When Does Software Useful Life Begin: Software Evolution, Obsolescence and Valuation

Boštjan Kežmah[1,*], Tina Beranič[1]

[1] University of Maribor, Faculty of electrical engineering and computer science, Koroška cesta 46, Maribor, 2000, Slovenia

Abstract

According to valuation methods, software valuation depends on the remaining useful life. The paper stems from a real-life problem faced by certified appraisers at the Slovenian Institute of Auditors. After developing a model for determining the initial useful life of software, we found that determining the lifetime is not sufficient, since the classification of software into a group does not also indicate the beginning from which the lifetime is counted. The problem is further exacerbated by the evolutionary nature of software, which is upgraded over time. The paper identifies this gap, indicates the direction of its resolution, and warns of the probable absence of empirical basis. It is both a problem statement and an invitation to the academic community to cooperate.

Keywords [1]

software valuation, useful life, beginning of useful life, software evolution

1. Introduction

This paper arises from a real-world professional problem raised by certified appraisers at the Slovenian Institute of Auditors. In their work, appraisers regularly value software as an intangible asset for various purposes: financial reporting, mergers and acquisitions, tax procedures, equity contributions, and forensic expertise. The remaining economic useful life of the software turns out to be the central and most sensitive variable of all applicable valuation methods.

Since there was no structured, repeatable framework available for its determination in practice, we were asked to prepare a model for appraisers to determine the initial useful life of software, based on established empirical studies and international accounting standards. However, when attempting to actually use this model, an obstacle emerged that we had not anticipated.

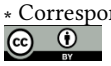
Determining the length of the useful life, that is, how many years the expected economic useful life of the software lasts, has proven to be a manageable part of the problem, which can be reasonably resolved by classifying software into performance tiers.

Significantly more demanding, but in practice indispensable, is the determination of the beginning from which this period is counted. Without a justified starting point, the calculated length of the period remains a floating date that cannot be converted into the remaining useful life, which is the actual input to the valuation. The problem is further complicated by the fact that software is repaired, supplemented and upgraded over time, which makes it questionable not only when the period began to run, but also whether and when this counting is reset when changes to the software are made. It is at this point that it has become clear that the model for determining the life span cannot be used in practice without resolving the issue of the beginning.

The purpose of this paper is therefore limited and precisely defined. We summarize the starting points of our methodology for determining the initial length of the period only to the extent necessary to identify the problem. This paper is not designed as a detailed description of this

[1]SQAMIA 2026: 13th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 16-18, 2026, Niš, Serbia

EMAIL: bostjan.kezmah@um.si (A. 1); tina.beranic@um.si (A. 2)
ORCID: 0009-0005-1104-4361 (A. 1); 0000-0001-6518-5876 (A. 2)
* Corresponding author.



© 2026 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-19

CEUR-WS.org/Vol-4336/paper3.pdf

methodology and does not set itself this goal. The focus is on the precise identification of the problem of determining the beginning of the useful life, on the analysis of the reasons why the classification of software into a group does not solve this problem, and on the aggravation brought about by the evolutionary nature of software. When identifying the problem, we also indicate a possible direction for its resolution, without developing it to the end in this paper, and we explicitly point out the probable absence of an empirical basis in the existing literature, which will significantly increase further work. In this way, the paper addresses the broader professional and academic community. Determining the beginning of the useful life is not just an operational detail of evaluation, but an open methodological question, the resolution of which exceeds the capabilities of an individual institution and requires the cooperation of researchers from the fields of evaluation, accounting and software engineering. The paper is therefore both a statement of the problem and an invitation to the academic community to participate in the development of a methodology, which is a necessary condition for further, methodologically supported evaluation of software as an intangible asset.

The rest of the paper is structured as follows. We first summarize the starting points of software valuation according to international valuation standards and define the concept of useful life as a central variable. We then briefly present the theoretical foundations and our model for determining the starting point of the lifespan. In the main chapter, we discuss the problem of determining the beginning of the lifespan with candidates for the starting point, the evolutionary nature of software, and the issue of resetting or extending the lifespan during upgrades. From it we derive a research gap with an indicated direction for its resolution. We conclude the paper with a summary of the findings and a definition of further steps.

2. Software valuation

Appraisers from the Slovenian Institute of Auditors are reporting that intangible assets represent an increasing share of the value of companies. Among these assets, software has a special place, which, as a non-physical but identifiable asset capable of generating future economic benefits, significantly contributes to the added value of organizations [1]. Software valuation is a complex and multidisciplinary process required in many business situations, including financial reporting, mergers and acquisitions, tax procedures, financing and intellectual property management [1].

