What Is the Value of Your Software?

Jelle de Groot*, Ariadi Nugroho*, Thomas Bäck† and Joost Visser*‡

*Software Improvement Group, The Netherlands – {j.degroot, a.nugroho, j.visser}@sig.eu
†LIACS – Leiden University, The Netherlands – baec@liacs.nl
‡Radboud University Nijmegen, The Netherlands

Abstract—Assessment of the economic value of software systems is useful in contexts such as capitalization on the balance sheet and due diligence prior to acquisition. Current accounting practice in determining software value is based on the cost spent in software development. This approach fails to account for the efficiency with which software has been produced or the quality of the product. This paper proposes three alternative models for determining the production value of software, based on the notions of technical debt and interest. We applied the models to 367 proprietary systems developed by a range of different organizations using a range of different programming languages. We present the valuation results and discuss the weaknesses and strengths of the models.

Keywords—software value; capitalization; due diligence; software quality; technical debt

I. INTRODUCTION

Nowadays software systems support a wide range of processes in business organizations. This massive use of software systems often requires considerable investments and costs [1]. Interestingly, regardless of doubts concerning the real value that software delivers to businesses [2], the amount of investments in software keeps growing. One reason of this development is that technology adoption is seen as a competitive advantage that keeps a company ahead of its competitors.

However, many investments in software are known to be troublesome. Often they do not deliver software according to expectations and consequently do not realize expected benefits. This problem stems from the fact that software development is a complex process involving technical and organizational issues, which in turn has a significant effect on the resulting software. Furthermore, software is an intangible product, and its value cannot be observed in a straightforward manner. Therefore, methods that can estimate the economic value of software systems may help IT executives better understand the value of IT investments.

Take as an example the current accounting practice of capitalizing software assets. In the current accounting practice, software value is determined based on development costs [3], [4]. However, we have seen that none of the existing techniques seem to offer a realistic assessment of software value. Furthermore, they leave much room for interpretation, which results in different approaches used by firms in determining the value of their software assets. Some firms capitalize all or a large amount of their software investments, which results in improved short-term financial statements. Other firms expense software investments, which results in improved long-term financial statements [5], [6]. Due to this discrepancy, it can be difficult to compare financial statements of different firms.

Our earlier study involving eight corporate executives confirms the need for a more realistic and objective software valuation method [7]. Considering the lack of methods to assess the value of software, we propose software valuation models that are based on the notion of technical debt [8]. Technical debt represents quality problems in software, and we use the quantification of such problems to determine the value of software.

This paper is organized as follows. In Section II we position our work with respect to various perspectives on software value and we discuss the notions of technical debt and interest that will be used in our own proposal for dealing with software value. In Section III we discuss how we use these notions to construct software valuation models that take inherent properties of software systems into account. In Section IV we present an explorative repository study where the models were applied to data of over 350 software systems. In Section V we discuss the results. In Section VI we discuss related work, and finally conclusions and future work are presented in Section VII.

II. ASSESSING THE VALUE OF SOFTWARE

In this section we discuss various perspectives on software value, current practices to measure it, and our proposal to deal with software value through the notions of technical debt and interest.

A. The Notion of Software Value

As is the case for any asset, the value of software assets can be viewed from three perspectives, namely production, exchange, and value of use. This is illustrated in the taxonomy of software value in Figure 1.

Under the use perspective of value, an asset derives its value from its current or future use, i.e. from the benefits that its owner can generate by its exploitation. Typical benefits of software assets include innovation through the introduction of new products or services, optimization of existing production processes, conformance to standards or regulations, and improvement of the quality of products. For
example, under this perspective, the value of a software system that supports a profitable business process can be said to amount to the profit that will be made during the lifetime of the software system. Determining such value of use may involve a high degree of uncertainty and may vary strongly depending on the owner. The value of use is sometimes called business value and is a typical ingredient in the business case for software investments.

Under the exchange perspective of value, an asset derives its value from the price that emerges from supply and demand on a competitive market place. This notion of value is difficult to apply to assets that are not generic, tradeable commodities but unique, one-of-a-kind artefacts. For software, we must make a distinction between licenses for the use of software, which are similar to tradeable commodities, and the software product itself, which is one-of-a-kind and is owned by a single organisation. As a consequence, software products do not have a price that emerges from exchange on an open market place, and there is no useful notion of exchange value of software products.

