WHITE PAPER | SEPTEMBER 2015

Cost, Complexity and Coverage

Using Test Case Optimization Technology to Project the Cost Delivering High Quality Applications

September 2015

ca technologies

Table of Contents

Executive Summary

There has been a concerted recent effort within the software development community to develop and embrace new working models that aim to address weaknesses in old methodologies. The most prominent example of this, perhaps, is the transition towards Agile development practices in response to the inherent inflexibility and lack of parallelism of more traditional Waterfall development models.

However, while the merits of newer, more flexible methodologies are theoretically and intuitively evident, there remains a notable lack of metrics by which to prove these assertions. So, how does one arrive at a projected cost for a software project using objective, reliable metrics? Judging by the fact that 61% of software projects are cancelled or challenged (delivered late, over-budget or missing functionality), it can be reasonably suggested that such a metric does not, in reality, currently exist.[1]

This paper discusses a reasonable approach to measuring development success, using requirements and metrics that can be defined in terms of those requirements, to be able to accurately project the cost of delivering quality software in a timely fashion. In addition, this article will make the case for one particular paradigm – the flowchart – illustrating how it can provide attractive solutions to the challenges of requirements design, cost projection and change management in Agile environments.

Requirements: The Genesis of Software

It is generally understood within the software industry that requirements are the backbone of the software development lifecycle (SDLC). They form the framework from which the rest of the process is predicated (one notable exception is testing, however, as explained later, some schools of thought are focused on correcting this). With this in mind, it stands to reason that, before any development starts, one can assume that a list of requirements will exist (in most cases, it will be the only information that exists at this stage). It is therefore reasonable to ask: since the cost of development needs to be projected as early as possible, can the requirements be of any use in estimating them?

Consider a software project – say an account management system for an online retailer – is at the point at which requirements are being drawn up. To begin with, a high-level design is put together, giving an overview of the major areas that need to be considered. For example:

Highest Level Requirements
Account Management
Security
Payments
Integration With Orders System

Once all the highest-level requirements have been identified, each section in turn is then dissected to find the next level of requirements. Continuing our example:

Highest Level Requirements	Next Level
Account Management	Add Account Delete Account ...
Security	Password security Payment details security ...

So far, this is consistent with accepted best practices. However, since we are approaching this from a cost perspective, how does this help us? Well, since the requirements have been subdivided according to granularity, it provides us with a methodology for calculating the total cost of development:

1. For each lowest level requirement, calculate the cost (and/or complexity) of implementing it
2. For each level above, its cost is simply the sum of the costs of the next level of requirements
3. Repeat step 2 until all highest-level requirement have a calculated cost
4. The total cost is simply the sum of the costs of the highest-level requirements

Using the previous example to illustrate, assume we have the following requirements:

Highest Level Requirements	Next Level	Lowest Level
Account Management	Add Account	Customer First Name Customer Last Name Customer Address ...
	Delete Account	Account Number
Security	Password security	AES Encryption SHA-256 password hashing ...
	Payment details security	RSA Encryption ...

Calculate and fill out the lowest level costs first (highlighted in pink):

Highest Level Requirements	Next Level	Lowest Level
Account Management	Add Account	Customer First Name **(2)** Customer Last Name **(2)** Customer Address **(6)** ... **(35)**
	Delete Account	Account Number **(2)**
Security	Password security	AES Encryption **(70)** SHA-256 password hashing **(40)** ... **(125)**
	Payment details security	RSA Encryption **(60)** ... **(250)**

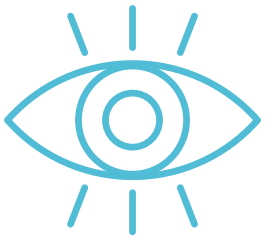
Once these have been identified and accounted for – for the next level, calculate the sum of the ones underneath it:

Highest Level Requirements	Next Level	Lowest Level
Account Management **(47)**	Add Account **(45)**	Customer First Name **(2)** Customer Last Name **(2)** Customer Address **(6)** ... **(35)**
	Delete Account **(2)**	Account Number **(2)**
Security **(545)**	Password security **(235)**	AES Encryption **(70)** SHA-256 password hashing **(40)** ... **(125)**
	Payment details security **(310)**	RSA Encryption **(60)** ... **(250)**

So, for this example, the total cost is 592 (i.e. 47 + 545). In practice, costs would usually be denoted in units with each unit assigned a fixed cost (e.g. 1 unit = $1,000). In this example, security = $545,000.