Due to its specific nature, software valuation requires an in-depth understanding of both financial and accounting laws and information technologies. Its value does not stem from its physical form, but from its functionality, ability to generate cash flows and a life cycle that is subject to rapid technological changes. This chapter provides a detailed analysis of the starting points for valuing software according to the International Valuation Standards (IVS), specific professional guidelines from practice, the role of modern technologies such as artificial intelligence, and the key issue that determines the reliability of any assessment - determining the useful life or the remaining useful life of software.

2.1. Valuing software according to the IVS standard

According to the International Valuation Standards (IVS 210 - Intangible Assets), software is classified as technology-related intangible assets. The standard specifies that three main methods can be used to value intangible assets: the market comparison method, the income-based method, and the cost method [2].

In practice, the market comparison method is used extremely rarely for software. The reason lies in the requirement for the existence of information on transactions with identical or similar assets between unrelated parties in an active market. Since software is often highly specific and is rarely traded separately from other company assets, suitable market comparables, especially in Slovenia, are generally not available. Therefore, appraisers most often resort to the income-based method and

the cost method, depending on the nature of the software itself and its ability to generate direct cash flows.

2.2. Specific requirements and professional explanations when valuing software

As the professional magazine SIR*IUS explains in detail, the choice of the appropriate valuation method is strictly conditioned by the type of software and its business model.

The income-based method is most often used for so-called product software or solutions (e.g. SaaS models, licensed products), which directly generate revenue on the market through licenses or subscriptions [1]. For this method to be used reliably, two conditions must be met: the software must generate future economic benefits and these benefits must be reliably measurable. The excess return method (MPEEM) is often used, which subtracts the required returns on all other contributing assets (e.g. computer equipment, working capital, brand) from the forecasted revenues, so that the remainder represents exclusively the value generated by the software itself [1]. A special challenge here is separating the value of the software itself from the value of the portfolio of so-called "locked-in customers", whose value stems from relationships and recurring revenues rather than from innovation [1].

The cost method (especially the replacement value method) is the primary choice for internally developed software that does not directly generate revenues, but rather supports the company's internal processes [1]. This method is based on the assumption that a market participant would not pay more for the asset than it would cost to re-produce the asset with the same functionality. This is where the professional dilemma arises regarding reliance on accounting records of development costs, which often do not reflect the full market value, intellectual capital and market potential.

2.3. Use of artificial intelligence in valuation

The IVS guidance also addresses the growing use of artificial intelligence (AI) and automated valuation models (AVM) in software valuation [3]: advanced tools are permissible, and in data collection and analysis even expected, but they must not replace professional judgment, and the appraiser must understand the operation and the limitations of any model used [2]. An assessment generated by an algorithm without critical expert review therefore violates the standards. For the problem addressed in this paper, such tools may eventually assist in assessing obsolescence, but they presuppose - rather than provide - a methodologically sound definition of the remaining useful life.

2.4. Determinin

g the useful life of software

Regardless of whether we use the income-based method (where we have to predict future cash flows) or the cost method (where we have to deduct the loss of value from the replacement cost), the key and most sensitive parameter in both methods is the remaining economic useful life or the useful life of the software [1][2].

Standards point out that, unlike physical machines, software does not deteriorate or become obsolete in the classic physical way, but loses value due to rapid technological changes and external circumstances.

Therefore, every valuation must professionally assess three forms of obsolescence that directly shorten the useful life of software [2]:

1. Functional obsolescence: Occurs when software no longer offers users an optimal user experience, does not have a modern interface, or does not support new business needs and regulatory requirements.
2. Technological obsolescence: A specific form of functional obsolescence where software is written in a deprecated programming language, is incompatible with modern hardware and operating systems (e.g., cloud computing), or has security vulnerabilities.
3. Economic obsolescence: A result of external market factors, such as strong competition from better and cheaper alternatives, that reduce the market potential of the existing solution.

Accurately determining these forms of obsolescence is essential for defining the useful life of software. An incorrect, overly optimistic, or overly pessimistic accounting estimate of the useful life can drastically distort the value of an asset on the balance sheet or in a transaction.

As pointed out by IVS 210, some software can be quickly replaced by cheaper alternatives, while others are renewed modularly, making it difficult to accurately determine the date of "economic death" [2].

The use of modern technologies (AI) and comparative market analyses can be helpful here, but a fundamental methodological gap remains in how to objectively translate these elements into an accurate number of years of useful use.

2.5. Scope of this paper within the overall valuation methodology

A review of the IVS requirements and the professional guidance published in SIR*IUS shows that all valid valuation methods - from discounting cash flows for market products to calculating replacement costs for internal solutions - depend at their core on the established remaining economic useful life. Without an objective and repeatable way to determine it, the resulting valuation loses its reliability and professional validity.