Under the production perspective of value, asset value is determined by the costs to produce it. Production cost typically is composed of material costs (the bricks and mortar for a house) and labour costs (the work of turning bricks and mortar into a house). For software assets, material costs correspond to licenses of third-party components that are used in the development of the software. Labour costs correspond to the effort of architects, programmers, testers, etc. to design, build, configure, test, and deploy the software. Both licences and development effort are commodities for which the notion of exchange value is valid.

The focus of this paper is on the value of software viewed from the production perspective. Therefore, the term software value used in the rest of this paper refers to the production value. Production value is not intended as a replacement for exchange value or value in use, but can be used as an anchor point when determining a fair price for a software product, both when acquiring the product from another organisation (exchange) or when investing into the development of a new software product for internal use.

B. Current Practice in Determining the Value of Software

Existing approaches to software valuation are based on the cost to develop the software, capped by possible future benefits that can be achieved with the software [3], [4]. These approaches fail to take into account the state of a system. Just as a house is worth more when it is well maintained, so is a software system. Currently development cost is used to identify software value but development cost does not necessarily add value. For example, sometimes features are developed that never make it into the final product. Or new technologies are experimented with, resulting in learning costs rather than product value. Expensive overheads for travel and accommodation of external specialists also increase project costs without an essential contribution to the value of the product. This means inefficient development would under current practise potentially lead to higher estimations of software asset value.

C. Technical Debt and Software Value

Cunningham defined technical debt as problems or quality issues in software that will cost organizations owning the software greater expenses if the problems are not resolved [8]. In our previous work, we defined two underlying aspects of technical debt, namely debt and interest [9]. Debt represents the costs to repair problems in software systems in order to achieve an ideal level of quality. The interest of a debt represents additional costs to maintain software systems due to the lack of quality.

Figure 2 visualizes the notions of technical debt. Technical debt might grow over time if not resolved, particularly if it concerns system parts that often change. Growing debt will subsequently lead to an increase in maintenance cost. Furthermore, we assume that there is an ideal level of quality, at which performing maintenance tasks will be most cost-effective. Technical interest is the cost difference between performing maintenance at the ideal level and any
levels below it. Therefore, paying technical debt to reach the ideal quality level will pay off in terms of zero interest (extra costs) in performing maintenance tasks.

III. THE SOFTWARE VALUATION PYRAMID

Our proposal for determining the production value of software in such a way that the inherent properties of the system is taken into account is depicted in the form of a pyramid in Figure 3. The pyramid has three levels. The Software Development level contains the inherent properties of a software system that serve as input for the valuation model. At the Application Portfolio Managment level, these inputs are translated to various effort estimates that can be used to inform portfolio-level investment decisions. At the Enterprise Management level, the various effort estimates for software assets are summarized in value estimations. We will discuss these levels in more detail below.

A. The Software Development Level

At the lowest level, the building blocks of software value are Quality, Volume, and Technology. These three aspects represent the technical state of a system and are the main concern of the software development team.

Technology: Technology concerns the programming languages, third-party components, development frameworks etc. that are used in the construction of a software system. Typically, software systems use more than a single technology. A stack of technologies is used because different aspects (layers, components) of a system need to be implemented using different approaches to reach an optimal solution. Every technology has its own strengths and weaknesses and comes with its own productivity characteristics.

Volume: The volume of a system concerns the technical size of the various components from which it is constructed. Typically, the volume is measured in terms of total lines of code. But some software artifacts might call for other units of measurement, such as the number of elements in a BPEL process definition. When the various volumes of software artefacts of different types are aggregated into a single volume measurement, the differences in measurement units need to be taken into account. When aggregating the volume of programs written in different programming languages, we take the productivity differences between these languages into account, as will be explained below.

Quality: We make a distinction between the functional quality of a software system, which is concerned with the degree to which it satisfies the functional and non-functional requirements of its end users, and its technical quality, which is concerned with the degree to which sound engineering principles have been applied in its construction. For determining the production value of a software system, we only take technical quality into account. To measure technical quality, we use the maintainability model of the Software Improvement Group (SIG) [10].