Bear in mind that, for the purposes of this example, the numbers are arbitrary and used for explanatory purposes only. In real application, it is necessary to calculate the cost of individual requirements (see the discussion on inner complexity which is set out further on in this article).

Above, we have provided a macro view on cost. The remainder of this article will deal with the frequently posed question: how does one calculate the cost of the lowest level requirements? This constitutes the micro view and a different approach.



The (Perceived) Problem with Requirements

Firstly, it is necessary to dispel the notion that requirements are subjective. Of all the engineering fields, this one appears to be totally isolated to software. For example, one would not expect engineers to proceed with the building of a bridge if there was any room for error in the underlying specifications. However, this is the expected norm of the software industry – the author's experience certainly attests to that fact – and how this came to be the case is an interesting (but separate) issue.

However, the previous comparison with civil engineering does raise the prospect of 'borrowing' methodologies from the more mature engineering disciplines.

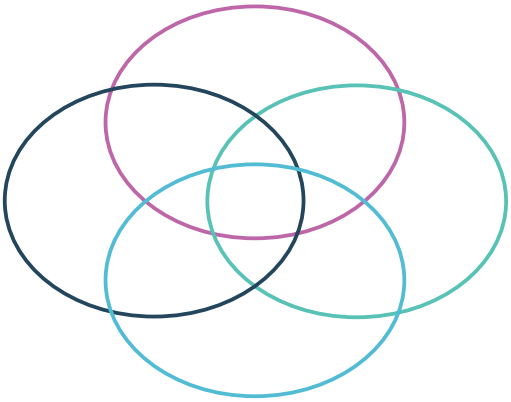
Requirements-Based Testing (RBT), which emerged in the early 1970s, is one such example. Based on the mathematics at the heart of hardware testing, it is a methodology that, because of the underlying logic, makes it possible to construct a (relatively small) suite of test cases to achieve 100% functional coverage. Those of you who are interested in its applicability to improving testing quality are invited to this reference.[2] However, for this discussion, we are more interested in what RBT has to say on ambiguity in requirements as it forms the foundation of the notion that we are dispelling.

To properly build logic models for use in RBT, it is necessary to purge any and all ambiguities from the requirement before it can be tested. The same is true here: in order to be able to accurately calculate the cost of each requirement, it is necessary to ensure that there is one and only one interpretation of the requirement. This particular topic is beyond the scope of this paper, but for an in-depth discussion of the ambiguity review process, the reader is invited to visit the following link.[3]

Bender, R., http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf

The reason for removing ambiguities is that minimizing and quantifying cost and complexity depend entirely on the interpretation of the requirement being consistent across every person who implements it. For instance, it is possible for a requirement to be interpreted in multiple ways (by the business analyst, by one or more developers, by testers etc.). Each misinterpretation involves an amount of rework to rectify the mistake, which reduces the accuracy of the cost projections. As interpretation of an ambiguous requirement is entirely subjective, it is not possible to predict the number of potential misinterpretations, only to give upper and lower bounds on this number.

If we consider 'interpretations' as a Venn diagram, it looks something like this:



The User knows what they want

The Analyst specifies what it is

The Programmer writes the code

The Tester tests the solution

Overlaps between the circles are the requirements that have been interpreted in a common way, while non-overlapping regions are those requirements, which have not. The key here is to maximize the overlaps so that each group has interpreted requirements the same way. The only way to achieve this, however, is to ensure that the requirements are unambiguous to begin with.