The methodology we are developing for the Slovenian Institute of Auditors addresses this dependence in two parts. The assessment of functional and technological obsolescence is anchored in the software product quality model of ISO/IEC 25010:2023 [4], whose quality characteristics provide the appraiser with a structured frame for identifying where and how a software asset lags behind current requirements. A detailed presentation of this part of the methodology is beyond the scope of this paper. What matters here is a design decision behind it: the methodology deliberately follows the established procedure for valuing machinery and equipment, in which the appraiser determines a baseline useful life, assesses obsolescence, and derives the remaining useful life. Appraisers can therefore apply a sequence they already know from tangible assets. This consistency with classical valuation practice is a hard requirement for industrial usability.

Equally hard is the cost constraint. Software appraisals are in practice low-priced engagements; in Slovenia, fees for smaller appraisals start at approximately EUR 500. The scope of work must remain proportionate to such fees, which excludes any method whose application requires a detailed inventory and analysis of the software asset. This includes the software inventory valuation practice developed in the licensing literature [5], which presupposes identification, registration, escrow deposit and audit of all software components, as well as exhaustive mappings of software value aspects developed to support product-management decisions [6]. Whatever their conceptual merits, such procedures are not industrially applicable at these price points. Approaches that infer the value of software from capital-market expectations about software firms [7] operate at the level of the enterprise rather than the individual asset and therefore do not yield the asset-level remaining useful life that IVS-based methods require. A recent systematic review of software valuation approaches [8] likewise concentrates on the choice among the cost, market and income approaches rather than on the temporal variables of the asset.

Within these constraints, the element that appraisers still cannot obtain is the remaining useful life itself, and it is at this point that the development of the definitive methodology currently stalls.

The remainder of this paper therefore focuses on the determination of the useful life - its baseline duration and, above all, its beginning.

3. Theoretical foundations for determining the software useful life

Determining an objective and reliable software useful life is a complex methodological challenge that requires consideration of both technical and technological as well as business and accounting factors. As a fundamental framework for the structured determination of the baseline useful life, we adopted Gartner's Pace-Layered Application Strategy model [9]. This model goes beyond traditional, unambiguous approaches and introduces an adaptive strategy that categorizes business capabilities and associated software into three separate layers according to the speed of change, added business value and level of adaptability.

The Pace-Layered Application Strategy model divides software into foundational, differentiating and innovative capabilities. Each of these layers is characterized by different levels of technological and economic obsolescence, which consequently dictates different baseline useful lives.

3.1. Foundational capabilities layer

The foundational capabilities layer includes software solutions that support basic, standardized business processes, the primary goal of which is operational efficiency and ensuring legal and regulatory compliance.

Typical representatives of this layer are the core modules of ERP systems, accounting software, basic human resources systems and general IT infrastructure [9].

This type of software is characterized by a high level of stability and an extremely low rate of change, and Gartner places it in the long-term planning horizon [9].

Scientific research confirms the long-term nature of these systems. Tamai and Torimitsu (1992) found through survival analysis that the average lifespan of mainframe software is approximately 10 years with a standard deviation of 6.2 years [10].

Similarly, Huang and Yasuda (2016) studied the life cycle of ERP systems and concluded that the post-implementation phase represents the central part of the life cycle, with these large-scale systems remaining in organizations for an average of about 14 years [11].

Based on these expert findings and taking into account the precautionary principle and the investment concept of long-term assets, an expected baseline lifespan of 10 years was selected for software in the foundational capabilities layer.

3.2. Differentiating capabilities layer

The differentiating capabilities layer includes software that enables organizations to implement unique or industry-specific processes that provide a direct competitive advantage in the marketplace [9].

Solutions such as advanced customer segmentation or dynamic pricing strategies are often extensions of core systems, are highly customized to the organization’s specific needs and follow a medium-fast change cycle. This exposes the software to a significantly higher risk of functional and technological obsolescence.

Empirical evidence of the shorter lifespan of these applications is evident from the research of Aier et al. (2009) [12]. Using the method of survival analysis of applications within the enterprise architecture, a high mortality rate of software was found, as according to the conducted case studies, only 38% to 57% of the considered applications survived a period of 4 to 10 years.

Additional professional guidance is provided by the International Accounting Standard (IAS) 38 - Intangible Assets [13], which clearly highlights that software and similar intangible assets are highly

subject to technological obsolescence due to rapid technological change, and therefore their useful life will often be short.

According to the provisions of the standard and accounting practice, such assets are typically depreciated over a period of 3 to 10 years, with a period of 5 years most often used [12].

Taking into account a moderate level of volatility and a prudent assessment that combines the findings of empirical research and accounting standards, an expected initial life of 5 years was selected for software in the differentiating capabilities layer.