The SIG maintainability model measures various intrinsic characteristics of a software system, such as the complexity of its code units, the degree of duplication in program text, coupling between modules, etc. The metrics for such directly observable properties are aggregated into a 5-star quality rating according to a layered model as shown in Figure 4. The ratings produced by the model are calibrated each year against a benchmark of hundreds of software systems, such that the best 5% of modern software system will be awarded with 5 stars, the worst 5% is awarded 1 star, and the systems inbetween these extremes are evenly distributed over 3, 4, and 5 star ratings [11].

B. The Application Portfolio Management Level

This level mainly concerns the costs to operate IT systems, and as such will be the main area of responsibility of managers who need to continuously maintain the efficiency of running IT systems.
Rebuild Value: The rebuild value (RV) of a software system is basically a technology-neutral measure for technical volume. Rebuild value is calculated in two steps. Firstly, the technical size measurement for each type of software artefact is multiplied with its productivity factor to obtain a size measurement in terms of the number of person-month effort that would be required to rebuild that artefact from scratch. Secondly, these effort estimations are summed to obtain a single rebuild value for the entire system. We use the word value because this effort estimation will serve as the first, unadjusted estimation of the production value of a system. Thus, rebuild value is based on Technology and Volume from the base level of the pyramid.

Repair Effort: Repair Effort (RE) is equal to the technical debt of a system. It represents the amount of effort needed to improve the quality of a system to the ideal level, and can be quantified as follows:

\[
RE = RF \times RV \times RA
\]

The rework fraction (RF) is an estimated percentage of lines of code that needs to be changed in order to improve the level of quality of a system to an ideal level (Figure 5)—the ideal quality level is chosen depending on specific needs of a project (in this paper we use 4-star as the ideal quality level). It is based on Quality input from the first level of the pyramid.

The rework fraction is multiplied with the rebuild value to obtain the technical volume (quantified in person-months) of code that needs repair. To account for the fact that this volume of code does not need to be rebuilt, but only needs to be refactored, we apply a Refactoring Adjustment (RA). This factor is a percentage discount which may vary with the technology, the available refactoring support, and available documentation.

Maintenance Effort: The maintenance effort (ME) is the yearly effort that is expected to be needed for regular maintenance of the system, including bug fixing and small enhancements. It is calculated as follows:

\[
ME = \frac{MF \times RV}{QF}
\]

The Maintenance Fraction (MF) is the percentage of lines of code that is expected to be modified due to maintenance on a yearly basis. Typical values for MF are between 5% and 15%. The MF percentage is multiplied with the technical volume of the entire system to obtain the technical volume (in terms of person-months) that needs annual change. To reflect that higher quality software require less effort to perform changes, we additionally apply a Quality Factor (QF). This factor ranges from 0.5 up to 2.0 from the lowest to the highest quality level (1 star - 5 star).

As explained earlier, the difference between maintenance effort for a system at its current quality level and at the ideal level can be regarded the technical interest that one pays over the unresolved technical debt in a software system.

Portfolio Management: Rebuild value, repair effort, and maintenance effort are important inputs for decision-making at the system and portfolio level. For example, they can be used to construct return-on-investment (ROI) calculations where initial repair effort (repaying technical debt) is compared against cumulative increased maintenance effort (technical interest), or against complete redevelopment of software systems (rebuild value). Examples of such calculations are provided elsewhere [9].

C. The Enterprise Management Level

At the enterprise management level, corporate executives consider software as assets that can be acquired, maintained and exploited, or sold. We discuss three alternative models by which the information of the lower levels of the software value pyramid can be aggregated into estimations of the production value of software assets. All models take the rebuild value of a system as starting point and then apply different impairments.

Model 1: Impairment based on Repair Effort: When buying a car with a dent, one would subtract the repair costs from the bid. Analogously, our first valuation model subtracts the repair effort from the rebuild value:

\[
V_1 = RV - RE
\]

Repair effort is equal to the amount of technical debt. Subtracting technical debt from the rebuild value essentially means discounting the cost of repair of problematic parts in the software from its production value.

