When to Stop Testing Software: Economic Approach

Thach Ngoc Nguyen

Banking University Ho Chi Minh City

Francisco Zapata

The University of Texas at El Paso

Olga Kosheleva (✉ olgak@utep.edu)

University of Texas at El Paso <https://orcid.org/0000-0003-2587-4209>

Research Article

Keywords: software testing, economic approach

Posted Date: April 6th, 2021

DOI: <https://doi.org/10.21203/rs.3.rs-389562/v1>

License: This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Version of Record: A version of this preprint was published at Soft Computing on April 26th, 2021. See the published version at <https://doi.org/10.1007/s00500-021-05818-x>.

When to Stop Testing Software: Economic Approach

Nguyen Ngoc Thach · Francisco Zapata · Olga Kosheleva

Received: date / Accepted: date

Abstract When should we stop testing software and release it? Many software engineering papers attempt to answer this question – from the computer science viewpoint. However, the ultimate objective of every company – and software companies are not an exception – is to bring profit. In view of this fact, in this paper, we analyze the problem of when to stop testing from the economic viewpoint. For the simplified first-approximation model, we provide an explicit answer, and we describe how this answer can be made more accurate by using more adequate models.

Keywords software testing · economic approach

1 Formulation of the Problem

Software testing: typical understanding. Nowadays, software is everywhere: it controls our cars, it controls street lights, it controls our planes, it controls how electricity from power stations gets to our homes, it controls how we type our papers and how we communicate.

Is it always working perfectly? No. There have been well publicized cases when software errors caused serious trouble. So why don't the software companies test it some more – to make sure that there are no bugs?

The authors are thankful to Hung T. Nguyen and Vladik Kreinovich for encouragement and valuable suggestions.

Nguyen Ngoc Thach
Institute for Research Science and Banking Technology
Banking University Ho Chi Minh City, Vietnam
E-mail: thachnn@buh.edu.vn

Francisco Zapata and Olga Kosheleva
University of Texas at El Paso, El Paso, TX 79968, USA
E-mail: fcozpt@outlook.com, olgak@utep.edu

If they are too greedy to do it on their own, maybe we should force them? Shame them?

This is what journalists write every time there is a software-related trouble, this is what many people think.

This typical understanding is naive. First of all, anyone who has studied Economics 101 understands that shaming is never a good economic strategy. If there is a problem in economy, it has to have an economics-based solution, shaming does not help.

For example, if in some area, food is too expensive, then shaming the store keepers and/or the farmers will not help, we need to analyze what causes the high prices. Maybe there are too many taxes on the farmers, so they cannot afford to sell at a lower price? Maybe taxes on imported food are too high, so cheaper food from neighboring countries cannot reach the customers? Maybe there are too many restrictions on selling land, so current landowners have a kind of monopoly? Maybe there are too many restrictions on opening new food stores – so no competing stores can enter the market? In all these cases, there are economic solutions. Without any serious changes along these lines, government restriction on prices will only lead to shortages and long lines – it has been tried many times before.

For software, there is an additional reason why the usual understanding is naive. The reason is that while it is, in principle, possible to make a small piece of code perfect and bug-free, there is no known way to make large million-lines-of-code software packages – like operating systems – completely bug-free. This is not just a theoretical idea: there are thousands of hackers out there, trying to find faults in the existing software, and in spite of all the efforts to protect, every year, they

manage to find a flaw and penetrate into the most supposedly secure and protected systems.

In general, this is not about software companies selling us buggy software because they want to save money on testing (although this, of course, happens too): because very expensive-to-design and supposedly secure military and intelligence databases and systems get hacked too, probably with similar frequency as much cheaper-to-design civilian systems.

Let us give another example. In the US, medical records are heavily guarded, there is a regulation that provides a \$1,000 fine for each violation of privacy – exactly the financial incentive that is supposed to force everyone to make software processing medical data flawless – and nevertheless, periodically, universities and companies handling this data get security breaches and have to pay million-dollar fines.

Resulting problem. Since we cannot completely get rid of software faults, no matter how much money and how much time we spend on testing, a natural question for each software company is: when to stop testing software?

There are many papers that deal with this questions on the software engineering level; see, e.g., [4] and references therein. What we do in this paper is consider this problem from the economic viewpoint.