3.3. Innovative capabilities layer

Innovative capabilities represent the highest and most dynamic layer in Gartner's model [9]. It covers concepts in the early stages of development and solutions that enable rapid experimentation with new technologies or business models, such as "proof-of-concept" (POC) applications and solutions based on generative artificial intelligence. Change is correspondingly frequent, with the delivery or update cycle often measured in days.

Organizations use these solutions as so-called "short-lived" (ephemeral) applications, created for a specific, narrowly focused goal and quickly discarded after its fulfillment or unsuccessful testing [9]. Because they are extremely short-term investments, the lifespan of this class of software is too short for long-term longitudinal research, and the scientific literature therefore provides no extensive quantitative data on their survival. If such a technology proves successful, its functionality stabilizes and gradually moves to the lower layers, where it is maintained in the longer term.

Due to the extremely high technological volatility, experimental nature, and planned short-term life, the expected baseline lifespan of software in the innovative capabilities layer is set at 1 year. The resulting three-level framework — ten, five and one year — is the baseline duration on which the remainder of this paper builds.

4. The beginning of software useful life

The framework presented so far answers the question of how long the initial useful life of software is, not the question of when it starts. The two questions are substantively independent: classifying software into a layer of foundational, differentiating, or innovative capabilities determines the length of the expected useful life interval, but says nothing about where on the time scale this interval begins. Specifying a duration without specifying a start thus leaves the initial useful life in the form of a "floating" interval — of known length but not fixed to a specific time base. This shortcoming makes the framework unsuitable for real-world valuations.

We introduce this distinction with a separate concept. Let t_0 define the moment from which we start counting the useful life of the asset, and let the initial useful life be L.

The remaining economic useful life at the valuation date t_v is in this case $L - (t_v - t_0)$. This equation is a core input to all valid valuation methods: in the income method it determines the horizon of the forecasted cash flows, and in the cost method it determines the extent of the obsolescence deduction from the replacement value [2]. The sensitivity of the result to t_0 is therefore the same as the sensitivity to L but t_0 is treated implicitly or not at all in existing methodological practice, while L is the subject of a developed framework.

The consequence of the oversight is measurable. Consider an enterprise resource planning (ERP) system, classified in the foundational capabilities layer, with a baseline life of L = 10 years, which is valued in 2026. If we take t_0 as the date of the first production launch in 2015, the remaining useful life is negative (10 - 11 = -1), which would mean an economically exhausted asset. If we take t_0 as the date of transition to a significantly updated version in 2021, the remaining useful life is five years. It is the same software, the same baseline life, and the same valuation date. The difference in the asset value stems solely from the choice of t_0. Without a methodologically sound rule for this choice, the

remaining useful life, and with it the value estimate, becomes a matter of arbitrary judgment, which is precisely what a professionally sound valuation must avoid.

The same mechanics apply at shorter time scales. Consider a custom pricing engine in the differentiating layer (L = 5): if t_0 is set at its first production launch in 2023, the remaining useful life in 2026 is two years; if a substantial 2025 rework is judged to have reset the asset's identity, it is four. On a five-year baseline, the two-year difference changes the value estimate materially more, in relative terms, than in the ERP case.

Determining t_0 turns out to be considerably more difficult than it seems at a first glance, for two separate reasons. The first is that there are several legitimate, but mutually inconsistent, candidates for the starting point (accounting, technical, and transactional), which give different dates for the initial introduction of the software. The second, more difficult reason is that software is not a static asset. It is repaired, supplemented, and upgraded over time, which makes it questionable not only when the period began, but also whether and when this count is reset or extended when changes occur. We discuss both reasons below.

4.1. Candidates for determining the starting point

Even at the initial introduction of the software, that is, before we even consider subsequent changes, there are several legitimate starting points for determining the starting point, which come from different disciplines and are not mutually exclusive. The choice between them is not a technical detail, as the gap between them can be several months, and in the case of large-scale systems, several years, which is directly transferred to the remaining useful life.

4.1.1. Accounting starting point

International Accounting Standard 38 states that amortization of an intangible asset begins when the asset is available for use, that is, when it is in the location and condition necessary for it to operate in the manner intended by management [2]. This criterion is relatively straightforward for physical assets, but for software it raises two problems. First, "availability for use" is not the same as actual use. An asset may be installed and operational, but not yet in productive use in practice due to parallel operation with the old system, gradual introduction by organizational units, or ongoing acceptance testing. Second, in the case of internally developed software, the very moment of completion of the development phase and thus the transition from recognition of development costs to availability for use [2] is the subject of assessment and not an objectively given date.