By using a mathematical construct known as a finite Hausdorff metric, it is possible to measure the 'distance' between each set of interpretations. Since measuring ambiguity in a consistent and objective manner is not yet achievable (apart from the processes referenced earlier in this section, which leave short the existence of a single metric to quantify the results of ambiguity reviews), this is beyond the scope of this article. However, the notion of a 'distance' between interpretations is sufficient to outline a mathematical argument as to how it affects the margins of error for cost projections (see appendix A). For this section, it is sufficient to consider the rework required bringing the interpretations closer together, and how this must be factored into cost projections. In appendix A, we relate how requirement ambiguities can be factored into the mathematical model as the margin of error on cost projections.

Inner Complexity

Other approaches[4] have proposed that the sizing of software is directly measurable from the number of 'testable requirements'; this is a laudable approach and a marked improvement on the archaic lines of code (LOC) and functional point (FP) measures. However, without wishing to single this particular instance out, it is necessary to point out a few shortcomings to date.

For the most part, previous approaches have ignored the inner (or inherent) complexity of each requirement. In other words, they have treated every testable requirement as having equivalent weighting. This is a rather naïve assumption and can lead to massive errors when calculating the overall cost. In the initial example, each requirement was given an individual weighting to reflect its complexity, and therefore, cost – if each requirement had been given equal weighting this would have not reflected the differences between them. Intuitively, this can be thought of as requiring more coding to implement the ones with higher costs.

Therefore, it is imperative that each requirement be assessed on its own merit. This includes avoiding assumptions such as 'if requirement X cost $1,000, then requirement Y should cost $2,000'. This is because, along with the cost, it is also necessary to factor in a 'margin of error' for each individual requirement. As you might expect, these 'margins for error' are very rarely consistent.

Prediction Power as a Measure of Quality

Bearing in mind the above, a question naturally arises: what can we say about the margins of error in the predicted cost and can this be used as an interpretation of quality?

Of course, a prediction is merely that, and it will always be hindsight that determines whether or not it has yielded fruit. Predictions also come with their own margin for error. However, a predicted figure will usually come within a high and low range, with the true value expected to lie somewhere between the two. Climate models are a good example of this, with an 'expected value' (i.e. the predicted true value) as well as a range of values that the model also factors in as its margin of error.

This concept can also be applied to our running example. Let's suppose that we factor in a 10% error margin for each requirement. How does this affect the predicted cost as a whole? The table below shows complexity metrics (red) as well as a 10% margin of error (upper bound is blue, lower is green):

Highest Level Requirements	Next Level	Lowest Level
Account Management **(47) (42.3) (51.7)**	Add Account **(45) (40.5) (49.5)**	Customer First Name **(2) (1.8) (2.2)** Customer Last Name **(2) (1.8) (2.2)** Customer Address **(6) (5.4) (6.6)** ... **(35) (31.5) (38.5)**
	Delete Account **(2) (1.8) (2.2)**	Account Number **(2) (1.8) (2.2)**
Security **(545) (490.5) (589.5)**	Password security **(235) (211.5) (258.5)**	AES Encryption **(70) (63) (77)** SHA-256 password hashing **(40) (36) (44)** ... **(125) (117.5) (187.5)**
	Payment details security **(310) (279) (341)**	RSA Encryption **(60) (54) (66)** ... **(250) (225) (275)**

This gives us the overall cost for the project of 592 ± 59.2 (i.e. a 10% overall margin of error).

From a statistical point of view, a margin of error is synonymous with a random variable with a given variance. So, for each cost, we associate an element of error, so that the cost is expressed as

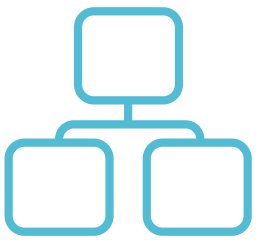
$$C = c + \varepsilon$$

(Where c is fixed and ε is an error element). Statistically speaking, the expected value of the error element is 0 – intuitively, you may consider this as the error having equal probability of being positive and negative. The magnitude of the error is denoted by its variance. To use a common modeling technique, we assume all the error elements are independent and identically distributed and all follow a normal (or Gaussian) distribution with expected values 0 and ε^2 (ε being the standard deviation). As outlined above, this margin of error will be proportional to the 'size' of the ambiguity (Appendix A).

It follows from an elementary result in statistics (the central limit theorem[5]) that:

$$\textit{Standard Deviation of Total Cost} = \sqrt{n} * \textit{Standard Deviation of Single Requirement}$$

Hence, the more requirements considered, the total margin of error will grow sub-linearly.



Application to Flowcharting Techniques—The Cost of Testing

One method of requirement specification, namely the flowcharting method, yields itself perfectly to leverage complexity metrics as outlined here. Firstly, there are three observations:

1. In mathematical terms, a flowchart is merely a directed graph – flowchart blocks become the vertices, and the arrows between them form the directed edges
2. Complexity metrics can be attached to each block – in mathematical terms, this translates to applying a 'weight' to each vertex in the graph
3. Test cases translate to paths through the graph

These three observations mean that we may leverage many tried-and-tested graph-theoretical concepts from mathematics. Graph theory is a branch of mathematics that exclusively deals with graph-like structures (i.e. any structure that can be described as a series of nodes connected with edges), and has many applications across a whole range of mathematical disciplines. An in-depth explanation of how graph theoretical concepts can be used as a basis for test case generation, as well as test case optimization can be found here.[6] For now, it is sufficient to use the above characterization of vertices, edges and paths for what follows.

One such discipline is the branch of mathematics known as operational research – this is the branch of mathematics that concerns itself with the application of algorithms to solve problems in scheduling and path optimization, among other things. For instance, the 'Travelling Salesman' problem is a very famous problem in operational research. Apart from the obvious applications of path-based algorithms to solve problems in test case design[7], existing algorithms for scheduling can be leveraged to make the project manager's life easier. Since we are focusing on quantifying testing costs, we shall defer this topic to a future article.

So far, we have only considered the case of implementation costs. Naturally, one needs also to consider the testing cost to gain a full picture of the total cost projections. Due to the third observation above, this is where graph theory really comes into its own. To this end, we will sub-divide the costs as detailed above into two separate costs: the cost of implementing and the costs of testing.

In our analysis, we shall consider the implementation cost i(Rj) of requirement j, and the testing cost t(Rj) of requirement j, so that the total cost C(Rj) will be equal to the sum of both, i.e.:

$$\boldsymbol{C(Rj) = i(Rj) + t(Rj)}$$

Since test cases are merely paths, we can express a path as an ordered set of edges:

$$\boldsymbol{p = (E1, E2, \ldots, Enp)}$$

Note that, since we are talking about complete paths, each path will start off on a start node and will finish on an end node. The above can be re-defined in terms of vertices (nodes) as:

$$\boldsymbol{p = (\,(\pi, v1), (v1, v2), \ldots, (vnp{-}1, vnp)\,, (vnp, \mu))}$$

(where π denotes a start node and μ denotes an end node).

The total testing cost for a path can be simply stated as the sum of the testing cost for each node, namely:

$$t(p) = t(\pi) + t(\mu) + \sum_{i=1}^{n_p} t(v_i)$$

The standard deviation of the cost, again assuming a normally distributed error factor, can be summarized as:

$$stddev(t(p)) = \left(\sqrt{n_p + 2}\right)\sigma$$

Now we have calculated the testing cost for each path, we turn our attention to the number of paths (i.e. the number of test cases). In the literature, a widely accepted notion for the number of test cases is the notion of cyclomatic complexity[8], or the number of linearly independent paths through a graph. Two paths in a graph are considered linearly independent if, and only if; they are comprised of unique sets of edges, disregarding loop back edges. There will be a further article reconciling this metric with a macro view (as it doesn't actually quite scale the same way, and McCabe originally designed this metric for code complexity, which is a micro view), however, for now it is sufficient as a lower bound on cost projections, due to its relationship with graph theory. For those who are mathematically inclined, this relationship is due to the formal definition of the cyclomatic complexity: it is defined as the size of the first homology (i.e. the first relative Betti number) of the graph with respect to its end nodes – which means that it is the number of linearly independent paths through the graph. This corresponds with the smallest number of test cases required to achieve 100% branch coverage.

For our analysis, let P(G) denote the total number of paths possible through the graph G, and let L(G) denote the smallest set of linearly independent paths through G. Clearly, the following hold:

$$L(G) \subseteq P(G)$$

$$M = |L(G)|$$

Where M is the cyclomatic complexity of G. Let us consider an arbitrary testing strategy that satisfies 100% branch coverage – denote this by S(G). Then, since it satisfies 100% branch coverage, the following holds:

$$L(G) \subseteq S(G) \subseteq P(G)$$

We define the cost over a set of paths to be the sum of the cost of each individual path, namely:

$$t(S(G)) = \sum_{i=1}^{|S(G)|} t(p_i)$$

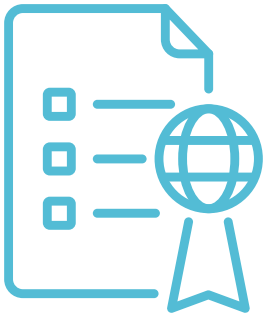
Since cost is a linear operator (it is merely the sum over a set of paths), then the following also holds:

$$t(L(G)) \leq t(S(G)) \leq t(P(G))$$

Also, tying back the cyclomatic complexity, the following upper and lower bounds can be derived straightforwardly:

$$M * \min_{p \in L(G)} t(p) \leq t(S(G)) \leq |P(G)| * \max_{p \in P(G)} t(p)$$

Simply speaking, the lower bound is the cyclomatic complexity M multiplied by the testing cost of the least expensive path, and the upper bound is the total number of paths multiplied by the testing cost of the most expensive path.



Applications in the Real World

So far, the discussion has been rather theoretical and has largely focused on building an argument as to how good requirements can translate to accurate cost projections. As any project manager knows, translating this into practice requires far more than a cogent argument – good process is absolutely key here. Since the applicability of this theory mostly surrounds flowcharting techniques, it makes sense to make the use of a flowcharting tool a key part of this process.

The key to a good process, where requirements are concerned, is having a medium where the business analyst can formulate thoughts and logic clearly. Note that the following is equally applicable regardless of the form of requirement (user story, old-style requirement, etc.):

1. If the requirements come in lengthy chunks of prose, it is imperative to first break them down into individual bullet points.
2. The process of breaking down should be iterative and the breaking down should continue until each bullet point has one (and only one) statement.
3. Since user stories and use cases map to paths in the flowchart paradigm, it is necessary to also consider how they interact – they will have some degree of commonality (e.g. common process blocks) and this will need to be factored into the overall flowchart design. This is imperative to the post-process of identifying missing test cases, as user stories rarely encompass all the possibilities.

Once the requirements have been formulated in succinct, clear bullet points, it is then rather trivial to map these bullet points as a flowchart – one bullet point should map very cleanly to one block. As an exercise, it is very informative to then re-build the text requirements back from the flowchart (whether this is done as a whole or on a path-by-path basis doesn't matter – the key is that flowcharts should be readable, and this is one of the ways to ensure that). Any good design tool should allow you to automatically rebuild the text requirement straight from the flowchart.

The next phase, once the flowcharts have been designed, is to run through all the paths generated by the tool to make sure they 'make sense'. This is known as the 'Test Case Review Process' (per (1)) and, in the author's experience, is an invaluable exercise in validating the requirements – in multiple cases, the author can attest that this exercise unearthed many critical omissions from the original requirements (i.e. use cases that the business analyst hadn't thought of and were 'nonsensical', but nevertheless possible per the requirements).

Once the flowchart is built, it is then necessary to evaluate the cost (or complexity) of each individual requirement (process or decision block). As outlined above, this 'cost' can be split into the cost of implementation and the cost of testing. This is the most critical, yet most difficult, part for the project manager to perform – if these costs are not evaluated with a degree of precision, then the overall cost estimates will also be inaccurate (see above). This will necessitate input from subject matter experts who are familiar with both the implementation and testing of these individual requirements. It should be pointed out here that, the more detailed and less ambiguous these requirements are, the easier it will be for the SME to put a cost (and a margin of error, if necessary) on them.

Once the individual costs are accounted for, the project manager can utilize the strategy highlighted in the first two sections to evaluate the overall cost, with margins of error to estimate lower and upper bounds. If there is a suitable requirements tool, then the chances are that it can perform this analysis quickly with no input from the user.

On Changing Requirements

Requirements are rarely static. In the process of clarifying and disambiguating requirements, it will inevitably be passed between business analysts and users, and changes will be made in each iteration. A question then arises: how can differences be accounted for in the cost metric?

A decent flowcharting tool should also give the facility to store and track changes to a set of requirements – you may think of this as a 'revision history' or 'versioning' capability. What this allows the business analyst to do is track changes in the requirements through 'versions', and also is able to adjust costs according to new information. As each version of the requirement progresses, the cost metric can be calculated for each version of the requirement and thus differences can be calculated between each. This is invaluable for the refinement process between business analyst and user, since the user can ask, "How much would it cost if I wanted to tweak a certain feature?", and the business analyst can factor this into the flowchart and come back with an amended cost.

A spin off benefit is also in an overall reduction in testing effort when requirements change. There is often a hidden cost in that tests teams find it easier to re-run entire packs of test cases rather than go to the trouble of working out which test cases need to be run, by applying this approach you can:

- Automatically identify which of your many tests need to be run rather than running them all
- Automatically repair tests, this currently a highly manual process and does not lend itself to
- Continuous development strategies
- Easily identify any new test cases that need to be created
- Finally, remove any redundant or duplicate tests. Testing teams never delete a test case as they worry about the overall effect on coverage

Summary

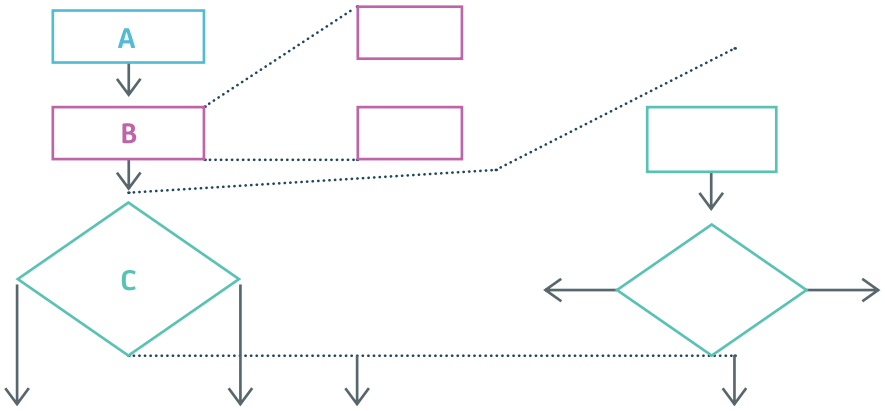
In this article, we have outlined how requirements affect cost projection for a project, and how having solid, unambiguous requirements narrow the margins of error since there is less information missing.

We have also outlined a strategy whereby testing costs are separated from implementation costs and how the cost of testing can be computed separately, and how both can be computed very easily given a flowchart. In addition, the killer question of "if the requirements change, how does the cost change?" also falls very elegantly from the overall strategy, making a solid case for the flowchart paradigm in requirements design.

Appendix A

Let a flowchart (with μ end nodes) consist of n nodes, $n_S = n_P + n_D + 1$ sub-flowcharts, where n_P is the number of process sub-flowcharts and n_D is the number of decision sub-flowcharts (so obviously, n > n_S). The plus 1 is there since the main flowchart is a sub-flowchart in and of itself.

As an illustration, consider the following simple example:



In this example, the main (unexpanded) flowchart is on the left, the sub-flowchart corresponding to B is in red in the center, and the sub-flowchart corresponding to C is in yellow on the right. In this example:

$$n_p = 1$$

$$n_d = 1$$

$$n_s = 3$$

Let each sub-flowchart F_i have N_i nodes and E_i edges, and number of connected components P_i. The number of connected components P_i is equal to the number of exit nodes, so for a decision with m outputs, P_i equals m.

Let M_i denote the cyclomatic complexity of each sub-flowchart – so that:

$$M_i = E_i - N_i + 2P_i$$

The total cyclomatic complexity of all the flowcharts is equal to:

$$M_T = \sum_{i=1}^{n} E_i - (\sum_{i=1}^{n} N_i - n_p - n_d) + 2$$

(since, for each decision and process sub-flowchart, one block is lost when composing it back to one big flowchart).

Re-arranging and introducing the P_i term:

$$M_T = \sum_{i=1}^{n} (E_i - N_i + 2P_i) - 2(\sum_{i=1}^{n} P_i) + n_p + n_d + 2$$

Since each decision has di exit points, and each process has just one, and the main flowchart has μ:

$$M_T = \sum_{i=1}^{n}(E_i - N_i + 2P_i) - 2\left(\sum_{i=1}^{n_d} d_i\right) - 2\left(\sum_{i=1}^{n_p} 1\right) - 2\mu + n_p + n_d + 2$$

Hence:

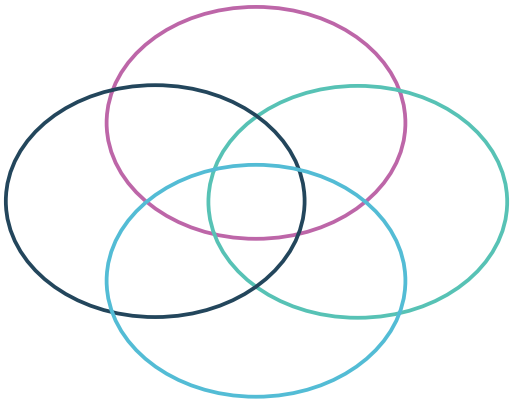
$$M_T = \sum_{i=1}^{n}(E_i - N_i + 2P_i) - 2\left(\sum_{i=1}^{n_d} d_i\right) - n_p + n_d + 2(1 - \mu)$$

$$M_T = \sum_{i=1}^{n} M_i - 2\left(\sum_{i=1}^{n_d} d_i\right) - n_p + n_d + 2(1 - \mu)$$

Appendix B: Applying the Hausdorff Metric to Cost Projection

In this article, we have outlined how requirements affect cost projection for a project, and how having solid, unambiguous requirements narrow the margins of error since there is less information missing.

We have also outlined a strategy whereby testing costs are separated from implementation costs and how the cost of testing can be computed separately, and how both can be computed very easily given a flowchart. In addition, the killer question of "if the requirements change, how does the cost change?" also falls very elegantly from the overall strategy, making a solid case for the flowchart paradigm in requirements design.



The User knows what they want

The Analyst knows what it is

The Programmer writes the code

The Tester tests the solution

For this analysis, we model 'requirement sets' simply as sets of requirements as understood by the particular group (e.g. the user has his set of interpretations, marked in red, while the analyst has his set of interpretations, marked in violet, and the intersection is what is understood commonly by the two). We will be introducing the Hausdorff metric to measure the differences between the interpretations of each group (basically, treating the sets as geometric spaces and then using the Hausdorff metric to measure distance between the spaces, as we would do a geometric space).

Let I_U, I_B, I_P and I_T denote the 'requirement set' or 'interpretation' of the user, BA, programmer and tester, respectively.

Let

$$\boldsymbol{\Theta} = \{\boldsymbol{IU}, \boldsymbol{IB}, \boldsymbol{IP}, \boldsymbol{IT}\}$$

(i.e. the set of all interpretations)

Then, for X, Y ∈ Θ, the Hausdorff metric is defined by:

$$\boldsymbol{d_H(X,Y) = \max\left\{\sup_{x \in X, y \in Y} inf\, d_H(x,y), \sup_{y \in Y, x \in X} inf\, d_H(x,y)\right\}}$$

Intuitively, this can be thought of as the maximum distance required to travel from subspace X to subspace Y given any arbitrary point.

Since X, Y are finite:

$$\boldsymbol{d_H(X,Y) = \max\left\{\max_{x \in X, y \in Y} d_H(x,y), \max_{y \in Y, x \in X} d_H(x,y)\right\}}$$

i.e. a restriction of d_H to finite subspaces, since in finite subspaces:

$$\boldsymbol{sup\ inf = sup\ sup = max}$$

For the entire space Θ, rework can be expressed as the effort required to minimize d_H (ideally, to zero) for each pair of subspaces.

Define the work $\omega_{X,Y}$ to be the amount of work it will take to minimize the Hausdorff metric between X and Y. It can be expressed as:

$$\boldsymbol{\omega_{X,Y} = c_{X,Y} d_H(X,Y)}$$

For some arbitrary work constant cX,Y Let P_θ be the set of all pairs in Θ under equivalence (i.e. the direct square of Θ minus duplicates, so (X,Y) ∈ Θ and (Y,X) ∈ Θ are equivalent). Basically P_θ is equivalent to the set of equivalence classes of the direct square of Θ under the relation (X,Y) = (Y,X). For this example:

$$\boldsymbol{P_\theta = \{(I_U, I_B), (I_U, I_P), (I_U, I_T), (I_B, I_P), (I_B, I_T), (I_P, I_T)\}}$$

So the total work ΩΘ can be expressed as:

$$\boldsymbol{\Omega_\theta = \sum_{(X,Y) \in P_\theta} \omega_{X,Y} = \sum_{(X,Y) \in P_\theta} c_{X,Y} d_H(X,Y)}$$

This forms the basis of the assertion that the re-work factor is proportional to the distances between each interpretation. This re-work factor will need to be factored into the variability of the overall cost – in our discussions, the standard deviations σ which we have been talking about are proportional to the re-work costs. In particular, given a single requirement R, the variability is proportional to:

$$\boldsymbol{\sigma_R \propto \Omega_{\theta_R}}$$

References

[1] 1 Standish Group, Chaos Manifesto, 2013

[3] AmbiguityProcess. benderrbt.com. [Online] **http://benderrbt.com/Ambiguityprocess.pdf**

[4] Wilson, Peter B. "Sizing Software With Testable Requirements

[5] Kallenberg, O. Foundations of Modern Probability. New York : Springer-Verlag, 1997.

[6] Jorgensen. Software Testing: A Craftman's Approach, 2002

[7] Leyzorek, M., et al. A Study of Model Techniques for Communication Systems. Cleveland, Ohio: Case Institute of Technology: Investigation of Model Techniques, 1957

[8] McCabe, Thomas J. (Snr.), A Complexity Measure, s.l.: IEEE Transactions on Software Engineering, 1976.

The CA Technologies Advantage

CA Technologies (NASDAQ: CA) provides IT management solutions that help customers manage and secure complex IT environments to support agile business services. Organizations leverage CA Technologies software and SaaS solutions to accelerate innovation, transform infrastructure and secure data and identities, from the data center to the cloud. CA Technologies is committed to ensuring our customers achieve their desired outcomes and expected business value through the use of our technology. To learn more about our customer success programs, visit **ca.com/customer-success**. For more information about CA Technologies go to **ca.com.**