2 Let Us Describe This Problem in Precise Terms

Main economics-related notations. As with every other economic situations, the decision on when to stop testing software is a trade-off between costs and benefits.

The cost, in this case, is the cost of testing software. In contrast to what many people think, software packages are extensively and thoroughly tested, and the cost of testing is a significant part of the overall cost. To gauge this cost, let us denote the cost of a single test by c_t . Then, if a company performs N_t tests before releasing the software, its overall testing cost is $N_t \cdot c_t$.

The benefit of extensive testing is avoiding penalties. Every time a fault is found in a software produced by a company, the company suffers financially. This may be the actual fine – as in the case of software that processes medical data. In other cases, the penalty comes from the need to spend resource on designing a patch – and from the potential loss of customers. Let us denote the overall penalty caused by a fault by P . The more we test, the smaller the overall penalty – this is the company's benefit from testing.

These are two economics-related notations corresponding to costs and benefits. To complete the cost-and-benefit analysis of the situation, we need to know when to expect the faults – depending on how many tests we have already performed.

Testing time. From the viewpoint of detecting bugs, the more times we run the software, the larger the possibility that we will find a bug.

During the testing, we run the software all the time – sometimes on parallel computers. Let us denote by Δt the average running time of this software. (We should not worry much about the testing time, since testing can be done in parallel, when several processors simultaneously test the software on different inputs.) So, after N_t tests, the software has run for an equivalent time of $N_t \cdot \Delta t$.

Once the software is released, we will have u uses per year – which is equivalent to u additional tests every year. So, during one year, the software will run for an equivalent time of $u \cdot \Delta t$.

How many faults will we find during a given time? Of course, different software packages are somewhat different. What we are looking for is a general description of such packages, a description that would be applicable – at least in the first rough approximation – to all kinds of packages.

What we want is to estimate the average time $t(n)$ at which the n -th bug will appear. At first glance, it therefore seems reasonable to look for a single universal function $t(n)$ describing this dependence. However, this would be too crude an approximation. First of all, we can start using software at different times. So, if we started using it t_0 years later, we will find all the bugs t_0 years later, so instead of the original function $t(n)$, we will have a function $t(n) + t_0$. Because of this, if $t(n)$ is a reasonable description of the time at which the n -th bug appears, then $t(n) + t_0$ is also a reasonable description of the same phenomenon. In other words, we cannot select a *single* function, we should be looking for the whole *family* of functions $\{t(n) + t_0\}$ corresponding to different values t_0 .

But this is not all. Different computers have different speed. We can run the tests on a faster or on a slower computer, and this will change the time at which the n -th fault will be detected. If the second computer is c times slower than the first one, then the time $t(n)$ for the first computer corresponds to the time $c \cdot t(n)$ for the slower second one. Thus, if $t(n)$ is a reasonable description of the times at which n -th fault surfaces, the function $c \cdot t(n)$ is also a good description of a similar situation.

If we take this into account, then we can conclude that instead of a single function $t(n)$, we should select a whole family of functions $\{c \cdot t(n) + t_0\}$ corresponding to different values t_0 and c .

Comment. The above family sounds like a reasonable first approximation to the desired dependence. In this and following section, we will analyze the main question of this paper – when to stop testing software – under this first-approximation model. In the last section, we analyze what happens if we use more accurate models.

What are reasonable families of functions: analysis of the problem. In principle, for different functions $t(n)$, we get different families $\{c \cdot t(n) + t_0\}$. Which of these families is the most reasonable?

To answer this question, we will follow the idea first described in [1,2]. Let us take into account that while we want a universal dependence, different people approach testing differently. Some people take a raw piece of code, and start testing it right away. Other people analyze the code attentively, check it – with pen and pencil – on simple inputs and only then, when they have found and corrected simple easy-to-find faults, only then they start the actual software testing.

In computer science classes, usually, students at first start with testing right away – since they may not yet have the skills to correctly manually trace the code. So, to incoming students, the ability of a teaching assistant or an instructor to look at the printout and see the problem looks like magic. Eventually, students learn this art, but still, the difference remains: some programmers do a very thorough manual checking and only then start automatic testing, while others do a perfunctory checking only and hope that automatic testing will help find the remaining faults.

The difference between these two approaches is that when the thorough programmer performs n tests, he had, in effect, already manually performed some additional number of tests; let us denote this additional number of tests by n_0 . The second programmer performs these additional tests automatically. So, at the moment when the first programmer perform n automatic tests, the second one performs $n + n_0$ of them.