4.1.2. Technical starting point

From the perspective of information systems management, the natural starting point is the moment of production launch, when the software moves from the development or test environment to the production one and begins to support actual business processes. This moment is also the milestone from which Huang and Yasuda (2016) consider the central, sustained phase of the system life cycle [11]. The technical baseline is often closest to the truth in terms of economic content, as the economic benefits of an asset only begin to be generated with actual use, but it can differ significantly from the accounting baseline. In the gradual introduction of a large-scale system, the "production start-up" is itself a stretched-out event without a single date.

4.1.3. Transactional baseline

For acquired (purchased) software, the date of purchase or acquisition is offered as the baseline, which is tied to the acquisition date for the purposes of allocating the purchase price in mergers and acquisitions (PPA) procedures. Here, the baseline reflects neither availability nor actual use, but the

legal transfer of the asset, which is appropriate for some valuation purposes, but misleading for others - purchased software may already be partially worn out at the time of acquisition if the seller has used it before.

4.1.4. Starting point

The discrepancy between these baselines is not an anomaly that could be eliminated by a more precise definition, but rather a consequence of the fact that each baseline measures a different economic content. The accounting baseline measures the readiness of the asset, the technical one its actual productivity, and the transactional one its legal transition. Which starting point is appropriate, therefore depends on the purpose of the valuation. In financial reporting, the accounting criterion of availability for use is binding. In forensic accounting and merger valuation, a technical or transactional starting point is often more appropriate, as it better reflects the actual economic condition of the asset. The starting point methodology must explicitly acknowledge this purpose-based conditionality and require the valuer to justify the choice of starting point in terms of purpose, rather than adopting a single universal definition.

This divergence is only the first, minor layer of the problem. All three starting points considered assume that the object of valuation is a stable artifact with a single, identifiable birth. Unfortunately, this assumption does not hold for software.

4.2. Software as an evolving artifact

The assumption of a stable artifact that from a single birth simply loses its value does not apply to an intangible asset of this type. Software that actually supports business processes is constantly being repaired, supplemented and upgraded throughout its entire life cycle, so the artifact on the valuation date is generally not the same artifact that existed at the chosen starting point.

This fact has direct consequences for determining the beginning and duration of the useful life and should be treated as a theoretical basis, not as a side issue. The systematic basis for this dynamic is provided by Lehman's laws of software evolution, originally formulated in the seventies and finally summarized in eight laws [14][15]. Lehman classified software into three types and showed that the laws apply to so-called E-type programs - programs embedded in the real world whose relevance depends on a changing environment. Three of them are essential for determining lifespan. According to the law of constant change, an E-type program must continuously adapt its operation to a changing environment, otherwise it becomes less and less relevant [14]. Hence, stagnation is not a state of value preservation, but a state of its decline. According to the law of constant growth, the functional content of a program must be constantly increased in order to maintain user satisfaction at the same level. According to the law of diminishing quality, the perceived quality of a program will seemingly decline if it does not adapt to changes in the environment.

The common implication of these three laws is that maintaining the economic usefulness of software is conditional on constant investment in change. That means that the very thing that preserves the useful life also changes the identity of the asset whose life we are trying to measure.

A complementary perspective is offered by Parnas's theory of software aging, which attributes aging to two different causes [16]. The first is the lack of adaptation. Software ages because the owner does not update it in accordance with changing needs, which causes it to lag behind the environment. This is aging due to dormancy. The second, more subtle reason for evaluation is interventions without understanding the original design. Changes made by people without knowledge of the original architecture gradually break down the structure of the software, increase its internal complexity and reduce its durability to future changes. This is economic and structural aging, to be distinguished from the same-named phenomenon of aging during execution treated in the software rejuvenation literature. Parnas's second reason explains the contradictory doctrine of maintenance. An upgrade can extend the useful life, but at the same time, if carried out without understanding the design, it increases structural obsolescence and thus shortens the useful life of future versions. Both

theories lead to a conclusion that redirects the consideration of the beginning of the useful life. If an artifact changes in content over time, the question is no longer just when the life began to run, but whether the object of evaluation remains the same asset at all. A sufficiently extensive upgrade may mean that we are no longer faced with an old asset in a later phase of the life cycle, but in an economic sense a new asset with its own starting point. A minor repair, however, has no such consequences. Between these two extremes lies a continuous range of changes for which the existing methodology has no rules on how it should affect the beginning and duration of the useful life. The operational typology of these changes and the resulting question of resetting or extending them is further discussed.

4.3. Maintenance and upgrades: reset, extension or neutrality

Software changes are not a single category. The ISO/IEC/IEEE 14764:2022 standard breaks down maintenance into four types, which it groups into two groups according to their purpose. The repair group includes corrective maintenance, which eliminates already detected errors, and preventive maintenance, which detects and eliminates latent errors before they manifest themselves in operation. The improvement group includes adaptive maintenance, which adapts the software to a changed or changing environment, and perfective maintenance, which improves performance, durability or other properties.