Model 2: Impairment based on Rework Fraction: Rather than fixing the dent in a car, one might opt for replacing the dented part altogether. Analogously, our second valuation model reduces the rebuild value by the fraction of the software system that is of suboptimal quality.

\[
V_2 = RV \times (1 - RF)
\]

In contrast to the previous model, this model does not make use of the refactoring adjustment (RA). As a consequence, the model is simpler, but also applies a more harsh impairment.

Model 3: Model based on Technical Interest: A third option is to keep the car in its dented state and accept higher running costs (e.g. due to increased air resistance) or higher maintenance costs (e.g. related rust under cracked paint). Our third valuation model therefore impairs rebuild value by the increased software maintenance costs due to suboptimal quality.

\[
V_3 = RV - \sum_{i=1}^{5} TI_i
\]

Here, \( TI_i \) is the technical interest incurred in year \( i \). We use a life expectancy of five years as this is a common practice in amortizing software assets in accounting.
IV. EXPLORATIVE STUDY

In order to study the behaviour of the models, we performed an explorative study where we applied the models to existing data of 367 software systems. The data was taken from the software analysis warehouse (SAW) of the Software Improvement Group (SIG). This software was provided by clients who requested quality assessment or monitoring of their software. For a few systems, we additionally approach the IT executives (CIOs and/or CFOs) in the client organizations in order to discuss the extent to which the valuation models provide realistic and useful results.

Some input parameters were set equal across systems to enable comparison, such as refactoring adjustment and yearly maintenance rate. Furthermore, we assume the cost of 1 FTE to be 100,000 euros annually.

A. Descriptive Statistics

The systems in the data set are comprised of multiple main technologies. The main or dominant technology is defined as the largest technology portion in a system and it exceeds 60% of the total code. Figure 6 shows the distribution of main technologies in the data set. A system is considered having a mixed technology if a dominant technology (above 60% of the total lines of code) is absent in the system.

Table I shows the descriptive statistics of the most important metrics for the 367 systems.

The size of the systems ranges from a tiny system of 290 lines of code up to 2.8 million lines of code. Half of the systems has 77 thousand lines of code or more, and 25% has 191 thousand lines of code or more. As expected, the distribution of sizes is highly skewed, as revealed by the large difference between median and mean system size.

All quality levels (from 1 to 5 stars) are represented in the sample, and the mean/median quality rating is 3 stars. The distribution of star levels is fairly symmetric.

The rebuild value ranges from a couple of weeks (0.03 man-years) to over 6 man-centuries, with similar skewness as the lines of code measurement from which it is calculated (taking technology-specific productivity factors into account).

The rework fraction ranges from 0% to 123%. This result indicates that some systems already reach the ideal level and some others require to change up to 123% of the code (this means refactoring might require working on the same code fragments multiple times) to achieve the ideal level. The median of the required rework fraction is 35%.

The repair effort ranges from none to over 5 man-centuries. Half of the systems requires a repair effort of under 2 man-years, while 25% requires a repair effort of over 9 man-years. The distribution of repair effort is skewed.

B. How Much Value Does Software Have?

Table II presents the descriptive statistics of the results from the three software valuation models measured in euros per line of code. By looking at the mean or median of the results obtained by the three models we can see that there is no significant difference between the results of the three models. However, we can notice that Model 1 gives a slightly higher value than Model 2 and 3. Therefore, we can conclude that Model 1 generally is more lenient than the other models.

The fact that Model 1 is more lenient might be explained by the use of Refactoring Adjustment as one of its components. Model 1 takes into account factors in a project that might reduce the effort in solving problems in software. In the contrary, Model 2 ignores environmental factors and directly excludes parts that need rework from value calculation.

The nature of Model 3 is rather different from the other models. It is based on a cumulative technical interest over a five years period. Nevertheless, we can see that Model 3 seems as stringent as Model 2. All in all, the three software valuation models give an average software value of €7.26 per LOC.

Note that the negative values in Model 1 and 2 indicate that repairing the software to the desired ideal level is going to cost more than rebuilding it from scratch. However, in Model 3 the negative value means that within 5 years of utilization, the total extra value spent on maintaining the software will exceed the cost to rebuild the system.

C. Software Value across Technologies

To further explore the results, we differentiate the calculation of software value across different technologies (see Figure 7). Systems are rarely composed of only one technology; each category in the figure represents systems built using a dominant technology, as defined previously.

There do not seem to be a significant difference between the results of the three valuation models. All models give
Table I

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
<th>1st Quantile</th>
<th>3rd Quantile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (KLOC)</td>
<td>0.29</td>
<td>2,883.00</td>
<td>213.00</td>
<td>77.00</td>
<td>21.00</td>
<td>191.00</td>
</tr>
<tr>
<td>Quality rating (stars)</td>
<td>1.25</td>
<td>5.26</td>
<td>3.09</td>
<td>3.05</td>
<td>2.43</td>
<td>3.78</td>
</tr>
<tr>
<td>Rebuild Value (Man-years)</td>
<td>0.03</td>
<td>630.59</td>
<td>26.38</td>
<td>7.88</td>
<td>2.21</td>
<td>21.27</td>
</tr>
<tr>
<td>Rework Fraction (%)</td>
<td>0.00</td>
<td>123.00</td>
<td>38.00</td>
<td>35.00</td>
<td>10.50</td>
<td>59.00</td>
</tr>
<tr>
<td>Repair Effort (Man-years)</td>
<td>0.00</td>
<td>542.29</td>
<td>16.01</td>
<td>1.87</td>
<td>0.14</td>
<td>9.06</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>Valuation Model</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
<th>1st Quantile</th>
<th>3rd Quantile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1</td>
<td>0.21</td>
<td>19.42</td>
<td>7.70</td>
<td>7.80</td>
<td>5.55</td>
<td>10.16</td>
</tr>
<tr>
<td>Model 2</td>
<td>-5.03</td>
<td>18.81</td>
<td>6.71</td>
<td>6.96</td>
<td>4.14</td>
<td>9.59</td>
</tr>
<tr>
<td>Model 3</td>
<td>-6.64</td>
<td>18.96</td>
<td>6.65</td>
<td>7.02</td>
<td>4.16</td>
<td>9.86</td>
</tr>
</tbody>
</table>

quite similar distributions of software value across technologies. However, similar to the results presented in Table II, Model 1 tends to give higher software value than the other two models.

Model 3 generally gives lower values, particularly for technologies where systems typically have poor quality levels (ABAP for example). Recall that Model 3 is based on a cumulative technical interest over 5 years. Yearly technical interest grows proportionally to the system size, the yearly growth rate of which is higher for poor quality systems. The most extreme case is ABAP systems that have a median quality rating of 1.5 stars. The median of software value of ABAP systems drops rather significantly because given their poor quality much higher interest is paid in five years. Negative software values in Model 3 indicate that after five years the total interest is higher than the rebuild value of the system.

Figure 7 also shows interesting insights about software value across technologies. We observe that systems built using Java and C# have the highest value, averaging €10 per LOC, compared to systems built using other technologies. We also see that ABAP scores the lowest amongst the other systems.

D. Technical Debt across Technologies

In Figure 8 we present the amount of technical debt per line of code across different technologies. Java systems are shown to have the lowest technical debt—averaging €1.6 per LOC, while ABAP and PL/SQL are the two technologies with the highest technical debt—averaging €15.4 and €13.5 per LOC respectively. Following ABAP and PL/SQL, COBOL systems have the third highest technical debt, averaging €4 per LOC.

These findings contradict with the report published by CAST [12]. CAST reported that ABAP and COBOL systems have the lowest technical debt per KLOC, while Java systems were found to have the highest technical debt. Considering plausible differences in the method used to calculate technical debt discrepancies in the absolute terms are expected. However, in relative terms it is counter intuitive to learn that modern technologies such as .NET and Java were found by CAST to have higher technical debt compared to old technologies like COBOL. In our approach, old technologies such as COBOL are found to have higher technical debt, which is closer to intuition.

V. Discussion

In this paper we have presented three models to assess the value of software. We apply the models to a large collection of software and found that the results obtained by the three models are quite similar. The question concerning which valuation model to use then becomes a matter of preference. Model 1 and 2 are quite similar in nature. However, one would prefer Model 1 if a more realistic estimate, which takes into account factors influencing repair costs, is preferred. Model 2 will generally be chosen for its simplicity. Model 3 has a different nature, and one would choose this model if a repair scenario is not an option.

To understand how practitioners think of the valuation models, we have conducted several case studies reported in [7]. We identify important insights concerning their views on the software valuation models. Firstly, we found that practitioners generally agree that an objective framework for assessing software value is needed and still lacking in the current practice. Secondly, they consider the three models as complementary to each other in looking at software value from different point of views. Finally, the proposed models are seen as an improvement of the current valuation practice, which is based on total development cost.

Several CFOs that participated in the study considered the models as an important instrument for valuation during due diligence. The buyer can make a good assessment of the value of software assets in a quick and simple way.
When rationalizing a software landscape, software valuation may be a guiding principle in investment decisions. Instead of examining the incurred development costs, systems may be preferred on the basis of the required repair costs and future maintenance costs. The proposed valuation models are very suitable for comparing software portfolios of various organizations. This makes it possible to benchmark organizations in an objective way.

VI. RELATED WORK

The work of Sneller and Bots [13] is a review of quantitative IT value research in which all IT value quantitative research and methods are listed and compared after the year 2000. In the paper, IT value is defined as research that impacts financial performance of IT. The forty research papers found by Sneller and Bots can be separated in four categories; capital market theory, microeconomic theory, resource-based view of the firm and analysis of financial statements. Generalizable from these studies is that research focuses on the impact of IT on financial aspects. For example, IT’s effect on equity price or firms productivity or financial ratios like ROI and ROA. Concluding from the study, IT value research focuses on financial return rather than risk-reward trade-offs.

To the best of our knowledge little work has been done to define and measure software value based on intrinsic values and actual measurements. In this section, the discussion on related work is focused on proposed approaches to quantify intrinsic software value. At present, software value is determined based on a sum of costs. Ben-Menachem and Gavious disagree with this method, and argue intrinsic value of a software asset cannot be only the cost incurred [14]. Example given is a software module that serves as an interface between several systems. Although the module has a relatively low production cost but the high degree of reuse of the module should enhance its value. A model is proposed to calculate the real intrinsic value based on total costs and premium/discount factors.

The premium/discount factors are based on a software sensitivity theory designed by the authors. Software sensitivity is used to calculate how software change can affect the business environment. Four quantifiable parameters affect sensitivity: reuse count, complexity, update difficulty, and interface implementation. Reuse count refers to the quantity of uses for this module. Complexity and update difficulty are technical concepts...
referring to how the module is constructed. Interface implementation refers to the connection a module can have to several systems. There are several limitations in the model proposed by Ben-Menachem and Gavious. Firstly, the model does not seem to clearly explain the underlying aspects behind software valuation, which make it difficult to use for root-cause analyses. Secondly, the model is not validated. Therefore, it is very difficult to rely on the conceptual equation in the paper of Ben-Menachem and Gavious. Thirdly, the final result is based on several expert estimates of the software system that could be biased.

VII. CONCLUSION AND FUTURE WORK

In this paper we present three models to determine software value based on the notions of technical debt. We use Rebuild Value (the cost to rebuild a system using a similar technology) as the base value of software. Impairment to this base value is done based on the notions of technical debt. The first model uses repair cost to adjust the value of software. The second model uses a similar approach but neglecting project environmental factors to impair software value. Finally, the third model uses technical interest (extra costs spent on maintenance) as an impairment factor of software value.

We apply the three valuation models to 367 proprietary software and learn the following insights:

- There is no significant difference in the results provided by the three models. The average software value resulted from the three model is around €7 per LOC.
- Valuation model that is based on repair effort is the most lenient, and it is due to its recognition of factors that might influence productivity in performing rework.
- Across the three models, C# systems are found to have the highest value, averaging €10 per LOC.
- Java systems are found to have the lowest technical debt, averaging €1.6 per LOC.

Future work is needed to investigate the extent to which the results given by the proposed valuation models differ from that of conservative approach that is based on the actual costs spent during software development. Performing such an investigation in proprietary software projects might be difficult due to the sensitivity of the required information. Nevertheless, this endeavor is worth investigating as it might reveal whether there is a general trend of overstating the value of software. More research is also needed to broaden the scope of technical debt to cover more aspects beyond maintainability such as reliability and usability.

Finally, further research should look into the business value of software—the business and financial benefits obtained from running software systems. Combining the notions of production value, business value, and technical debt in software will give a more comprehensive view of the value of software for organizations.

REFERENCES