The resulting expected number of bugs should not depend on whether the tests are performed first manually or whether, from the very beginning, they are performed automatically. Thus, by the time $t_1(n)$ when the first programmer finds the n -th fault, the second programmer already performed $n + n_0$ automatic tests, so $t_2(n + n_0) = t_1(n)$.

What are reasonable families: resulting mathematical problem. So, we can conclude that with each

reasonable function $t(n)$, functions $t(n + n_0)$ are also reasonable. We agreed that reasonable functions form a family $\{c \cdot t(n) + t_0\}$. Thus, we conclude that all shifted functions $t(n + n_0)$ – in particular, the function $t(n + 1)$ corresponding to $n_0 = 1$ – belong to this family, i.e.,

$$t(n + 1) = c_1 \cdot t(n) + t_1 \quad (1)$$

for some values c_1 and t_1 .

If $c_1 = 1$, then we have $t(n + 1) = t(n) + t_1$. This would imply that software faults appear with the same frequency, no matter how many tests we perform. This is not what we observe: in practice, the more we test, the longer the time to the next fault. Thus, $c_1 \neq 1$, and, as we will show, we can simplify the equation (1) by considering an auxiliary function $t'(n) \stackrel{\text{def}}{=} t(n) + C$, for some constant C . In terms of this new function, the original value $t(n)$ takes the form $t(n) = t'(n) - C$. Thus, from the formula (1), we conclude that

$$\begin{aligned} t'(n + 1) &= t(n + 1) + C = c_1 \cdot t(n) + t_1 + C = \\ &= c_1 \cdot (t'(n) - C) + t_1 + C = \\ &= c_1 \cdot t'(n) + (t_0 + C - c_1 \cdot C). \end{aligned} \quad (2)$$

In particular, if we select the constant C so that $t_1 + C - c_1 \cdot C = 0$, i.e., if we take $C = -\frac{t_1}{1 - c_1}$, then the formula (2) gets a simplified form

$$t'(n + 1) = c_1 \cdot t'(n). \quad (3)$$

The fact that the next bug appears after the previous one means that $t'(n + 1) > t'(n)$, i.e., that $c_1 > 1$.

A sequence that satisfies the formula (3) is known as a *geometric sequence* (or, alternatively, a *geometric progression*). It is known – and it is easy to prove by induction – that $t'(n) = t'(0) \cdot c_1^n$, and thus, $t(n) = t'(n) - C = t'(0) \cdot c_1^n - C$. Thus, all the functions $T(n)$ from the optimal family have the form

$$c \cdot t(n) + t_0 = c \cdot (t'(0) \cdot c_1^n - C) + t_0,$$

i.e., the form

$$T(n) = c' \cdot c_1^n + t', \quad (4)$$

where we denoted $c' \stackrel{\text{def}}{=} c \cdot t'(0)$ and $t' \stackrel{\text{def}}{=} c - c \cdot C + t_0$.

Conclusion of this section. The time $T(n)$ at which the n -th fault appears can be described by the formula (4).

How do we determine parameters of this formula? These parameters can be determined experimentally. The simplest idea is to look for the time $\Delta T(n) \stackrel{\text{def}}{=} T(n + 1) - T(n)$ between the two consequent faults. Based on the formula (4), this time has the form

$$\Delta T(n) = a \cdot c_1^n, \quad (5)$$

where $a \stackrel{\text{def}}{=} c' \cdot (c_1 - 1)$. To experimentally determine the values a and c_1 , it is convenient to take logarithm of both sides of the formula (5), then

$$\ln(\Delta T(n)) = n \cdot \ln(c_1) + \ln(a). \quad (6)$$

In this formula, the dependence on the unknowns $\ln(c_1)$ and $\ln(a)$ is linear, so we can use the usual linear regression methods – e.g., Least Squares; see, e.g., [3].