This typology is important for our research because it distinguishes between changes that merely maintain the asset in its originally intended state and changes that add content to it. Only the latter can logically affect the start or duration of the useful life.

A repair generally only maintains, not increases, the monetary benefit. From this breakdown, it follows that a change can have one of three effects on the start and duration of the useful life.

In a reset, the change is so extensive that on the valuation date the artifact is no longer the same asset as before, in economic terms, but a new asset with its own, later start of counting t_0. In an extension, the change adds functionality or moves the obsolescence back, but maintains significant continuity with the previous state, so it is appropriate to partially extend the duration without completely resetting the beginning.

In a neutral effect, the change merely maintains the asset in its intended state and does not affect the beginning or duration.

All three actions are legitimate in principle. The question remains open and unresolved as to which criterion should be used to classify a specific change as one of them. The first, seemingly simple option is to base this classification on the version label. That is, to distinguish between a major version, a minor version, and a revision. This approach does not hold up. Version labels are negotiable and are assigned by the provider according to its own technical or market criteria that are not related to the economic content of the change. A minor version may bring a major functional update, while a major version upgrade may be largely the result of a user interface redesign, a change in the supporting platform, or simply a marketing decision. The relationship between the version number and the actual scope of the change is therefore loose and unreliable, and the version number cannot serve as a criterion for deciding whether to reset, extend, or be neutral. Even the typology from the ISO/IEC/IEEE 14764:2022 standard, although necessary, is not sufficient in itself [17]. The typology classifies changes by their nature (what they are intended for), not by their scope (how much they change the assets). The scope is decisive for deciding when the useful life starts and how long it lasts. A major adaptive change may be transformative and warrant a reset, while a minor refinement of the same type may have negligible effects. The nature of the change is thus a necessary but not sufficient input.

The missing criterion is the quantification of the scope. The situation is further complicated by the fact that existing normative and technical frameworks address adjacent but different issues. Accounting standards deal with the issue of recognition for subsequent expenditures - whether to capitalize the expenditure or recognize it as an expense. For this they offer a qualitative criterion related to added functionality or increased economic benefits (ASC 350-40-25-7 and 350-40-25-8; IAS

38, provisions on subsequent expenditures). This criterion answers a different question than ours. It tells whether the cost of the change increases the carrying amount of the asset, but not whether and by how much the change advances the start or duration of the useful life. Software engineering, on the other hand, offers instruments for measuring the scope of the change. Among them the adjustment parameters of the COCOMO II model and the counting of functional points of additions according to the IFPUG method, which are not designed for valuation purposes and do not in themselves provide thresholds for deciding on the useful life. There is thus a methodological gap between qualitative accounting criteria and quantitative engineering instruments that existing practice cannot bridge, which we will focus on further.

4.4. Insufficient grouping

The layers of the problem discussed so far can be summarized into a single thesis, which is also the central position of this paper. The classification of software into a capability layer is a necessary but in itself insufficient condition for determining the remaining economic useful life. The Pace-Layered Application Strategy and the three-level framework derived from it (10 years for foundational, 5 years for differentiating, 1 year for innovative capabilities) answer exclusively the question of duration. The classification framework is silent on the two components that are necessary for calculating the remaining useful life, based on the start of the count and its possible reset during evolution.

This silence is not a shortcoming of the specific model that could be eliminated by a more precise classification or by an additional layer. It is a consequence of the fact that grouping and determining the beginning answer dimensionally different questions. Classification is the definition of a type of asset and assigns it an expected useful life. Determining the beginning is the definition of an event on a time axis and attaches an interval of this length to a specific moment. No refinement of the classification can produce a date, because classification is not the type of information from which a date would follow. It is therefore illusory to expect that the problem of the beginning would be solved by upgrading the classification framework. Both problems must be addressed separately, with their own criteria.

That classification is not sufficient is best seen if we assume that the classification is perfect. Suppose we can flawlessly place every piece of software in the appropriate layer and thereby assign it the correct initial duration. Two systems in the same layer, with the same duration, can still give completely different remaining useful lives. If the first was introduced ten years ago and has only been maintained since then, and the second was thoroughly overhauled two years ago. The classification treats them the same, but economic reality separates them. The difference does not stem from the type of asset, but from its temporal and evolutionary history, which the classification does not capture. A perfect classification therefore does not eliminate the uncertainty of the remaining life. This indeterminacy is only eliminated by a justified determination of the beginning together with a rule for its treatment in the case of upgrades.

