11

September 2010

testing experience

The Magazine for Professional Testers

printed in Germany

print version 8,00 €

free digital version

www.testingexperience.com

ISSN 1866-5705

Metrics

ONLINE TRAINING

English & German (Foundation)

ISTQB® Certified Tester Foundation Level

ISTQB® Certified Tester
Advanced Level Test Manager

ACCREDITED ISTQB TRAINING PROVIDER

ISTQB® International Software Testing Qualifications Board

Díaz Hilterscheid

te testing experience

Editorial

Dear readers,

Already when I did my thesis about metrics on software development there were different opinions about it due to the fact that having "numbers" you can compare and see what is good and what is not so good.

In this issue we face a lot of sights, visions and opinions about metrics. Once again, we received a lot of articles and like the last time it pains us not to be able to publish all of them, maybe in a later issue we will do.

This time we also included a very long article from Tom Gilb and Lindsey Brodie about requirements specifications and its value delivery. I hope you like it.

We did have a wonderful Testing & Finance with very good keynotes specially from Nitin Bhargava and my friend Bj Rollinson. If you weren't there you missed something good. In March 2011 we are planning the Testing & Finance in New York, I hope to see you there.

This year we still have two conferences Agile Testing Days in Berlin (www.agiletestingdays.com) and the Oz Agile Days in Sydney, Australia (www.ozagile.com). I hope you are able to attend one of them.

Last but not least, this year I went on holiday twice. Once with my kids in Gran Canaria and Lanzarote and the second time alone in Northern Spain. Now in Ribadeo writing the editorial I'm facing the last days of my Camino de Santiago which started in Bilbao. I did the northern path at the coast, with some excursions to Burgos and León. An amazing trip. I think that you can make this trip under different sights: religious, self-discovery, culture, meditation, life retrospective and of course the food and beverages culture. My main focus was culture, meditation and food!!! I met Amalia – a pastoress taking care of chicken, sheep and cows – who lives in a small room for 5 month alone in the mountains. I had a very nice talk with her. I bought some "cabrales" cheese and enjoyed the walk through the Picos de Europa by Covadonga. This can also be a very exciting life. You don't need to be a tester to be happy.

It doesn't matter what your motive is, the trip will help to switch off from your day to day routine and look at life differently. I will do this trip again. I will do it in stages, and for each stage a different friend. I'm looking forward to it.

Enjoy reading!

José Díaz

Contents

Counting defects

by Bert Wijgers

46

Ranorex®

Automate Your UI Testing with Ranorex

Object-based Capture & Replay Editor

- Maintainable recordings via the actions table editor
- Integration of Ranorex repositories

Automated Testing of Web & Windows Applications

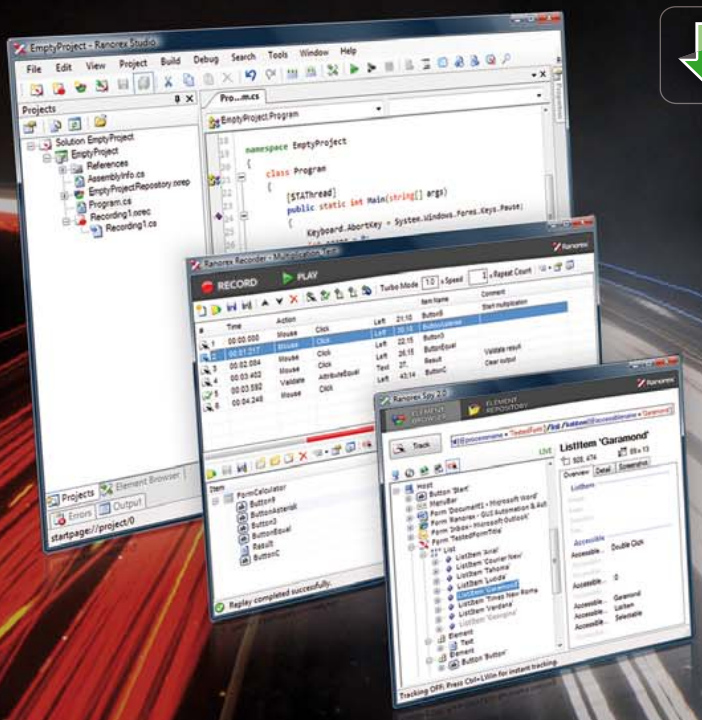
- Winforms / C# / VB.NET
- WPF / Silverlight / Win32 / MFC
- Flash / Flex / Web 2.0 / AJAX / IE / Firefox

Maintainable UI Object Repositories

- Easy to maintain all types of UI objects
- Separate test automation from UI identification

Write tests in C#, VB.NET and IronPython

- Automatic and flexible code generation in C#, VB.NET and IronPython
- All-on-one test environment with code editor, code completion and debugging



Get your 30-day Trial
www.ranorex.com

VISIT US!
29 Nov - 02 Dec
Copenhagen/Denmark
EuroSTAR 2010

The Value of Testing in 5 Dimensions

by Peter Zimmerer

Today most people in the software business agree that testing is important, but there is still a very diverse understanding of what testing is all about and what its value is. Unfortunately, there are still outdated definitions used for testing (see [IEEE 610.12-1990] or [IEEE 829-2008]) that limit testing just to "dynamically execute a piece of software to detect bugs". However, there is hope if we look at the definitions for testing as given in the ISTQB glossary (see [1])

The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

or as given by Cem Kaner (see [2]):

Testing is an empirical technical investigation of the product / system / artifact / service under test conducted to provide stakeholders with information about the quality.

The question is: How can we put these definitions into practice in real project life? And how can we find a way to improve the communication of the value of testing to the different stakeholders like managers and developers? One answer could be to break testing down into 5 dimensions to better visualize its mission, motivations, activities, and corresponding values (see figure 1).

Dimensions of testing

Demonstrate, check, confirm, verify, validate

First of all testing is about demonstrating in a constructive way that something works, at least to some extent. Further it is about checking and confirming artifacts like requirements, features or use cases. And it is also about verifying ("Are you building the thing *right*?") and validating ("Are you building the *right* thing?").

Detect, search

In testing we try to detect bugs as early as possible in a destructive way, and we search for unknown and therefore unspecified behavior in the system under test.

Mitigate, reduce risks, investigate, explore

Any testing should be based (at least to some extent) on risks in the system under test, so this is also a very important dimension in testing. Thereby we also investigate and explore in order to look deep inside the system under test.

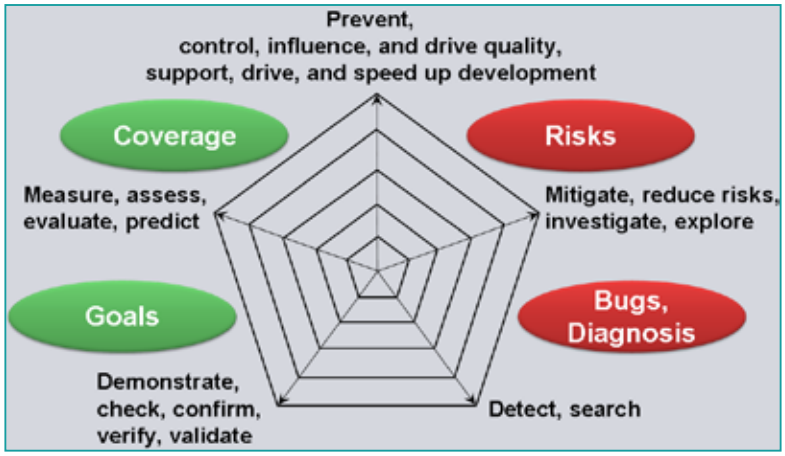


Figure 1: Dimensions of testing

Measure, assess, evaluate, predict

During testing we collect data and measure, for example to address quality attributes like performance, reliability or availability. Based on these data we know something about the current status, and we can hopefully make a good forecast for the future.

Prevent, control, influence, and drive quality, support, drive, and speed up development

Prevention of bugs should always be one direction in our testing activities. These could be approaches like systematic root cause analysis as well as early test case design by test-driven development – the idea is to use the information we get from testing directly to change or rather drive developments in the future. Furthermore, this testing dimension enables higher productivity and accelerated development by facilitating change, identifying any regressions as early as possible and decreasing maintenance effort.

All 5 dimensions are part of the ultimate mission of testing, which is to provide information and evidence about the quality of the product / system / artifact / service under test for the different stakeholders. If this information is effectively used, then we create real value for the business, i.e. the business value of testing lies in the savings that the organization can achieve from improvements based on the information that is provided by testing. These can be improvements

- in the product / system / artifact / service under test, for example by fixing a detected defect or removing an unspecified, unwanted behavior in the system
- in the decisions we make, for example by considering mitigated and residual risks in a release decision
- in the development approach (process) itself, for example by fostering prevention tasks.

A fundamental understanding of these dimensions is very useful in practice to effectively address typical questions in testing, as the following examples illustrate.

Developer: *My unit testing does not detect any important new bugs. Why should I do unit testing in the future?*

-> Do not miss the other 4 dimensions in understanding the real value of unit testing besides detecting bugs. Unit testing provides value within all 5 dimensions of testing.

Manager: *You have this big regression test suite. Why do we need to spend so much effort on just repeating these tests again and again?*

-> Map the value of the regression test suite to the different dimensions to emphasize its need and impact on the business to management. Often different parts of the regression test suite address the 5 dimensions to a different extent – that can be a starting point to prioritize the regression test cases based on the overall testing mission.

Project Lead: *You have all this great test automation in place, but the system still crashes quite often. What shall we do?*

-> This can be an indicator that by doing a lot of test automation the focus is currently too much on the "check" dimension only. Strengthen the testing on the "detect" and on the "mitigate risks" dimensions (including investigation and exploration) to expand and vary the scope of testing.

Frequently asked questions

Why are there just 5 dimensions and not more or less?

First of all, it should be easy to remember and easy to read. Based on psychological studies we know that a human being can remember 5±2 things at a point of time. And the number of different dimensions is always a trade-off between abstraction and detailing.

Don't some dimensions contain too many different things?

Again the intention is to avoid too many dimensions, i.e. to avoid fragmentation. Each dimension includes several activities with corresponding values that are related and are driven by the same motivation, which is why they have been clustered together into one dimension.

Are the different dimensions really disjoint?

No, not completely – but are totally disjoint dimensions possible at all? This means that dependent on the interpretation there is some overlap. For example "check" can also include "checking for some risk" and "investigate" can also mean to "measure some resource consumption". The important thing is to understand that every dimension visualizes a different, complementary view and perspective of the testing approach. Good testing always requires identifying and selecting an adequate mixture of these different dimensions to adequately cover the testing space.

The figure looks like a kiviat diagram. Is that intended?

Yes, definitely. You can think of an increasing intensity for every dimension from the center point to the edge. However, for perfect testing that does not mean that you should always be on the edges only, in fact that depends on the context. For example, if you develop a prototype for a tradeshow, the focus could be only on "demonstrate and check". If you develop a safety-critical system, you should also take "mitigate risks" and "measure" into account. So, for different projects the covered area in the diagram will be different. It is precisely the area covered which describes and visualizes the testing mission for a specific project.

Is there any semantics in the ordering of the dimensions in the figure?

Yes, there are some semantics concerning "bottom – top" and "left – right".

At the bottom are the more classical activities like "check" and "detect". As you go to the top of the diagram, maturity increases. That means more advanced testing approaches and processes as well as more experienced and professional people (!) are needed here, especially for the "prevent".

On the left side of the diagram are the more positive, constructive activities, which provide information on the achieved goals and coverage; on the right side are the more negative, destructive activities, which address things like bugs and risks.

The next step

Having this clear picture of the testing mission is very helpful to guide the overall testing approach and to define the testing strategy. That also includes the determination of the test design methods to be applied to create the right test cases for effective and efficient testing. So, an interesting question which comes up now is: "How do test design techniques support and foster these 5 dimensions of testing?"

To answer this question, we now cluster test design techniques into 5 related categories as well (see figure 2). Although there is no strict biunique mapping between all the corresponding edges of the diagrams in figures 1 and 2, there is a sound correlation between them. That means the 5 categories of test design techniques may contribute to several of the 5 dimensions of testing, but will do so with a focus on the corresponding edge as explained next.

Black-box -> Demonstrate, check, confirm, verify, validate

Demonstrating and checking is initially based on a black-box view of the system under test. So, the correlation between these edges is strong. Besides this, black-box test design techniques also contribute to other dimensions, e.g. to the "detect" and to the "measure" dimensions.

White-box -> Detect, search

Typically, a white-box view is needed (and beneficial!) to detect bugs and to identify strange, unknown behavior in the system under test. So, the correlation between these edges is also strong.

Grey-box -> Measure, assess, evaluate, predict

Perhaps here the direct relation between the two diagrams is not so strong and seems a little artificial, but a grey-box view is very helpful (if not needed!) for example in adequately measuring performance data or so.

Fault-based -> Mitigate, reduce risks, investigate, explore

Fault-based test design techniques directly address different types of risk, and they are a rich source to guide investigation and exploration, so again here is a strong correlation.

Regression -> Prevent, control, influence, and drive quality, support, drive, and speed up development

First of all the direct relation perhaps sounds a bit weak. However, if you have a regression test suite in place (preferable already from the beginning by doing test-driven development!), then you have a safety net that helps you to effectively prevent many defects in future and that enables you to control (and drive!) refactoring activities within development. Having these practices in place is also the starting point and a precondition for really improving productivity and accelerating development and maintenance.

For every category of the test design techniques given in figure 2, there is a number of specific test design techniques. A more detailed overview on these test design techniques can be found in [3].

Conclusion

Good testing requires many skills, capabilities, knowledge, experience, and creativeness. Therefore the diagrams as given in figures 1 and 2 should never be used too restrictively, but rather as a map and a general direction for the testing mission. Identifying the required intensity for each dimension and selecting an adequate mixture of them to adequately cover the testing space are the key for effective and efficient testing.

A good understanding of what testing is all about is only the first step. More difficult but even more important is to communicate this testing mission and its value in an appropriate manner to the different stakeholders like managers, project leads, developers, and even to other testers. For this purpose, characterizing testing in 5 dimensions to visualize its mission, motivations, activities, and corresponding values can be useful. The better we get at doing this, the better we will collaborate with the other stakeholders, and it will help us also to strengthen and sustain our own testing territory.

[1] ISTQB Standard Glossary of Terms used in Software Testing V.2.0, December 2007, http://www.istqb.org/downloads/glossary-current.pdf

[2] http://www.kaner.com/pdfs/ETatQAI.pdf

[3] P. Zimmerer, Systematic Test Design...All on One Page, STAREAST Conference 2008, http://www.stickyminds.com/s.asp?F=S14140_CP_2

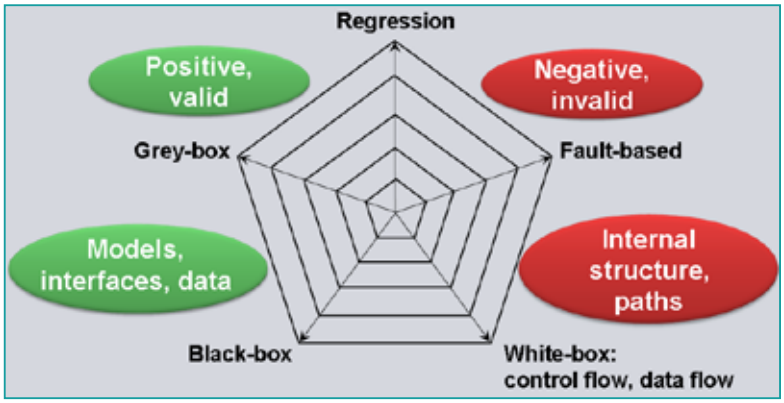


Figure 2: Categories of test design techniques

Biography

Peter Zimmerer is a Principal Engineer at Siemens AG, Corporate Technology, in Munich, Germany. He studied Computer Science at the University of Stuttgart, Germany and is an ISTQB® Certified Tester Full Advanced Level. For more than 19 years he has been working in the field of software testing and quality engineering for object-oriented, distributed, component-based, service-oriented, and embedded software. He was also involved in the design and development of several Siemens in-house testing tools for component and integration testing. At Siemens he performs consulting and training on test management and test engineering practices, including testing strategies, testing techniques, testing processes, test automation, and testing tools in real-world projects and drives research and innovation in this area. He is author of several journal and conference contributions and regular speaker at international conferences in Europe, Canada, and USA.

Scrat

Quality Center → TFS 2010

Migrate in hours instead of months

Convert existing Quality Center™ (QC) items to Microsoft Visual Studio 2010 (TFS)

- Migration of all QC elements to TS and their interrelationships links.
- The most time efficient and cost efficient migration process on the market today. Saves more than 90% of the migration time compared to other methods.
- Migrate projects to Team System 2010 in days instead of months.
- Based on Sela's migration expertise.
- Fully customizable (you decide what and how to migrate each item) or fully mirrored migration (migrate everything 'as-is').
- Reporting system with export to Ms-Excel.
- Support for multilingual data.

Europe and international
scrat@sela.co.il
+972-3-6176644

USA
scrat@sela.co.il
1-888-774-9471

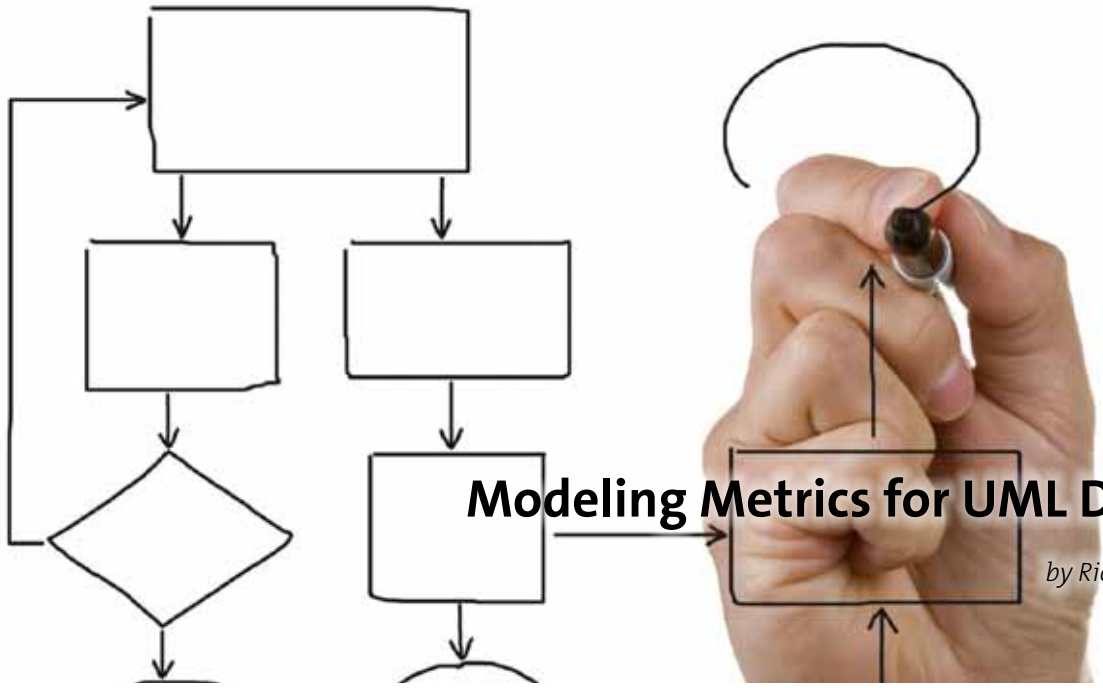
Canada
info@selacanada.ca
1-866-640-4754

India
info@sela.co.in
+91-20-6521-6466

SELA TECHNOLOGY CENTER

www.sela.co.il/scrat/main2.html

ISTQB Certified Tester



Modeling Metrics for UML Diagrams

by Richard Seidl & Harry Sneed

This article describes metrics are used to measure and evaluate UML models. The metrics are divided into four classes – quantity metrics, complexity metrics, quality metrics and size metrics. The quantity metrics are simple counts of the design entities and relationships. The complexity metrics measure the relations between design entities and the structure of the proposed system architecture. The quality metrics measure the relationship between the desired and the actual characteristics of the architecture. The size metrics transform the basic quantities into overall system size measures which can be used to estimate the costs of development and maintenance.

1. UML Design Quantity Metrics

Design quantity metrics are counts of the diagram and model types contained within the UML model. The model types are further subdivided into design entity counts and design relationship counts [BrMV96].

The 9 design diagram type counts are:

Number of...
- Use Case Diagrams
- Activity Diagrams
- Class Diagrams
- Sequence Diagrams
- Interaction Diagrams
- State Diagrams
- Component Diagrams
- Distribution Diagrams
- Design Diagrams in all

The 17 design entity type counts are:

Number of...
- Sub-Systems
- Use Cases
- Actors
- Components
- Interfaces
- Classes
- base/super classes
- Methods
- Parameters
- Attributes
- Activities
- Objects
- States
- Rules
- Stereotypes
- Design Entities in all
- Design Entities referenced

The 10 design relationship type counts are:

Number of...
- Usages
- Associations
- Generalizations
- Interactions
- Class Hierarchy Levels
- Method Invocations
- Activity Flows
- State Transitions
- Test Cases
- Design Relations in all

These quantities or element counts have been selected on the basis of their relation to the goals of object-oriented system design in accordance with the goal-question-metric method of Basili and Rombach [BrMV02].

2. UML Design Complexity Metrics

Design complexity metrics are computations for calculating selected complexities [HaMü87]. Complexity is defined here as the relation of entities to relationships. The size of a set is determined by the number elements in that set. The complexity of a set is a question of the number of relationships between the elements of that set. The more connections or dependencies there are relative to the number of elements, the greater the complexity [McBu89]. The complexity of a single entity is determined by the number of sub entities relative to the number of relationships between those sub entities. Overall design complexity can be simply stated as:

$$\frac{Number\ of\ design\ entities}{Number\ of\ design\ relationships}$$

Bearing this in mind, the following complexity types have been defined for UML models.

2.1 Object Interaction Complexity

$$1 - \left(\frac{\frac{Nr\_Objects}{Nr\_Object\_Interactions} + \frac{Nr\_Classes}{Nr\_Class\_Associations}}{2} \right)$$

The more interactions between objects and the more associations between classes there are, the higher will be the complexity. In this way both the abstract level of the class as well as the physical level of the objects is taken into consideration. This measure is an inverse coupling metric. It is based on empirical evidence that systems with many dependencies among their parts are difficult to maintain [Boocho8].

2.2 Class Hierarchical Complexity

$$\frac{Nr\_Class\_Levels * 2}{Nr\_Classes}$$

The more hierarchical levels there are in the class hierarchies, the more dependent the lower level classes are on the higher level ones. Deep inheritance has often been criticized for leading to increased complexity. This metric corresponds to the depth of tree metric from Chidamer and Kemerer [ChKe94]. It is based on empirical evidence that object-oriented systems with deep inheritance trees (e.g.> 3) are more error prone than others.

2.3 Class Data Complexity

$$1 - \left(\frac{Nr\_Classes * Minimum\_Nr\_Attributes}{Nr\_Class\_Attributes} \right)$$

The more data attributes a class has the higher its complexity. This corresponds to the class attribute metric in the Mood metrics [Hari98]. The design goal is to have many classes, each with a few data attributes, as opposed to having a few classes, each with many attributes. This goal is based on the assumption that it is easier to test and maintain smaller sets of data.

2.4 Class Functional Complexity

$$1 - \left(\frac{Nr\_Classes * Minimum\_Nr\_Methods}{Nr\_Class\_Methods} \right)$$

The more methods, i.e. functions, a class has, the higher its complexity, whereby it is assumed that each class has at least two implicit functions – a constructor and a destructor. This corresponds to the Number of Methods metric of Chidamer and Kemerer [ChKe94]. The design goal is to have many classes, each with a minimum number of functions, as opposed to having a few classes, each with many methods. This goal is based on the assumption that it is easier to maintain and test a system which is broken down into many small chunks of functionality.

2.5 Object State Complexity

$$1 - \frac{Nr\_Objects}{Nr\_Object\_States}$$

Objects are instances of a class. Objects have states. The more they have, the more complex they are. A simple class is a singleton with one object that has a static state. A complex class is one with multiple objects, each with several potential states. Neither the CK nor the MOOD metrics consider state complexity, even though it is a principle driver of test effort together with the cyclomatic complexity of the methods. The design goal is to have many classes, each with a minimum number of functions, as opposed to have as few object states as possible, but this is determined by the application. If an object such as an account has many states, e.g. opened, balanced, overdrawn, suspended, closed, etc., they all have to be created and tested.

2.6 State Transition Complexity

$$1 - \frac{Nr\_States}{Nr\_State\_Transitions + Nr\_Transition\_Conditions}$$

The connection lines of a state diagram represent the transitions from one state to another. A given state can have any number of successor states. The more there are, the higher the complexity of the state transition graph. As with the McCabe cyclomatic complexity measure, we are actually measuring here the relation of edges to nodes in a graph [McCa76]. Only here the nodes are not statements but states and the edges are not branches but transitions. The design goal is to have as few transitions as possible, since every state transition has to be tested at least once and that drives the test costs up.

2.7 Activity Control Flow Complexity

$$1 - \frac{Nr\_Activities}{Nr\_Activity\_Flows + Nr\_Activity\_Conditions}$$

The connection lines of an activity diagram represent the flow of control from one activity to another. They can be conditional or non-conditional. Conditional flows add to the complexity of the process being modeled. An activity can have any number of successor activities. The more there are and the more conditional ones there are, the higher the complexity of the process.

2.8 Use Case Complexity

$$1 - \frac{Nr\_UseCases}{Nr\_Usage\_Relations + Nr\_Actors}$$

Use cases as coined by Ivar Jacobson are instances of system usage [Jaco93]. A user or system actor invokes a use case. This is a case of usage. The relationships between use cases may have different meanings. They can mean usage or extension or include or inherits. The more relations there are, the higher the usage complexity. The design goal is to reduce complexity by restricting the number of dependencies between use cases. On the other hand, if the application requires it then they have to be included. Otherwise the complexity is only pushed off to another layer.

2.9 Actor Interaction Complexity

$$1 - \frac{Nr\_Actors}{Nr\_UseCases}$$

System actors trigger the use cases. Any one actor can start one or more use cases. The more use cases there are per actor, the more complex is the relation between actors and the system. From the viewpoint of an actor, a system is complex if he has to deal with many use cases. The more use cases there are per actor, the higher the complexity. A system which has only one use case per actor is simple because it is partitioned in accordance with the actors. The design goal is to restrict the number of use cases per actor. Of course by having more actors, the size of the system in use case points increases.

2.10 Overall Design Complexity

The overall design complexity is computed as the relation between the sum of all design entities and the sum of all design relationships.

$$1 - \frac{Nr\_Design\_Entities}{Nr\_Design\_Relationships}$$

A design in which each entity has only a few relationships can be considered less complex than a system design in which the number of relationships per entity is high. This reflects complexity as the relation of the number of relationships between elements of a set and the number of elements in a set. The more elements there are the larger the size of the set. The more relationships there are, the higher the complexity of the set. The design goal is to minimize the number of relationships between design entities.

3. UML Design Quality Metrics

The design quality metrics are computations for calculating selected qualities. Quality is defined here as the relation of that state the model is in, relative to the state it should be in [Card90]. Quality measurement presupposes a standard for the UML model. The actual state of the model is then compared with that standard. The closer the model is to fulfilling that standard, the higher is its quality. In German the overall design quality can be simply expressed by the ratio:

$$\frac{Ist\_Design}{Soll\_Design}$$

The upper bound of the metric is 1. If the IST exceeds the SOLL then the quality goal has been surpassed. A quotient coefficient of 0.5 indicates median quality. It should be remembered that quality is relative. By itself the ratio may not mean so much [Erdo08]. However in comparison with the ratio derived from another designs in exactly the same way, it indicates that the one design has a better or lower quality than the other, at least in respect to the quality characteristic measured. Since there is no absolute quality scale, the quality of a system design can only be assessed in relation to the quality of another [Romb90]. For assessing the quality of a UML model the following quality characteristics were selected.

3.1 Degree of Class Coupling

Class Coupling is the inverse of Interaction Complexity. It is computed by the equation:

$$\frac{\frac{Nr\_Objects}{Nr\_Object\_Interactions} + \frac{Nr\_Classes}{Nr\_Class\_Associoations}}{2}$$

The more interactions and associations there are between objects and classes the greater the dependency of those objects and classes upon one another. This mutual dependency is referred to as coupling. Classes with a high coupling have a greater impact domain. If they are changed there is a greater chance that the other classes will also be affected. The design goal is to have as few dependencies as possible, i.e. the coupling should be low. This quality characteristic is founded on empirical evidence that high coupling is associated with a greater impact domain, with a higher error rate and with more maintenance effort [Gymo09].

3.2 Degree of Class Cohesion

Class Cohesion is measured in terms of the number of data attributes in a class relative to the number of class methods. It is computed by the equation:

$$1 - \frac{Nr\_Attributes}{Nr\_Methods + Nr\_Attributes}$$

The notion of cohesion denotes the degree to which the functions of a module belong together [BiOt94]. Functions belong together when they process the same data. This can be referred to as data coupling. Thus, the fewer the data used by the same functions the better. Classes with a high cohesion have many methods and few attributes. Classes with many attributes and few methods have a lower cohesion. The design goal is to have as few common attributes for the same methods as possible. This quality characteristic is founded on the hypothesis that high cohesion is associated with high maintainability. This hypothesis has never really been proven.

3.3 Degree of Modularity

Modularity is a measure of decomposition. It expresses the degree to which a large system has been decomposed into many small pieces. The theory is that it is easier to deal with smaller units of code [SRK07]. The modularity of classes is determined by the number of attributes and methods a class has. It is expressed by the equation:

$$\frac{Nr\_Classes * Min\_Nr\_Methods\_Per\_Class}{Nr\_Methods}$$

There is a prevailing belief undermined by numerous field experiments that many smaller units of code are easier to change than fewer larger ones. The old roman principle of "divide et imperum" applies to software as well. It has not been proven that smaller modules will necessarily be more error free. Therefore, the justification for modularity is based on the ease of change. In measuring code, modularity can be determined by comparing the actual size of the code units in statements to some predefined maximum size. In an object-oriented design, the elementary units are the methods. The number of methods per class should not exceed a defined limit. In measuring the modularity of UML it is recommended here to compare the total number of methods with the minimum number of methods per class multiplied by the total number of classes. The design goal here is to have as few methods as possible per class so as to encourage the designer to create more and smaller classes.

3.4 Degree of Portability

Portability at the design level is a measure of the ease with which the architecture can be ported to another environment. It is influenced by the way the design is packaged. Many small packages can be more easily ported then a few large one. Therefore it is important to keep the size of the packages as small as possible. The package size is a question of the number of classes per package. At the same time packages should have only few dependencies on their environment. The fewer interfaces each package has, the better. The portability of a system is expressed in the equation:

Learn with the experts and... enjoy Madrid!

expoQA '10

- 40 lectures presented by international speakers from 10 countries.
- Tutorials, hands-on workshops, keynotes.
- Exhibition hall with leading companies from the QA & testing industry.
- Affordable: top quality, half the price of similar conferences!
- Social activities and networking dinner: meet the speakers in a friendly atmosphere and enjoy Spanish food and dance!

Madrid, North Convention Centre
15th to 18th November 2010

International Conference on
Software Quality and Testing

www.expoqa.com/en

organized by nexo qa

Promotional code:
TE01

10% discount for readers of Testing Experience

Promotional code:
TE02

€50 Amazon gift card for readers of Testing Experience

Promotional code:
TE03

40% discount on purchase of a second ticket

Endorsed by SOGETI

Platinum sponsor IBM

Gold sponsors
BULL Architect of an Open World
hp
MICRO FOCUS
steria
MÉTODOS Y TECNOLOGÍA
ORACLE
Microsoft

$$\frac{Nr\_Packages * Min\_Nr\_Classes}{Nr\_Classes + Interfaces}$$

The justification of this quality attribute goes along the same line as that of modularity. The number of classes per package should not exceed a given limit, nor should a package have more than a given number of interfaces with its environment, since interfaces bind a package with its environment. The design goal is to create packages with a minimum number of classes and interfaces.

3.5 Degree of Reusability

Reusability is a measure of the ease with which code units or design units can be taken out of their original environment and transplanted to another environment. That means there should be a minimum of dependency between design units [Sned97]. Dependencies are expressed in UML as generalizations, associations and interactions. Therefore, the equation for measuring the degree of dependency is:

$$\frac{Nr\_Classes + Nr\_Methods}{Nr\_Generalizations + Nr\_Associations + Nr\_Interactions}$$

The more generalizations, associations and interactions there are, the more difficult it is to take out individual classes and methods from the current architecture and to reuse them in another. As with plants, if their roots are entangled with the roots of neighboring plants, it is difficult to transplant them. The entangled roots have to be severed. This applies to software as well. The degree of dependency should be as low as possible. Inheritance and interaction with other classes raises the level of dependency and lowers the degree of reusability. The design goal here is to have as few dependencies as possible.

3.6 Degree of Testability

Testability is a measure of the effort required to test a system relative to the size of that system [SnJu06]. The less effort is required, the higher the degree of testability. Test effort is driven by the number of test cases required to test as well as by the width of the interfaces, whereby that width is expressed as the number of parameters per interface. The equation for computing testability is:

$$1 - \frac{Nr\_Methods + Nr\_Interfaces}{Nr\_Paths + Nr\_Parameters}$$

The number of test cases required is computed based on the number of possible paths thru the system architecture. To test an interface the parameters of that interface have to be set to different combinations. The more parameters it contains, the more combinations have to be tested. Field experience has proven that it is easier to test several narrow interfaces, i.e. interfaces with few parameters, than to test a few wide interfaces, i.e. interfaces with many parameters. Thus, not only the number of test cases but also the width of the interfaces affects the test effort. The design goal here is to design an architecture which can be tested with the least possible effort. This can be achieved by minimizing the possible paths through the system and by modularizing the interfaces.

3.7 Degree of Conformity

Conformity is a measure of the extent to which design rules are adhered to. Every software project should have a convention for naming entities. There should be prescribed names for data attributes and interfaces as well as for classes and methods. It is the responsibility of the project management to see that these naming conventions are made available. It is the responsibility of the quality assurance to ensure that they are adhered to. The equation for conformity is very simple:

$$1 - \frac{Nr\_NonConventional\_Names}{Total\_Names\_Used}$$

Incomprehensible names are the greatest barrier to code comprehension. No matter how well the code is structured it will remain incomprehensible as long as the code content is blurred by inadequate data and procedure names. The names assigned in the UML diagrams will be carried over into the code. Therefore, they should be selected with great care and conform to a rigid naming convention. The design goal here is to get the designers to use meaningful, standardized names in their design documentation.

3.8 Degree of Consistency

Consistency in design implies that the design documents agree with one another. One should not refer to a class or method in a sequence diagram, which is not also contained within a class diagram. To do so is to be inconsistent. The same applies to the methods in the activity diagrams. They should correspond to the methods in the sequence and class diagrams. The parameters passed in the sequence diagrams should also be the parameters assigned to the methods in the class diagrams. Thus, the class diagrams are the base diagrams. All of the other diagrams should agree with them. If not, there is a consistency problem. The equation for computing consistency is:

$$1 - \frac{Undefined\_Methods\_References + Undefined\_Parameter\_References}{Nr\_Defined\_Methods + Nr\_Defined\_Parameters}$$

When measuring the degree of consistency, we encounter one of the greatest weaknesses of the UML design language. It is in itself inconsistent. That is because it was pasted together out of many different design diagram types, each with its own origin. State diagrams, activity diagrams and collaboration diagrams existed long before UML was born. They were taken over from structured design. The basis of the object-oriented design is the class diagram from Grady Booch [Booch86]. Use case and sequence diagrams were added later by Ivar Jacobson. So there was never a consistent design of the UML language. The designer has the possibility of creating the diverse diagram types totally independent of one another. If the UML design tool does not check this, it leads to inconsistent naming. The design goal here is to force the designers to use a common name space for all diagrams and to ensure that the methods, parameters and attributes referenced are defined in the class diagrams.

3.9 Degree of Completeness

Completeness of a design could mean that all of the requirements and use cases specified in the requirement document are covered by the design documentation. To check that would require a link with the requirement repository and to require that the same names are used for the same entities in the design as are used in the requirement text. Unfortunately the state of information technology is far removed from this ideal. Hardly any IT projects have a common name space for all of their documents let alone a common repository. Therefore, what is measured here is only formal completeness, i.e. that all of the diagrams required are also present. Degree of completeness is a simple relation of finished documents to required documents.

$$\frac{Finished\_Diagram\_Types}{Required\_Diagram\_Types}$$

The design goal here is to ensure that all UML diagram types required for the project are actually available. As witnessed all of the UML projects ever tested by this author, the design is never completed. The pressure to start coding is too great and once the coding is started the design becomes obsolete.

3.10 Degree of Compliance

The ultimate quality of a system design is that it fulfills the requirements. Not everything which is measured is important and much of what is important is not measurable [EbDu07]. That certainly applies here. Whether or not the user requirements are really fulfilled, can only be determined by testing the final product against the requirements. The most that can be done here is to compare the actors and use cases in the design with those specified in the requirements. Every functional requirement should be assigned to a use case in the requirement document. Assuming this to be the case the use cases in the requirement document should cover all functional requirements. If the number of use cases in the design matches the number of use cases in the requirements, we can consider the design to be compliant with the requirements, at least formally. This can be expressed in the coefficient:

$$\frac{Nr\_Designed\_UseCases}{Nr\_Required\_UseCases}$$

If there are more use cases designed than were required, this only shows that the solution is greater than the problem. If there are less use cases in the design, then the design is obviously not compliant. The design goal here is to design a system which covers all requirements, at least at the use case level.

4. UML Design Size Metrics

The design size metrics are computed values for representing the size of a system. Of course what is being measured here is not the system itself, but a model of the system. The system itself will only be measurable when it is finished. One needs size measures at an early stage in order to predict the effort that will be required to produce and test a system. Those size measures can be derived from the requirements by analyzing the requirement texts or at design time by analyzing the design diagrams. Both measurements can, of course, be only as good as the requirements and/or the design being measured. Since the design is more detailed and more likely to be complete, the design size metrics will lead to a more reliable estimate. However, the design is complete much later than the requirements. That means the original cost estimation has to be based on the requirements. If the design based estimation surpasses the original one, it will be necessary to delete functionality, i.e. to leave out less important use cases and objects. If the design based estimation varies significantly from the original one, it will be necessary to stop the project and to renegotiate the proposed time and costs. In any case the project should be recalculated when the design is finished.

There are several methods for estimating software project costs [Sned96]. Each is based on a different size metric. When estimating a project one should always estimate with at least three different methods. For that reason five measures are taken to give the estimator a choice. The five size measurements taken are:

- Data-Points
- Function-Points
- Object-Points
- Use Case-Points
- Test-Cases

4.1 Data-Points

Data-Points is a size measure originally published by Sneed in 1990 [Sned90]. It is intended to measure the size of a system based solely on its data model but including the user interfaces. It is a product of the 4th Generation software development where the applications are built around the existing data model. The data model in UML is expressed in the class diagrams. The user interfaces may be identified in the use case diagrams. This leads to the following computation of data-points:

$$Data\text{-}Points = Nr\_System\_Interactions*8+Nr\_Classes*4+Nr\_Parameters*2+Nr\_Attributes$$

4.2 Function-Points

Function-Points is a size measure originally introduced by Albrecht at IBM in 1979 [Albr79]. It is intended to measure the size of a system based on its inputs and outputs together with its data files and interfaces. Inputs are weighted from 3 to 6, outputs from 4 to 7, files from 7 to 15 and system interfaces from 5 to 10. This method of system sizing was based on the structured systems analysis and design technique. It has evolved over the years but the basic counting scheme has remained unchanged [GaHe00]. It was never intended for object-oriented systems but it can be adapted. In a UML design, the closest to the logical files are the classes. The closest to the user inputs and outputs are the actor/usecase interactions. The interfaces between classes can be interpreted as system interfaces. With that rough approximation we come to the following computation of function-points:

$$Function\text{-}Points = Nr\_Actor\_System\_Interactions*5+Nr\_Interfaces*7+Nr\_Classes*10$$

4.3 Object-Points

Object-Points were designed specifically for measuring the size of object-oriented systems by Sneed in 1996 [Sned96]. The idea was to find a size measure which could readily be taken from an object-design. As such it fits perfectly to the UML design. Object-Points are obviously the best size measure of an object model. Classes weigh 4 points, methods weigh 3 points, interfaces weigh 2 points and attributes/parameters weigh one point. That way object-points are computed as:

$$Object\text{-}Points = Nr\_Classes*4+Nr\_Methods*3+Nr\_Interfaces*2+Nr\_Attributes+Nr\_Parameters$$

4.4 Usecase-Points

UC-Points were introduced by a Swedish student working at Ericsson by the name of G. Karner in 1993 [Karn93]. The idea here was to estimate the size of a software system based on the number of actors and the number and complexity of the use cases. Both actors and use cases were classified in three levels – simple, medium and difficult. Actors are rated on a scale of 1 to 3. Use Cases are rated now on a scale of 5 to 15 [Ribu01]. The two are multiplied together to give the unadjusted use-case points. This method is also appropriate for measuring the size of a UML design provided the use cases and actors are all specified. Here the median levels are used to classify all actors and use cases, but are enhanced by the number of actor to use case interactions.

$$UseCase\text{-}Points = Nr\_Actors*2+Nr\_UseCases*10+Nr\_Actor\_UseCase\_Interactions$$

4.5 Test-Cases

Test-Cases were first used as a size measure by Sneed in 1978 for estimating the effort to test the Siemens Integrated Transport System – ITS. The motivation behind it was to charge the module test on the basis of tested cases. A test case was defined to be equivalent to a path through the test object. Much later the method was revitalized to estimate the costs of testing systems [SBS08]. In testing systems a test cases is equivalent to a path through the system. It begins at the interaction between an actor and the system and either follows a path through the activity

Testmetriken im Testmanagement

am **22. & 23.11.2010** in **Berlin**

Intensivtraining Testmetriken im Testmanagement

- Wie lange müssen wir noch testen?
- Wieviele Probleme werden wir in Produktion bekommen?
- Wo stehen wir mit der Qualität unserer Applikation?
- Wie effizient ist unser Testprozess?

Sind Sie mit der einen oder anderen Frage dieser Art bereits konfrontiert worden und konnten keine fundierte Antwort geben? Mit unserem Intensivtraining Testmetriken im Testmanagement bieten wir Ihnen eine Übersicht, welche Testmetriken sich in der Praxis bewähren und wie Sie diese Metriken sinnvoll einsetzen und interpretieren können.

Die Trainingsinhalte sind:

- Identifikation von Metrikquellen, Schaffung von Metrikquellen
- Metriken zur Größen/Komplexitätsbestimmung mit/ohne existierendem Sourcecode/Referenzprojekt
- Metriken zur Feststellung des Qualitätszustandes
- Metriken für den Problemmengenforecast, zur Fundierung der Teststrategie, zur Kalkulation des Testaufwandes, zur Messung der Testeffizienz und „Gefährliche" Metriken
- Wann welche Metrik einsetzen
- Iterative Metrikoptimierung

Die Metriken werden sowohl theoretisch vermittelt, als auch praktisch anhand eines Beispielprojektes demonstriert.

http://training.diazhilterscheid.com

998,00 € zzgl. MwSt.

Anforderungsmanagement

am **25.10.2010** in **Berlin**

Qualität beginnt bereits bei der Anforderung. Je früher die Anforderungsqualität sichergestellt wird, desto günstiger gestaltet sich der Verlauf des gesamten Entwicklungsprojekts.

Anforderungsmanagement hilft relevante Informationen zu sammeln und dabei eine stabile und breite Basis als Grundlage für Soll – Ist Vergleiche zu schaffen. Sie erwerben bei diesem Intensivtraining umfassende Kenntnisse zu folgenden Bereichen:

- Systematisierung von Anforderungen (Funktionale und nicht funktionale Anforderungen)
- Charakterisierung von Analyseergebnissen
- Geschäftsattribute, Geschäftsregeln, Geschäftsprozesse und Aktivitäten
- Designergebnisse (Anwendungsfälle (use cases), Oberflächen (screens)) sowie deren effektive Beschreibung
- Dokumentation von Anforderungen
- Anforderungsinhalte und deren Prüfung auf Vollständigkeit, Korrektheit
- Review von Anforderungen
- Workflow, Kommunikation, z.B. Interview Technik, ...

Danach sind Sie in der Lage kompetent folgende Fragestellungen für Ihr Unternehmen und Ihre Prozesse zu beantworten:

- Wann sollten Anforderungen erfasst werden?
- Wer definiert die Anforderungen und welche Qualitätskriterien sind verbindlich?
- Welchen Nutzen haben zentral erfasste und systematisierte Anforderungen?
- Change Prozess: Anforderungen können sich ändern, wie reagiert man darauf?
- Wer ist für die Inhalte der Anforderungen zuständig?
- u.v.m.

http://training.diazhilterscheid.com

499,00 € zzgl. MwSt.

SEQIS
The Austrian Software Test Experts!

Díaz Hilterscheid

diagrams or transcends the sequence diagrams via interactions between the classes. There should be one test case for every path through the interaction diagrams as well as for every object state specified in the state diagrams. Thus, the number of test cases can be derived from the use case interactions times the number of class interactions times the number of object states. It is computed as follows:

*Test - Cases = Nr_Actor_UseCase_Interactions*Nr_Class_Interactions*Nr_Activity_Flows*Nr_States*

5. Automated Analysis of UML Designs with UMLAudit

To measure UML-designs the tool UMLAudit was developed. UMLAudit is a member of the SoftAudit automated quality assurance tool set. This tool set also includes analysis tools for English and German requirements texts as well as for all of the leading programming languages, the most popular database schema languages and several user and system interface definition languages. UMLAudit contains an XML Parser which parses the XML files produced by the UML modeling tool to represent the diagrams [Bocko3]. There the diagram and model entity types, names and relationships are included as attributes, which are readily recognized by their model types and names. The object of measurement is the XML schema of the UML-2 model with its model types as specified by the OMG [OMG03]

The first step of UMLAudit is to collect the design types and names from the XML files and store them in tables. The second step is to go through the tables and count them. The third step is to check the names against the name convention templates. The final step is to check the referential consistency by comparing the entities referenced with the entities defined. As a result two outputs are produced:

- a UML deficiency report and
- a UML metric report.

The UML Deficiency Report is a log of rule violations and discrepancies listed by diagram. At present there are only two types of deficiencies:

- Inconsistent references and
- Name rule violations.

If a diagram such as a state, activity or sequence diagram references a class, method or parameter not defined in the class diagram, an inconsistent reference is reported. If the name of an entity deviates from the naming rules for that entity type, a naming violation is reported. These deficiencies are summed up and compared with the number of model types and type names to give the design conformance.

The UML Metric Report lists out the quantity, complexity and quality metrics at the file and system level. The quantity metrics are further subdivided into diagram quantities, structural quantities, relationship quantities and size metrics.

6. Conclusion

Judging a software system by its design is like judging a book by its table of contents. If the table is very fine grained you will be able to assess the structure and composition of the book and to make assumptions about the content. The same is the case for UML. If the UML design is fine grained down to a detailed level, it is possible to make an assessment and to estimate costs based on the design [Mill02]. If it is only coarse grained, the assessment of the system will be superficial and the estimation unreliable. Measuring size, complexity and quality of anything can only be as accurate as the thing one is measuring. UML is only a model and models do not necessarily reflect reality [Seli03]. In fact they seldom do. UML models are in practice often incomplete and inconsistent, making it difficult to base a test upon them. This remains the biggest obstacle to model-based testing.

References:

[Albr79] Albrecht, A.: "Measuring Application Development Productivity", Proc of Joint SHARE, GUIDE and IBM Symposium, Philadephia, Oct. 1979, p. 83

[BiOt94] Bieman, J./Ott, L.: "Measuring Functional Cohesion", IEEE Trans on S.E., Vol. 20, No. 8, August 1994, p. 644

[BrMV02] Briand, L./Morasca,S./Basili,V.: "An Operational Process for Goal-Driven Definition of Measures", IEEE Trans. On S.E. Vol. 28, No. 12, Dec. 2002, p. 1106

[BrMV96] Briand, L./Morasca,S./Basili,V.: "Property-based Software Engineering Measurement", IEE Trans. On S.E. Vol. 22, No. 1, Jan. 1996, p. 68

[Card90] Card, D./Glass,R.: Measuring Software Design Quality, Prentice-Hall, Englewood Cliffs, 1990, p. 42

[Bocko3] Bock, C.: "UML without Pictures", IEEE Software, Sept. 2003, p. 35

[Booch86] Booch, G.: "Object-oriented Development" IEEE Trans. On S.E., Vol. 12, No. 2, March 1986, p. 211

[Booch08] Booch, G.: "Measuring Architectural Complexity", IEEE Software, July 2008, p. 14

[ChKe94] Chidamer, S./Kemerer, C.: „"A Metrics Suite for object-oriented Design", IEEE Trans on S.E., Vol. 20, No. 6. 1994, p. 476

[EbDu07] Ebert, C./Dumke, R.: "Software Measurement, Springer Verlag, Berlin, 2007, p. 1

[Erdo08] Erdogmus, H.: "The infamous Ratio Measure", IEEE Software, May 2008, p. 4 [GaHe00] Garmus, D.; Herron, D.: Function-Point Analysis: Measurement Process for successful Software Projects, Addison-Wesley, Reading MA., December 15, 2000.

[Gymo08] Gyimothy, T: "Metrics to measure Software Design Quality", Proc. of CSMR2008, IEEE Computer Society Press, Kaiserslautern, March 2009, p. 3

[Hari98] Harrison,R./Counsel,S./Reuben, V.: "An Evaluation of the MOOD Set of Object-oriented Software Metrics", IEEE Trans. On S.E., Vol. 24, No. 6, June 1998, p. 491

[HaMü87] Hausen, H.-L./Müllerburg, M.: "Über das Prüfen, Messen und Bewerten von Software", Informatik Spektrum, No. 10, 1987, p. 123

[Jaco93] Jacobson, I., a.o: Object-Oriented Software Engineering – A Use Case driven Approach, Addison-Wesley Pub., Wokingham, G.B., 1993, p. 153

[Karn93] Karner, G.: Metrics for Objectory, Diplomarbeit, University of Linköping, Sweden, Nr. Lith-IDA-Ex-9344:21, Dec. 1993

[McCa76] McCabe, T.: "A Complexity Measure", IEEE Trans S.E., Vol. 2, No. 6, 1976, p.308

[McBu89] McCabe, T./Butler, C.: "Design Complexity Measurement and Testing", Comm. Of ACM, Vol.32, No. 12, Dec. 1989, p. 1415

[Mill02] Miller, J.: "What UML should be", Comm. Of ACM, Vol. 45, No. 11, Nov. 2002, p. 67

[OMG03] Object Management Group, "UML 2 XML Schema, Version 2.1, April 2003, www.omg.org/cgi-bin/doc?ad/03-04-02

[Ribu01] Ribu, K.: Estimating Object-Oriented Projects with Use Cases, Masters Thesis, University of Oslo, Norway, November 2001

[Romb90] Rombach, H.-D.; "Design Measurement – Lessons learned", IEEE Software, March 1990, p. 17

[SRK07] Sarkar,S./Rama,G./Kak,A.: "Information-theoretic Metrics

for measuring the Quality of Software Modularization", Vol. 33, No. 1, Jan. 2007, p. 14

[Seli03] Selic, B.: "The Pragmatics of Model-Driven Development", IEEE Software, Sept. 2003, p. 19

[SBS08] Sneed, H./Baumgartner,M./Seidl,R.: Der Systemtest, Hanser Verlag, München/Wien, 2008, p. 59

[Sned90] Sneed, H.: "Die Data-Point Methode", Online, Zeitschrift für Datenverarbeitung, No. 5, May 1990, p. 48

[Sned96] Sneed, H.: "Schätzung der Entwicklungskosten von objektorientierter Software", Informatikspektrum, Band 19, No. 3, June 1996, p. 133

[Sned97] Sneed, H.M.: „Metriken für die Wiederverwendbarkeit von Softwaresystemen", in Informatikspektrum, Vol. 6, S. 18-20 (1997)

[SnJu06] Sneed, H./ Jungmayr, S.: „Produkt- und Prozessmetriken für den Softwaretest", Informatikspektrum, Band 29, Nr. 1, p. 23, (2006)

Biography

After graduating in communications engineering, **Richard Seidl** began working as software engineer and tester in the environment of banking. Since early 2005 Richard Seidl is test expert and test manager at ANECON GmbH, Vienna. His work focuses on planning and realization of test projects mainly in the environment of banking and egovernment. In 2006 he completed the Full Advanced Level Certification of ISTQB. In the following year he qualified as IREB Certified Professional for Requirements Engineering and as Quality Assurance Management Professional (QAMP). He published the book „Der Systemtest" together with Harry M. Sneed and Manfred Baumgartner. In 09/2010 his second book with Sneed and Baumgartner - „Software in Zahlen - Vermessen von Applikationen" - will be published.

Harry M. Sneed has a Master's Degree in Information Sciences from the University of Maryland, 1969. He has been working in testing since 1977 when he took over the position of test manager for the Siemens ITS project. At this time he developed the first European module test bed – PrüfStand – and founded together with Dr. Ed Miller the first commercial test laboratory in Budapest. Since then he has developed more than 20 different test tools for various environments from embedded real time systems to integrated information systems on the main frame and internet web applications. At the beginning of his career Sneed worked himself as a test project leader. Now, at the end of his long career, he has returned to the role of a software tester for the ANECON GmbH in Vienna. Parallel to his project work Harry Sneed has written over 200 technical articles and written 18 books including 4 on testing. He also teaches software engineering at the university of Regensburg, software maintenance at the technical high school in Linz, and software measurement, reengineering and test at the universities of Koblenz and Szeged. In 2005 Sneed was appointed by the German Gesellschaft für Informatik as a GI Fellow and served as general chair for the international software maintenance conference in Budapest. In 1996 Sneed was awarded by the IEEE for his achievements in the field of software reengineering and in 2008 he received the Stevens Award for his pioneering work in software maintenance. Sneed is a certified tester and an active member in the Austrian and the Hungarian testing boards. In 09/2010 his new book with Sneed and Baumgartner - „Software in Zahlen - Vermessen von Applikationen" - will be published.

iseb Accredited Training Organisation

1. akkreditiertes Unternehmen im deutschsprachigen Raum

ISEB Intermediate (deutsch)

Der ISEB Intermediate Kurs ist das Bindeglied zwischen dem ISTQB Certified Tester Foundation Level und dem Advanced Level. Er erweitert die Inhalte des Foundation Levels, ohne dass man sich bereits für eine Spezialisierung - Test Management, technisches Testen oder funktionales Testen - entscheiden muss. In drei Tagen werden Reviews, risikobasiertes Testen, Test Management und Testanalyse vertieft; zahlreiche Übungsbeispiele erlauben die direkte Anwendung des Gelernten.

Eine einstündige Prüfung mit ca. 25 szenario-basierten Fragen schließt den Kurs ab. Das „**ISEB Intermediate Certificate in Software Testing**“ erhält man ab 60% korrekter Antworten.

Voraussetzungen

Für die Zulassung zur Prüfung zum “Intermediate Certificate in Software Testing“ muss der Teilnehmer die Prüfung zum Certified Tester Foundation Level (ISEB/ISTQB) bestanden haben UND entweder mindestens 18 Monate Erfahrung im Bereich Software Testing ODER den akkreditierten Trainingskurs “ISEB Intermediate” abgeschlossen haben - vorzugsweise alle drei Anforderungen.

Termine

02.11. – 04.11.2010

06.12. – 08.12.2010

€1600,00

plus Prüfungsgebühr €200 zzgl. MwSt.

http://training.diazhilterscheid.com

Díaz Hilterscheid

Metrics For Test Automation Effort Estimations

by Michael T. Pilawa

The question about whether or not a manual regression test is going to become automated is often not only a question about necessities or time to execute the tests, but also about return on investment and thus about the question:"How much is my test automation system going to cost?". This article will explain the metrics and methods used to make the effort of functional system test automation more transparent for both test engineers and the test managers who offer the test automation system to their clients.

Motivation

Among many other activities in the software lifecycle management the IT project management usually has to know well in advance about how much budget has to be dedicated to the testing effort in order to assure a measurable software and system quality {1}. While testing efforts add up when being repeated manually for each software release cycle, the question soon extends towards test automation efforts that have to pay off within a decent period of time liberating some of the time being used for manual regression testing for other tasks like explorative testing in order to find new areas of functional weakness. The method to predict the effort of building a black box functional test automation system is also known as the function point analysis that has been extended by the test point analysis and is fully described in the literature about the TMAP test management approach {2}. This article describes the usage of Tmap´s FPA/TPA method {3} implemented in a downloadable spreadsheet calculator {4} and the HP Quality Centre / QuickTest Professional based architecture for automated functional regression testing.

Rationale for test automation effort estimation

The rationale for a test automation implementation effort forecast are the mandatory project management tasks to answer questions like „Can we predict the initial effort of building a scaleable test automation system?" or „Can we calculate the return of investment (ROI) considering the initial effort to build the test architecture and the continued costs for the maintenance effort to adapt and extend the test automation system?" The answer is „Yes, but it takes some time of preparation to predict the effort and some practical experience in programming common function library routines for test automation in order to validate and to verify the correctness of the estimations." For small and medium size projects it should be worth the effort. For large scale projects there is often not enough detailed knowledge about the size of the test automation system at early stages of the project in order to make a sound estimate of the total test automation effort for the entire project duration. Therefore the metrics of small sized portions of a large scale project will be of great advantage in order to extrapolate the numbers for later and larger more complex endeavours of functional test automation.

Test automation architecture

It turns out that the choice of the implementation architecture, i.e. programmed versus captured scripts, data and keyword driven versus of mixed data/script/object captured tests, will be determining about the predictability of the implementation and maintenance effort. The right choice of the test architecture for test automation must underlie the principle „Write once, use many times!"

Therefore it is mandatory to know in advance what kind of test automation architecture will be the foundation of the test script implementation. For a high degree of re-usability and scalability of your tests, think about test automation structures that consist of reusable functions with parameterized values for data and keyword driven test case execution. The choice of the functional testing tool does matter very much, because the tool has to be flexible enough to be adapted to the needs of the test engineer to implement the test architecture and to make the interface to the application under test as feasible as needed to be fast enough in implementing and executing the interfacing functions. Choosing a three layer test architecture for most of the GUI and non-GUI test automation projects leads first to a parameterized main test script at the top of the function library pyramid with parameters that determine the functional coverage of the test, secondly to a middle scripted function library layer consisting of the business functions containing the business workflow with use cases according to the test input parameters and finally to a third level function library script layer containing the interface functions to the system under test for execution of tiny input and output and verification commands. The functions cover the initialization of the test environment, the input, action triggering and synchronization functions, the output collecting and verification steps. Since data and functions and object descriptions are stored in

tables, scripts and libraries commonly used by all tests, adding new tests is merely a matter of copy & paste in HPQC's test plan tree view and adapting the main function call parameters either within the test script's expert view or within the test configuration in HPQC and HPQTP. These parameters should lead to an automatic selection of a subset of the global test data table stored as a loadable attachment in the HPQC test plan branch and used by all tests within that branch. This will ease the maintenance of test cases, if also the common VBscript function script library and the HPQTP object library are stored like attachments within the same branch and merely linked to each HPQTP test within that branch. As of Version 10 HPQC contains a versioning system and the historization of all artefacts can be managed within the test resource module most efficiently and effectively. For HP TestDirector and HPQC Version 9.x the versioning has to be done either within the file system, MS source safe Add-In for HPQC or another versioning system like subversion.

Three spreadsheet approach

Handling the metrics for test automation effort estimation need three calculating table sheets, one for calculating the amount of functional steps and their degree of re-usability, one for the function point analysis derived from the amount of test steps to be implemented and taking into account the complexity of the test functions and the test data and finally one sheet for the test point analysis that take into account the amount of function points and the environmental factors for building the test automation system. The first spreadsheet contains columns to reference the test name, the test priority, the number of steps contained in the test, the number of reusable steps, the percentage of reusable test steps, the percentage of the test implementation effort and some columns for each test step that has to be implemented. While the percentages are merely there to optimize and manage the test automation project, the total step count and the reusable step count columns are needed to calculate a total number of test functions for the function point analysis.

Defining the test step size

This first spreadsheet has to be filled with the knowledge about the tests that should be automated. Therefore the test steps of the tests within HPQC's manual test have to be analysed first and must be broken into ideally equal sized test steps with the future implementation of the test step in mind. This analysis of the step size will facilitate the calculations of the function points later because it avoids calculations for many different types of function complexity. For example a defined step may contain either a single data input or a single data output verification or one initiation of a Non-GUI functionality by using a batch command line string or an URL that starts the default browser, or one action on a graphical user interface item like a button or a menu or one kind of a file handling activity. For the sake of calculating the reusability of test steps by many different tests the choice of the step size with the lowest possible complexity of the function to be implemented adds some more work to the estimation process, but gives more precise estimations and pays off during the implementation phase due to a clear understanding of what has to be done and in which amount of time because each of these simple functions will become the foundation of the second layer's high level business logic functions that will be executed by calling the third layer parameterized functions which are reading the test case data from the HPQC attached Excel table.

The first spreadsheet for the test steps

With these architectural details in mind, every test step will receive a short description consisting of at least four or five characters and filling a cell of the spreadsheet in the same row that contains the test name, that step and in the column for the test step. Example: The first row of the test script contains the names of the columns like „test name", „step 1", „step 2", „step 3", etc. The second row contains the data for counting the function points, e.g. „My first Web application test", „Close any open browser except HPQC explorer and start the default browser application with an URL given by the test data case and entered into the window Run dialog", „Enter Login user name", „Enter Login password", „Push the Login button", „Choose the menu abc/defg", „Wait for website label ABC to appear", etc.

The second row does contain another test. Not just another test case of the first test, because that will not increase the effort of the test automation engineer to implement new functions significantly if the test can be driven by test data. But if the second test has the same behaviour to initialize the system under test, we can use the first spreadsheet to calculate the re-usability of the test functions. Hence the third row of the first spreadsheet might look like „My second Web application test", „0101", „0102", „0103", „0104","0105","Wait for the grid list of items to appear with at least one item named XYZ", „Click on the detail button of the first item in the list", „Verify the detail data section appears under the grid", etc.

This second test lacks the details of the first five test steps but references the test steps of test „01" in row 2 (assuming that the first row contains the column names) with step „01", „02", „03", using the names „0101", „0102", etc as an abbreviation for „first test first step", „first test second step" etc instead. Using this kind of short reference can be of advantage to count the number of referenced test step for each test and calculate the percentage of reusable steps. In Excel this can be done using the function COUNTA to calculate the total amount of steps in a test row and FREQUENCY to calculate the amount of reusable steps. The latter made it necessary to have a shadow row further down in the sheet representing the cells with a reference number as 1 and those without 0, but perhaps there is a more elegant way of doing this.

Why is there a need to know the percentage of re-usability at all? Well, this is what test automation is all about! If you cannot reuse test steps by executing parameterized script code that is fed with test case data, why bother about a test automation system at all? A test tool that writes some code by itself is okay for some occasional SOA test or GUI test, but maintaining these scripts can also be very painful if the test automation system has to deal with several hundred tests for many different test environments and business objectives because of the changes that have to be applied sooner or later and that would have to be applied to all the scripts effected by the software change instead of changing just one reusable shared function library used by all tests involved. Or you have to record the scripts once again or hope that an update function can do the job for you. However, if you want predictable efforts for scalable test automation systems, you better stick with a test architecture that you know and that gives you reproducible test results. In the example above the second test will have a reusability of 50% of all test steps if the second test would have a total count of 10 steps of which the first five are in common with the first test. Thus, you calculate five reusable steps in the appropriate column for this test instead of 10 like in the row of the first

test. The example project had a total re-usability of 74% of all test steps, thus being 26% cheaper to produce than a system with no common function library.

By adding the total number of steps of each test row and subtracting the total number of reusable test steps we know the number of test steps to be implemented into the function library. The differences between the referenced test steps and the original test step will be implemented into the test case data of that test. Remember that each test has got a reference parameter given for the main function and referencing a test specific section of test cases, i.e. rows in the global test data sheet. E.g. if the step reference „0101" of the second test has to start the browser with another URL than the first test, the URL can be found in the appropriate column of the test data sheet and has almost no influence on the total amount of test steps to increase the effort estimated within the subsequent FPA/TPA sheets. If the test step size is declared as something of approximately equal complexity, we can now proceed into the second spreadsheet containing the function point analysis.

The second spreadsheet for the function point analysis

The second spreadsheet takes the complexity of the test steps that have to be implemented into account for counting the function points. Since the test steps have to be reusable by as many tests and test cases as necessary in order to cover the regression test requirements the test data used for the test cases can add some complexity to the functions being implemented for each test step. In order to calculate the function points for each test steps the type of step and the complexity of its parameterized functionality has to be classified. The following table shows three types of test steps:

- Data input,
- data output and queries,
- database and interfaces.

For each of these step types the amount of data types is considered as well as the amount of data fields. The lookup tables below will help to classify a test step as "simple","medium" or "complex".

		amount of distinct data fields**		
		... in data input		
data / step*		< 5 data fields	5-15 data fields	> 15 data fields
	< 2 data types	simple	simple	medium
	2 data types	simple	medium	complex
	> 2 data types	medium	complex	complex
		... in data output and queries		
data / step*		< 6 data fields	6-19 data fields	> 19 data fields
	< 2 data types	simple	simple	medium
	2-3 data types	simple	medium	complex
	> 3 data types	medium	complex	complex
		... in database and interfaces		
groups per step***		< 20 data fields	20-50 data fields	> 50 data fields
	< 2 field groups	simple	simple	medium
	2-5 field groups	simple	medium	complex
	> 5 field groups	medium	complex	complex

* data base (file types referenced – FTR)
** data fields (data element types – DET)
*** field groups (record element types – RET)

Once the complexity per step type is known, the amount of steps per step type can be used to calculate the sum of unadjusted function points UFP according to the following table which contains some example counting for a small project.

step type	complexity	X	o		X			X		
	simple			medium			complex			sum
External Input	o	3	0	6	4	24	o	6	0	24
External Output	o	4	0	o	5	0	o	7	0	0
External Inquiry	o	3	0	o	4	0	o	6	0	0
External Interface File	o	5	0	o	7	0	o	10	0	0
Logical Internal File	1	7	7	o	10	0	1	15	15	22
									UFP	46

The UFP will be multiplied by the value adjustment factor VAF to become function points FPs.

```
VAF = 0.65 + 0.01 x TDI
```

The total degree of influence TDI is the sum of all influences from data communication, distributed functions, performance requirements to the test automation system, load on hardware, transaction rate requirement, online data input, online data change, user interface efficiency, complexity of data processing, reuse-ability of test automation system, ease of installation of the test automation system, ease of usability of the test automation system for the tester, portability, ease of change and extension of the test automation system. Each of these influences should be valued by a number ranging from 0 (no influence) to 5 (strong influence) with 3 being average influence.

For the UFP 46 example project the influence of the performance was 3, of the complexity was 3, of the re-usability was 5, the ease of installation was 1, of usability was 2 and of changeability was 5, thus TDI = 3+3+5+1+2+5=19.

```
VAF= 0.65+0.01x19= 0.84 as value adjust-
ment factor
FP = UFP x VAF = 46 x 0.84 = 38.64 as
function points
```

The third spreadsheet for the test point analysis

The influence of the test engineering environment U, the domain knowledge K and the amount of test points TP influence the time TP spent primarily to implement the test automation system for functional regression testing and the total time T can be calculated taking test management TM into account.

T = PT x ((100 + TM)/100) is the total amount of time used to implement the test automation system which has been almost 10 full time equivalent working days in our example, with PT=68hours and TM=11 percent test management effort, including the time used for this FPA/TPA analysis.

The following section will only give a brief description of what has been considered and may be considered in addition to the example project. A full description of the factors involved in test point analysis and estimation can be found in the literature {5}.

PT = TP x K x E is the amount of primary testing hours for the test points TP, knowledge K and environment E.

Set TP to the test points that will be calculated later in this paper with TP=38.16 as in our example project, K=1.5 and E=1.19, PT=68 hours.

Set K to 1.5 if you have a test engineer in your team, if not set it to 3, but in this case you cannot make a test automation system and you merely get the estimation of a testing effort for a single manual test run.

```
E= ((TT + VT + TB + TA + TU + TW) / 21) is the sum
of environment factors
```

Set TT to 1 if both a tool for test design and a tool for capture & replay can be used or set it to 2, if you can use only either of these tools. Set TT to 4 if you have to order these tools first before you can start to work with them, but in that case the estimation cannot take into account the amount of time it takes to learn efficient and effective usage of these tools or to work on a proof of concept for a given test design method or test automation architecture.

Set VT to 2 if the tests that you want to automate have already designed and executed as manual tests, thus allowing you to reuse the test case data and expected values of the test results. Set VT to 4 if you merely have the test plan but no results of the test execution. Set VT to 8 if there is no test plan available. This is the case if you have to start from scratch with the first spreadsheet described in this paper.

Set TB to 3 if your test base provides standards and templates for documentation and reviews have been taking place, which is usually the case in a well organized test factory. Set TB to 6 if no reviews have been done yet, but the standards and templates are already available. Otherwise set TB to 12.

Set TA to 4 if your test automation tool contains a syntax checker and a pre-compiler for semantic error checking on the scripts that you're going to write and if it also checks for errors on the parameters that you use within the function library, actions and tests. Set TA to 2 if the test automation tool inhibits test execution in case of those scripting or parameter errors. Set TA to 8 in any other case but be aware that you hardly can find errors automatically, effectively and efficiently in software under test if your own software for test automation is full of unknown static and dynamic bugs that cannot be found with the help of your test automation tool. In that case you better invest some more time and money into the tool evaluation process because it will pay off in the long run.

Set TU to 1 if the test bed has been used already several times before, set TU to 2 if there is experience with similar test beds but this kind of test bed is being used the first time. Set TU to 4 if the test bed is in an experimental state.

Set TW to 1 if the test-ware is available in a generalized and centralized yet usable form, i.e. the test cases are specified and ready for becoming automated. Set TW to 2 if just a usable and centralized base situation is given, but the test cases have not been specified yet. Set TW to 4 if there is no test-ware or if you have to search for it in the first place and review it to start with spreadsheet 1.

In the example project we had E= (2+8+3+4+4+4)/21) =1.19

TM = 12 + S + MT + TO is the test management effort depending on size S of the team, management tools MT and test organisation TO.

Set S to 1 if only up to 4 persons have to be managed in your team, 2 if the team has five to nine members and 3 if team size is at least ten persons.

Set MT to -1 if you can take advantage of at least 3 automated utilities for test management, 0 for one or two automated tools and 5 if you do not have any tools available.

Set TO to be -2 if you are working in a test factory as a permanent test organisation with established test tools and tool usage standards, test processes and documentation guidelines, test project administration and test tool support and test environment management and test base management.

Set TO to 0 if you are not working in a test factory.

Now that the factors influencing the test environment are estimated, the test points have to be derived from the function points according to dynamically measurable quality characteristics Qd, some function dependent factors Af and test point factors TPf, statically measured quality characteristics Qc.

```
TP=sum(TPf) + (min(FP,500) x Qs) / 500
```

Qs=Fx+Tx+Sx+Cx+Cy is the statically measurable quality characteristic

Set Fx to 16 if the system under test has to be tested for flexibility. Otherwise set Fx to 0.

Set Tx to 16 if the system under test has to be tested for testability. Otherwise set Tx to 0.

Set Sx to 16 if the system under test has to be tested for security. Otherwise set Sx to 0.

Set Cx to 16 if the system under test has to be tested for continuity. Otherwise set Cx to 0.

Set Cy to 16 if the system under test has to be tested for controllability. Otherwise set Cy to 0.

Set Qs to 0 if you are not going to test any of these quality criteria automatically with the test automation system for functional regression testing. In our example project we did not test any of these criteria automatically, thus Qs=0.

Qd=Qde+Qdi is the dynamically tested quality characteristic, estimated for explicit testable criteria and implicit testable criteria.

```
Qde=Fkt*0.75/4+Sec*0.05/4+Eff*0.1/4+Pwr*0.05/4+Po
rt*0.05/4
```

Set Fkt to 4, if a normal functionality is to be tested, 0, if no functional test will be performed (doesn't make sense here), 3, if low priority is given to functionality, 5, if high priority is given, 6, if extremely high priority is given to functional quality of the system under test.

Set Sec to 0, if no security testing is performed by the test automation system. Else set it to 3, 4, 5 or 6 in analogy to the factors described for Fkt.

Set Eff to 0, of no effectiveness is tested. Also set it to 3, 4, 5 or 6 for low, normal, high and extremely importance of effectiveness.

Set Pwr to 4, if the power and performance of the system under test is of normal importance.

Since the test automation system needs to be synchronized to the application under test, there will be time-out implemented into the test steps. Depending on how fast the test automation system has to drive the system under test, time-out may vary between very sloppy, i.e. Pwr=0 and very tight, i.e. Pwr=6.

Set Port to 0 if the portability of the system under test will not be tested automatically. In case the test automation system will test a web application for different browsers, there will be some substantial effort be dedicated to interface the browser specific characteristics of GUI widget by the test tool. In that case you better assume Port to be 5 or 6.

In our small example projects we set Fkt to 4, Sec to 0, Eff to 5, Pwr to 4 and Port to 5 because the test automation system had to be portable to different kind of tests rather than to test the portability of the system under test. Therefore we had Qde=0.9875.

```
Qdi=Usy+Efc+Pow+Mnt
```

Set the implicit tested usability Usy, efficiency Efc, power Pow and maintainability Mnt to 0, if there are no such requirements for your functional test automation system, thus Qdi becoming 0 and Qd=0.9875.

Tpf=Fpf x Af x Qd are the test points depending on the functionality of the system under test

Af = ((Ur + Ui + I + C) / 20) * U is the balanced influence of the functions under test.

Ur is the user relevance of the functions under test, which can be low (3), neutral (6) or high (12).

Ui is the use intensity of the functions under test, which can be low (2), neutral (4) or high (8).

I is the influence of changes in one function on the other functions of the system, which can be low (2), neutral (4) or high (8).

C is the complexity of the functions under test, which can be low (3), neutral (6) or high (12).

U is the uniformity of the functions under test, it is either 1 (default) or 0.6 in case of a dummy function that will be used in a

similar form at least twice, e.g. a help menu function or error message.

Here's the table for our example project that did not consider anything else but the functionality aspects, not the error messages nor the menu structure of the system under test.

		FP from sheet 2						
	test points							
functional factor	**TPf**	**FPf**	**Br**	**Ni**	**As**	**C**	**E**	**Af**
error messages	0.00	0	6	4	4	6	0.6	0.60
help screens	0.00	0	6	4	4	6	0.6	0.60
menu structure	0.00	0	6	4	4	6	0.6	0.60
functionality	38.16	38.64	6	4	4	6	1	1.00

Conclusion

Once the excel sheet to calculate the influences of the granularity and uniformity and complexity of test steps is known to the designer of the test automation system, the re-usability of the test steps can be calculated. In the example project we had 74% parameterized reusable test steps. Based on the simplicity of the chosen approached in the first spreadsheet, the more complex function point analysis could be used in an easier manner by just counting single sized test step, omitting some of the more complex types of classifications. With the number of function points at hand, the test point analysis was straight forward and gave a rough result about the estimated working hours to build the system. The return on investment was estimated to break even after 3 cycles based in the estimated time saved with the automated regression test in comparison with the manual regression test. In the example project we were able to implement 99% of all steps within the estimated 10 days. The proof of the return of investment is still due, because the test automation system has been given to an external test team for usage while the author went on to the next project.

References

[1] Productivity grid from information in Capers Jones' book "Estimating Software Costs"

http://ipma-wa.com/~ipmacoa1/sites/default/files//page/2003/03/fp_sample_for_sharing.xls

[2] Tmap test management approach http://eng.tmap.net/Home/

[3] Standard glossary of terms used in Software Testing http://www.astqb.org/educational-resources/glossary.php

[4] Tool "Test Point Analysis" http://eng.tmap.net/Home/TMap/Downloads/index.jsp

[5] Test point analysis: details of the technique, page 161ff in Software Testing, A guide to the TMap approach, by Martin Pohl, Ruud Teunissen, Erik van Veenendaal, Addison-Wesley, ISBN 0.201.74571.2

Biography

Michael T. Pilawa serves as an independent test management consultant since 2001 with contributions to the testing community of customers in different branches like pharmaceutical industries, life insurances and electronic equipment manufacturers. He holds a master of science in nuclear physics from the University of Siegen, Germany, since 1988.

What's Fundamentally Wrong?

Improving Our Approach Towards Capturing Value In Requirements Specification

by Tom Gilb & Lindsey Brodie

We are all aware that many of our IT projects fail and disappoint: the poor state of requirements practice is frequently stated as a contributing factor. This article proposes a fundamental cause is that we think like programmers, not engineers and managers. We fail to concentrate on value delivery, and instead focus on functions, on use-cases and on code delivery. Our requirements specification practices fail to adequately address capturing value-related information. Compounding this problem, senior management is not taking its responsibility to make things better: managers are not effectively communicating about value and demanding value delivery. This article outlines some practical suggestions aimed at tackling these problems and improving the quality of requirements specification.

1. Introduction

We know many of our IT projects fail and disappoint, and that the overall picture is not dramatically improving [1] [2]. We also know that the poor state of requirements practice is frequently stated as one of the contributing failure factors [3] [4]. However, maybe a more fundamental cause can be proposed? A cause, which to date has received little recognition, and that certainly fails to be addressed by many well known and widely taught methods. What is this fundamental cause? In a nutshell: that we think like programmers, and not as engineers and managers. In other words, we do not concentrate on value delivery, but instead focus on functions, on use cases and on code delivery. As a result, we pay too little attention to capturing value and value-related information in our requirements specifications. We fail to capture the information that allows us to adequately consider priorities, and engineer and manage stakeholder-valued solutions.

This article outlines some practical suggestions aimed at tackling these problems and improving the quality of requirements specification. It focuses on 'raising the bar' for communicating about value within our requirements. Of course, there is much still to be learnt about specifying value, but we can make a start – and achieve substantial improvement in IT project delivery – by applying what is already known to be good practice.

Note there is little that is new in what follows, and much of what is said can be simply regarded as commonsense. However, since IT projects continue not to grasp the significance of the approach advocated, and as there are people who have yet to encounter this way of thinking, it is worth repeating!

2. Definition of Value

The whole point of a project is achieving 'realized value' (also known as 'benefits'), for the stakeholders: it is not the defined functionality, and not the user stories that actually count. Value can be defined as 'the benefit we think we get from something' [5, page 435]. See Figure 1.

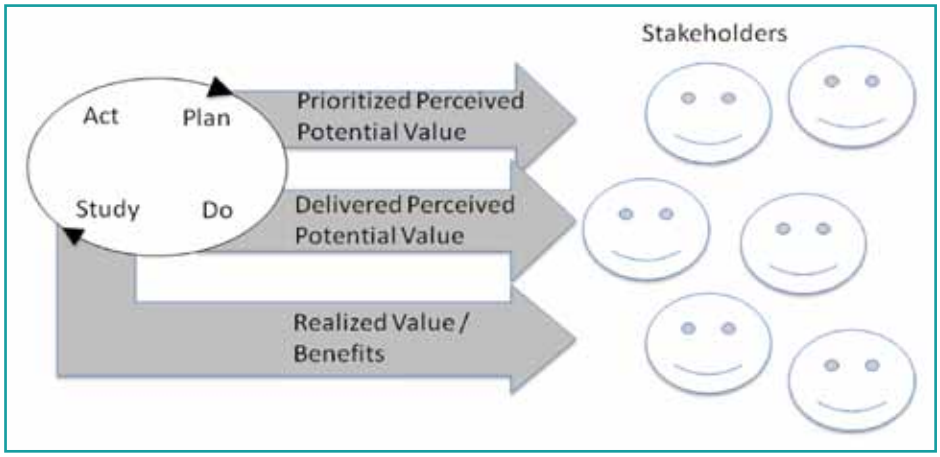


Figure 1. Value can be delivered gradually to stakeholders. Different stakeholders will perceive different value.

Notice the subtle distinction between initially perceived value ('I think that would be useful'), and realized value: effective and factual value ('this was in practice more valuable than we thought it would be, because ...'). Realized value has dependencies on the stakeholders actually utilizing a project's deliverables.

The issue with much of the conventional requirements thinking is that it is not closely enough coupled with 'value'. IT business analysts frequently fail to gather the information supporting

a more precise understanding and/or the calculation of value. Moreover, the business people when stating their requirements frequently fail to justify them using value. The danger if requirements are not closely tied to value is that we lack the basic information allowing us to engineer and prioritize implementation to achieve value delivery, and we risk failure to deliver the required expected value, even if the 'requirements' are satisfied.

It is worth pointing out that 'value' is multi-dimensional. A given requirement can have financial value, environmental value, competitive advantage value, architectural value, as well as many other dimensions of value. Certainly value requires much more explicit definition than the priority groups used by MoSCoW ('Must Have', 'Should Have', 'Could Have', and 'Would like to Have/Won't Have This Time') [6] or by the Planning Game ('Essential', 'Less Essential' and 'Nice To Have') [7] for prioritizing requirements. Further, for an IT project, engineering 'value' also involves consideration of not just the requirements, but also the optional designs and the resources available: tradeoffs are needed. However, these are topics for future articles, this article focuses on the initial improvements needed in requirements specification to start to move towards value thinking.

3. Definition of Requirement

Do we all have a shared notion of what a 'requirement' is? This is another of our problems. Everybody has an opinion, and many of the opinions about the meaning of the concept 'requirement' are at variance: few of the popular definitions are correct or useful - especially when you consider the concept of 'value' alongside them. We have decided to define a requirement as a "stakeholder-valued end state". You possibly will not accept, or use this definition yet, but we have chosen it to emphasize the 'point' of IT systems engineering.

In previous work, we have identified, and defined a large number of requirement concepts [5, see Glossary, pages 321-438]. A sample of these concepts is given in Figure 2. You can use these concepts and the notion of a "stakeholder-valued end state" to re-examine your current requirements specifications. In the rest of this article, we provide more detailed discussion about some of the key points (the "key principles") you should consider.

4. The Key Principles

The key principles are summarized in Figure 3. Let's now examine these principles in more detail.

Note, unless otherwise specified, further details on all aspects of Planguage (a planning language developed by one of the authors, Tom Gilb) can be found in [5].

Ten Key Principles for Successful Requirements

1. Understand the top level critical objectives
2. Think stakeholders: not just users and customers!
3. Focus on the required system quality, not just its functionality
4. Quantify quality requirements as a basis for software engineering
5. Don't mix ends and means
6. Capture explicit information about value
7. Ensure there is 'rich specification': requirement specifications need far more information than the requirement itself!
8. Carry out specification quality control (SQC)
9. Consider the total lifecycle and apply systems-thinking - not just a focus on software
10. Recognize that requirements change: use feedback and update requirements as necessary

Figure 3. Ten Key Principles for Successful Requirements

Principle 1. Understand the top-level critical objectives

The 'worst requirement sin of all' is found in almost all the IT projects we look at, and this applies internationally. Time and again, the high-level requirements – also known as the top-level critical objectives (the ones that fund the project), are vaguely stated, and ignored by the project team. Such requirements frequently look like the example given in Figure 4 (which has been slightly edited to retain anonymity). These requirements are for a real project that ran for eight years and cost over 100 million US dollars. The project failed to deliver any of them. However, the main problem is that these are not top-level critical objectives: they fail to explain in sufficient detail what the business is trying to achieve: there are no real pointers to indicate the business aims and priorities. There are additional problems as well that will be discussed further later (such as lack of quantification, mixing optional designs into the requirements, and insufficient background description).

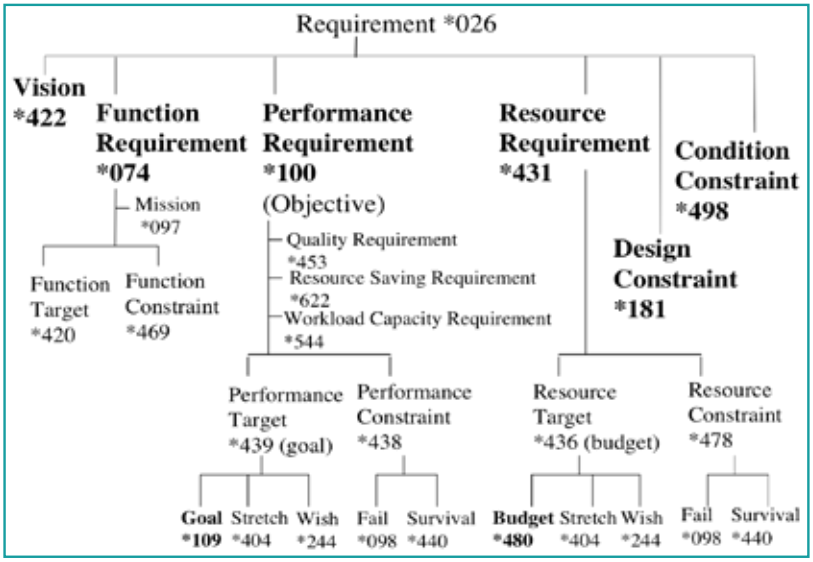


Figure 2: Example of Planguage requirements concepts

Example of Initial Weak Top-Level Critical Objectives

1. Central to the corporation's business strategy is to be the world's premier integrated <domain> service provider
2. Will provide a much more efficient user experience
3. Dramatically scale back the time frequently needed after the last data is acquired to time align, depth correct, splice, merge, recomputed and/or do whatever else is needed to generate the desired products
4. Make the system much easier to understand and use than has been the case with the previous system
5. A primary goal is to provide a much more productive system development environment then was previously the case
6. Will provide a richer set of functionality for supporting next generation logging tools and applications
7. Robustness is an essential system requirement
8. Major improvements in data quality over current practices

Figure 4. Example of Initial Weak Top Level Critical Objectives

Management at the CEO, CTO and CIO level did not take the trouble to clarify these critical objectives. In fact, the CIO told me that the CEO actively rejected the idea of clarification! So management lost control of the project at the very beginning. Further, none of the technical 'experts' reacted to the situation. They happily spent $100 million on all the many suggested architecture solutions that were mixed in with the objectives.

It actually took less than an hour to rewrite one of these objectives, "Robustness", so that it was clear, measurable, and quantified (see later). So in one day's work the project could have clarified the objectives, and perhaps avoided some of the eight years of wasted time and effort.

Principle 2. Think stakeholders: not just users and customers!

Too many requirements specifications limit their scope to being too narrowly focused on user or customer needs. The broader area of stakeholder needs and values should be considered, where a 'stakeholder' is anyone or anything that has an interest in the system [5, page 420]. It is not just the users and customers that must be considered: IT development, IT maintenance, senior management, operational management, regulators, government, as well as other stakeholders can matter. The different stakeholders will have different viewpoints on the requirements and their associated value. Further, the stakeholders will be "experts" in different areas of the requirements. These different viewpoints will potentially lead to differences in opinion over the implementation priorities.

Principle 3. Focus on the required system quality, not just its functionality

Far too much attention is paid to what the system must do (function) and far too little attention is given to how well it should do it (qualities). Many requirements specifications consist of detailed explanation of the functionality with only brief description of the required system quality. This is in spite of the fact that quality improvements tend to be the major drivers for new projects.

In contrast, here's an example, the Confirmit case study [8], where the focus of the project was not on functionality, but on driving up the system quality. By focusing on the "Usability" and "Performance" quality requirements the project achieved a great deal! See Table 1.

Table 1. Extract from Confirmit Case Study [8]

Description of requirement/work task	Past	Current Status
Usability.Productivity: Time for the system to generate a survey	7200 sec	15 sec
Usability.Productivity: Time to set up a typical market research report	65 min	20 min
Usability.Productivity: Time to grant a set of end-users access to a report set and distribute report login information	80 min	5 min
Usability.Intuitiveness: The time in minutes it takes a medium-experienced programmer to define a complete and correct data transfer definition with Confirmit Web Services without any user documentation or any other aid	15 min	5 min
Performance.Runtime.Concurrency: Maximum number of simultaneous respondents executing a survey with a click rate of 20 sec and a response time < 500ms given a defined [Survey Complexity] and a defined [Server Configuration, Typical]	250 users	6000

By system quality we mean all the "-ilities" and other qualities that a system can express. Some system developers limit system quality to referring to bug levels in code. However, a broader definition should be used. System qualities include availability, usability, portability, and any other quality that a stakeholder is interested in, like intuitiveness or robustness. See Figure 5, which shows a set of quality requirements. It also shows the notion that resources are "input" or used by a function, which in turn "outputs" or expresses system qualities. Sometimes the system qualities are mis-termed "non-functional requirements (NFRs)", but as can be seen in this figure, the system qualities are completely linked to the system functionality. In fact, different parts of the system functionality are likely to require different system qualities.

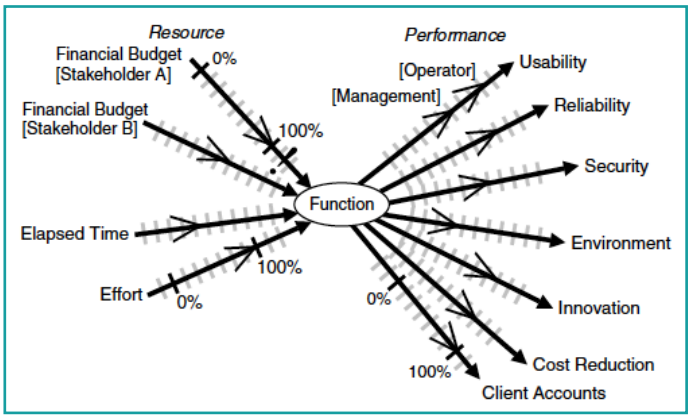


Figure 5. A way of visualizing qualities in relation to function and cost. Qualities and costs are scalar variables, so we can define scales of measure in order to discuss them numerically. The arrows on the scale arrows represent interesting points, such as the requirement levels. The requirement is not 'security' as such, but a defined, and testable degree of security [5, page 163]

Principle 4. Quantify quality requirements as a basis for software engineering

Frequently we fail to practice "software engineering" in the sense of real engineering as described by engineering professors, like Koen [9]. All too often quality requirements specifications consist merely of words. No numbers, just nice sounding words; good

enough to fool managers into spending millions for nothing (for example, "a much more efficient user experience").

We seem to almost totally avoid the practice of quantifying qualities. Yet we need quantification in order to make the quality requirements clearly understood, and also to lay the basis for measuring and tracking our progress in improvement towards meeting them. Further, it is the quantification that is the key to a better understanding of cost and value – different levels of quality have different associated cost and value.

The key idea for quantification is to define, or reuse a definition, of a scale of measure. For example, for a quality "Intuitiveness", a sub-component of "Usability":

Usability.Intuitiveness:
Type: Marketing Product Quality Requirement.
Ambition: Any potential user, any age, can immediately discover and correctly use all functions of the product, without training, help from friends, or external documentation.
Scale: % chance that defined [User] can successfully complete defined [Tasks] <immediately> with no external help.
Meter: Consumer reports tests all tasks for all defined user types, and gives public report.
Goal [Market = USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, When = 2012]: 80% ±10% <- Draft Marketing Plan.

Figure 6. A simple example of quantifying a quality requirement, 'Intuitiveness'.

To give some explanation of the key quantification features in Figure 6:

1. Ambition is a high-level summary of the requirement. One that is easy to agree to, and understand roughly.

2. Scale is the formal definition of our chosen scale of measure. The parameters [User] and [Task] allow us to generalize here, while becoming more specific in detail below (see later). They also encourage and permit the reuse of the Scale, as a sort of 'pattern'.

3. Meter provides a defined measuring process. There can be more than one for different occasions.

4. Goal is one of many possible requirement levels (see earlier detail in Figure 2 for some others: Stretch, Wish, Fail and Survival). We are defining a stakeholder-valued future state (for example: 80% ± 10%).

One *stakeholder* is 'USA Seniors'. The *future* is 2012. The requirement level type, Goal, is defined as a very high priority, budgeted promise of delivery. It is of higher priority than a Stretch or Wish level. Note other priorities may conflict and prevent this particular requirement from being delivered in practice.

If you know the *conventional* state of requirements methods, then you will now, from this example alone, begin to appreciate the difference proposed by such quantification - especially for quality requirements. IT projects already quantify time, cost, response time, burn rate, and bug density – but there is much *more to achieve system engineering*!

Here is another example of quantification (see Figure 7). It is the initial stage of the rewrite of Robustness from the Figure 4 example. First we determined that Robustness is complex and composed of many different attributes, such as Testability.

Robustness:
Type: Complex Product Quality Requirement.
Includes: {Software Downtime, Restore Speed, Testability, Fault Prevention Capability, Fault Isolation Capability, Fault Analysis Capability, Hardware Debugging Capability}.

Figure 7. Definition of a complex quality requirement, Robustness

Then we defined Testability in more detail (see Figure 8).

Testability:
Type: Software Quality Requirement.
Version: Oct 20, 2006.
Status: Draft.
Stakeholder: {Operator, Tester}.
Ambition: Rapid duration automatic testing of <critical complex tests> with extreme operator setup and initiation.
Scale: The duration of a defined [Volume] of testing or a defined [Type of Testing] by a defined [Skill Level] of system operator under defined [Operating Conditions].
Goal [All Customer Use, Volume = 1,000,000 data items, Type of Testing = WireXXXX vs. DXX, Skill Level = First Time Novice, Operating Conditions = Field]: < 10 minutes.
Design: Tool simulators, reverse cracking tool, generation of simulated telemetry frames entirely in software, application specific sophistication for drilling – recorded mode simulation by playing back the dump file, application test harness console <- 6.2.1 HFS.

Figure 8. Quantitative definition of Testability, an attribute of Robustness

Note this example shows the notion of there being different levels of requirements. Principle 1 also has relevance here as it is concerned with top-level objectives (requirements). The different levels that can be identified include: corporate requirements, the top-level critical few project or product requirements, system requirements and software requirements. We need to clearly document the level and the interactions amongst these requirements.

An additional notion is that of 'sets of requirements'. Any given stakeholder is likely to have a set of requirements rather than just an isolated single requirement. In fact, achieving value could depend on meeting an entire set of requirements.

Principle 5. Don't mix ends and means

"Perfection of means and confusion of ends seem to characterize our age." Albert Einstein. 1879-1955

The problem of confusing ends and means is clearly an old one, and deeply rooted. We specify a solution, design and/or architecture, instead of what we really value – our real requirement. There are explanatory reasons for this – for example solutions are more concrete, and what we want (qualities) are more abstract for us (because we have not yet learned to make them measurable).

The problems occur when we do confuse them: if we do specify the means, and not our true ends. As the saying goes: "Be careful what you ask for, you might just get it" (unknown source). The problems include:

- You might not get what you really want

- The solution you have specified might *cost too much* or have bad *side effects*, even if you do get what you want

- There may be much *better solutions* you don't know about yet.

So how do we find the 'right requirement', the 'real requirement' [10] that is being 'masked' by the solution? *Assume* that there probably is a better formulation, which is a more accurate expression of our real values and needs. Search for it by asking 'Why?' Why do I want X, it is because I really want Y, and assume I will get it through X. But, then why do I want Y? Because I really want Z and assume that is the best way to get X. Continue the process until it seems reasonable to stop. This is a slight variation on the '5 Whys' technique [11], which is normally used to identify root causes of problems (rather than high-level requirements).

Assume that our stakeholders will *usually* state their values in terms of some perceived means to get what they really value. Help them to identify (The 5 Whys?) and to acknowledge what they really want, and make that the 'official' requirement. Don't insult them by telling them that they don't know what they want. But explain that you will help them more-certainly get what they more deeply want, with better and cheaper solutions, perhaps new technology, if they will go through the '5 Whys?' process with you. See Figure 9.

Why do you require a 'password'? For Security!
What kind of security do you want? Against stolen information.
What level of strength of security against stolen information are you willing to pay for? At least a 99% chance that hackers cannot break in within 1 hour of trying! Whatever that level costs up to €1 million.
So that is your real requirement? Yep.
Can we make that the official requirement, and leave the security design to both our security experts, and leave it to proof by measurement to decide what is really the right design? Of course!
The aim being that whatever technology we choose, it gets you the 99%?
Sure, thanks for helping me articulate that!

Figure 9. Example of the requirement, not the design feature, being the real requirement

Note that this separation of designs from the requirements does not mean that you ignore the solutions/designs/architecture when software engineering. It is just that you must separate your requirements - including any mandatory means - from any optional means. The key thing is to understand what is optional so that you consider alternative solutions. See Figure 10, which shows two alternative solutions: Design A with Designs B and C, or Design A with Design D. Assuming that say, Design B was mandatory, could distort your project planning.

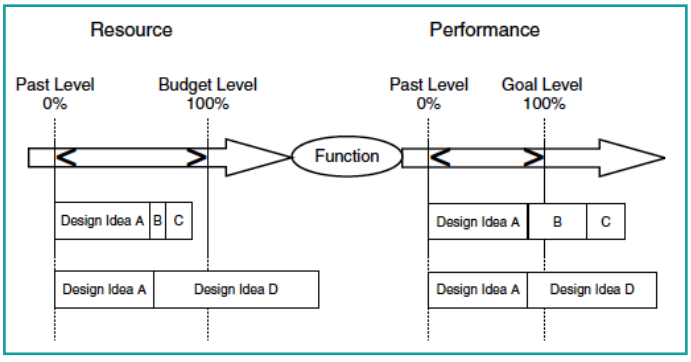


Figure 10. A graphical way of understanding performance attributes (which include all qualities) in relation to function, design and resources. Design ideas cost some resources, and design ideas deliver performance (including system qualities) for given functions. Source [5, page 192].

Principle 6. Capture explicit information about value

How can we articulate and document notions of value in a requirement specification? See the example for Intuitiveness, a component quality of Usability, given in Figure 11, which expands on Figure 6.

Usability.Intuitiveness:
Type: Marketing Product Requirement.
Stakeholders: {Marketing Director, Support Manager, Training Center}.
Impacts: {Product Sales, Support Costs, Training Effort, Documentation Design}.
Supports: Corporate Quality Policy 2.3.
Ambition: Any potential user, any age, can immediately discover and correctly use all functions of the product, without training, help from friends, or external documentation.
Scale: % chance that a defined [User] can successfully complete the defined [Tasks] <immediately>, with no external help.
Meter: Consumer Reports tests all tasks for all defined user types, and gives public report.
---------- Analysis ----------
Trend [Market = Asia, User = {Teenager, Early Adopters}, Product = Main Competitor, Projection = 2013]: 95%±3% <- Market Analysis.
Past [Market = USA, User = Seniors, Product = Old Version, Task = Photo Tasks Set, When = 2010]: 70% ±10% <- Our Labs Measures.
Record [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Record Set = January 2010]: 98% ±1% <- Secret Report.
---------- Our Product Plans ----------
Goal [Market = USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, When = 2012]: 80% ±10% <- Draft Marketing Plan.
Value [Market =USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, Time Period = 2012]: 2M USD.
Tolerable [Market = Asia, User = {Teenager, Early Adopters}, Product = Our New Version, Deadline = 2013]: 97%±3% <- Marketing Director Speech.
Fail [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Product Release 9.0]: Less Than 95%.
Value [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Time Period = 2013]: 30K USD.

Figure 11. A fictitious Planguage example, designed to display ways of making the value of a requirement clear

For brevity, a detailed explanation is not given here. Hopefully, the language specification is reasonably understandable without detailed explanation. For example, the Goal statement (80%) specifies which market ("USA") and users ("Seniors") it is intended for, which set of tasks are valued (the "Photo Tasks Set"), and when it would be valuable to get it delivered ("2012"). This 'qualifier' information in all the statements, helps document where, who, what, and when the quality level applies. The additional Value parameter specifies the perceived value of achieving 100% of the requirement. Of course, more could be said about value and its specification, this is merely a 'wake-up call' that explicit value

needs to be captured within requirements. It is better than the more common specifications of the Usability requirement that we often see, such as: "The product will be more user-friendly, using Windows".

So who is going to make these value statements in requirements specifications? I don't expect developers to care much about value statements. Their job is to deliver the requirement levels that someone else has determined are valued. Deciding what sets of requirements are valuable is a Product Owner (Scrum) or Marketing Management function. Certainly, the IT staff should only determine the value related to IT stakeholder requirements!

Principle 7. Ensure there is 'rich specification': requirement specifications need far more information than the requirement itself!

Far too much emphasis is often placed on the requirement itself; and far too little concurrent information is gathered about its background, for example: who wants this requirement and when? The requirement itself might be less than 10% of a complete requirement specification that includes the background information. It should be a corporate standard to specify this related background information, and to ensure it is intimately and immediately tied into the requirement itself.

Such background information is useful related information, but is not central (core) to the implementation, and nor is it commentary. The central information includes: Scale, Meter, Goal, Definition and Constraint.

Background specification includes: benchmarks {Past, Record, Trend}, Owner, Version, Stakeholders, Gist (brief description), Ambition, Impacts, and Supports. The rationale for background information is as follows:

- To help judge the value of the requirement
- To help prioritize the requirement
- To help understand the risks associated with the requirement
- To help present the requirement in more or less detail for various audiences and different purposes
- To give us help when updating a requirement
- To synchronize the relationships between different but related levels of the requirements
- To assist in quality control of the requirements
- To improve the clarity of the requirement.

Commentary is any detail that probably will not have any economic, quality or effort consequences if it is incorrect, for example, notes and comments.

See Figure 12 for an example, which illustrates the help given by background information regarding risks.

Reliability:
Type: Performance Quality.
Owner: Quality Director. Author: John Engineer.
Stakeholders: {Users, Shops, Repair Centers}.
Scale: Mean Time Between Failure.
Goal [Users]: 20,000 hours <- Customer Survey, 2004.
Rationale: Anything less would be uncompetitive.
Assumption: Our main competitor does not improve more than 10%.
Issues: New competitors might appear.
Risks: The technology costs to reach this level might be excessive.
Design Suggestion: Triple redundant software and database system.
Goal [Shops]: 30,000 hours <- Quality Director.
Rationale: Customer contract specification.
Assumption: This is technically possible today.
Issues: The necessary technology might cause undesired schedule delays.
Risks: The customer might merge with a competitor chain and leave us to foot the costs for the component parts that they might no longer require.
Design Suggestion: Simplification and reuse of known components.

Figure 12. A requirement specification can be embellished with many background specifications that will help us to understand risks associated with one or more elements of the requirement specification [12].

Background information must not be scattered around in different documents and meeting notes. It needs to be directly integrated into a sole master reusable requirement specification object. Otherwise it will not be available when it is needed: it will not be updated, or shown to be inconsistent with emerging improvements in the requirement specification.

See Figure 13 for a requirement template for function specification [5, page 106], which hints at the richness possible for background information.

Principle 8. Carry out specification quality control (SQC)

There is far too little quality control of requirements against relevant standards. All requirements specifications ought to pass their quality control checks before they are released for use by the next processes. Initial quality control of requirements specification, where there has been no previous use of specification quality control (SQC) (also known as Inspection), using three simple quality-checking rules ('unambiguous to readers', 'testable' and 'no optional designs present'), typically identifies 80 to 200+ words per 300 words of requirement text as ambiguous or unclear to intended readers! [13]

Principle 9. Consider the total lifecycle and apply systems-thinking - not just a focus on software

If we don't consider the total lifecycle of the system, we risk failing to think about all the things that are necessary prerequisites to actually delivering *full value* to real *stakeholders* on time. For example, if we want better maintainability then it has to be designed into the system. If we are really engineering costs, then we need to think about the total operational costs over time. This is much more than just considering the programming aspects.

You must take into account the nature of the system: an exploratory web application doesn't need the same level of software engineering as a real-time banking system!

TEMPLATE FOR FUNCTION SPECIFICATION <with hints>
Tag: <Tag name for the function>.
Type: <{Function Specification, Function (Target) Requirement, Function Constraint}>.
========== Basic Information ==========
Version: <Date or other version number>.
Status: <{Draft, SQC Exited, Approved, Rejected}>.
Quality Level: <Maximum remaining major defects/page, sample size, date>.
Owner: <Name the role/email/person responsible for changes and updates to this specification>.
Stakeholders: <Name any stakeholders with an interest in this specification>.
Gist: <Give a 5 to 20 word summary of the nature of this function>.
Description: <Give a detailed, unambiguous description of the function, or a tag reference to someplace where it is detailed. Remember to include definitions of any local terms>.
========== Relationships ==========
Supra-functions: <List tag of function/mission, which this function is a part of. A hierarchy of tags, such as A.B.C, is even more illuminating. Note: an alternative way of expressing supra-function is to use Is Part Of>.
Sub-functions: <List the tags of any immediate sub-functions (that is, the next level down), of this function. Note: alternative ways of expressing sub-functions are Includes and Consists Of>.
Is Impacted By: <List the tags of any design ideas or Evo steps delivering, or capable of delivering, this function. The actual function is NOT modified by the design idea, but its presence in the system is, or can be, altered in some way. This is an Impact Estimation table relationship>.
Linked To: <List names or tags of any other system specifications, which this one is related to intimately, in addition to the above specified hierarchical function relations and IE-related links. Note: an alternative way is to express such a relationship is to use Supports or Is Supported By, as appropriate>.
========== Measurement ==========
Test: <Refer to tags of any test plan or/and test cases, which deal with this function>.
========== Priority and Risk Management ==========
Rationale: < Justify the existence of this function. Why is this function necessary? >.
Value: <Name [Stakeholder, time, place, event>]: <Quantify, or express in words, the value claimed as a result of delivering the requirement>.
Assumptions: <Specify, or refer to tags of any assumptions in connection with this function, which could cause problems if they were not true, or later became invalid>.
Dependencies: <Using text or tags, name anything, which is dependent on this function in any significant way, or which this function itself, is dependent on in any significant way>.
Risks: <List or refer to tags of anything, which could cause malfunction, delay, or negative impacts on plans, requirements and expected results>.
Priority: <Name, using tags, any system elements, which this function can clearly be done after or must clearly be done before. Give any relevant reasons>.
Issues: <State any known issues>.
========== Specific Budgets ==========
Financial Budget: <Refer to the allocated money for planning and implementation (which includes test) of this function>.

Figure 13. A template for function specification [5, page 106]

Principle 10. Recognize that requirements change: use feedback and update requirements as necessary

Ideally requirements must be developed based on on-going feedback from stakeholders, as to their real value. System development methods, such as the agile methods, enable this to occur. Stakeholders can give feedback about their perception of value, based on the *realities* of actually using the system. The requirements must be *evolved* based on this realistic experience. The whole process is a 'Plan Do Study Act' Shewhart cyclical learning process involving many complex factors, including factors from outside the system, such as politics, law, international differences, economics, and technology change.

Attempts to fix the requirements in advance of feedback, are typically wasted energy (unless the requirements are completely known upfront, which might be the case in a straightforward system rewrite with no system changes). Committing to fixed requirements specifications in contracts is not realistic.

5. Who or What Will Change Things?

Everybody talks about requirements, but few people seem to be making progress to enhance the quality of their requirements specifications and improve support for software engineering. Yes, there are internationally competitive businesses, like HP and Intel that have long since improved their practices because of their competitive nature and necessity [8, 14]. But they are very different from the majority of organizations building software. The vast majority of IT systems development teams we encounter are not highly motivated to learn or practice first class requirements (or anything else!). Neither the managers nor the systems developers seem strongly motivated to improve. The reason is that they get by with, and even get well paid for, failed projects.

The universities certainly do not train IT/computer science students well in requirements, and the business schools also certainly do not train managers about such matters [15]. The fashion now seems to be to learn oversimplified methods, and/or methods prescribed by some certification or standardization body. Perhaps insurance companies and lawmakers might demand better industry practices, but I fear that even that would be corrupted in practice if history is any guide (for example, think of CMMI and the various organization certified as being at Level 5).

6. Summary

Current requirements specification practice is often woefully inadequate for today's critical and complex systems. Yet we do

know a considerable amount (Not all!) about good practice. The main question is whether your 'requirements' actually capture the true breadth of information that is needed to make a start on engineering value for your stakeholders.

Here are some specific questions for you to ask about your current IT project's requirements specification:

Do you have a list of top-level critical objectives?

Do you consider multiple stakeholder viewpoints?

Do you know the expected stakeholder value to be delivered?

Have you quantified your top five quality attributes? Are they all testable? What are the current levels for these quality attributes?

Are there any optional designs in your requirements?

Can you state the source of each of your requirements?

What is the quality level of your requirements documentation? That is, the number of major defects remaining per page?

When are you planning to deliver stakeholder value? To which stakeholders?

If you can't answer these questions with the 'right' answers, then you have work to do! And you might also better understand why your IT project is drifting from delivering its requirements. The good news is that the approach outlined in this article should allow you to focus rapidly on what really matters to your stakeholders: value delivery.

References

1. Thomas Carper, Report Card to the Senate Hearing "Off-Line and Off-Budget: The Dismal State of Federal Information Technology Planing", July 31, 2008. See http://uscpt.net/CPT_InTheNews.aspx [Last Accessed: August 2010].

2. The Standish Group, "Chaos Summary 2009", 2009. See http://www.standishgroup.com/newsroom/chaos_2009.php [Last Accessed: August 2010].

3. John McManus and Trevor Wood-Harper, "A Study in Project Failure", June 2008. See http://www.bcs.org/server.php?show=ConWebDoc.19584 [Last Accessed: August 2010].

4. David Yardley, Successful IT Project Delivery, Addison-Wesley, 2002. ISBN 0201756064.

5. Tom Gilb, "Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering using Planguage", Elsevier Butterworth-Heinemann, 2005.

6. Jennifer Stapleton (Editor), DSDM: Business Focused Development (2nd Edition), Addison Wesley, 2003. ISBN 0321112245. First edition published in 1997.

7. Mike Cohn, User Stories Applied: For Agile Software Development, Addison Wesley, 2004. ISBN 0321205685.

8. Trond Johansen and Tom Gilb, From Waterfall to Evolutionary Development (Evo): How we created faster, more user-friendly, more productive software products for a multi-national market, Proceedings of INCOSE, 2005. See http://www.gilb.com/tiki-download_file.php?fileId=32

9. Dr. Billy Vaughn Koen, "Discussion of the Method: Conducting the Engineer's Approach to Problem Solving", Oxford University Press, 2003.

10. Tom Gilb, Real Requirements, see http://www.gilb.com/tiki-download_file.php?fileId=28

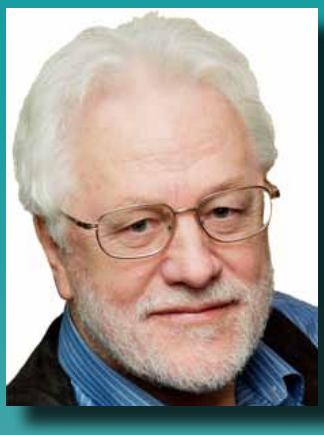
11. Taiichi Ohno, "Toyota production system: beyond large-scale production", Productivity Press, 1988.

12. Tom Gilb, "Rich Requirement Specs: The use of Planguage to clarify requirements", see http://www.gilb.com/tiki-download_file.php?fileId=44

13. Tom Gilb, Agile Specification Quality Control, Testing Experience, March 2009. Download from www.testingexperience.com/testing experience01_08.pdf [Last Accessed: August 2010].

14. Top Level Objectives: A slide collection of case studies. See, http://www.gilb.com/tiki-download_file.php?fileId=180

15. Kenneth Hopper and William Hopper, "The Puritan Gift", I. B. Taurus and Co. Ltd., 2007.



Biography

Tom Gilb (born 1940, California) moved to UK 1956, and to Norway since 1958. He is the author of 9 published books, including Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, 2005. Edited by this paper co-author Lindsey Brodie.

He has taught and consulted world-wide for decades, including having direct corporate methods-change influence at major corporations such as Intel, HP, IBM, Nokia. He has had documented his founding influence in Agile Culture, especially with the key common idea of iterative development. He coined the term 'Software Metrics' with his 1976 book of that title. He is co-author with Dorothy Graham of the static testing method book 'Software Inspection' (1993). He is known for consistently avoiding the oversimplified pop culture that regularly entices immature programmers to waste time and fail on their projects.

More detail at www.gilb.com.

Lindsey Brodie is currently carrying out research on prioritization of stakeholder value, and teaching part-time at Middlesex University. She has an MSc in Information Systems Design from Kingston Polytechnic. Her first degree was Joint Honours Physics and Chemistry from King's College, London University. Lindsey worked in industry for many years, mainly for ICL. Initially, Lindsey worked on project teams on customer sites (including the Inland Revenue, Barclays Bank, and J. Sainsbury's) providing technical support and developing customised software for operations. From there, she progressed to product support of mainframe operating systems and data management software: databases, data dictionary and 4th generation applications. Having completed her Masters, she transferred to systems development - writing feasibility studies and user requirements specifications, before working in corporate IT strategy and business process re-engineering. Lindsey has collaborated with Tom Gilb and edited his book, "Competitive Engineering". She has also co-authored a student textbook, "Successful IT Projects" with Darren Dalcher (National Centre for Project Management). She is a member of the BCS and a Chartered IT Practitioner (CITP).

Es gibt nur einen Weg zum Glück.

gebrauchtwagen.de
Autos zum Verlieben.

Implementing a metrics program to guide quality improvements

by Harish Narayan & Rex Black

Do you have questions on to how to establish and implement a quality metrics program? How to define quality metrics within a program? What are some real-world metrics one could implement in their organization to guide improvement or transformation initiatives? If so, this article will be of value to you.

In this article, we will share an overview of a metrics taxonomy based on three broad categories (product or service, process and project) and two intended uses (indicative and diagnostic) with some specific examples. We will outline a five step metrics program with an illustrative thread. Finally, we will share a case study from a financial services company to show how metrics helped guide a transformation effort.

A simple metrics taxonomy

Implementing a metrics program to guide quality improvements – the title has two words that are ambiguous many a time. How do you define 'metric' and 'quality'? The definition of metric is less ambiguous. Classically, a metric is a measurable indication of some kind. Usually, it is a quantitative aspect of a product, service, process or project. Tom DeMarco's simple definition is that a metric is a number attached to an idea.

The use of metrics is however undisputed. Metrics are used in almost every walk of life. In software technology, metrics are typically used as follows:

- to monitor how something is performing
- to analyze the information provided to drive improvements
- to predict what will happen in the future by comparing plans to actuals, or analyzing trends over time.

Quality is a more complex concept and is misunderstood due to the levels of abstraction – being referred to in a broad sense or as a very specific attribute – as in, 'Japanese-made car' to indicate quality or 'mean-time between transmission failure' to measure a specific quality attribute. Broadly around the world, quality is usually perceived and interpreted in different ways and hence challenging to quantify. The notion of quality also changes over time. What may be considered by a customer as a 'nice to have' today may become a 'must have' tomorrow.

In the discipline of quality engineering and as applied to software technology, quality and its attributes can be operationally defined, measured and managed. Crosby defined quality as 'conformance to requirements' and Juran as 'fitness for use'. Quality is also a customer's viewpoint of perceived value. Specific definitions of quality in software are centered on defects found in the software. However, attributes such as usability, performance, reliability etc., and notions of customer satisfaction are all part of the quality paradigm. In our view quality needs to be defined based on the context of use. In this article, we use the term quality in two ways. One, as a broad qualifier in the context of transforming a quality assurance organization and the other, to share specific quality metrics used in such organizations.

Metrics used to measure, monitor and improve quality are broadly termed as quality metrics. Software quality metrics can be classified along these broad categories – product or service, process and project. Table 1 defines each metric type and provides some examples.

Quality Metric Type	Definition	Examples
Product / Service	Product quality metrics describe the characteristics or attributes of a product such as features, complexity, performance and defects. Service metrics describe the attributes of a service like effectiveness, customer experience and performance.	• *Defect Density* • *Mean Time To Failure (MTTR)* • *Customer Issues* • *Customer Satisfaction Score*

Quality Metric Type	Definition	Examples
Process	Process quality metrics describe the behavior of a process and can be used to improve such processes.	• *Test Effectiveness* • *Phase Containment* • *Defect Removal Efficiency* • *Fix Response Time* • *Issue Backlog* • *Cost of Quality* • *Test Cycle Time*
Project	Project quality metrics describe the characteristics of a project and help manage risk on a project from planning to execution to closure.	• *Project Cost of Quality* • *Requirements Coverage* • *Defects Found in a project* • *Tests Passed vs. Failed* • *Most process or product metrics at a project level*

Table 1: Metrics Taxonomy

Indicative metrics are used to monitor performance or get an initial assessment of anything that is measured. Typical presentations include time-based trends, aggregated customer satisfaction scores etc. The example illustrated in Figure 1 below is a monthly bar chart of customer issues coming into support.

Figure 1: Incoming Customer Issues

The example shows that while the trend of number of incoming customer issues is declining, and while the number of incoming severity 1 and 2 issues are also declining, severity 1 and 2 issues are increasing as a percentage of all incoming.

Diagnostic metrics include information that helps in drilling down further, to identify root cause, and directly help in addressing issues or driving improvement. Typical presentations include a pareto chart of defects by feature, testing time classified by setup, execution, triage, reporting and closure etc. An example of defects by severity and feature is illustrated in Figure 2 below. Based on the chart in figure 1, if we are interested in a question such as, "Where are severity 1 & 2 defects coming from and what set of defects should be reduced?", then such a chart would help in further analysis.

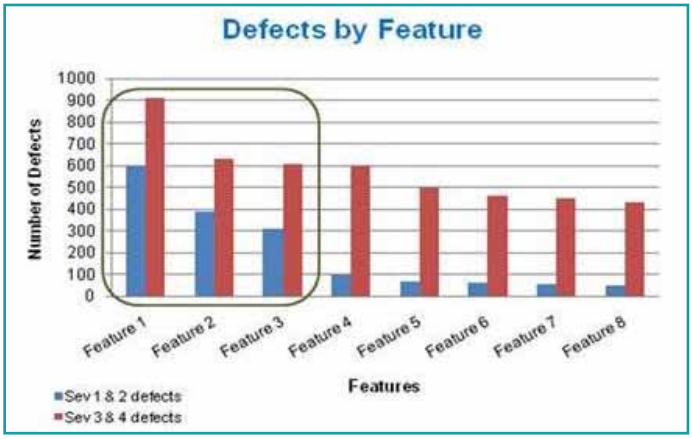


Figure 2: Defects by Feature and Severity

The example above shows the total defects for say, a year, charted as a pareto of severity 1 and 2 defects by feature, along with severity 3 and 4 defects. It shows that features 1, 2 and 3 contribute to 80% of all severity 1 and 2 defects. It also shows that in relative terms, severity 1 and 2 defects in features 4 through 8 are a much smaller percentage compared to severity 3 and 4 defects. Such information can be used to prioritize quality improvement efforts.

A combination of various metrics, when aggregated, can be used to indicate and drive improvement in organizations. Metrics are valuable when used as an integral program to strategically support any organization's quality goals to dramatically change or improve.

A five-step metrics program

Many organizations understand and realize the importance of metrics in evaluating and managing an organization's performance and quality. However, most are not clear on what metrics to use, what measures to collect, how to collect measurements, and how to use collected measures effectively. One quick note – metrics are derived from a set of measures, and measurements are data collected from various organization information sources.

Establishing and operating a metrics program is a very essential component to the successful management and improvement of a quality organization. The basis of the program we outline draws from the Goal-Question-Metric (GQM) paradigm developed by Victor Basili. We now describe a five-step process that can be used

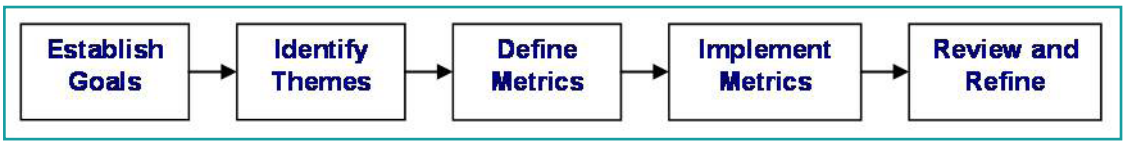


Figure 3: Five-step Metrics Program

to build and operate a metrics program. Figure 3 graphically illustrates this process.

The steps are:

1. **Establish business-aligned goals** - Goals can be broad or more specific but should be aligned to meet specific business needs. As seen with a quality-tinted lens, here are some illustrative goals:

- Increase customer satisfaction by 20% in 2 years
- Reduce cost of quality by 10% in 3 years
- Improve quality in production year over year
- Reduce mean time to repair (MTTR) by 3 days in 1 year

2. **Identify themes and associated metrics** - Themes are useful when defining and presenting information back to stakeholders. Goals will be measured and tracked by metrics and metrics are broadly defined based on questions one can ask about how to accomplish a goal. One such set is illustrated below:

- Goal: Improve quality in production year over year
- Questions:
  - What is production quality of the system as it is today?
  - How many issues escape the development and test process into production?
  - How many issues are found in testing and does the process allow for most number of issues to be found early in the test lifecycle?
- Metrics: (Theme - Effectiveness)
  - Production quality can be defined by new software defects found in production and by the number of total defects found in production
  - Issue escape can be quantified by defect containment effectiveness of the development and test process
  - Issues found early in testing can be indirectly measured by a metric like first-pass run rate.

3. **Operationally define the metrics** - This will involve defining the measures and metrics, the collection process, identifying the intended audience, guidelines on how to use the metrics and report them to stakeholders. Measurements should answer the questions related to the goals, have information content that truly adds value, be easy to collect and report, and be able to be modeled statistically. Shown below is an illustration of one such definition

- Operational Definition: Software defects in production are the number of software defects found in production, categorized by severity 1 and 2 defects. These measures are collected after every release into production and reported every quarter. The metrics are charted quarterly as a bar chart of severity 1 and 2 defects. Improvement is indicated by seeing a reduction in severity 1 and 2 defects over time. Any anomaly in a trend can be used to ask additional questions about the processes that affect this metric and any special cause variation in the process.

4. **Collect, analyze and report the metrics** - The collection process involves collecting and capturing measurement data, analyzing the quality of measurement, deriving metrics from the measurement data, charting the metrics if required, analyzing the metrics, reporting the metrics and analysis to the intended audience, and using the metrics to monitor and manage against the set goals.

- Continuing the example above, defects found in production can be queried from the organization's defect management system or business intelligence information store. These measures can then be charted every quarter manually or automated post-collection. If the trend of quarterly defects is going down, then it can be implied that the quality processes being employed in the development and test lifecycle are being effective. If the trend is going up, one would need to further analyze the defects, identify root causes, and then drive the improvements. Such a chart can be part of a balanced scorecard or management dashboard used to communicate results to the organization's management team.

5. **Review and refine the program periodically** - As with any approach, as the metrics program matures, or as the organization's quality needs change, the program and its constituents would have to be refined.

- Continuing the example above, quarterly reporting of defect metrics in production may not be frequent enough to implement improvements quickly. So a refinement would be to chart these metrics monthly.

The old management adage, 'if you can't measure it, you can't manage it,' is the very reason a metrics program is key to managing either processes or projects, or managing quality improvements. Organization behavior many a time models the outcomes desired from metrics rather than working to achieve the overall goals. So, it is crucial to have a balance of complementary metrics to encourage the right behavior. For instance, let's presume reducing customer issues is a quality performance metric. If reduction of issue backlog is the only metric, then managers can increase resources to reduce the backlog thereby addressing the symptom only. If reducing or maintaining cost of quality is also a performance metric, then increasing resources will increase the cost. It will then encourage managers to look at other ways to reduce backlog, one of which could be to reduce defects that are released to production.

Case Study: Quality transformation at a financial services company

This case study is based on the transformation of a quality assurance organization and the quality function within a financial services company over a three-year span. The broad objective was to improve the capabilities of the quality organization while increasing effectiveness and efficiency.

The following were key challenges seen initially:

- Inconsistent delivery and varying quality assurance capabilities
- Increasing trend of QA cost and cycle time supporting the organization growth
- Low initial quality and issues identified late in the lifecycle or production
- Unclear value-add from the QA function.

The goals for the three year transformation were around the following themes – Efficiency, Effectiveness and Capabilities. Based on the goals and the associated questions about how to achieve the goals, key practices and metrics were defined to monitor and report progress and manage the quality transformation. The goals, themes and metrics are shown in table 2 below.

Goals	Themes	Metrics
Establish a global organization standardized on process and technology	**Capabilities**	• *Number of people trained in quality related practices* • *Percent of projects employing appropriate QA roles* • *Percent of applications whose tests and defects are stored in a single system*
Labor growth for QA should be 50% less than growth of the development team	**Efficiency**	• *Percent QA team growth compared to percent of development team growth*
Reduce cost of quality by 10% while supporting business growth	**Efficiency**	• *Cost of quality as a percent of project cost*
Reduce test cycle time by 10% without affecting quality	**Efficiency**	• *Test cycle time as a percent of project cycle time*
Improve quality of releases to production	**Effectiveness**	• *High severity defects in production over time* • *New software defects in production over time* • *Ninety-day defect containment* • *First-time pass rate*

Table 2: Goals – Themes – Metrics

The initiatives and accomplishments, guided by metrics, are summarized below.

Capabilities - Initiatives implemented were along the dimensions of people, process and technology.

> ***People****: Established software engineering and QA architecture functions to build and implement consistent quality practices. The QA architect role was employed across large projects to help strategize testing across multiple releases and improve quality delivered into production. Implemented a training program aimed at lifting the verification and validation capabilities of the organization.*
>
> ***Process****: Established informal SLAs relative to QA, deployed risk based assurance and test practices, and peer reviews tailored by project size, duration and complexity.*
>
> ***Technology****: As an organization, standardized on HP Quality Center for test management and IBM ClearQuest for defect management.*

In the three year span from 2006 to 2008, the accomplishments as reported using the defined metrics were the following:

- Over 800 people were trained in quality related practices
- Percentage of projects employing appropriate QA roles increased from 40% to 80%
- Tests for core applications were stored in Quality Center and defects for all applications were managed in ClearQuest.

Efficiency - Initiatives implemented were in the areas of distributed operations, organization design, risk-based testing and automation.

> ***Distributed Operations****: In addition to consolidating locations, team growth was limited to two global locations. One offered the best mix of talent around distributed technologies, the other around mainframe technologies while both lowered the cost of operations.*
>
> ***Organization Design****: Designed an organization with QA managers engaged as business/technology relationship managers, QA specialists as business domain experts, and with QA practitioners having multiple-application knowledge. Optimized management span of control by flattening the team structure and employed a role-based engagement model.*
>
> ***Risk Based Test Management****: Implemented risk based test planning and execution to determine priority and sequence of the QA and test schedule, and excelled at delivering against aggressive time constraints.*
>
> ***Automation****: Achieved automation on 30% of the core appli-*

cations, by focusing on marquee features, multi-year legacy application regression and test data automation.

The accomplishments as reported using the defined metrics were the following:

- QA team grew by only 15% supporting a 35% development team growth
- Cost of quality in 2008 reduced by up to 33% - between 20% and 25% of project cost depending on project characteristics, from around 30% in 2006
- Test cycle time in 2008 reduced by up to 28% - between 18% and 20% of project cycle time depending on project characteristics, from around 25% in 2006.

Effectiveness - Initiatives implemented were in the areas of business acumen, end-to-end testing, peer reviews and development testing.

***Business Acumen**: Chartered QA specialists to understand the key processes and products for each business unit, and how technology enabled or supported these processes. By imparting this knowledge across the teams, we ensured that the QA team focused on the features and scenarios that mattered the most to the business.*

***End-to-end Test Strategy**: Introduced risk based testing as a QA strategy and discipline to clearly understand the failure impact on core business features and requirements, and the likelihood of finding high severity defects in production. Implemented practices that ensured that core business processes and end-to-end user scenarios were system tested prior to business acceptance, which helped identify major system integration issues early in the lifecycle.*

***Peer Reviews and Development Testing**: Increased the use of peer reviews to contain defects in design and code, and automated parts of unit and build testing to identify defects before release to test.*

The accomplishments were as follows:

- High severity defects in production decreased by 25%
- New software defects in production reduced by 35%
- On average, 90-day defect containment was between 90% and 95%, up from around 80%
- First-time pass rate improved by 20%.

Value-add from the QA function - A balanced quality scorecard approach was used to communicate progress against the transformation objectives. A balanced scorecard is a performance management tool that helps managers and organizations monitor and manage results balanced across four categories – financial, customers, processes and customers.

The balanced quality scorecard format used was a four-quadrant scorecard along the following dimensions – efficiency, effectiveness and capabilities. The top two quadrants focused on efficiency, the bottom left on effectiveness and the bottom right on capabilities. This format was used to periodically communicate progress and results to the CIO and senior leadership team. In addition, project metrics were communicated consistently, focused on coverage against requirements and solution quality. This type of scorecard is a very effective format to showcase value-add with simplicity and clarity. It is balanced by the themes important to the organization and the metrics show how the organization is performing along those themes, supporting the organization's goals. An illustrative balanced scorecard is shown in figure 4 below. In each of the charts, dev means development, SIT is QA testing and UAT is user acceptance testing.

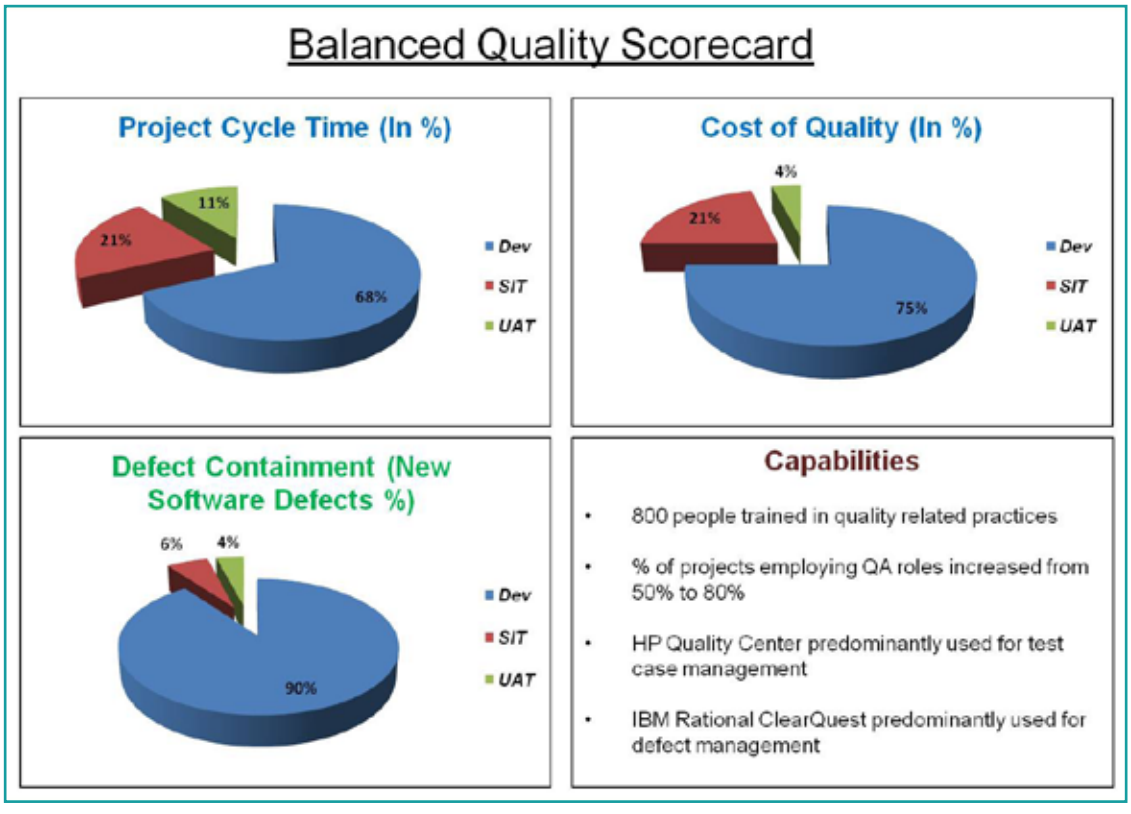


Figure 4: Balanced Quality Scorecard

Summary - Over a three year span, quality transformation at this financial services company led to significant cost avoidance and savings. An organized and deliberate approach - of aligning goals, defining metrics, implementing them to monitor progress and make decisions over time, and using them to share accomplishments – was a key enabler to this transformation.

In this article, we presented an overview of a metrics taxonomy. We outlined a five-step metrics program with an illustrative thread and shared a case study showcasing how a well-implemented metrics program helped guide initiatives through an organization's quality transformation.

To establish a metrics program, start by understanding the business goals and the business context. Ask pertinent questions about how you would know if the goals will be achieved. Identify themes and metrics that will help answer these questions. Define the metrics in an operational context and implement them. Use the metrics as indicators and as an analysis and decision guide as initiatives are executed to achieve the business goals, and to periodically report accomplishments. Remember to balance the metrics so as to not encourage unintended behavior in the organization. Keep the metrics program simple and use stakeholder feedback and inputs to refine and improve the metrics over time.

Biography

Harish Narayan is currently a director of technology at Vistaprint, focused on quality assurance and automation. He is a software engineering and information technology leader and has worked in the field since 1995. He has experience building and leading world-class quality functions, and championing enterprise-wide process and capability improvements. Harish has led and been part of technology transformations in multiple companies. He is passionate about instilling a culture of quality in organizations and has been a proven business partner in diverse industries including financial services, telecommunications, enterprise software and consumer products. He also brings years of strategic planning, global operations, project management, performance management and team-building skills and experience.

With a quarter-century of software and systems engineering experience, Rex Black is President of RBCS (www.rbcs-us.com), a leader in software, hardware, and systems testing. For over fifteen years, RBCS has delivered software and hardware testing services in consulting, outsourcing and training. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups and hires testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to start-ups, RBCS clients save time and money through improved product development, decreased tech support calls, improved corporate reputation and more. As the leader of RBCS, Rex is the most prolific author practicing in the field of software testing today. His popular first book, Managing the Testing Process, has sold over 35,000 copies around the world, including Japanese, Chinese, and Indian releases, and is now in its third edition. His five other books on testing, Advanced Software Testing: Volume I, Advanced Software Testing: Volume II, Critical Testing Processes, Foundations of Software Testing, and Pragmatic Software Testing, have also sold tens of thousands of copies, including Hebrew, Indian, Chinese, Japanese and Russian editions. He has written over thirty articles, presented hundreds of papers, workshops, and seminars, and given about fifty keynotes and other speeches at conferences and events around the world. Rex is the immediate past President of the International Software Testing Qualifications Board and a Director of the American Software Testing Qualifications Board.

Transforming Quality Metrics into Strategic Business Value

by David Gehringer

Many network device manufacturers collect testing metrics to better gauge their efficiency and identify areas for improvement. But tracking metrics such as the speed and number of tests run often doesn't tell the whole story. That is, it doesn't tell us whether the QA team will be able to meet its quality objectives by the ship date.

Faced with an increasingly competitive marketplace, network device manufacturers are under pressure to deliver the highest-quality products to market first. By analyzing the right quality process metrics, today's QA teams can predict whether they will meet their time-to-market goals. More important, they can provide executives with greater visibility, allowing them to make smarter, more strategic business decisions.

Putting Quality Metrics in Context

Looking at individual testing metrics, such as the number of automated tests, pass/fail rates, or bug severity, will not directly tell QA teams whether quality is improving—or whether they will meet their shipping deadlines. Instead of focusing on a few isolated process metrics, QA teams need to track the level of quality throughout the product development life cycle. This involves taking a holistic approach and analyzing how numerous process metrics relate to one another in the overall quality cycle.

For example, after measuring how many bugs are found/fixed each day, QA teams should also look at the found/fixed rate in the context of the severity of the bugs and the number of tests run. Without the right context, these metrics tell only half the story. QA teams might think that their find/fix rate is improving only to realize a week later that the number of tests executing was cut in half.

Thus, each metric measured and reported within the right context can provide insight into actual product quality throughout the process. Still, this is only the first step.

Determining the Projected Ship Date

Whether employing an agile or waterfall testing model, QA managers need a way to take process metrics and put them in the context of product quality and normal convergence. That way, they can determine whether the ship date is realistic. QA teams can accomplish this in three easy steps.

First, using an automated testing tool with robust reporting capabilities, QA teams should gauge their current quality metric. They can do this by tracking their normal distribution bugs (at the end of each release) against the rate at which the bugs can be fixed. Second, they need to determine how many bugs they can fix per week before and after code completion. Third, they need to establish a working convergence model. To build this model, the QA manager must examine the history of quality and document his insights. Next, he and the engineering manager need to adjust the model until they agree that it is 70 to 80 percent accurate (nothing will be 100 percent). This model will be used as a predictor for each subsequent release and refined over time.

By reviewing the current quality metric against the normal convergence curve, QA teams should be able to determine the projected shipping date at the desired level of quality. For example, imagine that a QA manager expects to have 500 normal distribution bugs (70 percent, Priority 1; 20 percent, Priority 2; and 10 percent, Priority 3). Once coding is completed, his team can realistically fix 100 bugs per week until they reach the last 100, which are more difficult and will require three additional weeks. Based on these data points, the QA manager knows that his team will need six weeks to get the product ready to ship.

QA teams can build this curve early in the production cycle and compare the amount of development time remaining with the time required to fix all of their bugs. Based on this information, they can determine well in advance whether the quality goals are compatible with the ship date. The last step is communicating this information to people outside the QA team.

Providing a Strategic Business Tool

Experienced QA managers often know months in advance whether they will miss the ship date based on their analysis of key process metrics. Usually, though, they keep this information to themselves. As a result, executives are seldom aware of these quality and time-to-market conflicts until engineering has missed the ship date—and it's too late to act on this knowledge.

Strong leadership within the QA team is critical to ensuring that

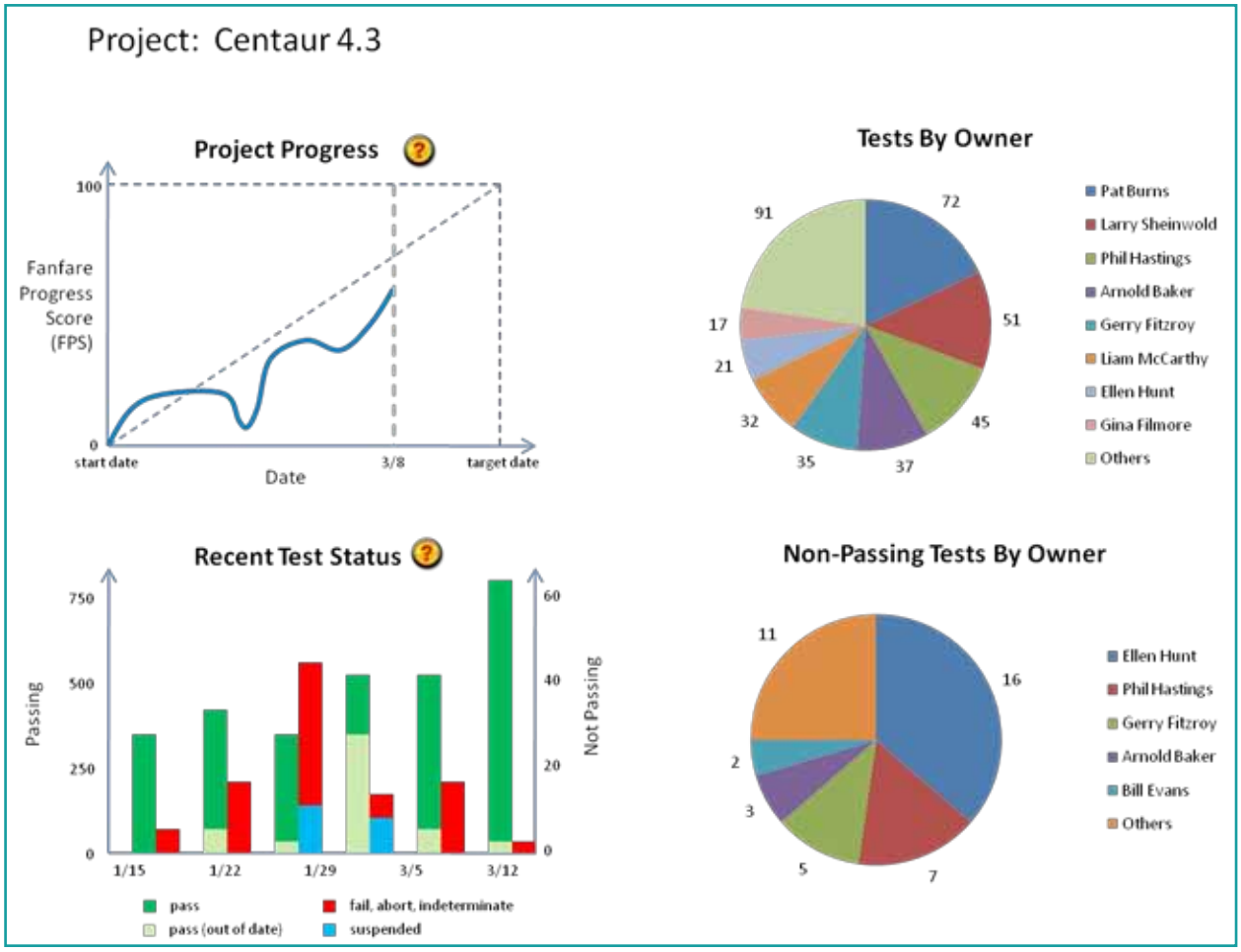


Figure 1.
This dashboard provides a good balance of metrics for measuring product quality and predicting ship dates, as well as the necessary tactics of daily execution and bug fixing.

this information reaches the executive level. Modern automated testing tools can help QA teams document their time-to-market findings, but it takes a confident QA manager to deliver the results to executive management, so they can take preemptive actions to protect and/or enhance the organization's bottom line.

Today, testing plays a crucial role in every product release—accounting for nearly 50 percent of the time to market—yet it often plays second fiddle to development. Soon, however, this might change. By leveraging process metrics, QA teams can offer strategic business value and drive competitive advantage.

Biography

David Gehringer is the Vice President of Marketing at Fanfare Software. He brings over 15 years of marketing and product management experience to this role. David earned bachelor's degrees in mechanical engineering and aeronautical engineering, both from University of California, Davis.

Counting defects

by Bert Wijgers

Defect counts are often considered as measurements of product quality. However, the most important defect count in that respect is by definition unknown; the number of undiscovered errors. Defect counts can be used as indicators of process quality. In doing so, one should avoid assuming simple causal relations. Defect counts can provide useful information but have to be presented with care.

In one of the first projects I joined it was my task to bring the defect process under control. There were hundreds of open defects and they kept coming. At the end of every week I exported the data from the defect management tool and made a few cross-tables in a spreadsheet to show the numbers of defects still open for the next release. We had a weekly meeting to classify and assign new defects and to re-assign old ones. Whenever the workload for the next release became too heavy we would assign some defects to the release after the next. In fact, we spend more than half of the meeting time changing the numbers of the release in which defects ought to ship. As a result, the weekly status report was always optimistic. After a few months my services were no longer needed. I showed someone how to generate the desired reports and the project continued for another year and a half before the plug was pulled and the project had to start all over. In retrospect it is clear to me that things had been going in the wrong direction long before I joined the project and that the weekly status reports hadn't helped one bit to change the course of events. I ease my conscience with the thought that I didn't know then what I know now.

The idea that "you can't control what you can't measure" (DeMarco, 1982) has been abandoned for some time now. DeMarco himself apologizes for overstating the importance of all kinds of software metrics in 1995: "I can only think of one metric that is worth collecting now and forever: defect count. Any organization that fails to track and type defects is running at less than its optimal level. There are many other metrics that are worth collecting for a while."

In this article I want to focus on the use of defect counts. Therefore we need to realize what they are and what they are not. We also have to realize why we do and do not need them. Only then defect counts can be used effectively in decision making.

What a defect count is and what it is not

A defect count is a number of defects that have been discovered. In order to be included in a count a defect has to be logged and classified. The number of severe and still open defects caused by specification errors and found during system test is an example of a specific defect count that might be of interest to somebody.

What does a defect count measure? Things are not always what they seem to be. Kaner and Bond (1994) call this "construct validity". This is a well known concept in social sciences. Does an IQ test really measure intelligence or does it only measure the ability to do IQ tests? Software development is a social activity and that implies lots of variables that affect defect counts.

Let's take the example of defects found per week. Typically, in the first weeks of a software development project only few defects are found. Often this is because the test environment is not yet up and running and the test team is preparing itself. Then the test team gets going and there is a peak. After a while the number of defects found per week diminishes. Often this is taken as evidence that there are only a few defects left to discover in the software and that the product is therefore ready for release. This is not valid. Just like there were other causes for the low defect counts in the first weeks there are other causes for the fact that defect counts drop in the last weeks.

Just before a deadline the focus of the test team shifts from bug hunting to other tasks. The testers have to write their reports and conserve test scripts. They elaborate on old defects to help developers fix these. In fact, they have to shift their focus because defect counts are supposed to drop when the deadline is nearing. Bug hunting is discouraged at these stages of the software development process. The number of defects per week is at least strongly influenced by the focus of the test team.

Counting defects is not the same as measuring product quality. There are a lot of definitions of quality but as Kaner (2000) points out: "The essence of 'quality' is 'qualitative.'" This can never be captured by objective quantitative measures. Quality involves the satisfaction of the user. This is something else than the absence of defects. Even if we define software quality as the absence of defects after release, the number and nature of the defects that

occur during the production phase of the software can only be indicative. The defects that are still hidden in the software might or might not surface after shipment. At the moment of release their number is and will always be unknown. The only thing that defect counts directly measure is the amount of defects in the defect management database.

When the user is taken into account, measurement of quality has to be subjective in nature. Therefore, it is not advisable to formulate acceptance criteria in terms of defect counts. Obviously, a piece of software cannot go live with one or more showstoppers, but the number of allowable minor defects cannot be agreed upon in advance since there is no telling how the actual unsolved defects will influence the behavior of the system in interaction with actual users.

Roughly the same metric that is used for measuring product quality is used for measuring the quality of testing. The number of defects found per week or per working day is sometimes used to evaluate the performance of individual testers or testing teams. As pointed out earlier defect counts are influenced by the dynamics of the software production process. Undoubtedly, motivation and testing skills play an important role as well. But so do product quality, testing strategy and a lot of other things. The dangers associated with the use of defect counts in evaluating testers and their teams will be addressed in the next section.

Why we do and don't need defect counts

Defect counts can be very useful in telling us something about the software production process. They can be compared between phases or timely intervals (weeks, months) in order to reveal tendencies in the process. Defects can be counted separately for different modules of the same system. This information can be used to allocate resources to certain modules or phases. That is, defect counts can be used as management information. In doing so, there are a few pitfalls to be avoided.

When using defect counts to evaluate individuals or teams, this should only be done for coaching purposes. When people notice that they are being judged and rewarded on the basis of the numbers of defects they find, their main focus will be on improving these numbers. Testers might be tempted to log different variations of the same defect. If one logged defect is enough to expose a problem and get it solved then there is no need to log additional defects that are caused by the same problem. Testers might be discouraged to look for more complex defects or for defects in more stable areas of the system. It takes more time to look deeper, but the defects that can be found there might well be more significant. Testers might be discouraged to work as a team and to work on process optimization. By improving their individual defect counts they will diminish the overall quality of the service provided, individually and as a team. Therefore, defect counts should never be considered as goals in themselves.

When management doesn't act upon the metrics provided from the defect management database there is no use in providing the metrics in the first place. Far too often the tables and charts with defect information are only used to talk about in the weekly meetings. The Goal Question Metric approach (Basili and others, 1994) provides a framework to define defect counts that will be used. By clarifying the goals of the project and the questions that should be answered defect information can be presented in a way that satisfies the true needs of management.

The first step is to identify the goals. Every goal has a purpose, an issue, and an object. The fourth element of a goal, the viewpoint, doesn't play a role in defect counts, since these are objective metrics that do not depend on point of view. An object may be a product, a process or a resource. An example of a goal is: To improve (purpose) the efficiency (issue) of testing (object). Take your time to identify the goals and questions at the appropriate level. Usually that is the people that you will be sending your reports to. Aim as high as you can. When deadlines are getting close, defect reports go way up in the organization. In order to make these reports you need to identify these high level goals. The next step is to pose the questions that have to be answered. For example: Are the logged defects valid? Or: How many defects are found? The final step is to define the metrics. For example: the number of valid defects divided by the number of logged defects. Or: the number of defects found per testing day. There is a number of ways to come up with these goals, questions and metrics but every which way, they should be verified regularly. Goals and questions can change. If so, the metrics probably have to change as well.

Defect counts can be used to guide resource allocation. When there is a disproportional amount of defects found in a certain subsystem then it might be wise to assign an extra or a more experienced developer to that subsystem. However, Fenton and Neil (1999) point out that a high defect density, that is a high number of defects per line of code, is probably a sign of good testing rather than a sign of poor quality. So it might be wise to assign extra testers when there is a low defect density. Defect counts are only indicators of process quality. They reveal correlations, not causal relations.

Defect counts can be helpful in planning future projects. Let's take the example of the number of defects per injection phase. Generally speaking, the longer a defect remains undiscovered the more it will cost to fix it. Testing effort can be allocated according to the observed distribution for a similar product. If lots of defects were injected in the design phase, then extensive reviewing of design documents can be planned.

Management information

There are endless possibilities to define defect counts and they can all be useful under certain circumstances. It is our responsibility to provide the numbers that really matter. When managers ask for information from the defect management database, make them sweat. The Goal Question Metric approach is a good way to get commitment from managers. They will be forced to think about what they really need to know and in the process they will gain insight in the complexity behind the numbers. What is true for most things in life is true for defect counts; what is given too easily is often not fully appreciated. Managers have to realize that defect counts are not for free. It takes time and effort to define and present the right metrics and they should be valued accordingly. Only then defect counts can become management information.

References

Basili, V.R., Caldiera, G and Rombach, H.D., 1994. The Goal Question Metric Approach, in Marciniak, J.J. (ed.), Encyclopedia of Software Engineering, volume 1, pages 528-532. John Wiley & Sons.

DeMarco, T., 1982. Controlling Software Projects: Management, Measurement, and Estimates. Yourdon Press, New York.

DeMarco, T., 1995. Mad About Measurement, in Why Does Software Cost So Much?, pages 14-44. Dorset House, New York.

Fenton, N.E. and Neil, M., 1999. Software metrics: successes, failures and new directions. Journal of Systems and Software 47(2-3), pages 149-157.

Kaner, C., 2000. Rethinking software metrics. Software Testing & Quality Engineering, Volume 2, Issue 2.

Kaner, C. and Bond, W.P., 2004. Software Engineering Metrics: What Do They Measure and How Do We Know? 10th International Software Metrics Symposium.

Biography

Bert Wijgers is a test consultant with Squerist, a service company in the Netherlands. Squerist focuses on software quality assurance and process optimization. Its core values are innovation, inspiration and confidence.

Bert holds a university degree in experimental psychology for which he did research in the areas of human-computer interaction and ergonomic aspects of the workspace. He has worked as a teacher and trainer in different settings before he started an international company in web hosting and design for which a web generator was developed. There he got the first taste of testing and came to understand the importance of software quality assurance. Bert has a special interest for and the social aspects of software development.
He uses psychological and business perspectives to complement his testing expertise.

In recent years Bert has worked for Squerist in financial and public organizations as a software tester, coordinator and consultant. In these assignments he has made extensive use of metrics, notably defect counts.

Testing: 'What do we really know?'

Column

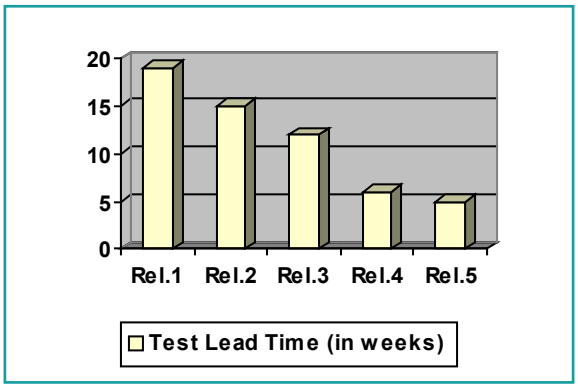
by Erik van Veenendaal

State-of-the-practice

The focus of this issue of Testing Experience is on test metrics. A topic within the testing discipline that is still in its infancy. Many papers (and even books) have been written addressing test metrics, but (too) few organizations have really practiced test measurement programs. Beware, I'm not discussing metrics in the context of running a test project, but metrics in the context of a test process or even test improvement project. If you think about it, it's amazing how much we do not know regarding testing. Many decisions about starting (and thus investing time and money!) to use a new method or technique are taken based on gut feeling and, sadly enough, not on hard factual data. (We are by the way far from being unique in this within the ICT-industry!) Even with relatively straightforward questions, we are most often lost. Examples of such questions being "Which test design technique will guide us in finding most defects (in a certain situation)?", "What is a reasonable level of defect finding effectiveness?", "What are quantitatively measured benefits when acquiring a test management tool?", or "How many testers per developer in a SCRUM team get the job done?" Of course, measurement is far from easy, and interpreting data is even harder, but wouldn't it be nice if we had at least some clues.

Part of test process improvement

Nowadays many test professionals are involved in test process improvement based on either TPI (Next) or TMMi. I often encounter so-called success stories of testers that tell me the great things they have achieved; „We are now doing risk-based testing, we have implemented CTM, etc". When I ask them for concrete data to convince management for long-term commitment, they often go quiet. What contribution does test process improvement provide towards achieving business objectives, and does it really have a return-on-investment? That it can be measured is shown by the graph where an organization had reduction of time-to-market as its major business driver for improving the testing process (using TMMi). The graph clearly shows a reduction of the lead time for the test execution phase for the various releases over time.



Test measurement programs are (too) difficult

Full-blown test measurement programs are too far off for most organizations. Remember measurement programs are an investment, too! At TMMi the test measurement process area is at level 4; most of us are not even close yet. Only when you are mature enough, when you have a stable and defined process, when there is real management commitment towards measurement, can a full test measurement program be successful. Most organizations that I have seen trying to implement a test measurement program do not meet the constraints and therefore fail sooner or later.

Does that mean we cannot and should not do anything on measurement? I largely disagree! Of course, it is more difficult if you are a level 1 organization, of course data is less reliable, but when you want to get a management buy-in for testing and test process improvement, you shall convince by showing metrics they understand. Initially you need to find someone who is a believer; someone who will support you in starting test process improvement. However, after a while this supporting manager (or his boss) wants to know what you are doing and achieving with all the effort and money that is being put in.

Getting started

So how do we start, what can we do? I like to keep things simple, because making things too difficult will surely lead to failure. Perhaps you have heard the saying "Less is more". Here is my 5-step light-weight test measurement process:

1. *Identify business objectives* – Test process improvement does not have self-fulfilling reasons. There needs to be a business problem why we are doing it. Are our customers complaining about the level of quality? Are we missing a market window because testing takes too long? We must have a very clear understanding what the business *problem is that we are targeting.*

2. *Define test performance indicators* – Related to the business objectives, define two or three (maximum!!) metrics that we will use to show management that we are contributing to business objectives. Use metrics (performance indicators) that are easy to understand for management and can be shown in a simple graph. For example, if the business problem is product quality then the Defect Detection Percentage (DDP) would be an ideal candidate as a test performance indicator.

3. *Establish a data gathering template* – Of course, one needs to gather the base data that is required to calculate the test performance indicators/metrics. Notice, I do not mention the term process or procedure. Again, start low profile! If test projects already have a test project closure phase in place, then it is easy to link the template to a test evaluation report (template). Keep the data to be submitted to the minimum level possible. Recently, I was provided a highly sophisticated 5-page Excel sheet to submit data. After trying for 30 minutes, I gave up.....

4. *Analyse and discuss draft performance indicators* – Since you're doing this for the first time and data can easily be incorrect, be very careful. In measurement what you see is often not what you get..... It is recommended to perform data integrity checks as close to the source of the data as possible. Checks can include scans for missing data, out-of-bounds data values, and unusual patterns and correlation across measures. It is also appropriate to review initial interpretations of the results and the way in which they are presented before disseminating and communicating them more widely. Reviewing the initial results before their release may prevent needless misunderstandings and lead to improvements in the data analysis and communication.

5. *Present results to management* - and to all other stakeholders and interested parties. Now the time has come to show the added business value of test process improvement. Keep relevant stakeholders informed of results on a timely basis and assist them in understanding the results. Discuss the result in so-called feedback sessions. Be open, create awareness, and talk about testing. This is when you can achieve management buy-in. If possible make an A3 print and show it at coffee corners. Of course, corrective and improvement actions can be defined based on the analyzed results.

Benchmarking

Of course, this is not the perfect or most reliable way to gather data and metrics, but 15 years after the first release of TMap, 10 years after the start of the ISTQB certification program, isn't it about time we start some measurements program, and preferably in a practical and feasible manner. After all, we are (very) slowly getting more mature. We would all benefit if we shared our results openly. It will allow benchmarking, which is something that we are all looking for, but is still lacking. I challenge you to start with some simple metrics, learn, create management commitment, and publish the results openly. Thanks on behalf of the testing community!

Erik van Veenendaal is a leading international consultant and trainer, and recognized expert is the area of software testing and quality management. He is the director of Improve Quality Services BV. At EuroStar 1999, 2002 and 2005, he was awarded the best tutorial presentation. In 2007 he received the European Testing Excellence Award for his contribution to the testing profession over the years. He has been working as a test manager and consultant in software quality for almost 20 years.
He has written numerous papers and a number of books, including "The Testing Practitioner", "ISTQB Foundations of Software Testing" and "Testing according to TMap". Erik is also a former part-time senior lecturer at the Eindhoven University of Technology, the vice-president of the International Software Testing Qualifications Board and the vice chair of the TMMi Foundation.

Probador Certificado Nivel Básico

Tester profesional de Software

Formación para el Probador Certificado - Nivel Básico
de acuerdo al programa de estudios del ISTQBª

República Argentina

Docente: Sergio Emanuel Cusmai

Co - Founder and QA Manager en QAustral S.A.
Gte. Gral. de Nimbuzz Argentina S.A.
Docente en diplomatura de Testing de Software de UTN - 2009.
Titular y Creador de la Diplomatura en Testing de Software de la UES XXI - 2007 y 2008.
(Primer diplomatura de testing avalada por el ministerio de Educación de Argentina).
Team Leader en Lastminute.com de Reino Unido en 2004/2006.
Premio a la mejor performance en Lastminute.com 2004.
Foundation Certificate in Software Testing by BCS - ISTQB. London – UK.
Nasper - Harvard Business school. Delhi – India.

Casa Central: + 54 0351 4817951 - **Tel Bs As Argentina:** 011 53685847 - **Tel Londres UK:** 00442033183814
Tel San Francisco. EEUU: 001 14157041515 - **E-mail: info@qaustral.com**

What should we measure during testing?

by Yogesh Singh & Ruchika Malhotra

"What cannot be measured, cannot be controlled" is a reality in this world. If we want to control something, we should first be able to measure it. Therefore, everything should be measurable. If a thing is not measurable, we should make an effort to make it measurable. The area of measurement is very important in every field, and we have mature and established metrics to quantify various things. However, in software engineering this "area of measurement" is still in its developing stage and may require significant effort to make it mature, scientific and effective.

Software metrics are applicable in all phases of the software development life cycle. At the requirements and analysis phase, where the output is the SRS document, we may have to estimate the cost, manpower requirement and development time for the software. The customer may want to know the cost of the software and development time before signing the contract. As we all know, the SRS document acts as a contract between customer and developer. The readability and effectiveness of the SRS document may help to increase the confidence level of the customer and may provide a better foundation for designing the product. Some metrics are for cost and size estimation like COCOMO, Putnam resource allocation model, function point estimation model etc, whilst others relate to the SRS document, like number of mistakes found during verification, change request frequency, readability etc. In the design phase, we may like to measure stability of a design, coupling amongst modules, cohesion of a module etc. We may also like to measure the amount of data that is input to a software, processed by the software and also produced by the software. A count of the amount of data input to, processed in, and output from software is called a data structure metric. Many such metrics are available, like number of variables, number of operators, number of operands, number of live variables, variable spans, module weakness etc. Some information flow metrics are also popular like FAN IN, FAN OUT etc.

Use cases may also be used to design metrics like counting actors, counting use cases, counting number of links etc. Some metrics may be designed for various applications of websites, like number of static web pages, number of dynamic web pages, number of internal page links, word count, number of static and dynamic content objects, time taken to search a web page and retrieve the desired information, similarity of web pages etc. Software metrics have number of applications during the implementation phase and after the completion of such a phase. Halstead software size measures are applicable after coding, like token count, program length, program volume, program level, difficulty, estimation of time and effort, language level etc. Some complexity measures are also popular, like cyclomatic complexity, knot count, feature count etc. Software metrics have found good number of applications during testing. One area is the reliability estimation, where popular models are Musa's basic execution time model and the Logarithmic Poisson execution time model. Source code coverage metrics are available that calculate the percentage of source code covered during testing. Test suite effectiveness may also be measured. Number of failures experienced per unit of time, number of paths, number of independent paths, number of "define-use" paths, percentage of statement coverage, percentage of branch conditions covered are also useful software metrics. The maintenance phase may have many metrics, like the number of faults reported per year, number of requests for changes per year, percentage of source code modified per year, percentage of obsolete source code per year etc.

We may find a number of applications of software metrics in every phase of the software development life cycle. They provide meaningful and timely information, which may help us to take corrective actions as and when required. Effective implementation of metrics may improve the quality of the software and may help us to deliver the software in time and within budget.

1. Categories of Metrics

There are two broad categories of software metrics, namely product metrics and process metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc. Process metrics describe the effectiveness and quality of the processes that produce the software product. Examples are effort required in the process, time to produce the product, effectiveness of defect removal during development, number of defects found during testing, maturity of the process.

1. Product metrics for testing

These metrics provide information about the testing status of a software product. The data for such metrics are also generated during testing and may help us to know the quality of the product. Some of the basic metrics are given as:

(i.) Number of failures experienced in a time interval
(ii.) Time interval between failures
(iii.) Cumulative failures experienced up to a specified time
(iv.) Time of failure
(v.) Estimated time for testing
(vi.) Actual testing time

With these basic metrics, we may find some additional metrics as given below:

(i.) $$\% \text{ of time spent} = \frac{\text{Actual time spent}}{\text{Estimated testing time}} \times 100$$

(ii.) Average time interval between failures
(iii.) Maximum and minimum failures experienced in any time interval
(iv.) Average number of failures experienced in time intervals
(v.) Time remaining to complete the testing

We may design similar metrics to find the indications about the quality of the product.

1.2 Process metrics for testing

These metrics are developed to monitor the progress of testing, status of design and development of test cases and outcome of test cases after execution.

Some of the basic process metrics are given below:

(i.) Number of test cases designed
(ii.) Number of test cases executed
(iii.) Number of test cases passed
(iv.) Number of test cases failed
(v.) Test case execution time
(vi.) Total execution time
(vii.) Time spent for the development of a test case
(viii.) Total time spent for the development of all test cases

On the basis of the above direct measures, we may design the following additional metrics, which may convert the base metric data into more useful information.

(i.) % of test cases executed
(ii.) % of test cases passed
(iii.) % of test cases failed
(iv.) Total actual execution time / total estimated execution time
(v.) Average execution time of a test case

These metrics, although simple, may help us to know the progress of testing and may provide meaningful information to the testers and project manager.

An effective test plan may force us to capture data and convert it into useful metrics both for process and product. This document also guides the organization for future projects and may also suggest changes in the existing processes in order to produce a good-quality maintainable software product.

2. Testing metrics

Testing metrics may help us to measure the current performance of any project. The collected data may become historical data for future projects. This data is very important because in the absence of historical data, all estimates are just the guesses. Hence, it is essential to record the key information about the current projects. Test metrics may become an important indicator of the effectiveness and efficiency of a software testing process and may also identify risky areas that may need more testing.

2.1 Time

We may measure many things during testing with respect to time and some of them are given as:
1) Time required to run a test case
2) Total time required to run a test suite
3) Time available for testing
4) Time interval between failures
5) Cumulative failures experienced up to a given time
6) Time of failure
7) Failures experienced in a time interval

A test case requires some time for its execution. A measurement of this time may help to estimate the total time required to execute a test suite. This is the simplest metric and may help to estimate the testing effort. We may calculate the time available for testing at any point in time during testing, if we know the total time assigned for testing. Generally, the unit of time is seconds, minutes or hours, per test case. Total testing time may be defined in terms of hours. Time needed to execute a planned test suite may also be defined in terms of hours.

The failure pattern may help us to define the following:
1) Time taken to experience 'n' failures
2) Number of failures in a particular time interval
3) Total number of failures experienced after a specified time
4) Maximum / minimum number of failures experienced in any regular time interval.

2.2 Quality of source code

We may know the quality of the delivered source code after a reasonable# time of release using the following formula:

$$QSC = \frac{WD_B + WD_A}{S}$$

Where
WDB: Number of weighted defects found before release
WDA: Number of weighted defects found after release
S: Size of the source code in terms of KLOC.

The weight for each defect is defined on the basis of defect severity and removal cost. A severity is assigned to each defect by testers based on how important or serious the defect is. Lower values of this metric indicate fewer errors detected or the detection of less serious errors.

We may also calculate the number of defects per execution test case. This may also be used as an indicator of source code quality, as the source code progressed through the series of test activities.

2.3 Source code coverage

We may like to execute every statement of a program at least once before its release to the customer. Hence, the percentage of source code coverage may be calculated as:

$$\% \text{ of source code coverage} = \frac{\text{Number of statements of a source code covered by test suite}}{\text{Number of statements of a source code}} \times 100$$

Higher values of this metric give confidence about the effectiveness of a test suite. We should write additional test cases to cover the uncovered portions of the source code.

2.4 Test case defect density

This metric may help us to know the efficiency and effectiveness of our test cases.

$$\text{Test case defect density} = \frac{\text{Number of failed tests}}{\text{Number of executed test cases}} \times 100$$

Where Failed test case: A test case that, when executed, produced an undesired output.
Passed test case: A test case that, when executed, produced a desired output.

Higher values of this metric indicate that the test cases are effective and efficient, because they are able to detect more number of defects.

2.5 Review efficiency

Review efficiency is a metric that gives insight into the quality of the review process carried out during verification.

$$\text{Review} = \frac{\text{Total number of defects found during review}}{\text{Total number of project defects}} \times 100$$

The higher the value of this metric, the better is the review efficiency.

3. Conclusion

There are many schools of thought about the usefulness and applications of software metrics. However, every school of thought accepts the old quote of software engineering that says "You cannot improve what you cannot measure; and you cannot control what you cannot measure". In order to control and improve various activities, we should have "something" to measure such activities. This "something" differs from one school of thought to another. Despite different views, most of us feel that software metrics help to improve productivity and quality. Software process metrics are widely used, such as in the capability maturity model for software (CMM-SW) and in ISO9001. Organizations put serious effort into implementing these metrics.

Biography

Yogesh Singh
He is a professor at the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. He is also Controller of Examinations at the Guru Gobind Singh Indraprastha University, Delhi, India. He was founder Head (1999-2001) and Dean (2001-2006) of the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. He received his master's degree and doctorate from the National Institute of Technology, Kurukshetra, India. His research interests include software engineering focusing on planning, testing, metrics, and neural networks. He is coauthor of a book on software engineering, and is a Fellow of IETE and member of IEEE. He has more than 200 publications in international and national journals and conferences. Singh can be contacted by e-mail at ys66@rediffmail.com.

Ruchika Malhotra
She is an assistant professor at the Department of Software Engineering, Delhi Technological University (formerly known as Delhi College of Engineering), Delhi, India, She was an assistant professor at the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Prior to joining the school, she worked as full-time research scholar and received a doctoral research fellowship from the University School of Information Technology, Guru Gobind Singh Indraprastha Delhi, India. She received her master's and doctorate degree in software engineering from the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Her research interests are in software testing, improving software quality, statistical and adaptive prediction models, software metrics, neural nets modeling, and the definition and validation of software metrics. She has published more for than 37 research papers in international journals and conferences. Malhotra can be contacted by e-mail at ruchikamalhotra2004@yahoo.com.

ISTQB® Certified Tester-Foundation Level

in Paris, in French

Test&Planet: votre avenir au présent.

Venez suivre le Cours « Testeur Certifié ISTQB – Niveau fondamental » à Paris, en français !

Test&Planet, centre de formation agréé DIF, vous propose ce cours D&H en exclusivité.

Apprenez à mieux choisir Quoi, Quand et Comment tester vos applications. Découvrez les meilleurs techniques et pratiques du test logiciel, pour aboutir à une meilleure qualité logicielle à moindre coût.

Possibilités de cours sur mesure.

Pour vous inscrire ou pour plus d'information: www.testplanet.fr

Test&Planet

Diaz Hilterscheid

Advocating Code Review for Testers

by Chetan Giridhar

According to Wikipedia, "Code review is systematic examination (often as peer review) of computer source code intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills."

Code Review has been a good practice featuring as an important part of the software development life cycle. Not only has the practice helped in finding or rather preventing critical defects early in the cycle (thereby reducing the bug fixing costs), but it has also been instrumental in improving developers' knowledge and build exposure to the other components of the software.

However, have we ever seen QA professionals participating actively in code reviews? With Quality Assurance being a proactive process aimed at preventing defects, wouldn't code review feature in the QA realm as well? Would it make sense to make code reviews an integral part of QA processes? Are there any risks associated with doing so?

This article aims at breaking the myth, or at least triggering a thought process, of why testers couldn't be or shouldn't be an imperative part of code reviews. During the course of this article, the author substantiates the merits of the thought and also throws light on the flip side of it. At the end, the author hopes that the readers (independent of their role in the software industry) give this philosophy sufficient thought and evaluate whether including testers in code reviews would be beneficial as a process and if it would be helpful for their team context or not.

Involve testers in code review!

Historically, code reviews have been done by developers. The general idea has been that it's the developers that write code and it's them who can best review it. Testers were not expected to write code or have good programming knowledge. Coding skills were good to have rather than being desirable. And even if testers were required to code, it was more for automation. A quick program in a scripting language like Perl or VB would do the job.

Of late, the scenario has changed. Industry prefers test engineers competent in development skills. Programming knowledge not only helps testers in developing better designed automation frameworks (that in turn help to reduce testing cycle time), but also adds a development perspective in tests in terms of white-box testing, thus ensuring more coverage. Given the kind of knowledge testers have about code, it makes more sense that testers could or should be involved in code reviews.

The following section will give some reasons for including software testers in the code review sessions conducted by the development team.

Difference in the mindsets of Dev and QA

Development teams often code in a way to make the software workable. They are more bothered that the functionalities work fine. They work with the intent of building the system.

QA, however, has a completely different mindset. QA would think about ways of breaking the system. They would consider all possibilities where the functionalities wouldn't work. The developers may oversee some of the minute mistakes in the code, which a tester may find out.

QA builds product knowledge

During code review, QA would go through the features that are already developed. This will help them gain knowledge about the features from the product's perspective.

Often, QA is only aware of the functionalities to be tested from the requirement specification documents that are sometimes not well documented. Going through an implementation of the requirement would definitely help QA bridge gaps between their perception of the requirement and the actual implementation. This would mean that QA understands the product better which is imperative for testing the product.

Improved test documentation

It is often observed that when QA starts testing, they may find that the tests designed by them do not cover all the scenarios. QA will then have to add more tests to their test design and might even have to change their test plans accordingly. All this has to be done during testing.

If QA were to be involved during code reviews, they would get sufficient ideas on what kind of tests will be required for finding

defects that they might not have been able to find had they not reviewed the code implementation beforehand.

Code reviews can also be seen as an opportunity where QA can get feedback on what type of tests Dev expects for the functionalities implemented. This helps in improving the quality of test planning and test design. Even test automation can be started based on the discussions in code reviews.

Reduced testing time and costs

Traditionally, a frozen requirement specification document was the basis on which Dev and QA would plan their tasks. While the Dev were busy implementing the code, QA would get involved in activities pertaining to test planning, test design, test automation and setting up test environments. Once the implementation was completed, a build was released to QA for formal testing, often known as the RTQA (Release to QA) build.

If QA were to participate in code reviews, they wouldn't have to wait for the RTQA build to get a feel of how the requirements are implemented. This would in turn speed up testing and reduce the overall product life cycle time.

More information from QA to Dev on where the tests fail

Usually, QA starts testing on the RTQA build and finds defects after running their tests. QA logs these defects in a defect tracking system with information on 'Steps to Reproduce' and the 'Actual Results'. When the Dev team receives the defect, they first try to reproduce the bug, and then start debugging where the defect lies.

If QA attends code reviews, they would have information about code changes that have gone into a particular build. When any test fails, intuitively, QA would try to zero on the point where the test failed as they would be aware of the implementation. Getting exact information in terms of the line of code, where the test has failed, saves a lot of time that Dev invests in investigation. This time could instead be used by them to fix similar issues, if any, in any other parts of the software code. A definite win-win situation!

Involvement of QA in Root Cause Analysis (RCA)

When customers observe any defect in the field, these are reported back to the engineering team. Typically, it's the development that is responsible for a root cause analysis as they are aware of the code. If QA participates in code reviews, they would be cognizant of all the functionalities and would have knowledge on how they were implemented. Based on this knowledge, they could assist in performing root cause analysis of defects that have been reported in the field.

Improved quality of code documentation and coding guidelines

The idea here is that developer egos can be used to boost the quality of code.

If QA is involved in code reviews, the developers would know that their code is going to be examined by a group of QA members and they will take extra care while coding.

Developers will include better comments so that they don't have to answer too many questions from the reviewers.

During review, if QA is looking for inconsistencies between comments and code, developers would make sure that code documentation is appropriate and get motivated to follow coding guidelines as per the standards specified by the regulatory authority.

Identifying more scenarios for testing

If testers are involved in code review, they could get a chance to review the code from a non-functional testing point of view.

Security testing

One of the most important test types that is not included in code reviews is security testing. It can often be observed that developers are either not trained in secure coding practices or security is not given much importance. Testers, if involved in code reviews, may find out security flaws in the code. Issues relating to a buffer overflow or memory leaks would get caught early in the development.

Globalization testing

During code review, testers may find scenarios where strings are being operated upon. Testers could find opportunities of additional testing this in the context of globalization testing based on the specifications made in the requirements document.

Aids in white-box testing

Taking part in code review would definitely help testers in improving and adding more tests in white-box testing.

Let's consider a case where a tester was involved in the code review of an API. The API under review accepts values of data type UINT16 (i.e., values ranging from 0 to 2^16 – 1). While reviewing this API, the tester finds an opportunity of trying out boundary value tests for this API during white-box testing. Thus he could write white-box tests that check if the API exists gracefully or how the values (-1, 0, 2^16-1 and 2^16) are handled by the API. This test would stand more chance of finding a defect.

Help for a newly set-up development team

You must have noticed or experienced cases where a new development team is set up to work on a research project, but the QA team has been in place for some time and has good experience of working on similar projects prior to this. Suggestions from senior QA members given to Dev as formal/informal inputs would definitely make developers comfortable in understanding the technologies better that are completely new to them. Code review meetings can be seen as an opportunity for such discussions. This would be a win-win situation for both Dev and QA.

Improving rapport between Dev and QA

Code review meetings would also help in improving rapport between the two teams. As code reviews are helpful for both Dev and QA, they would definitely be beneficial for developing a healthy environment in the teams.

Code review is not for testers!

Do you belong to the group that believes that code review shouldn't be done by testers? Well, you have reasons to say that affirmatively! We have already seen the benefits of involving testers in code review. Obviously, there are cases when you may not consider it as a viable option. Here are some of the cases where this process would not yield results.

Steeper learning curve

Code review for testers may sound like a good idea, but it is imperative to evaluate the development skills of the QA team working on the project. It may take longer for a QA team that is inexperi-

enced in development or white-box testing skills to start understanding the code and then actually review it. Benefits would come eventually, but it may take a considerable amount of time.

Extra effort required by the testing team

If QA is involved in code review, they have to be given more time for testing activities. This may essentially impact QA deadlines and would ultimately lead to project deadline slippages. This situation would not be favorable from the project perspective.

Testing based on implementation rather than specification

If testers get involved in code reviews frequently, it may happen that they start testing the product based on implementation knowledge and not based on the specification. This defeats the whole purpose, as QA starts justifying the behavior of the product based on product knowledge while they should be validating the product based on user specifications.

Code complexity/simplicity decides the testing strategy

Depending on the complexity of the code, testers may change their testing strategy. QA may feel that if it's a simple piece of code, it may not require exhaustive testing. On similar lines, for a complex code snippet, QA may feel it would require lot of testing effort. However, this isn't always true. So this may lead to the development of an incorrect test strategy, which would be considered fatal from the QA and the project perspectives.

Over-involvement of testers understanding complex code

While performing code reviews, testers may get too interested in understanding a complex code snippet. This would mean that a tester spends too much time appreciating the code and spends less time in testing. This change in focus won't help the testing team.

You know what's best for you!

As you would have thought, it would be a good idea if testers are included in code reviews.

Not only do they help QA in improving the product knowledge, but also errors found by the QA team in code reviews can be fixed earlier in the cycle rather than be fixed after a formal build release to QA.

Code reviews also help in improving test documentation and give QA an opportunity to identify more areas in testing based on the knowledge of the code.

Moreover, developers also benefit as they get a chance to get inputs on the products from experienced QA who have worked on similar technologies before. As developers know that the code will be reviewed by professionals less aware of development, they get motivated to write code that is well documented and abide by coding standards.

Even though involving testers in code reviews looks like an attractive proposition, it should be evaluated. Does it suit my needs? Does it solve my purpose? These are some of the questions that need to be answered. It may work marvels, but at other times it could fail.

I would definitely advocate it! It's worth a try!

Biography

Chetan Giridhar works as a Senior Engineer at McAfee Labs, McAfee India and has an experience of around 5 years as an Automation, Manual and White-box Test Engineer.
Chetan is an avid blogger and has a blogspot (http://tediousscripts.blogspot.com/) which he updates with testing tips, useful testing scripts in Perl, Python and testing frameworks that he develops.
He has co-authored a book on 'Design patterns in Python' (http://technobeans.wordpress.com/books/) and written articles about testing tools in collaboration with Rahul Verma.

Agile
TESTING DAYS

Agile Testing Days 2010

4–7 October 2010, Berlin

The **Agile Testing Days** is the European conference for the worldwide professionals involved in the agile world.

We chose Berlin, one of the best-connected capitals in Europe to host this event due to the affordability/service ratio for hotels and flights. Berlin also offers a lot of fabulous sights to visit in your spare time.

Please have a look at the program at **www.agiletestingdays.com** and enjoy the conference!

Agile TESTING DAY

Berlin, Germany

October 4

Tutorials


Lisa Crispin


Janet Gregory


Elisabeth Hendrickson


Stuart Reid


Isabel Evans


Linda Rising


Michael Bolton


Jennitta Andrea


Anko Tijman


Eric Jimmink


Pekka Klärck


Juha Rantanen


Janne Härkönen

October 5

Conference

Day 1	Track 1	Track 2	Track 3	Track 4	Track 5 – Vendor Track
08:00	Registration				
09:25	Opening				
09:30	Keynote: Agile Defect Management – *Lisa Crispin*				
10:30	Incremental Scenario Testing: Beyond Exploratory Testing *Matthias Ratert*	Experiences on test planning practices in Agile mode of operation *Eveliina Vuolli*	Testability and Agility – "Get your quality for nothing and your checks for free" *Mike Scott*	Testing Web Applications in practice using Robot Framework, Selenium and Hudson *Thomas Jaspers*	Continuous Deployment and Agile Testing *Alexander Grosse*
11:30	Break				
11:50	Test Driven Migration of Complex Systems *Dr. Martin Wagner and Thomas Scherm*	Making GUI Testing Agile and Productive *Geoffrey Bache*	Real World Test Driven Development *Emanuele DelBono*	Acceptance Test Driven Development using Robot Framework *Pekka Klärck*	Talk 5.2
12:50	Lunch				
14:20	Keynote: Deception and Estimation: How We Fool Ourselves – *Linda Rising*				
15:20	Test Center and agile testers – a contradiction? *Dr. Erhardt Wunderlich*	Reinvigorate Your Project Retrospectives *Jennitta Andrea*	Error-driven development *Alexandra Imrie*	Flexible testing environments in the cloud *Jonas Hermansson*	Talk 5.3
16:20	Break				
16:40	Solving the puzzles of agile testing *Matthew Steer*	Structures kill testing creativity *Rob Lambert*	Integration Test in agile development *Anne Kramer*	Waterfall to an Agile Testing Culture *Ray Arell*	Talk 5.4
17:40	Keynote: Lessons Learned from 100+ Simulated Agile Transitions – *Elisabeth Hendrickson*				
19:00	Chill Out/Event				

Agile TESTING DAY

Berlin, Germany

October 6

Conference

Day 2	Track 1	Track 2	Track 3	Track 4	Track 5 – Vendor Track
08:00	Registration				
09:25	Opening				
09:30	Keynote: From Checking to Testing: Getting Testers Out of the Quality Assurance Business – *Michael Bolton*				
10:30	Top 10 reasons why teams fail with ATDD, and how to avoid them *Gojko Adzic*	Agile hits Formal – Development meets Testing *Matthias Zieger*	A lucky shot at agile? *Zeger Van Hese*	The Leaner Tester: Providing true Quality Assurance *Hemal Kuntawala*	Talk 5.1
11:30	Break				
11:50	TDD vs BDD: from developers to customers *Alessandro Melchiori*	How can the use of the People Capability Maturity Model® (People CMM®) improve your agile test process? *Cecile Davis*	Mitigating Agile Testing Pitfalls *Anko Tijman*	Maximizing Feedback – On the importance of Feedback in an Agile Life Cycle *Lior Friedman*	Talk 5.2
12:50	Lunch				
14:20	Keynote: Agile Testing Certification – how could that be useful? – *Stuart Reid*				
15:20	Alternative paths for self-education in software testing *Cirilo Wortel*	Hitting a Moving Target – Fixing Quality on Unfixed Scope *David Evans*	Fully Integrated Efficiency Testing in Agile Environment *Ralph van Roosmalen*	The Flexible Art of Managing a Multi-layered Agile team *Shoubhik Sanyal*	How to Brew a Tasty Agile Test Strategy *Dr. Alexander Schwartz*
16:20	Break				
16:40	Mastering Start-up Chaos: Implementing Agile Testing on the Fly *Christiane Philipps*	Building Agile testing culture *Emily Bache and Fredrik Wendt*	Implementing collective test ownership *Eric Jimminik*	Alternative paths for self-education in software testing *Markus Gärtner*	Talk 5.4
17:40	Keynote: About Learning – *Janet Gregory*				
19:00	Chill Out/Event				

October 7

Open Space

Open Space hosted by **Brett L. Schuchert**.

Day 3	Track
08:00	Registration
09:25	Opening
09:30	Keynote – Isabel Evans
10:30	*Open Space*
11:30	Break
11:50	*Open Space*
12:50	Lunch
14:20	Keynote – Jennitta Andrea
15:20	*Open Space*
17:20	Keynote – Tom Gilb
18:20	Closing Session

Exhibitors

Díaz Hilterscheid

testing experience

NOKIA ovi

iSQI International Software Quality Institute

Agile Record

Microsoft Visual Studio Test Professional 2010

c:t:co Dynamic World. Stable Solutions.

Please fax this form to +49 (0)30 74 76 28 99
or go to http://www.agiletestingdays.com/onlinereg.html.

Agile TESTING DAY
Berlin, Germany

Participant

Company: _____
First Name: _____
Last Name: _____
Street: _____
Post Code: _____
City, State: _____
Country: _____
Phone/Fax: _____
E-mail: _____

Billing Address (if different from participant)

Company: _____
First Name: _____
Last Name: _____
Street: _____
Post Code: _____
City, State: _____
Country: _____
Phone/Fax: _____
E-mail: _____

Remarks/Code: _____

Tutorial (4 October 2010)

- ☐ Making Test Automation Work on Agile Teams *by Lisa Crispin*
- ☐ A Rapid Introduction to Rapid Software Testing *by Michael Bolton*
- ☐ A Tester's Guide to Navigating an Iteration *by Janet Gregory*
- ☐ Patterns for Improved Customer Interaction/Influence Strategies for Practitioners *by Linda Rising*
- ☐ The Foundations of Agile Software Development *by Jennitta Andrea*
- ☐ Testing2.0 – agile testing in practice *by Anko Tijman & Eric Jimmink*
- ☐ Managing Testing in Agile Projects *by Stuart Reid & Isabel Evans*
- ☐ Agile Transitions *by Elisabeth Hendrickson*
- ☐ Executable Requirements in Practice *by Pekka Klärck, Juha Rantanen & Janne Härkönen*

750,– EUR
(plus VAT)

Conference (5–7 October 2010)

- ☐ 1 day (first day) 550,– EUR
- ☐ 1 day (second day) 550,– EUR
- ☐ 1 day (third day) 350,– EUR
- ☐ 3 days 1.350,– EUR
- ☐ 2 days 1.100,– EUR

1.350,– EUR
(plus VAT)

Included in the package: The participation at the exhibition, the social event and the catering during the event.

Notice of Cancellation
No fee is charged for cancellation up to 60 days prior to the event. Up to 15 days prior to the event a payment of 50 % of the course fee becomes due and after this a payment of 100 % of the course fee becomes due. An alternative participant can be designated at any time at no extra cost.

Settlement Date
Payment becomes due no later than at the start of the event.

Liability
Except in the event of premeditation or gross negligence, the course holders and Díaz & Hilterscheid GmbH reject any liability either for themselves or for those they employ. This particularly includes any damage which may occur as a result of computer viruses.

Applicable Law and Place of Jurisdiction
Berlin is considered to be the place of jurisdiction for exercising German law in all disputes arising from enrolling for or participating in events by Díaz & Hilterscheid GmbH.

Date

Signature, Company Stamp

Ethno-metrics

by Thomas Hijl

Column

Dear readers, imagine a group of people in white clothes. What do you associate them with? A wedding? Mourning? Maybe something else?

Years ago (1996), when I first went to Eastern Europe, cultural differences were immediately obvious to me. However, I always found it hard to explain to others what those differences exactly are. After a few years I discovered that those differences are mostly small things that add up to quite firm differences in practice. It's very interesting to look into those tiny differences and discover the influence on people's daily activities.

To give you an example:

When you want to meet people in a restaurant in Romania and you go there at the average Dutch dinner time (18:00) you won't find anybody there. Empty,... desolate place. A Dutch person might think that Romanian people don't go to restaurants and they all eat at home. Wrong, ...you need to go there at 21:00 at the earliest.

Another example:

A true black / white cultural difference can be found in black and white clothes, like I mentioned in the first sentence of my column. People in China might wear white clothes when mourning. In Western cultures it is associated with happiness and weddings.

Do you still remember the term ethno-marketing from the 90's? Specific campaigns for specific groups of people that were divided based on ethnic background. Asian people in one group, Latin-Americans in another, Western people in yet another etc. etc. The term ethno-marketing was very popular and meant that you focused your marketing campaign specifically on an ethnic group.

Over the years of project, test and acceptance management, I found myself in projects where this is no different when talking about test metrics. "Ethno-metrics!" As I named it. Each organization has its own culture and so it makes a huge difference how you communicate your metrics to the groups of people within that specific organization.

When you tell a project manager that there is a way around a major bug in the software, he will be happy. When you tell it to the person that needs to work with it, she or he can be very unhappy when it takes her/him 5 extra clicks.

So, how can we communicate the right things to the right people, when all kinds of different groups exist within only one organization? You have sales people, support people, managers etc.

Lessons to be learned:

Most easy lesson to learn is to never use the same presentation for different groups. A second lesson is to be open to those different cultures and feel empathy for the group you want to communicate your metrics to.

What to be aware of (the toolbox):

Cultural differences, in general, manifest themselves in different ways:

- Symbols

What kind of presentation do you use? Which words, pictures, graphs will reach your target audience.

- Heroes

Who is the hero in the group? When she/he gets the message they all will.

- Rituals

What does the group expect you to act like, look like, sound like etc? Present the metrics in a way they understand.

- Values.

Do's and don'ts within the organization. Know them well!

The metrics message:

Whether you are successful in getting the metrics message across, will only be proven by how your metrics were interpreted by the group and what will happen next.

My advise would be: Take on the adventure of new cultures, learn and teach at the same time and get the metrics interpreted as you want them to be interpreted... by being one of them (your audience). Be honest, don't sell!

Good luck ;-)

Thomas Hijl is acceptance manager and partner at Qwince BV. He started in 1996 as consultant / project manager for Philips Semiconductors (a.k.a. NXP) working on technology projects in the EMEA, APAC and AMEC regions. He switched to test management because of his ambition to improve project deliverables. Now, he serves clients with managing acceptance throughout the project lifecycle.

Benchmarking Project Performance using Test Metrics

by Nishant Pandey & Leepa Mohanty

It is a common practice to evaluate project performance using testing phase metrics. The use of metrics enables the test manager and other stakeholders to gauge performance of the test team, the test environment and the system under test. While the metrics of individual projects provide a lot of interesting and useful information, it is the comparison and analysis of the test metrics over various releases or periods that can help the test manager in deriving trends and addressing any issues that are being repeated over time. This can be a particularly effective tool especially in a release based environment. However, sometimes the metrics and trends might not be easily comparable across projects. The complexity, criticality, skills of the involved teams and management support can have an impact on the factors that govern these performance metrics.

How do we then determine the metrics performance of a particular project, relative to the data from past projects? How can we be sure that we are making a valid comparison and that our inferences from such a comparison are valid?

It is possible to benchmark the performance of processes and teams in projects using testing metrics. The exercise of benchmarking testing metrics with data collected from past releases is a useful tool for comparing, understanding and analyzing performance. It is also possible to extrapolate the idea to drive software product benchmarking and software process benchmarking using metric data obtained from the testing team. There are many available tools and techniques for benchmarking software products, processes and performances. One simple yet effective method for benchmarking testing project's performance is related to the concept descriptive statistics.

Preparing for Benchmarking Test Metrics

In order to enable effective benchmarking using testing metrics it is recommended that a preparation be started in advance. It is highly recommended that metric data from individual projects be maintained and updated in metrics database. It is also recommended that sufficient planning time be allocated to allow selection of appropriate metrics for the benchmarking exercise. Once you have data from the past project sample and the current project's metrics, you are ready to perform the benchmarking exercise. A box-and-whisker diagram can serve as an effective tool for graphically depicting groups of numerical data.

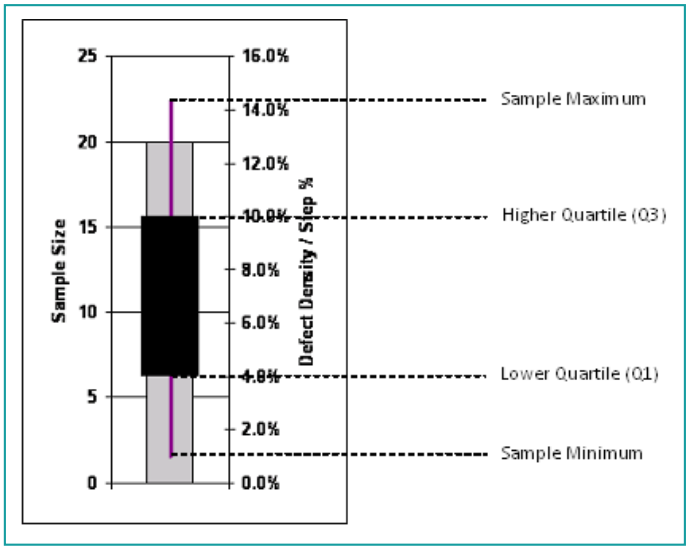
Benchmarking Test Metrics

After the metric to be benchmarked has been selected, analysis of the values in the sample from past data needs to be performed. For the purpose of this article, let's assume that we plan on benchmarking defect density of a recently tested code against the It's important to collect information on sample size, sample maximum and minimum, upper and lower quartiles as shown in the above table to be able to plot a box and whisker plot. Box and whisker diagrams are a convenient tool for displaying differences between populations without making assumptions about their underlying statistical distribution. The spacing between the different parts of the box represents the dispersion (spread) of the data, and is useful in identifying outliers. sample size of 25 past projects.

Measurement	Description	Value
Sample Size	The total number of projects that were considered in this benchmarking exercise	25
Sample Max	The maximum value of the particular metric in the sample	14.2%
High Quartile	The 75 percentile mark obtained by sorting collected metrics in order of magnitude	10%
Low Quartile	The 25 percentile mark obtained by sorting collected metrics in order of magnitude	4%
Sample Min	The minimum value of the particular metric in the sample	1%

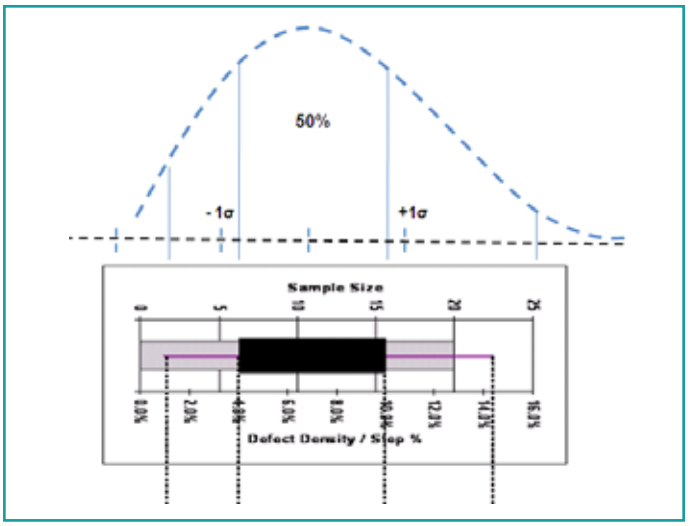
It's important to collect information on sample size, sample maximum and minimum, upper and lower quartiles as shown in the above table to be able to plot a box and whisker plot. Box and whisker diagrams are a convenient tool for displaying differences between populations without making assumptions about their underlying statistical distribution. The spacing between the different parts of the box represents the dispersion (spread) of the data, and is useful in identifying outliers.

Below is the graphical representation of the metric data of the sample discussed.



The value of the current metric (the one to be evaluated) is then superimposed on the above plot along with the mean and the median of the data. This data and its representation in the box format can empower the test manager and other stakeholders for making meaningful inferences. The relative position of median value and the metric (to each other and to the bulk of 'most likely' sample values) and provide insight into the relative merit of the performance gauged by the metric being benchmarked.

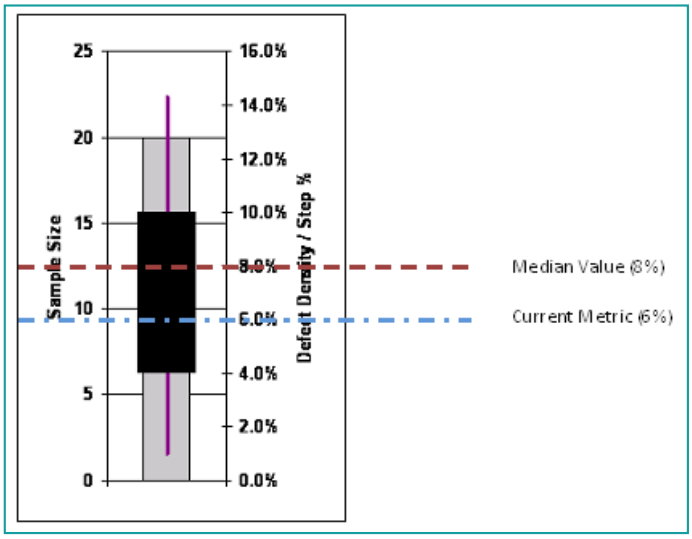
The box whisker plot is a convenient way of comparing and analyzing data points and their statistical distribution. One can compare the box plot against the probability density function for a normal distribution to understand the concept.



Additionally, while making inferences from the box whisker plot, the size of the box and the relative positions of median value and the current value play an important role. The examples below depict few inferences that can be made depending upon the relative positions and box sizes.

Example:

Defect density of the current project is measured as 6%. The test team wants to use benchmarking technique to compare this value to defect density in the past releases. The test manager refers to the metric database and is able to find data pertaining to Defect Density of past 25 projects. Using this sample size of 25, the box-whisker plot is prepared as shown below.



Following Inferences can be made from the above:

- In the past, the Defect Density has varied from 1% to 14% (Range)
- Outliers can be identified and eliminated - Values greater than 10% and less than 4% are rare
- The current metric falls under the majority range and is not an outlier
- Current metric is lesser than the Median value indicating that the code under test for this project has a lower defect density than most projects completed in the past.
- The whisker depicts the spread of the data and the box depicts its dispersion.

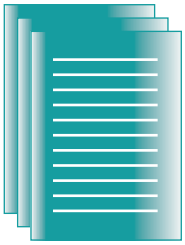
If the current metric falls in the outliers, then it might make sense to investigate the reasons behind the fact. Also at times plotting both the median and the mean can lead to insights related to skew in the data sample. Including this in the equation can help an even more precise analysis of the metric being benchmarked, however such degree of precision might not be needed in most cases.

So next time when you are ready to perform closure of your testing project, you might want to attempt benchmarking on key metrics that you use. The box whisker plot can be prepared using Microsoft Excel and quite a number of samples are available on the internet.

Biography

Leepa is a Lead Consultant with Capgemini. She is a seasoned Business Analyst and is a certified PMP and Test Manager. She has managed IT projects in Singapore, US, Middle East and India. Currently she is based out of Orlando (Florida), where she manages a team of Business Analysts for Capgemini.

Nishant is a Lead Consultant with Capgemini. He is a certified Project Manager (PMP) and Test Manager (ISTQB Advanced Certificate in Test Management). Nishant also holds the ITIL V3 Foundation certification and a certificate in IT Benchmarking from Stanford University. He is based out of Orlando (Florida), where he works for a prestigious client of Capgemini, managing the Capgemini Test Engagement.



Test Encapsulation – Enabling Powerful Automated Test Cases

by Rahul Verma

1. About the Series

Test Automation Frameworks (**TAFs**) do not get the treatment in terms of design that they deserve. Spending time on understanding the precise requirements and then designing/choosing the framework becomes much more important when a general-purpose framework is required. Such a situation is unavoidable when your organization is looking at the possibility of a single underlying TAF that could be employed by multiple teams involved in strikingly different products; their testers are conversant with different programming languages, and they choose or need to employ different tools to support testing.

This series is focused on the technical aspects of designing a scalable, general-purpose, object-oriented TAF. This in turn means that the contents would mainly comprise concepts that directly translate into an implementation in an object-oriented language. One could choose to write the same thing in the traditional programming way, but there would be further work required on part of the reader to translate the design.

The series will not attempt to answer the questions of why to automate, when to automate, which tests to automate, automated tests versus manual tests, ROI of automation, etc. There are quite a few books available on the subject of test automation, and these topics are discussed at length in them. So, if the reader chooses to follow this series, it is assumed that he/she already understands the importance of test automation. This series focuses on how to automate, whereby the "how part" is not about using an existing tool or framework, but rather about designing one.

The name of the series "Notes on Test Automation" purposefully does not use any adjectives to indicate how universally efficient or effective its ideas are. This series will evolve over time, and I am going to share practical nuggets of test automation knowledge with you that I have learnt from my practical experience. We will occasionally strike out a sentence or two which revokes our initial understanding, and you might find me challenging my own ideas from the previous or even from the same article.

2. The Problem of Dumb Tests

In the design of TAFs, most of the important information that a test requires is kept outside the TAF, as many times such frameworks are designed using traditional programming rather than employing object-oriented programming. A simple example is grouping tests into platform-based folders, after which the scheduling logic copies in only the tests meant for a test execution machine based on a specific platform. There are multiple problems with this approach. One is redundancy, multiple copies of tests across folders. If you think about it, we could have a common test platform and then specific platform folders, but my question is how many such "commons" would you have? Even if you have the patience, what happens when a test gets modified and is not meant to be run on Platform A? How do you know to which folder you should go to change it? So, there should be an external means to map tests to folders! By not making the information available within the test about which platforms it should (or should not) run on, you have the following situation:

- You need a complex folder structure, which is virtually impossible to maintain as the number of supported platforms grows
- You need an external means to map tests to the above folder structure.
- If the test writer is not the test framework administrator, it means the person who requires this is different from the one who implements it.
- You have a "dumb" test, which does not know on which platform it can run. So, if anything goes wrong, you have an incorrect test case that could crash the system or, at worst, lead to an incorrect result.

So far, we have only talked about the platforms. We have not talked about categorization of tests, priority values associated with tests, known bugs, API version checks. Can you see the complex and impossible folder structures that these options would lead to if kept outside the test?

This article focuses on the concept that a test is meant to be the most complex part of the framework in terms of its power and flexibility. Test encapsulation is at the heart of building a general-purpose framework, which can be used by testing and development teams as a common test automation platform for white-box and black-box tests. To know about what exactly is meant by test encapsulation, its approach and benefits, keep reading!

3. Introduction to Test Automation Frameworks

As usual in the software testing world, even the term "test automation framework" is interpreted differently by different individuals and in different contexts. Here are some general points which arise when one discusses terms:

- **Data-driven test automation approach**: This separates test data from the test execution code. This is helpful when the same scenario is to be driven with different sets of data. This is also helpful in enabling changes being made to the test data for an existing scenario without modifying the test execution code.
- **Keyword-driven test automation approach**: This is a command-based model. Commands exist outside the test execution code in the form of plain text files with corresponding parameters. The execution logic reads from the files, interprets the commands and calls corresponding functions in the test automation framework. This is mostly used in combination with the data-driven approach.
- **Generation and mutation based test automation approach**: Instead of relying on hard-coded data provided, these frameworks rely on data generated at run-time. Another difference from traditional data-driven frameworks is that these con-

***IREB®** Certified Professional for Requirements Engineering (English, German)*
***ISTQB®** Certified Tester Foundation Level (English, German, Spanish, Russian)*
***ISEB®** Intermediate Certificate in Software Testing (German)*
***ISTQB®** Certified Tester Advanced Level – Test Manager (English, German, Spanish)*
***ISTQB®** Certified Tester Advanced Level – Test Analyst (English)*
***ISTQB®** Certified Tester Advanced Level – Technical Test Analyst (English)*
***ISSECO®** Certified Professional for Secure Software Engineering (English)*

Why choose E-EXams?

X More Flexibility in Scheduling:

- choose your **favourite test center** around the corner
- decide to take the exam at the **time** most convenient for you

X Convenient System Assistance throughout the Exam:

- "**flagging**": you can mark questions and go back to these flagged ones at the end of the exam
- "**incomplete**": you can review the incompletely answered questions at the end of the exam
- "**incongruence**": If you try to check too many replies for one question the system will notify you

X Immediate Notification: passed or failed

* Availability of E-Exams is subject to country restrictions

www.isqi.org | certification@isqi.org
CERTiFYING PEOPLE

iSQI®
International Software Quality Institute

tain commands for mutations directly within the input configuration files, thereby incorporating a flavor of key-word driven frameworks as well. These are more complex than both the above categories and their implementation is usually found in the world of "fuzzing" frameworks.

- **Multiple test nodes on a single test runner machine**: (Test node is the thread or process running tests on a test machine). This is mostly seen in performance testing frameworks, which use a single test runner machine and multiple test nodes connected via multi-threading (mostly) or multi-processing. This is made possible only for non-GUI testing and testing in which no system-wide changes are made by a test.
- **Central execution control**: The framework should be able to trigger, control and report the results of multiple tests from a single point of control. For example, CPPUnitLite, a unit testing framework for C++, can execute all tests via make command.
- **Distributed test execution**: Testing frameworks can also be thought of as providing the functionality of central execution control from one or more "controller" machines, which can trigger, control and report the results from multiple test node machines.

- **Enhanced reporting via GUI**: As an extension, frameworks can support GUIs (stand-alone executables or web-based) to support customized reporting. In performance testing frameworks, for example, features for plotting and analysis of performance statistics is essential. Also, different stakeholders would have different needs in terms of the test report they want to see with respect to details and contents.
- **Hardware/operating system infrastructure**: In some contexts, where hardware and operating system requirements are precise (or even otherwise), some refer to the complete set-up (i.e. test automation code + hardware and operating system set-up) as the test automation framework. This can be seen, for example, where the functioning of the TAF depends on external tools that are pre-installed on the test runner machines, which cannot be done as part of the test set-up.

Note: *Some texts enumerate the first two bulleted points above as the "Types of test automation frameworks". I beg to differ. I do not consider these as types of test automation frameworks, because that would suggest they cannot co-exist in a single framework. The first three are approaches towards a TAF design in terms of enabling the TAF to execute tests via data and commands present in clear text in various forms, such as plain text files, XML files, databases etc. All three can be built on top of a general-purpose framework. In simple words, a test framework can provide any and all of the features mentioned above, by choice.*

Following is the list of other general features, on which decisions have to be made when choosing/building a test automation framework:

- Which language should be chosen to build the base frameworks? For which languages would extensions be supported?
- Should the test automation framework be generic or specific to a product under test, testing tool or a language, in which tests can be written?
- How much complexity do you want to hide from the end user (the one writing/executing/reporting a test). Many times, hiding the complexity might result in providing less flexibility as well. Some performance testing tools, for example, provide for the recording of the test scenario as a GUI-based tree representation with a limited amount of controls for customization – ease of use with less flexibility. Others might choose to provide both.
- Should it be cross-platform? What browsers it will support? Which versions of the product can it test?
- How do you manage framework files (e.g. core library, configuration files) and the user contributed files (scripts)? Are you going to opt for version management? What would be the protocol? Should it exist along with the development repository for the product under test, or be separate?
- What kind of regression strategies would it provide? Does it provide for bug regression, priority based regression, author based regression, creation date based regression, API version based regression, etc.?
- Should it support physical and/or virtual test runner machines?

The above is an incomplete and a very high-level list of features that one might need to consider when building in a test framework. So, it shouldn't be a surprise if one sees testers dissatisfied with the testing tool/framework they currently employ, because there would be one or the other feature missing.

4. Did We Miss Discussing Something?

In the discussion of this complex and long list of requirements, we missed the most important thing. What's the final purpose of a TAF? With all the "noise-features", which help in scheduling, executing and reporting with an icing of user-friendliness, at the core of it, *a TAF is about running one or more tests*. Where in our discussion did we talk about what the test should look like? How often does this aspect get discussed when we talk about TAFs?

TAFs are like onions. Their tastiest part is their core covered with lots of wrappers/layers. For a TAF the tastiest (the most important) part is the code written by a tester which is going to execute the actual test on the system. In our TAF discussions, we are often so caught up in the wrappers and layers, that by the time we reach the heart of the problem, we have made it powerless and small, putting most of the things in the hands of the surrounding wrappers/layers.

If the "test" part of a TAF is not designed well, all other features will be useless. All user-defined configurations, default framework configurations, platform conditions, tool decisions, pass/fail decisions etc. have to be translated into how the test gets executed and how it should report the results back.

Because of this faulty approach, most of the information, which should enable the test to take decisions at runtime, is made available outside of the test, so that some outside component and not the test itself makes these decisions. As mentioned in the introduction, this can become highly problematic when you try to add more and more features to the framework.

5. Encapsulation

As per Grady Booch, encapsulation is – "The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

If that sounds complex, let's settle for this:

When we encapsulate X, we provide inside the X:

- All data that defines the state of X as well as the data it requires to do its job.
- All methods that X needs to use and manipulate the data available.

6. Test Encapsulation

With test encapsulation you provide inside the test all the data it needs for test execution and all the methods related to setting-up, running, reporting and clean-up.

6.1 What does a test need to KNOW?

Let's personify test and see what it asks:

6.1.1 Meta data

Every test is unique in that it would have at least one thing that's different from the other tests in that framework. (If it is not, please check your review process!). It means, every test should be identifiable as a separate entity.

A test could ask the following questions in order to find out what it is:

- What is my name?
- What is my identification number? (a single test could have multiple IDs in different contexts)
- What is my purpose?
- Who is my author?
- What is my birth (creation) date? On which date, did I take my present shape? (modification date)
- What is my version number?
- Who are my parents and grandparents? (parent group/sub-groups)
- What type of test am I?
    - At which level do I work? (unit/component/system)
    - What kind of software attribute do I test? (functional/performance/security)
    - When should I get executed? (BVT/acceptance/main)
    - Am I based on custom extensions? (e.g. CPPUnitLite based test)

6.1.2 Logging options

Logging of test results, progress of execution and dumping key information at precise locations is one of the key aspects of test automation.

The following are questions that a test might ask:

- Do I use the centralized logging mechanism or my own?
- Am I running in debug mode, thereby increasing what I log and changing where I log?
- Should I log performance statistics and local resource utilization statistics?

6.1.3 For runtime

Prior to test execution, a test must know some important things about itself to take decisions at runtime. Compare this to a game show, in which the person asking the questions would request that only those that satisfy a particular condition would be eligible for that round of the game. So, when he shouts "All those in black shirts", the ones in black shirts would rush towards the podium. But before getting up, the person must know that he is wearing a black shirt!

On the same lines, at runtime, the test would come to know about configuration settings chosen and the test environment around itself. At that time, to make important runtime decisions, a test would ask:

- How important am I? What is my priority?
- What is my version? Yes! I will utilize this when version based regression is on.
- Did I uncover some bugs so far? If so, what are their IDs?
- For which API versions of the product can I and can't I run?
- On which platforms can I or can't I run?
- Do I need stubs to run? Can I run at all if stub mode is on?

  (There could be a requirement that you want to use stubs instead of actual components, which is usually the case with unit and component testing. There could be tests, which are meant to be run only with stubs, others which can run with and without stubs, and still others which would not run with stubs.)

6.1.4 From runtime configuration

There are various runtime configurations, which can be configured by the user or are set as defaults by the framework. As discussed in the previous section, these should get passed on by the framework to the test so that it can match them against its properties to take runtime decisions. One important example of such a decision is whether the test should get executed or not.

Let's see what kind of questions the test might ask:

- What is the API version under test?
- What is the platform on which I am being asked to run?
- What are the regression options? Are you looking for priority based regression, bug regression, version regression, author based regression or any other form?
- I have been asked to always log performance data. Do you want to override this?
- Do you want me to run in debug mode?
- What is the base directory reference path from which I am getting executed?
- What is the current mode of execution? Are you using stubs?

6.1.5 Execution properties

Once the test takes the decision that it is meant to execute (i.e. the criterion received from the runtime framework configuration settings match its own filter properties), there are execution related questions, which the test would need to ask:

- Where is the build under test located?
- Where are the tools that I need?
- How many threads/processes should I launch?
- The previous test has just completed. Should I delay execution by going into sleep mode?
- What configuration should I use for the tools that I plan to use?
- Where are my input files?
- To whom should I hand over the test results? Can I talk to the database directly or is there a middleman?

6.1.6 During and post execution

There are properties of the test which are set during and after execution of the test. These make up the basis of the following questions:

- How much time did I take to execute?
- Where am I in terms of execution? What is the current step I am executing?
- Where should I maintain my state information if I am asking for a reboot now?
- Did I ask for a system reboot/hibernate at a previous step?
- How many assertions have I made? How many of them passed or failed?
- Did any of the assertions relate to errors in execution?
- What is my final say about the test I executed? Did it pass or

Berlin, Germany

IT Law
Contract Law

German
English
Spanish
French

www.kanzlei-hilterscheid.de
info@kanzlei-hilterscheid.de

k a n z l e i h i l t e r s c h e i d

fail?

- Should I send notification to the administrator that I potentially succeeded in finding a bug?
- Should I send notification that I couldn't execute due to error in execution?

6.2 What does a test need to DO?

In section 5.1, we talked about all that a test needs to KNOW at runtime. Let's now look at all it needs to DO at runtime:

6.2.1 Prepare

A test needs to prepare itself. This means that at runtime the test needs to load all properties that were set for it statically by the writer of the test. It should also get hold of the runtime configuration and test environment settings passed to it by the test framework.

6.2.2 Set-up

It needs to do the initial set-up for the test that needs to be executed. For example, if the test needs something to be installed so that it can run, the set-up step should, as a pre-cursor to the test, complete the installation.

6.2.3 Run

This step in the test would include the code that executes the actual test. This could comprise of multiple assertions (sub-tests).

6.2.4 Report

The test would report the results of the test as per the configuration, e.g. to a chosen file or database, or publish the results to an object, or return them as part of the function call.

6.2.5 Clean-up

The test would clean up anything it created specifically for itself, so that the system is returned to its previous state ready for the next test.

7. How to Achieve Test Encapsulation

What we discussed in Section 5 might seem like a lot to be put in a test, might it not? Relax! The fact that a test needs to know and do a lot does not mean that the solution has to be complex. You can still hide most of the complexity from the end user, thereby making the test scripts simple, yet powerful. The following are some quick tips:

- If you take a careful note of what a test should know, you will observe that most of it is optional and would be used only for certain test cases.
- You can set the meta data and other properties in a container to which the tests is registered. This means that this would not appear for every test case written, but still can be overridden at the test level, if needed. For example, a test group can have the author name as the XYZ, to which 30 test cases are subscribed and not even one of them needs to have the author property set by the script writer. Later, if another tester adds a test case, the author property can be overridden for the 31st test case.
- From all that the test should do, you'll find that, apart from prepare() and run(), all other commands are optional and can be set for the container if the setup() and tear-down() are common. In fact, grouping the tests like this into containers would make it easy to manage.

Similarly, the runtime checking for whether a test is "runnable" or not is hidden from the end user, and is placed within the core library.

8. Benefits of Test Encapsulation

As a first benefit to reap once you have taken a decision to proceed with test encapsulation, you have already crossed the first hurdle by making the tests the most complex and powerful part of your framework.

In the introduction section, I mentioned the problem of running a test on multiple platforms and difficulties with the approach of folder wise grouping. Let's try to solve that with test encapsulation.

- Imagine that you created a test that knows on which platforms it is meant to run. Let's call it the MEANT-FOR-PLATFORMS property of the test, which is list or array of platform strings. For this example, let's say it is: ["WINXP_X86","WIN7_AMD64"], wherein the values signify that this test is meant to be executed on 32-bit Windows XP and 64-bit Windows 7 platforms.
- All tests reside together in a single folder structure; no redundant copies are created for the above-mentioned platforms.
- In the test cycle, when the TAF is running tests on a given test runner machine, it already knows the platform of that machine. While it is looping over the tests, it would pass this information to the test, for example, by setting the test's RUNNER_PLATFORM property.
- Now when the TAF asks the test to execute, the test can search for RUNNER_PLATFORM in its MEANT-FOR-PLATFORMS array, and only if it is found in the list, will the test agree to execute. Accordingly, the test would execute or the TAF would skip to the next test in queue.

In a similar way, you can add support for any runtime decisions. The approach is pretty much the same:

- Encapsulate the properties that form the basis of runtime decision in the test
- Make the base value available in advance
- Make the value against which comparison should be done at runtime available at runtime!

The second benefit that you get with test encapsulation is that your framework does not dictate to a test which product it is testing, which tools it should use with what parameters/configuration, how it should analyze the results or how it should report back etc. You can still provide these features which the test writer is free to use or ignore at will. This puts a lot of flexibility into the hands of the test writer, and you are free to enforce rules if the test writer chooses to use a given framework feature. An example is that a test writer might choose to write the report in a format and place of choice, but if he wants to use the web reporting platform provided by your framework (if it's there), then he has to provide the results in a pre-defined format.

The third benefit is that whatever your test does can be kept independent of other tests. It can set up and clean up any changes to the system. Also, you can put exception handling in the caller covering all individual test methods being called. This will enable the framework to continue execution, even if there's an error in a given test which has been called. These exceptions can be reported in the final test report generated by the framework and would help in providing accurate results as well as point testers to the areas which need investigation.

Over the course of the next articles in this series, we will observe how test encapsulation can help in implementing many of the essential features of the TAF in an easy manner.

9. Execution Time Overhead of Test Encapsulation

As you have probably observed by now, to skip a test with test encapsulation, involves a runtime decision. This means any skipping of the test would have to be done at runtime. Before this happens, the TAF must have already called the test constructor.

Being from the performance testing background, I can relate to any concerns relating to overhead, but this is not as much as it sounds. It doesn't take minutes or even seconds to create a test object. The time taken is a tiny fraction of a second. I designed a framework with this approach, and my requirement was to measure the time taken by the test to execute. For some tests, the values were not captured even with a precision of 6 decimal places when the unit was seconds.

Having said that, you can reduce this overhead still further:

- As discussed in section 6, you can group tests in containers. You could put the filter at the container level rather than at individual test level. For example, you can set the MEANT-FOR-PLATFORMS property for ABCTestGroup to which all tests of ABC category subscribe.
- When using this approach you have unlimited test filters. Look into the order in which filtering is done. Place the most commonly used filters first. For example, if the probability of an author based regression or a creation date based regression is much lower than bug regression in your context, filtering based on empty bug lists should come first.
- Test encapsulation has given you the capability to take runtime decisions, but it does not stop you from making careful exceptions. If you feel that most of your tests can be easily split up into platform-wise categories and there's minimal overlap, you could still group them into folders. Then resort to scheduling only specific tests for specific platforms. If over a period of time you observe that the scope of overlapping tests is increasing considerably, all you have to do is set the optimum platform property for tests and start relying on runtime checks.

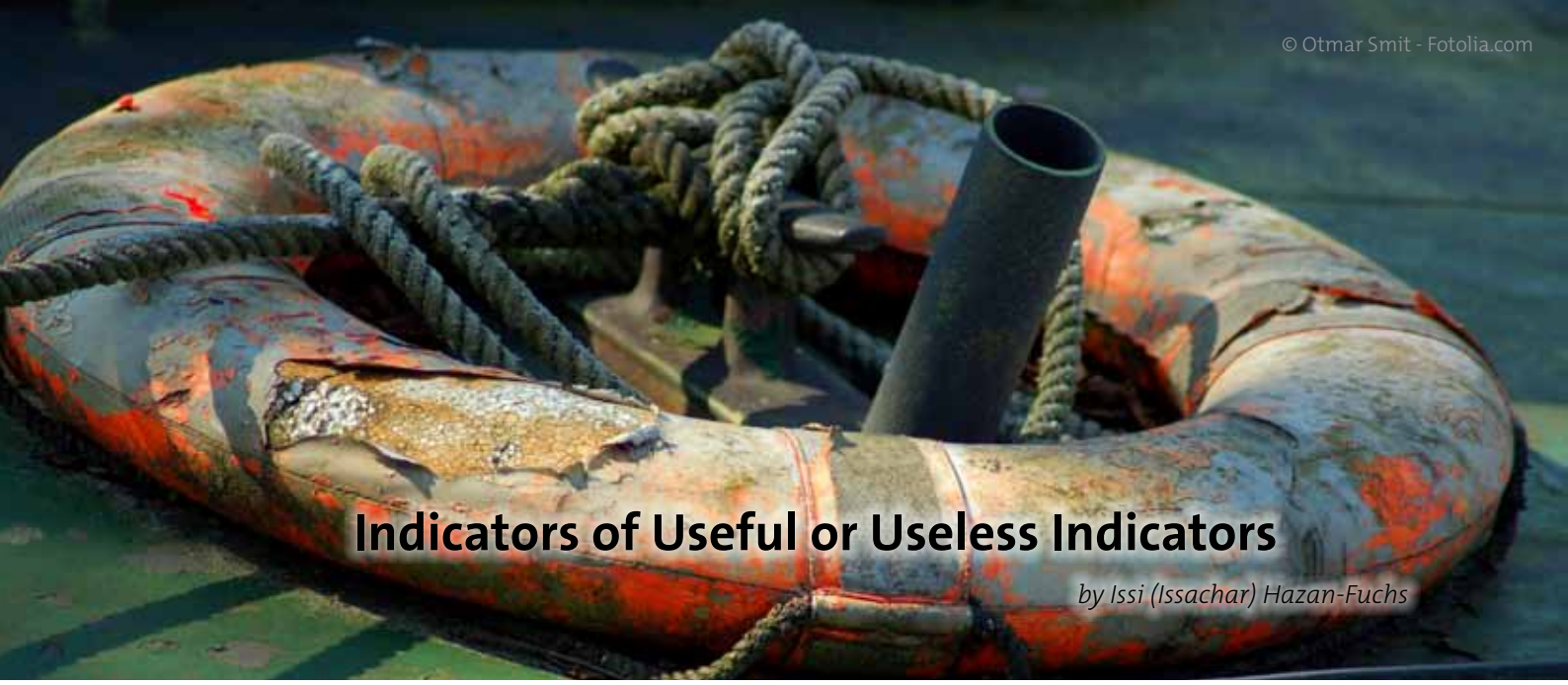
So, the overall benefit of test encapsulation is that once you have carefully built it into your framework and created the underlying skeletal support, you can still go ahead with your traditional approach to test automation. What's Next?

This was the first in a series of articles that I plan to write under the general heading of "Notes on Test Automation". Next, I will write on distributed testing execution and discuss two simple models.

As you can see, this article was focused on concepts and did not have any code snippets. This will be true for the next couple of articles as well. Once we start tying all concepts together, I'll move on to discuss the subject with pseudo-code/Python code as needed.

Rahul Verma is a Senior QA Technical Lead with the Anti-Malware core team of McAfee Labs, India. He runs the Testing Perspective (www.testingperspective.com) and Python+Testing (www.pythontesting.com) websites.. He has special interest in performance testing, security testing, Python and design of test automation frameworks. Rahul has presented at several conferences, organizations and academic institutions, including CONQUEST, STeP-IN, ISQT, TEST2008, Yahoo! India, McAfee, Applabs, IIT Madras and STIG. He is a member of the working party for ISTQB's Certified Tester, Advanced Level.

rahul_verma@testingperspective.com



Indicators of Useful or Useless Indicators

by Issi (Issachar) Hazan-Fuchs

During the past few years, I have been involved in producing and reporting testing indicators. I have observed good and less good indicators, learned from my own and others successes and mistakes, while reaching agreement with stakeholders on format.

I would like to share my own experience, both good and bad, and give indicators to assist you in evaluating the usefulness of your own testing indicators.

I am going to specifically discuss my experiences with quality indicators (bug counts) and with test execution indicators.

Before we start, we must define what "useful" means in our case. My understanding of "useful" indicators has been shaped by trial and error and by reading the thoughts of others[1].

The most simple definition that I can define as a baseline for this article is:

Useful indicators aid the decision-making process

Note that I used the term **indicators** rather than metrics, to emphasize the fact that metrics are not employed to measure for the sake of measurement, but to aid decision making and the resulting actions. I also highlighted the verb **aid** to mark that useful indicators are not a direct baseline for decisions, but triggers for inquiries that will be the baseline for making decisions. The inquiry itself cannot be done by any formula, but by interviews of people, observations and analysis[2].

I will try to list a few indications that I found to be signs of useful indicators, and indications, that I found to be signs of non-useful indicators.

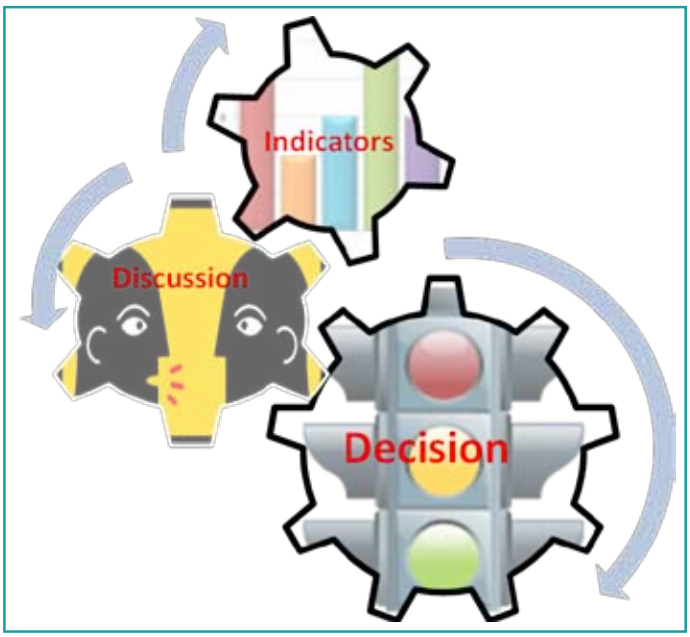
In addition to the play on words in the headline of this articles, describing the "Indicators of useful or useless indicators" from my own experience is an opportunity to share my own learning process of observing an indication and deriving a heuristic from it[3].

Note: Following the spirit of this article, these are only indicators and not absolute facts. Like any indicators, you should use them only as aids for the evaluation of your indicators.

Useful Indicators:

- **Indicators** produce a **discussion** that initiates **decisions**

A good indication that our indicators are useful is when these three parts - Indicators, Discussion and Decision - exist and connect in a cause and effect connection. Otherwise indicators are just wasting time for the people who produce or view them. Discussions without decisions might indicate that the indicators were interesting. but did not provide the data for decision making (unless we have a decision-making problem). If we have indicators which initiate decisions by themselves, we probably have a problem of over- valuing the indicators.



1 *I highly recommend reading Michael Bolton's post "Meaningful Metrics" which describes an approach for meaningful metrics. See http://www.developsense.com/blog/2009/01/meaningful-metrics/*

2 *Ibid.*

3 *Thanks to my friend and colleague Shmuel Gershon for reviewing the draft of this article.*

Although I have never heard of a machine which automatically makes the decision to release a build according to the number of reported bugs on the code base, the problem of viewing the numbers as targets instead of aids to the decision making process, is common.

Discussions and decisions will not happen each time we extract the indicators. When defining the indicator sets, imagining what-if scenarios and assessing whether they are capable of inducing discussion that will lead to decisions, can be a good way to assess them.

Useless (or less useful) Indicators' Indications:

- Indicators are happy while the project is sad

While corrections, exceptions and complimentary information are essential to complete the picture drawn from indicators, when time after time you have to complete the picture by drawing an **opposite** picture to the one that is apparent from the quantities indicators, your indicators might be wrong, or the whole activity is going in the wrong direction. For example, week after week getting green "pass" bars for the test execution while you have to express your concerns that the product itself is not useable or too buggy can indicate that you are publishing the less important indicators, or that a major activity of your test team is focused on a less critical path of your product quality.

I could say the same about "Sad Indicators in a happy project". While working on this article a software developer that I know told me about a project she was involved in. The project manager asked to delay the release because the high-severity defect numbers were too high, but inquiry proved that the defects were not classified correctly and they were only minor cosmetic ones. The project manager understood his mistake and suggested to release the product, but the developer objected. Not due to the number of open bugs, but because the type of bugs showed that the test team spent the time on finding minor, redundant bugs instead of testing the relevant functionality.

- Extracting the indicators consumes more than a few minutes

Although there is no direct relationship between the time it takes to extract the indicators and their quality. In cases where extracting the indicators becomes a long task that consumes too much precious time, this can indicate that the indicator set is too complicated or too large. Automating the indicators is a good idea in such cases, but before you set off, you should first revisit your approach, in order that you will help your process and not just solve the time problem whilst preserving the over-complicated approach.

- Indicator set continuously change during the project

Although being responsive and tuned to the stakeholders' needs is a good thing, so is being flexible about your own process. In cases where there are repetitive change requests to the indicator set during the project, it can indicate that you have a fundamental problem with the indicator set. Either the stakeholders do not understand or agree on the purpose of the indicators, or your indicators are not defined in a useful manner.

- Indicator set remains the same for ages

As opposed to the continuously changing indicator set described above, the opposite could also occur. Where the indicator set is an old one that you inherited without questioning, there is a good chance that it is out of date for your up-to-date needs.

"Use with caution" indications

- When a second mathematical function is involved

Some indicators introduce a weight function that unifies a few items that are measured together. For example, if there are some components of our software which are more risky than others, we will use a function that will assign a weight to components according to our risk level analysis.

This approach can be useful since the indicators looks more unified and complete, but the mix between the subjective interpretations and the numbers can be misleading. In the correct context it is definitely OK, but in some contexts, especially when indicators are a sensitive subject, it's better to have the bare numbers followed by a verbal interpretation for discussion, so that the numbers themselves will not be subject to debate.

I hope that I have given you some good indicators and food for thought so that in future you will ask your own indicators, "Do you work for me (and the project's success)? Or do I work for you?" – Such questioning can be a good start to better and more useful indicators.

Biography

Issi (Issachar) Hazan-Fuchs has been testing drivers, firmware and software for more than 10 years in the Functional Testing Lab of Intel® Design center in Jerusalem, Israel.
During the last years, Issi was testing technical lead of several projects, and was involved in defining and producing indicators for the product development team and other stakeholders.
Issi publishes his thoughts about testing in his blog at: http://testermindset.blogspot.com

Go Lean on Your Software Testing Metrics

by Jayakrishnan Nair

There is something almost funny about software testing metrics. Everybody wants them, believes he or she knows about them, and often have them (frequently in large quantities!) but few really understand or utilize the information they see in their metrics reports to do anything useful. Having worked with several organizations in my career as a tester and consultant, this has been a clear discernible pattern.

On the other hand, a common complaint that you hear in many of these places, especially at the executive level, is that there is no clear visibility into the state of testing or the quality of the software tested: "We have no idea if we are testing enough. Are we letting defects leak into production? Maybe we have just been lucky so far." Interestingly, some of these companies have extensive metrics reports with pages of colorful charts and tables! Why do they still have this problem?

The answer lies in the planning and design of the metrics program. There are many things to consider, of which one of the most significant has to do with choosing a parsimonious set of metrics parameters.

This article has a narrow focus. It begins by describing in brief a general approach for selecting metrics for software testing. It then introduces a basic set of metrics that can be considered for use in many organizations. For each of the metrics parameters presented, I will explain what it really means, why it makes sense and how to properly interpret the information it represents. While the discussion is set primarily in the context of software development, it has broad applicability wherever software testing is performed.

So, Where Do You Want to Go Today?

Simply taking measurements will not be of any use unless they are evaluated against goals. A good practice is to start with the business goals of your organization that are relevant for the testing function. These goals may be related to revenue growth, profitability, operational efficiency, employee satisfaction, customer satisfaction etc. You need to align your metrics program with these goals, for a simple reason: if you do not know where you want to go, it does not really matter whether or not you measure progress, does it?

So how do you get to the metrics from the goals? Well, there is more than one way to bell the metrics cat. A popular and effective approach is to apply the Goal – Question – Metric (GQM) method, illustrated in the diagram below. Reference materials on the GQM approach are plentiful in the public domain; it is not the purpose of this article to delve into details.

Too Much of a Good Thing?

Yes, it is possible. This is especially true of metrics, and if you have this situation in your organization, you may face problems such as the following:

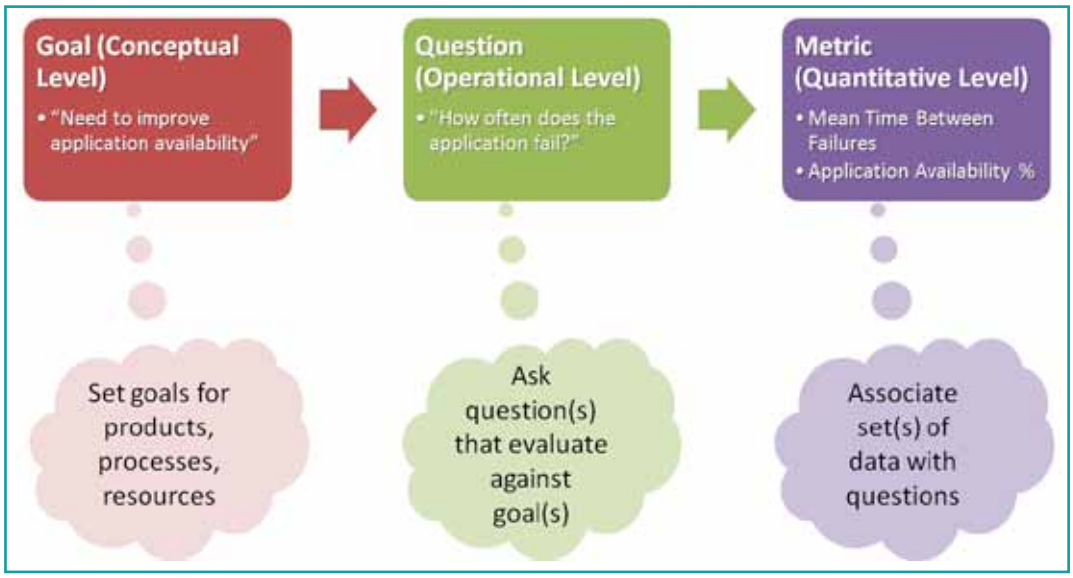


Figure 1: GQM Method

- If you have a very long list of metrics to track, you probably haven't spent the effort to choose the right ones. Do you need them all? Do they really serve the intended purpose?
- You run the risk of information overload. Relevant information may be hidden by the tons of "junk" data all around. In the worst case, consequences can be severe if a decision maker misses a critical piece of data.
- The more the data you have to collect and manage, the more the overheads in the process and the greater the risk of non-compliance and inaccuracies.

Therefore, my advice is to go lean. *Parsimony* is the mantra.

A Basic Set of Software Testing Metrics

A small set of basic metrics for software testing that are useful for many organizations is listed below.

- Test Coverage
- Productivity
- Defect Detection Effectiveness
- Defect Acceptance Ratio
- Estimation Accuracy
- Schedule Variation

A caveat, though you may consider these as potential candidates when you reach the "M" stage of your GQM journey, but by no means should you adopt them blindly. While they are of general applicability, they are not necessarily the most suitable ones for your context, nor do they represent a "full" set. Please also note that variations of terminology or definitions are possible across organizations. The main intent here is to illustrate using this set how to understand and properly apply software testing metrics and explain how the use of such metrics ultimately helps the organization meet its business goals.

Let us now consider each of these metrics in detail.

Test Coverage

Simply put, this useful but rather controversial metric, tells you what fraction of "everything that should be tested" has actually been tested. The controversy arises from the fact that "everything" is all that the software is supposed to be and do, and strictly speaking, it defies definition.

So what do you do? You approximate. A good approximation is to decide that "everything" is the set of requirements specified for the software. If you have comprehensive, well documented requirements, this is a reasonable assumption to make.

The key specifications of the Test Coverage metric are:

Metric	Definition	Calculation	Unit
Test Coverage	Percentage of requirements that are covered in test execution	= (Number of Requirements Covered in Test Execution) / (Number of Requirements Specified) * 100	%

Let us now see why this metric makes sense and what it tells us.

The Test Coverage metric measures the extent to which testing has covered the requirements of the application under test. If this is not close to 100%, it means that portions of the application remain untested and therefore undetected defects may be present in the product. Therefore it is a leading indicator of quality of the final deliverable and therefore on customer satisfaction (which is very likely a business goal for the company). Additional time spent fixing defects in production can impact time to market the product. Poor customer satisfaction due to quality issues or a delayed product launch can adversely impact future revenues (another business goal). Since the cost to fix defects in production is typically quite high, Test Coverage is also an indicator of the risk to profitable operations (yet another business goal that just about every organization has).

Productivity

Productivity is a fundamental metric in software engineering; it tells you how fast work gets done. In the context of testing, Productivity can be defined for test design and execution as shown in the table below. (It can also be defined similarly for other testing tasks such as test data set up).

Metric	Definition	Calculation	Unit
Productivity of Test Design	Number of Test Cases developed per person hour of effort	= (Number of Test Cases Developed) / (Effort for Test Case Development)	/hour
Productivity of Test Execution, or Test Execution Rate	Number of Test Cases executed per person hour of effort	= (Number of Test Cases Executed) / (Effort for Test Execution)	/hour

Higher Productivity values are desirable. Monitoring Productivity daily or weekly provides a leading indication of whether the project is at risk of meeting the schedule.

In most projects that are in trouble, someone in the team would have been aware of impending trouble in advance. If the Test Manager is surprised to see low Productivity values in his project, it may be that the team is facing bottlenecks but did not communicating this to him.

Low Productivity levels can mean one or more of many things, and the causes must be investigated. It may, for instance, indicate that project teams have a competency gap, are not equipped with the right tools, and/or that the process being followed is not effective. Addressing these issues (for example, test automation may be a solution in some cases; training may suffice in others) will help reduce costs by improving the productivity and effectiveness of the testers. Productivity is a leading indicator of employee satisfaction. Issues with Productivity usually also lead to quality issues and therefore may have an impact on customer satisfaction.

When measuring Productivity, care must be taken to ensure that the team records accurate effort information, and this is not always easy (for a variety of reasons). If there is rework, the effort

spent on it should also be counted. Historical Productivity values are required for effort estimation and task scheduling; therefore if Productivity data have not been captured accurately, it can impact the ability to estimate effort and schedule testing activities in future projects.

Defect Detection Effectiveness

Defect Detection Effectiveness (DDE) is a measure of how effective the testing process is in detecting defects and not letting them pass through to the next stage in the software lifecycle. It is defined as shown below.

Metric	Definition	Calculation	Unit
Defect Detection Effectiveness (DDE)	Percentage of the total number of defects reported for the application that are reported during the testing stage	= (Number of Defects Reported during testing and Accepted) / (Number of Defects Reported during testing and Accepted + Number of Defects Reported after the testing stage and Accepted) x 100	%

In my opinion, this is the single most important metric for software testing effectiveness at an organizational level. Since it requires data from downstream stages in the lifecycle for its calculation, it is a lagging indicator. This means that you only get to know if there is a problem after the fact. It is still valuable because it can help fix problems in the process, which is beneficial for future testing efforts.

A low value for DDE indicates that testing has not been effective - too many defects have passed through undetected. This has a direct impact on customer satisfaction. Since resources and time are utilized for testing without producing adequate output, and because of the cost involved in fixing production defects, DDE is an indicator of the risk to profitable operations. A low DDE may point to competency gaps in the testing team.

Defect Acceptance Ratio

Defect Acceptance Ratio (DAR) tells you what percentage of the defects reported by the testing team turn out to be valid. Its key specifications are:

Metric	Definition	Calculation	Unit
Defect Acceptance Ratio	Percentage of defects reported that are accepted as valid	= (Number of Defects Accepted as Valid) / (Number of Defects Reported)	%

If the DAR value is too low, it means that testers are reporting too many defects that are invalid. This has a direct impact on Productivity. When an invalid defect is recorded, effort is wasted not only by the testing team but also by the development team as they have to process the defect record anyway and prove that it is invalid. (If very frequent, this may negatively affect the relationship between testers and developers, again adversely impacting the project.) A large number of invalid defects may clog the defect tracking system, which makes it hard to locate useful information and may add to the maintenance costs of the tool. A low DAR may be an indication that the defect tracking mechanism is not effective. For example, if it is not easy for testers to look up past defects, they may end up recording a large number of duplicate defects. In some cases, it may be an indicator of competency or knowledge gaps in the testing team.

Estimation Accuracy

Estimation Accuracy measures how closely the actual effort spent in testing tracks the effort estimated in the beginning:

Metric	Definition	Calculation	Unit
Estimation Accuracy	Ratio of estimated effort to the actual effort for testing	= (Estimated Effort) / (Actual Effort) x 100	%

In large projects, measuring Estimation Accuracy periodically will help in planning for the remaining tasks and making corrections if the existing plan, based on previous estimates, is found to be off track. Large inaccuracies in estimation can put the project at serious risk of failure. If this happens frequently in your organization, it tells you that there is a need to improve on the estimation techniques used.

Deviations in Schedule Variation (see below) may be explained by Estimation Accuracy, since an accurate estimation is a pre-requisite for building a realistic schedule. Wrong estimates can lead to schedule issues, which in turn lead to customer dissatisfaction. Projects that have been under-estimated may also see employee dissatisfaction as people will have to work extra hours to meet delivery commitments. It may also point to competency gaps in the team.

Schedule Variation

Schedule Variation provides a measure of the deviation of the actual duration of testing activities from the planned duration.

Metric	Definition	Calculation	Unit
Schedule Variation	Difference between planned and actual durations of testing in the project, expressed as a percentage of actual duration	= ((Actual Duration) - (Planned Duration)) / (Planned Duration) x 100	%

Even if the effort has been estimated right, if the testing schedule is not realistic, the project is at risk. How well the team is able to track to the schedule tells us whether the schedule has been built right and whether the Test Manager is able to control the influencing factors adequately in order to keep the testing activities on track. It may also point to competency gaps in the team. Monitoring Schedule Variation early on will indicate whether there is a risk of not having enough time for testing in the end. It helps in the causal analysis of

Test Coverage issues. Just as for Estimation Accuracy, a large Schedule Variation may also be a leading indicator for customer and employee satisfaction (or dissatisfaction!)

Defect Density

All of the metrics we have discussed so far are indicators of the effectiveness or efficiency of the testing or test planning process. Defect Density, or the count of defects per unit size of software, on the other hand is a very important measure of the quality of the software itself. It is specified as follows:

Metric	Definition	Calculation	Unit
Defect Density	Number of defects reported and accepted per unit size of the software	= (Number of Defects Reported and Accepted) / (Size of Software Tested)	/FP, / KLOC*

* FP – Function Points, KLOC – Kilo Lines of Code

Much more than the absolute count of defects in the software, it is Defect Density that is a true and universal measure of software quality because it normalizes software for size. That is, it allows for software of different sizes to be compared for quality.

If Defect Density is large, there is the risk that the project may slip on the schedule because of the time taken to fix the defects. It may be indicative of competency gaps in the development team. A very large Defect Density may indicate problems with the design, which are usually much harder to fix than programming errors and can put the project at serious risk of failure.

Analyzing Defect Density by functional area is an effective way of pin-pointing quality issues in specific parts of the software and facilitates defect prevention and corrective measures. It also helps in prioritizing or re-prioritizing the remaining tests, since there is an axiom in software engineering that defects tend to concentrate in particular areas – the higher the Defect Density in a particular functionality, the higher the chances of there being even more defects. A high Defect Density in the development lifecycle may indicate that the possibility of defects manifesting in production is also high.

However, this is not an easy metric to use, primarily because of the difficulties involved in measuring software size, which is in fact one of the most challenging problems in software metrics implementation. Another reason is that in order to make perfect sense, defect counts have to be weighed by severity, and there is no easy way to do it. (This applies to all metrics that use defect counts.)

Conclusion

We have seen how a small set of carefully chosen metrics can provide useful insights into the health of testing and the quality of software, and how to meaningfully interpret the information they convey. Given the constraints of space, this is by no means an exhaustive discussion. For instance, there are other metrics such as Defect Age that are highly useful, and there is a lot to be discussed about applying statistical techniques to analyze distributions and trends in metrics data and correlations between them, which are required to unleash the full power of your metrics program.

Now, give the title of this article careful consideration. The message is: Go lean on your software testing metrics, because then, you can go lean on them to manage your testing activities!

Biography

Jayakrishnan Nair is Director of Quality Services with UST Global® (www.ust-global.com), a leading end-to-end IT and BPO services provider for Global 2000 companies. In this role, he has had the opportunity to provide software testing and quality assurance consulting services and solutions for some of the largest corporations in the world across multiple industry verticals. He also heads the Software Engineering Process Group (SEPG), which drives the institutionalization and continuous improvement of software engineering processes across the company.

Prior to joining UST Global in 2000, he was working in the aerospace industry for 9 years, where he built extensive expertise in quality and reliability engineering of electronic devices, development of automated test equipment and qualification of manufacturing facilities.

He is based in the New York metro area. He can be reached at jayakrishnan.nair@ust-global.com.

OZ Agile Days

2010
November 22–24
Sydney, Australia

November 22

Tutorials

Patterns for Improved Customer Interaction & Influence Strategies for Practitioners
by Linda Rising

A Tester's Guide to Navigating an Iteration
by Janet Gregory

Principles of Agile Methods
by Leanne Howard

Managing Testing in Agile Projects
by Stuart Reid

OZ Agile Days is for all IT professionals and software testers interested in learning about Agile, or just staying ahead of the curve.

Learn from the best. Over three thought-provoking days, explore new developments and key concepts as local and internationally recognised authorities share their unique experience in Agile – including North American experts Janet Gregory and Linda Rising and UK expert Stuart Reid.

Please have a look at the program at **www.ozagile.com** and enjoy the conference!

Supporting Organisations:

Díaz Hilterscheid

Planit Software Testing

iSQI International Software Quality Institute

Agile Record

testing experience

A Díaz & Hilterscheid and Planit Conference

Conference Managed by ICE Australia

November 23 Conference

Start	End	Track 1	Track 2	Vendor Track
08:30	08:50	Registration		
08:50	09:00	Conference Opening		
09:00	10:30	Keynote: About Learning **Janet Gregory, Co-author of Agile Testing**		
10:30	11:00	Morning Tea		
11:00	12:00	Waterfall vs. Agile – As a Tester Are They Really that Different? Is it All in the Mind? **Andrew Hammond**	User Stories Are a Testers Best Friend **Leanne Howard, Planit**	Vendor 1
12:00	01:00	Moving a Major Project From Waterfall Methodology to Agile – A Testing Experience **Linda Brunker**	Stay Agile with Model-Based Testing **Jelle Calsbeek**	Vendor 2
01:00	02:00	Lunch		
02:00	03:00	Agile Methodologies and SoX Requirements **Saurabh Chharia**	It's Really Hard to Assess Soft Skills **Werner Lieblang**	Vendor 3
03:00	03:30	Afternoon Tea		
03:30	05:00	Keynote: *TBD* **Shane Parkinson, Planit**		

November 24 Conference

Start	End	Track 1	Track 2	Vendor Track
09:00	10:30	Keynote: Agile Testing Certification – How Could That Be Useful? **Stuart Reid, CTO, Testing Solutions Group**		
10:30	11:00	Morning Tea		
11:00	12:00	Zen and the Art of Agile Testing or Being Agile without Doing Agile **Matt Mansell**	Rapid-Result Tools for Test Automation: Code Generators – A Good Solution for Agile? **Mark Feldman**	Vendor 4
12:00	01:00	The Creation of an Agile Vendor **Mitch Denny**	Agile Tester: A Profession with Profile **Stephan Goericke**	Vendor 5
01:00	02:00	Lunch		
02:00	03:00	Panel Session		
03:00	03:30	Afternoon Tea		
03:30	05:00	Keynote: Deception and Estimation: How We Fool Ourselves **Linda Rising, PHD in Object-based design metrics**		

A programmers' nightmare and the goal to write bug-free software

by Stefan David

The Nightmare

It was one of those days, I got to bed at around 3am and it was one of those warm summer nights out there. I was sweating and could not manage to fall asleep, although I was tired and exhausted. There was the nightmare again: We spent the whole evening chasing bugs and it became a nightshift ..., as it does so often when we feel the cold breath of the project deadline coming closer and closer in our necks. Tomorrow, I am sure of that, we will succeed and we will find this nasty little sporadic bug that we weren't able to reproduce yet! Because we are good, we are probably a bunch of the best-embedded software developers the whole company has ever seen. But what if we can't? Well, at least we have tried everything, really: We did execute our unit and integration tests and performed static analysis, checked our coding guidelines and conducted code reviews as well. We even achieved 100% decision coverage[1], so we know that we structurally tested the code very well and besides other metrics, cyclomatic complexity[2] measures are far better than the average. Moreover, I can tell you, those were not easy to achieve. It took us a lot of time putting our good relation with the QA people on the line!

So, taking all those activities and metrics we know, we've achieved great quality and to be honest, it's probably not so bad either! It is just a little sporadic bug, causing the system to crash in operation may be only once in a year per item we sell. Once a year? This is nothing, and probably it will occur even less often! Anyway, everybody knows that Software cannot be bug-free. Even the famous Edsger W. Dijkstra[3] said "Program testing can be used to show the presence of bugs, but never to show their absence!" (1)

Damn, why can't I sleep well then, with all these recurring nightmares?

Software and confidence

This is just a little story, but honestly it could have certainly happened in this or a similar way. Most of the above points are still valid for many software projects. The activities and metrics mentioned are well introduced in the software business. Nonetheless there's obviously still a lack of confidence in our software that doesn't let us sleep well as developers, simply because we still can't be sure that our code won't fail during execution. As users exposed to software, which is undoubtedly almost everybody these days, this is not really satisfying either. Specifically when it comes to software operated in safety related systems such as cars, airplanes or medical equipment. The point is, everybody would like to have the confidence that the software one is using day to day has been tested the best possible way.

So what about metrics that are able to provide confidence in software? Are there metrics that can measure confidence? Well I guess not, but is there a metrics indicating correctness with the goal to write bug-free code that at least will give us some confidence?

Measuring code correctness

In theoretical computer science, correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification (2). In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of algorithms underlying a system with respect to a certain formal specification or property, using formal methods (3).

Formal methods application example: Abstract Interpretation

Over the past years, tools based on abstract interpretation have been widely adopted specifically in the embedded systems domain. In computer science, abstract interpretation is a theory of sound approximation of the semantics of computer programs. It can be viewed as a partial execution of a computer program that gains information about its semantics (e.g. control structure, flow of information) without performing all the calculations. This approach can be used to prove certain properties of software, e.g. prove that the software will or will not exhibit certain run-time errors (4), (5). Run-time errors are latent faults such as NULL-pointer de-reference, out of bounds array access, access to non-initialized data, division by zero, overflow, etc. which may lead to crashes, inconsistent data and in general to undefined or random systems behavior which might impact reliability and safety.

Abstraction simplifies problems up to problems amenable to efficient automatic solutions. One crucial requirement is to diminish precision so as to make problems manageable while still retaining enough precision for answering the important questions,

1 ***Decision Coverage*** *is a form of code coverage. It describes the degree to which the source code of a program has been tested based on the question if each control structure (e.g. IF) has evaluated both to true and false?*

2 ***Cyclomatic complexity*** *is a software metric developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program.*

3 ***Edsger W. Dijkstra*** *(1930-05-11 – 2002-08-06) was a Dutch computer scientist, and winner of the 1972 Turing Award.*

such as „may the program crash?“ (4).

The following exercise demonstrates the effectiveness of abstract interpretation. Consider the function:

```
1       int where_are_errors(int input)
2       {
3       int x, y, k;
4
5       k = input / 100;
6       x = 2;
7       y = k + 5;
8       while (x < 10)
9               {
10              x++;
11              y = y + 3;
12              }
13
14      if ((3*k + 100) > 43)
15              {
16              y++;
17              x = x / (x - y);
18              }
19
20      return x;
21      }
```

The goal is to identify run time errors in the function where_are_errors(). The function performs various mathematical computations, includes a while loop, and an IF statement. Note that all variables are used. On line 17 a potential divide by zero could occur if x=y. Now the question is: Given the control structures and mathematical operations on x and y, can we have x=y?

As shown in Fig 1, for the code underlined in green, an abstract interpretation based tool[4] has been used and proven there are no run-time errors in this code. This is because line 17 is executed only when the condition (3*k + 100 > 43) evaluates to true. Moreover, since the value of y is dependent on that of k, the tool determines that at line 17, while x is equal to 10, y will always have a value greater than 10. As a result, there cannot be a divide-by-zero error at this line.

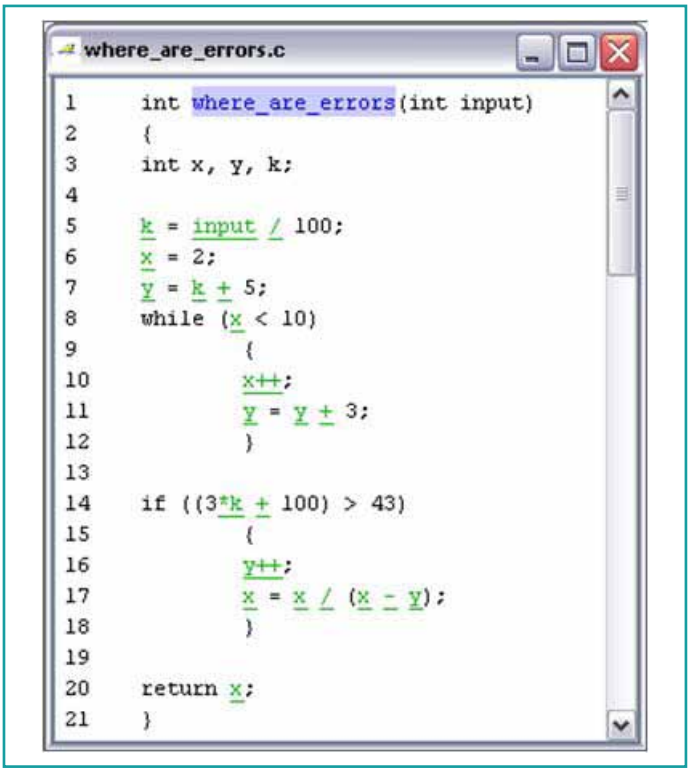


Fig 1: example of verification results highlighting correct code

This result is determined efficiently without the need to execute code, write test-cases, add instrumentation in source code, or debug the code. The tool also identifies all aspects of code that could potentially have a run-time error. These are underlined (see Fig 1). For this example, since there are no run-time errors, the underlined code shows up in green. For example, on line 17 the underlined green on the division '/' operator proves safety of this operation with respect to overflow as well as division by zero (6). Other colors are used to highlight proven violations, dead code and unproven states.

Abstract interpretation, reliability, and maturity

Now we need to establish the link from proving correctness of code to code quality. We want to take advantage of the software quality model ISO/IEC 9126 that provides a framework of metrics, quality characteristics and sub-characteristics, focusing on reliability of software. As stated in ISO/IEC 9126-2 a "reliability metric should be able to provide attributes related to the behaviors of the system of which the software is a part during execution to indicate the extent of reliability of the software in that system during operation." A "maturity metric should be able to measure such attributes as the software freedom from failures caused by faults existing in the software itself." (7)

This is also important, as maturity is a sub-characteristic of reliability according to ISO/IEC 9126. Tools based on abstract interpretation are able to generate metrics that, by default, provide "worst-case", i.e. program robustness measures. For supported run-time checks, these tools can verify existence or absence of run-time errors that would lead to unpredictable behavior of a

	A	B	C	D	E	F	G
1	RTE Statistics						
2	Check category	Check detail	R	O	Gy	Gr	% proved
3	OBAI	Out of Bounds Array Index	1	1	0	1	66,67%
4	NIVL	Uninitialized Local Variable	0	8	36	115	94,97%
5	IDP	Illegal Dereference of Pointer	1	2	2	9	85,71%
6	NIP	Uninitialized Pointer	0	0	2	16	100,00%
7	NIV	Uninitialized Variable	0	0	0	32	100,00%
8	IRV	Initialized Value Returned	0	0	6	34	100,00%
9	COR	Other Correctness Conditions	0	0	0	13	100,00%
10	ASRT	User Assertion Failure	1	6	12	0	68,42%
11	POW	Power Must Be Positive	0	0	0	0	N/A
12	ZDV	Division by Zero	0	1	5	15	95,24%
13	SHF	Shift Amount Within Bounds	0	0	0	0	N/A
14	OVFL	Overflow	0	9	21	37	86,57%
15	UNFL	Underflow	0	0	0	0	N/A
16	UOVFL	Underflow or Overflow	0	0	0	0	N/A
17	EXCP	Arithmetic Exceptions	0	0	0	0	N/A
18	NTC	Non Termination of Call	5	0	0	0	100,00%
19	k-NTC	Known Non Termination of Call	0	0	0	0	N/A
20	NTL	Non Termination of Loop	1	0	0	0	100,00%
21	UNR	Unreachable Code	0	0	6	0	100,00%
22	UNP	Uncalled Procedure	0	0	0	0	N/A
23	IPT	Inspection Point	0	0	0	0	N/A
24	OTH	other checks	0	0	0	0	N/A
25	EXC	Exception handling	0	0	0	0	N/A
26	OOP	Object Oriented Programming	0	0	0	0	N/A
27	CPP	C++	0	0	0	0	N/A
28	NNR	Non Null Receiver	0	0	0	0	N/A
29	FRV	Function Returns a Value	0	0	0	0	N/A
30	INF	Informative check	0	0	0	0	N/A
31	Total :		9	27	90	272	93,22%

Fig 2: example of metrics by run-time check category

4 *The commercial tool Polyspace generated metrics based on C/C++ for the displayed examples.*

program even under functionally unexpected conditions (see Fig 2). Using these metrics, to have valuable software measures for maturity (see Fig 3) and other characteristics such as portability, is the logical consequence (8).

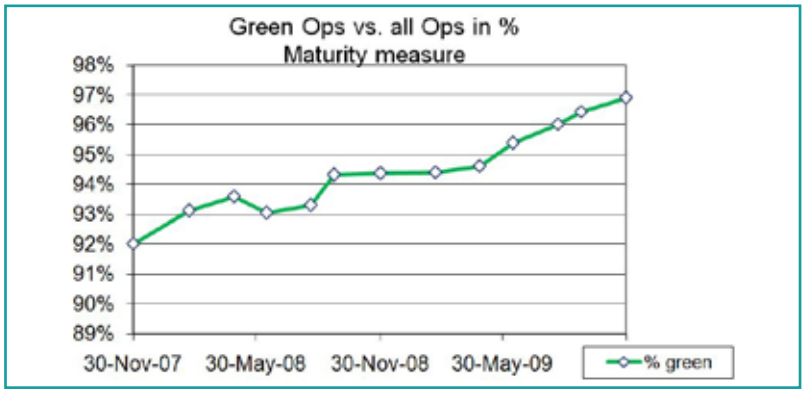


Fig 3: increase of run-time error free (green) operations over time

Proof -> Code correctness metrics -> Confidence -> Better sleep

Formal approaches like abstract interpretation may not be applicable for every piece of code. They address a subset of software errors and might not be able to solve all problems. However, they have proven their value over many years of industrial use, specifically in the embedded systems area and especially where quality or safety matters.

Abstract interpretation based tools can be applied at an early stage in the development cycle as soon as source code exists, providing metrics streamlining unit and integration tests as well as code inspection, reducing verification costs. This is the case because you simply do not need to spend time in verifying code e.g., through reviews or tests, already proven correct. You may also proceed with functional tests with a much higher confidence level, as you know the code is robust and mature.

For sure, this would have helped the software developer getting rid of his nightmares, helping him catch the sporadic bug he was chasing, adding more confidence so that he at least would have known how to spend his efforts more efficiently on unproven issues or other sources of errors.

References

[1] Dijkstra, Dahl, Hoare. Structured Programming. s.l. : Academic Press, 1972.

[2] Correctness. Wikipedia. [Online] July 1, 2010. [Cited: August 4, 2010.] http://en.wikipedia.org/wiki/Correctness.

[3] Program Verification. Wikipedia. [Online] July 7, 2010. [Cited: August 4, 2010.] http://en.wikipedia.org/wiki/Program_verification.

[4] Abstract interpretation. Wikipedia. [Online] June 1, 2010. [Cited: August 4, 2010.] http://en.wikipedia.org/wiki/Abstract_interpretation.

[5] Cousot. "Abstract Interpretation". s.l. : ACM, 1996.

[6] Sound Verification Techniques for Developing High-Integrity Medical Device Software. Abraham, Jetley, Jones. San Jose, California, USA : s.n., 2009. Embedded Systems Conference.

[7] ISO/IEC TR 9126-2:2003(E). Software engineering — Product quality — Part 2: External metrics. s.l. : ISO, July 1, 2003.

[8] Beyond Reliability: Measuring High Integrity Embedded Software Quality. David, Abraham, Chapple. Nuremberg, Germany : s.n., 2010. Embedded World.

Biography

Stefan David works as Application Engineer at MathWorks in the realm of Verification and Validation solutions since 2007. Before joining MathWorks, he worked for PolySpace Technologies for 4 years as consultant and trainer in the area of static and formal verification techniques for hand-written and automatically generated embedded software. He received his Dipl.-Ing. (BA) degree from the University of Cooperative education in Mannheim/Germany and a B.Sc. from the Open University/UK in Information Technology. He is ISTQB-certified tester and has several years of experience in software development and test.

EUROPE 2011

Testing & Finance

The Conference for Testing & Finance Professionals

May 9–10, 2011
in Bad Homburg (near Frankfurt am Main), Germany

The conference for Testing & Finance professionals includes speeches and field reports of professionals for professionals in the areas of software testing, new developments and processes. Futhermore there will be field reports for recent projects in financial institutions and theoretical speeches for regulatory reporting, risk- and profit based reporting.

Call for Papers

Infos at **www.testingfinance.com** or contact us at **info@testingfinance.com**.

The Call for Papers ends by midnight on September 30, 2010 (GMT+1).

Supported by:

Díaz Hilterscheid

testing experience

VOB SERVICE

A Díaz & Hilterscheid Conference

Quality Profiles - approach of efficient quantitatively management of software quality

by Yasna Milkova & Sergey Abramov

List below are some problems/potential problems related to quality that could be applicable for many software companies:

- The only one established quality criteria is acceptance criteria (it is not always enough to determine requirements to quality of the product)
- Lack of quality requirements for products or components purchased from third parties to implement in products
- Non functional requirements will not always be taken into account when determining the criteria of the release of the product, as a result there is a risk that they wouldn't be controlled properly.
- The quality of the product monitored only at the final stage of development.

Approach proposed in this article - Quality Profiles, help for an earlier stage of the project to develop requirements for quality, assess the expected quality level, forecast achievement / non-achievement of the delivered level of quality and, if necessary, will facilitate the timely development and implementation of corrective and / or preventive actions.

The aim of creating Quality Profiles is to describe the quality and its levels, as well as forming the basis of their profiles to ensure quality assessment, control and monitoring during all phases of the software life cycle.

Quality Profile's place in a software development lifecycle is demonstrated in figure 1.

Quality profile implementation consists of 3 main steps:

- Step 1. Determine metrics (quality attributes) that are important for the product
- Step 2. Determine target level (from 1 till 5) for each metric (quality attribute)
- Step 3. Monitor selected metrics (quality attributes). Execute corrective and preventive actions if necessary.

Step 1. Determine metrics (quality attributes) that are important for the product

According to recommendations of ISO 9126 standard there are 4 different groups of quality attributes:

- process quality attributes
- internal quality attributes of product
- external quality attributes of product
- product quality-in-use attributes

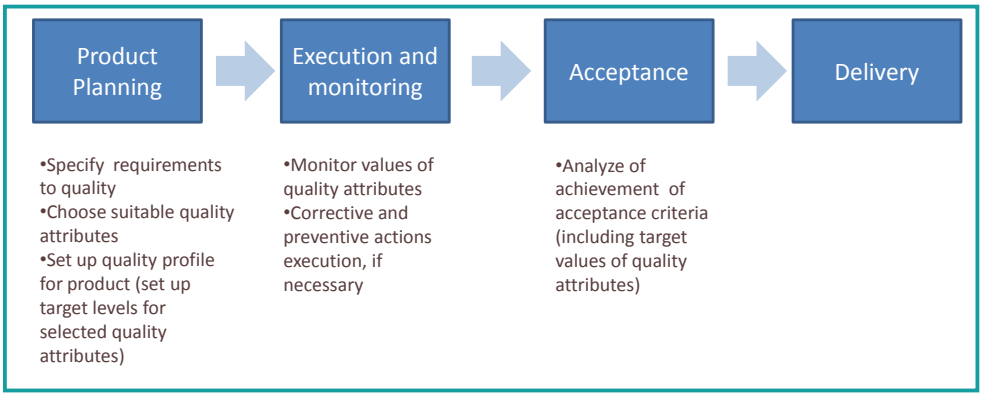


Fig.1: Quality Profile in a software development lifecycle

Process quality attributes characterize the main parameters of product development processes taking place within the company during product development.

Internal quality attributes are those attributes that are not visible to the end user, are used and can be elicited only by company specialists and influence the user's evaluation of product quality indirectly.

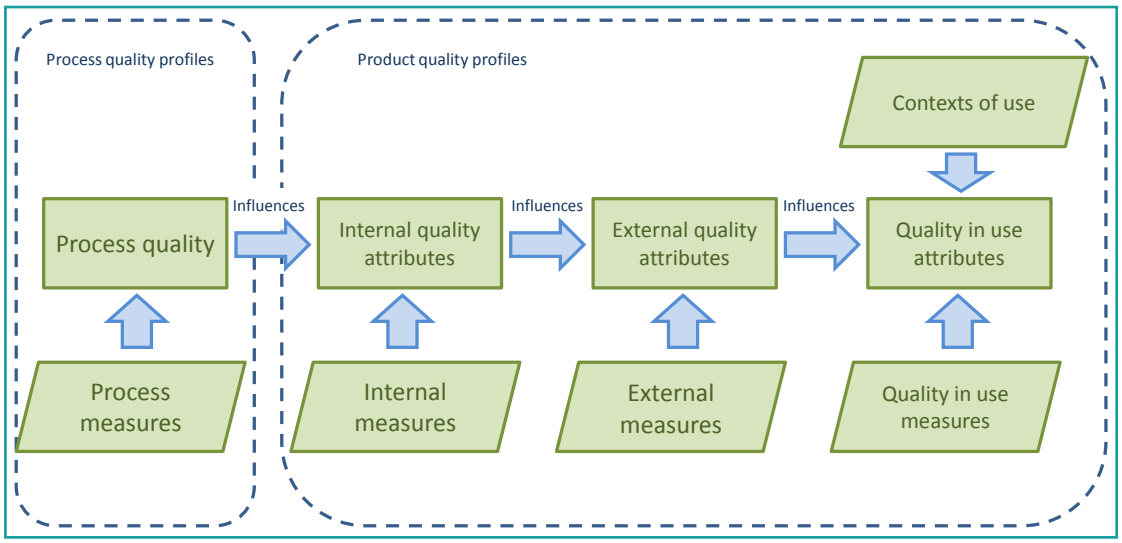


Fig.2: Quality attributes interaction

External quality attributes include those attributes that are visible to the end user and can be evaluated outside the company.

Quality-in-use attributes include those attributes that can be evaluated in the process of product usage by end users.

The figure 2 shows how different groups of quality attributes influence one another.

To ensure efficient monitoring for each group of attributes appropriate quality metrics should be developed.

Below example of 4 metrics related to each group of quality is represented (in real project you can indicate those metrics that would be important for your product and for your customer).

For each metric the following responsible role should be determined:

Quality attribute group	Quality attribute	Formula	How to use
Quality in use	Defect removal efficiency	DRE= (B/(B+C))*100% B -number of acknowledged defects found before the release (delivery) C -number of acknowledged defects found after release (delivery)	Help to evaluate quality of the delivered product
External quality	Operability on different configurations	ODC=(B/C)*100% B – number of configurations where application execute standard operations C – number of configurations where application should execute standard operations	Help to evaluate opportunities of the product to work on different configurations
Internal quality	Defect density	DD=(B/C)*1000 B – number of acknowledged defects found before the release (delivery) C – number of lines of added and modified code	Help to evaluate quality of the product before delivery
Process quality	Tested Requirements rate	TRR=(B/C)*100 B-number of requirements, test cases for with were run C -total number of requirements	Help to evaluate fullness of the testing and evaluate rate of requirements that were tested

Table 1

- responsible role(s) for the establishment of target value of metric
- responsible role(s) for the achievement of the target value of metric
- responsible role(s) for the measurement and calculation of the metric
- role(s) that carries risks in case of target value of metric would not be achieved

If at least one role is not determined for proposed metric, it should be additionally analyzed for the opportunity to eliminating from the Quality Profile.

Also analysis of efforts for measuring metrics should be performed. If measurement is not automatized and is required a lot of manual efforts than measurement would not be economically efficient. If metric is very important for the customer and very expensive for measurement, the company should analyze risks and cost of measurements and decide whether to measure it or not.

Thus it is recommended to include in the Quality Profile only those metrics that :

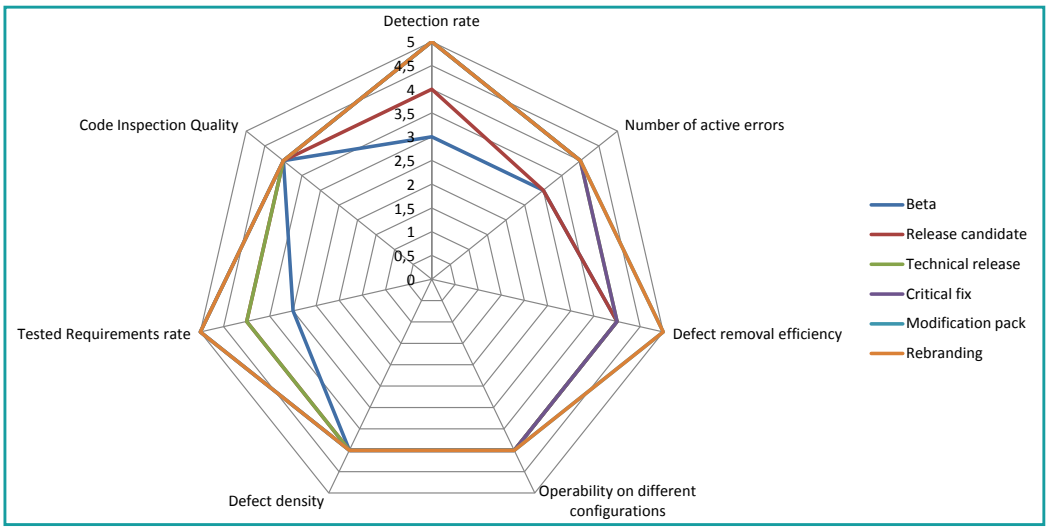


Fig.3: Quality Profile example

- are really important for the customer
- measurement of metrics is economically efficient (automatized measurement is preferred)
- all responsible roles are determined

Step 2.Determine target level (from 1 till 5) for each metric (quality attribute)

Target level for each metric should be established at the planning phase. According to the requirements for quality, different number of efforts could be planned (and spent) for the product development.

It is assumed that each indicator of quality should have 5 levels. An approach based on the definition of 5 levels helps managers to better control the target values, without going into details of the metrics. The first level indicates a low („bad") indicator value and the fifth level - the high („good") indicator value.

Example of target meaning of DRE for each level is represented in table 2

	Target meaning of DRE for each level
1 level	DRE<= 50%
2 level	50%< DRE <= 60%
3 level	60%< DRE>= 70%
4 level	70%< DRE>= 80%
5 level	80%< DRE>= 90%

Table 2. DRE target values for 5 levels

Common rules to determine the target value for a particular product could be as follows:

- target values of external quality attributes and quality-in-use attributes are developed by the head of the department who is the internal customer that ordered the product being developed
- target values of internal quality attributes and process quality attributes are developed by the head of the department who is responsible for managing the development of the products

Example of quality profile for particular product is represented on figure 3.

Step 3. Monitor selected metrics (quality attributes).

All metrics listed on step 1 are interconnected to each other. It allows execution monitor process quality and internal quality and predict weather external quality and quality in use would be achieved or not, at the earlier stage of the project. Monitoring should be executed till the end of the project. In case of problems/potential problems appearing appropriate corrective/preventive actions should be executed.

The accomplishment of target attributes is controlled by employees who set the target values of them. In turn, at checkpoints they inform their supervisors about the progress made in accomplishing the target values of the quality profile.

Conclusion

Thus Quality Profiles helps understanding of the objectives to be achieved at any stage of the software life cycle. This would help to concentrate resources on the priorities that are important for this stage of the life cycle, under lack of resources pressure. Quality Profiles also could be used like an acceptance criteria and ensure delivery of high quality products.

Biography

Yasna Milkova is in the position of Senior Business Process Manager of Quality Control unit at Kaspersky Lab. She has more than 5 years experience in software improvement.
Key areas of experience:

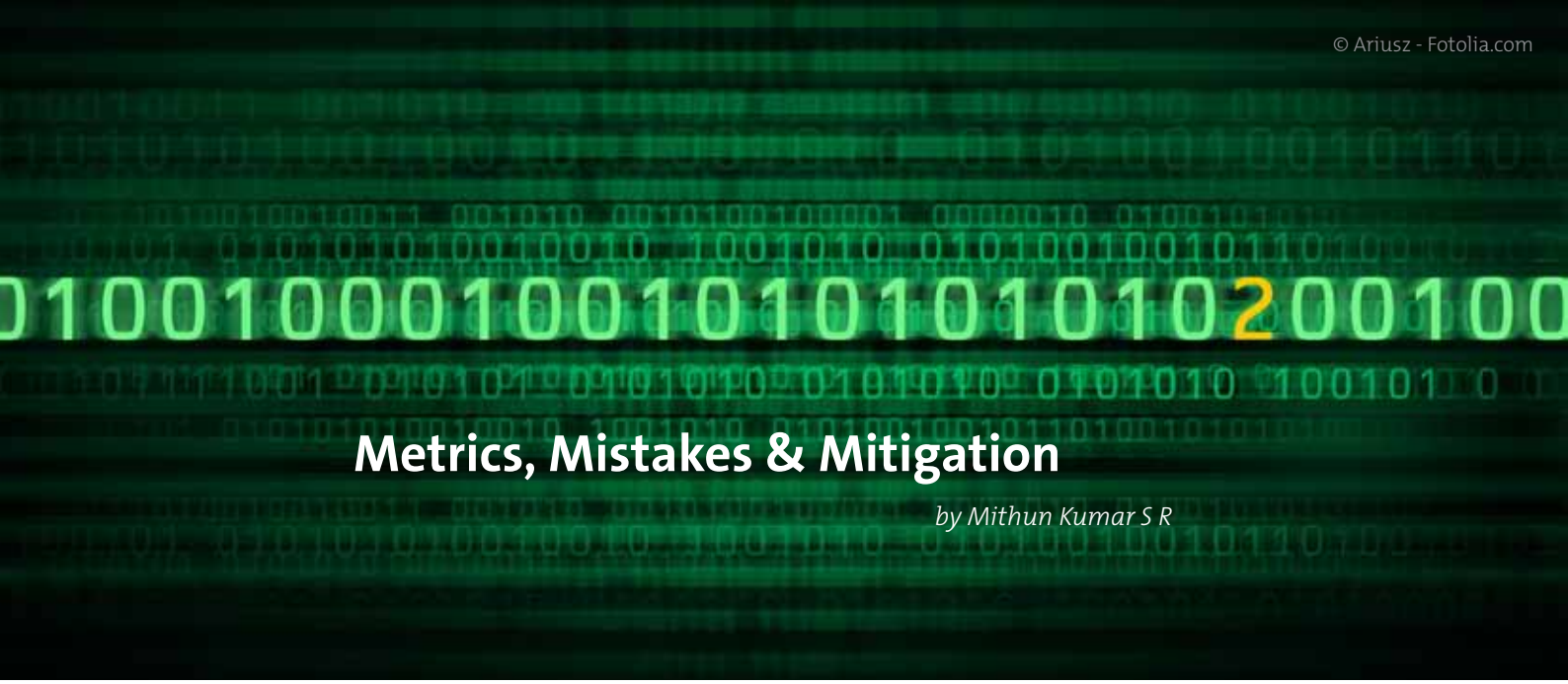
- Project measurements and analysis
- Project audits and consulting
- Process description and improvement
- Implementation practices of CMMI (including level 5) & participation in appraisal team
- Quality Management System implementation and improvement

Education- Moscow Institute of Electronic Engineering

Sergey Abramov is in the position of Head of Quality Assurance at Kaspersky Lab. He has more than 10 years experience in software improvement and testing
Key areas of experience:

- Managing quality of processes, guidelines, tools, metrics
- Manage testing
- Creating test requirements, test cases, test reports
- Creating performance & stress testing methodic
- Integrated test management system & auto testing system

Education- Moscow state institute of Electronics & Mathematics



Metrics, Mistakes & Mitigation

by Mithun Kumar S R

Amongst the myriad available metrics, only a few are suitable for projects. But even these fail to guide the project due to the mistakes committed either in its selection, evaluation or modification. Let us look into the common mistakes committed while using metrics and also ways and means to mitigate them.

Metrics derived using: Cyclomatic Complexity.

Mistake -> Same weight being assigned to nested and non-nested loops.

Mitigation -> Cyclomatic complexity Number (CCN) is measure of a program's structural complexity but not data complexity. It must be used with care as it may give a misleading figure with regard to frequency of simple comparisons and decision structures. Else a simple case statement can brand a method with high CCN value.

Metrics derived using: Test case Execution

Mistake -> Tracking test progress only on the number of test cases executed over a time frame.

Mitigation -> This fails to represent the true project state because clustering of functions is common over many applications. Around 75% of the execution surrounds pre-requisite and test environment setup. Reaching this function cluster takes time. Many managers do not wish to consider the setup time as a milestone in testing. Testers and Reviewers can overcome the problem by spreading-even the functionality across test cases rather than concentrating it in the last few test cases. Projects can also consider assigning proportionate weight according to complexity of the program.

Metrics derived using: Defects

Mistake -> Counting defects right from Unit Testing (UT).

Mitigation -> Defects per KLOC considered from UT phase may not yield desired output for monitoring due to constant change of the application, especially with low level of independence, where defects are fixed by the developer on the run-time. