

Starting small with one pilot team is a good idea to determine whether or not the framework would yield positive results, and in the happy end scenario to use the results of this experiment to push for organization-wide adoption.

REFERENCES

Books:

[1] Tripathy, Priyadarshi "Software evolution and maintenance: a practitioner's approach", ISBN:0-470-60341-0, 978-0-470-60341-3, John Wiley & Sons Inc, 01/01/2015

[2] April, A. and Abran, A., 2012. Software maintenance management: evaluation and continuous improvement (Vol. 67). John Wiley & Sons.

[3] Pressman, R.S., 2005. Software engineering: a practitioner's approach. Palgrave Macmillan.

[4] Tsui, F., Karam, O. and Bernal, B., 2013. Essentials of software engineering. Jones & Bartlett Publishers.

[5] Bass, L., Weber, I. and Zhu, L., 2015. DevOps: A Software Architect's Perspective. Addison-Wesley Professional.

Articles:

[6] Lehman, M.M., 1979. On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software, 1, pp.213-221.

[7] Moonen, L. and Pollock, L., 2016. Introduction to the special issue on software maintenance and evolution. Journal of Software: Evolution and Process, 28(7), pp.510-511.

[8] Bennett, K., 1996. Software evolution: past, present and future. Information and software technology, 38(11), pp.673-680.

[9] Penta, M.D. and Maletic, J.I., 2015. Guest editorial: special section on software maintenance and evolution. Empirical Software Engineering, 20(2), pp.410-412

[10] Niessink, F. and Van Vliet, H., 2000. Software maintenance from a service perspective. Journal of Software Maintenance, 12(2), pp.103-120.

[11] April, A., Huffman Hayes, J., Abran, A. and Dumke, R., 2005. Software Maintenance Maturity Model (SMmm): the software maintenance process model. Journal of Software Maintenance and Evolution: Research and Practice, 17(3), pp.197-223.

[12] Rajlich, V.T. and Bennett, K.H., 2000. A staged model for the software life cycle. Computer, 33(7), pp.66-71.

[13] Sneed, H. and Huang, S., 2010. Value-Driven Software Maintenance. International Journal of Computers and Applications, 32(2), pp.215-221.

[14] Ulziit, B., Warraich, Z.A., Gencel, C. and Petersen, K., 2015. A conceptual framework of challenges and solutions for managing global software maintenance. Journal of Software: Evolution and Process, 27(10), pp.763-792.

[15] Kurtel, K., 2013, October. Measuring and Monitoring Software Maintenance Services: An Industrial Experience. In Software Measurement and the 2013 Eighth International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2013 Joint Conference of the 23rd International Workshop on (pp. 247-252). IEEE.

[16] Boehm, B., 2006, May. A view of 20th and 21st century software engineering. In Proceedings of the 28th international conference on Software engineering (pp. 12-29). ACM.

[17] Davidson, S.J. and Holloway, G.K., 2004. Software Services and Maintenance Agreements. 23rd Annual Institute on Computer Law.