In evolution, this insufficiency is intensified. The classification framework is static. It places an asset in a layer and assigns it a duration, but it does not assume that the asset's identity changes over time. As the previous subsections showed, this is exactly what happens. A sufficiently extensive upgrade can mean the emergence of an economically new asset with its own beginning. The classification framework does not detect this event and does not offer a criterion for whether the beginning of the count should be reset, extended, or left unchanged upon upgrade. Classification into a group is thus not only insufficient in the static case, but completely silent in the dynamic one, which in practice is the rule rather than the exception.

This is not a rejection of the classification approach: the three-level framework remains a sensible and empirically supported basis for determining the initial duration, and we maintain it. It solves one of the three components of a reliable assessment, but by its nature cannot address the other two — the beginning and its treatment during evolution — which is the research gap we formulate next.

4.5. Definition of the research gap

From the discussion so far, the research gap can be formulated. The existing software valuation methodology develops a criterion for the duration of the initial useful life, but does not offer a substantiated criterion for either starting to count this period or for its treatment during upgrades and additions.

The first, static layer of the gap is represented by the divergence between the accounting, technical and transactional starting points, which cannot be eliminated by a more precise definition, because each starting point measures a different economic content and its relevance is conditioned by the purpose of the valuation.

The second, dynamic and more difficult layer is represented by the absence of a criterion according to which a specific software change would be classified as a reset of the beginning, a partial extension of the duration or a neutral intervention. It is this second layer that is systematically not addressed in the literature at the level of valuation methodology and constitutes the original contribution to which this paper calls.

The direction of the solution is evident from the structure of the problem, although its implementation exceeds the scope of this paper. Since the version designation has proven to be unreliable and the maintenance typology according to the ISO/IEC/IEEE 14764:2022 standard is necessary but insufficient, the decision criterion must be based on the quantification of the extent of the change relative to the existing asset.

Such a criterion would build on engineering instruments for measuring extent of change - the adjustment parameters of cost estimation models and the counting of function points of additions. Those would need to be mapped into a synthetic extent indicator and calibrated with thresholds that distinguish between the three effects on the start and duration of the useful life. For a small extent of change, the effect would remain neutral, for a moderate extent the duration would be partially extended without shifting the start, and for a large extent the change would mean the creation of an economically new asset with its own, reset start.

A threshold model would bridge the gap between qualitative accounting metrics and quantitative engineering tools that current practice leaves open, and would translate the question of when to start from an arbitrary judgment into a traceable, repeatable process. Calibrating the thresholds of such an indicator and determining how much the duration is actually extended at a given rate of change requires an empirical basis that seems to be lacking in the existing literature. The available empirical studies of software lifespan, including Tamai and Torimitsu (1992), Aier et al. (2009), and Huang and Yasuda (2016), measure the duration of systems, not the impact of individual upgrades on that duration. They were designed as analyses of the survival of the artifact as a whole and do not dissect how a single evolutionary event shifts or resets the lifespan. There is very scarce (or virtually no) data in the available literature that would link the measured extent of a specific upgrade with the subsequent extension of the useful life, as such measurements require longitudinal monitoring of the same system through several evolutionary cycles while simultaneously recording the extent of each change, a rare combination or absent in published research. Predictive models for estimating remaining useful life, as developed in prognostics for physical assets, do not remove this obstacle: they derive their estimates from measurable degradation signals collected on the individual asset, whereas software exhibits no physical degradation, and the longitudinal per-asset data such models would need is precisely what the literature lacks. Their per-asset data collection would, moreover, exceed the cost constraint described in Section 2.5. Predictive approaches may become valuable once the empirical basis described here exists, but they cannot substitute for it.

The next phase of research will not be able to simply summarize and rearrange existing findings, but will have to establish the empirical basis. This is a significantly greater research effort compared to synthesizing the already available data, collecting longitudinal data on the evolution of real systems, operationalizing the measurement of the extent of changes in these systems, and calibrating

thresholds on the material thus obtained. Before this empirical work, it makes sense to take a separate step: a systematic literature review that formally confirms and delimits the existence of the gap in order to avoid duplication of any existing but scattered findings. It is precisely the absence of definitive empirical answers that places the determination of the beginning of the software useful life among the open and research-relevant questions of intangible asset valuation.

5. Conclusions and key takeaways

The elements presented in this paper combine into a single valuation model as follows:

4. Classification of the software into a pace layer yields the baseline useful life L (ten, five or one year), supported by empirical lifespan studies and accounting standards. This component is available today.
5. A purpose-conditional rule selects and justifies the starting point t_0 among the accounting, technical and transactional candidates. This component is formulated here as a requirement; its operationalization is open.
6. A change-extent threshold rule classifies each subsequent upgrade as neutral, duration-extending or identity-resetting, and updates t_0 and L accordingly. This component constitutes the core of the identified research gap.
7. The remaining useful life L - (t_v - t_0) then enters the income or cost method as required by IVS, with obsolescence assessed against the software product quality model of ISO/IEC 25010:2023 [4] within the broader methodology outlined in Section 2.5.

The gap concentrated in components (2) and (3) cannot be closed by refining the classification framework, because classification and dating answer dimensionally different questions. Closing it will require an empirical basis that the existing literature does not appear to provide: longitudinal data linking the measured extent of individual upgrades to subsequent extensions of useful life. We plan to begin with a systematic literature review that formally confirms and delimits the gap, and we invite researchers from valuation, accounting and software engineering to join this effort, within the industrial constraints of consistency with classical valuation practice and proportionate cost of engagement.

Declaration on Generative AI

During the preparation of this work, the author(s) used **Google Translate** in order to: **Text Translation**, and **Google Gemini** in order to: **Improve writing style and Grammar and spelling check**. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Končar, K. Lušnic, D. Mikolič, S. Javornik, Ocenjevanje vrednosti programske opreme, Sir*ius 4 (2025). https://sirius.si-revizija.si/articles/1179/show (accessed June 20, 2026).

[2] International Valuation Standards Council, International Valuation Standards, 2025.

[3] Slovenian Institute of Auditors, Uporaba umetne inteligence pri ocenjevanju vrednosti: priložnosti, tveganja in standardi, Sir*ius 3 (2026). https://sirius.si-revizija.si/articles/1347/show (accessed June 20, 2026).

[4] International Organization for Standardization, ISO/IEC 25010:2023 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model, Geneva, 2023.

[5] D. Olson, Software & Valuation in the Information Society, Part Four: The Software Inventory Valuation—TOV (MV), Les Nouvelles — Journal of the Licensing Executives Society International (2009) 171–179.
[6] U. binti A. Rahman, A.F. binti Arbain, W.M.N. bin W. Kadir, D.N. bte. A. Jawawi, C.F. binti M. Foozy, E. Sudarmilah, Valuation of software product: a systematic literature review, Proceedings: Computer Science 1 (2023). https://doi.org/10.55092/pcs2023020049.
[7] R. Gómez-Cram, A. Lawrence, The Value of Software, American Economic Review 115 (2025) 3514–3558. https://doi.org/10.1257/aer.20230463.
[8] M. Khurum, T. Gorschek, M. Wilson, The software value map — an exhaustive collection of value aspects for the development of software intensive products, Journal of Software: Evolution and Process 25 (2013) 711–741. https://doi.org/10.1002/smr.1560.
[9] T. Iles, L. Mangi, D. Gaughan, Apply Pace Layers to Your Application Strategy to Align With Business Needs, Gartner, Inc., Stamford, CT, Rep. G00825745, 2025.
[10] T. Tamai, Y. Torimitsu, Software lifetime and its evolution process over generations, in: Proceedings Conference on Software Maintenance 1992, IEEE Comput. Soc. Press, 1992: pp. 63–69. https://doi.org/10.1109/ICSM.1992.242557.
[11] T. Huang, K. Yasuda, Reinventing ERP Life Cycle Model: From Go-Live To Withdrawal, Journal of Enterprise Resource Planning Studies (2016) 1–21. https://doi.org/10.5171/2016.331270.
[12] S. Aier, S. Buckl, U. Franke, B. Gleichauf, P. Johnson, C.M. Schweda, J. Ullberg, A Survival Analysis of Application Life Spans based on Enterprise Architecture Models, in: EMISA 2009, LNI P-152, 2009.
[13] Evropska komisija, Uredba Komisije (EU) 2023/1803 z dne 13. septembra 2023 o sprejetju nekaterih mednarodnih računovodskih standardov v skladu z Uredbo (ES) št. 1606/2002 Evropskega parlamenta in Sveta (Konsolidirano besedilo: 01.01.2025), 2023. https://eur-lex.europa.eu/legal-content/SL/TXT/HTML/?uri=CELEX:02023R1803-20250101 (accessed June 20, 2026).
[14] M.M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (1980) 1060–1076. https://doi.org/10.1109/PROC.1980.11805.
[15] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, W.M. Turski, Metrics and laws of software evolution-the nineties view, in: Proceedings Fourth International Software Metrics Symposium, IEEE Comput. Soc, 1997: pp. 20–32. https://doi.org/10.1109/METRIC.1997.637156.
[16] D.L. Parnas, Software aging, in: Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society Press, Washington, DC, USA, 1994: pp. 279–287.
[17] International Organization for Standardization, ISO/IEC/IEEE 14764:2022 — Software engineering — Software life cycle processes — Maintenance, 2022. https://www.iso.org/standard/80710.html (accessed August 14, 2026).