How many bugs did we find during testing. We denoted the number of tests performed by the software company by N_t , so overall, these tests are equivalent to running the software for time $N_t \cdot \Delta t$. We started at the moment corresponding to $n = 0$, for which $T(0) = t'$. After the time $N_t \cdot \Delta t$, we are at the moment $t' + N_t \cdot \Delta t$. We can use the formula (4) to find the number of faults n_t that we have discovered during this time. The formula (4) takes the form

$$T(n_t) = t' + N \cdot \Delta t = c' \cdot c_1^{n_t} + t', \quad (5)$$

hence

$$c' \cdot c_1^{n_t} = N_t \cdot \Delta t. \quad (6)$$

The next fault will be detected after the overall running time

$$\begin{aligned} T(n_t + 1) &= c' \cdot c_1^{n_t+1} + t' = \\ &= (c' \cdot c_1^{n_t} + t') + c' \cdot c_1^{n_t} \cdot (c_1 - 1) = \\ &= (c' \cdot c_1^{n_t} + t') + N_t \cdot \Delta t \cdot (c_1 - 1) \end{aligned} \quad (7)$$

i.e., at running time

$$N_t \cdot \Delta t \cdot (c_1 - 1) \quad (8)$$

after the software release.

According to our notations, the equivalent running time is $u \cdot \Delta t$ per calendar year. Thus, to get the calendar time t_1 from the software release to the first bug, we need to divide the expression (8) by the amount $u \cdot \Delta t$ of running time per calendar year. Then, we get the value

$$t_1 = \frac{N_t}{u} \cdot (c_1 - 1). \quad (9)$$

Similarly, the second next fault will be detected after the overall running time

$$\begin{aligned} T(n_t + 2) &= c' \cdot c_1^{n_t+2} + t' = \\ &= (c' \cdot c_1^{n_t} + t') + c' \cdot c_1^{n_t} \cdot (c_1^2 - 1) = \\ &= (c' \cdot c_1^{n_t} + t') + N_t \cdot \Delta t \cdot (c_1^2 - 1), \end{aligned} \quad (10)$$

so the corresponding calendar time is equal to

$$t_2 = \frac{N_t}{u} \cdot (c_1^2 - 1). \quad (11)$$

In general, the k -th fault will be discovered at the time

$$t_k = \frac{N_t}{u} \cdot (c_1^k - 1). \quad (12)$$

Comment. Now, we are ready to formulate the problem in precise terms – and to explain how to solve it.

3 Precise Formulation of the Problem and the Resulting Solution

Precise formulation of the problem: what is given. Suppose that we have performed N_t tests, and the cost of each test is c_t . During testing, we analyzed how the amount of faults decreases with time by fitting the times $t(n)$ at which we discovered the n th fault to the formula $t(n) \sim c_1^n$ for some constant $c_1 > 1$.

We expect that after the software release, our software will be used u times per year. Eventually, there will be faults, and the expected penalty for each fault is P . We also know the discounting r , so that a penalty of P one year from now is equivalent to the amount $r \cdot P$ now, the penalty 2 years from now is equivalent to $r^2 \cdot P$ now, etc. This makes sense: if we place the amount $r \cdot P < P$ into, e.g., a bank account now, then in a year it will grow to a larger value.

What we want to minimize. We want to find the value N_t for which the overall cost of testing and paying penalties is the smallest possible, i.e., for which we minimize the following expression:

$$N_t \cdot c_t + P \cdot r^{t_1} + P \cdot r^{t_2} + \dots + P \cdot r^{t_k} + \dots, \quad (13)$$

where the values t_k are determined by the formula (12).

This optimization problem determines when to stop testing software.

A case when an analytical solution is possible. In general, the formula (13) is complicated, so we need to perform numerical optimization. However, in some practically important cases, this formula can be simplified.

This possibility to simplify is related to the fact that the released software is usually reasonably safe: it may take several years for a fault to be detected. In this case, the main penalty comes from the first fault. Other faults are far in the future that their influence – because of the discounting – can be safely ignored. In this case, the formula (13) takes a simplified form

$$N_t \cdot c_t + P \cdot r^{N_t \cdot z}, \quad (14)$$

where we denoted

$$z \stackrel{\text{def}}{=} \frac{c_1 - 1}{u}. \quad (15)$$

Differentiating this expression with respect to t_1 and equating the derivative to 0, we conclude that

$$c_t + P \cdot \frac{c_1 - 1}{u} \cdot r^{N_t \cdot z} \cdot \ln(r) = 0, \quad (16)$$

hence

$$r^{z \cdot N_t} = \frac{c_t \cdot u}{P \cdot (c_1 - 1) \cdot |\ln(r)|}, \quad (17)$$

and the optimal number of tests is therefore equal to

$$N_t = \quad (18)$$

$$\begin{aligned} \frac{u}{c_1 - 1} \cdot \frac{\ln(c_t) + \ln(u) - \ln(P) - \ln(c_1 - 1) + \ln(r)}{\ln(r)} = \\ \frac{u}{c_1 - 1} \cdot \left(1 + \frac{\ln(c_t) + \ln(u) - \ln(P) - \ln(c_1 - 1)}{\ln(r)} \right). \end{aligned}$$

Discussion. Interestingly, the smaller r – i.e., the larger interest rate – the fewer faults should be detected. From the economic viewpoint, this makes perfect sense: when the interest rate is high, we do not worry that much about future losses.

It is known that interest rates grow in the boom periods, and drop down during recession. So, in the boom period, it makes sense to release not-fully-perfect software, while in the recession period, everything should be checked very carefully before the release.

4 Towards More Accurate Models: Idea

Idea. In the above text, we approximated the actual dependence of the time $t(n)$ of detecting the n -th bug on n by functions from a 2-parametric family

$$\{c \cdot t(n) + t_0\}, \quad (19)$$

with two parameters c and t_0 . To get a more accurate approximation, a natural idea is to use families with more parameters, i.e., families of the type

$$\{c_0 + c_1 \cdot t_1(n) + \dots + c_k \cdot t_k(n)\}, \quad (20)$$

where $c_0, c_1, \dots, c_k$ are parameters, and $t_1(n), \dots, t_k(n)$ are given functions.

Which families should we use? Similar to the 2-parametric case, we can conclude that for each i , the shifted function $t_i(n+1)$ should belong to the same family. In other words, for appropriate constant $c_{i,j}$, we should have:

$$\begin{aligned} t_1(n+1) &= c_{1,0} + c_{1,1} \cdot t_1(k) + \dots + c_{1,k} \cdot t_k(n); \\ &\dots \end{aligned} \quad (21)$$

$$t_k(n+1) = c_{k,0} + c_{k,1} \cdot t_1(k) + \dots + c_{k,k} \cdot t_k(n).$$

In other words, the vector

$$v(n+1) \stackrel{\text{def}}{=} (1, t_1(n+1), \dots, t_k(n+1)) \quad (22)$$

is obtained from the vector

$$v(n) \stackrel{\text{def}}{=} (1, t_1(n), \dots, t_k(n)) \quad (23)$$

by the formula

$$v(n+1) = C v(n), \quad (24)$$

where C is the matrix with coefficients $c_{i,j}$. Thus, by induction, we get

$$v(n) = C^n v(0). \quad (25)$$

If we transform the matrix into a diagonal or almost diagonal form – by describing it in terms of its eigenvectors – we conclude that each function $t_i(n)$ has the form $n^p \cdot \exp(\lambda \cdot n)$, where the natural number p is different from 0 if we have degenerate eigenvalues, and $\lambda = a + b \cdot i$ is the corresponding eigenvalue – which is, in general, a complex number. In real-number terms, each function $t_i(n)$ – and thus, each approximating linear combination

$$T(n) = c_0 + c_1 \cdot t_1(n) + \dots + c_k \cdot t_k(n), \quad (26)$$

is a linear combination of the functions $n^p \cdot \exp(a \cdot n)$ (similar to what we had in the 2-parametric case) and oscillating functions $n^p \cdot \exp(a \cdot n) \cdot \cos(b \cdot n + \varphi)$; see, e.g., [2] for details.

How we can use this family. We can use this model to predict the time of each next fault – and thus, to find the optimal number of tests, i.e., the number of tests N_t for which the overall losses are the smallest possible.

Author contributions All the authors contributed equally to this research paper.

Conflict of interest The authors declare that they have no conflict of interest.

Human and animal ethical standards This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. V. Kreinovich, Th. Swenson, and A. Elentukh, “Interval approach to testing software”, *Interval Computations*, 1994, No. 2, pp. 90–109.
2. H. T. Nguyen and V. Kreinovich, *Applications of continuous mathematics to computer science*, Kluwer, Dordrecht, 1997.
3. D. J. Sheskin, *Handbook of Parametric and Non-Parametric Statistical Procedures*, Chapman & Hall/CRC, London, UK, 2011.
4. I. Sommerville, *Software Engineering*, Pearson, Harlow, Essex, UK, 2021.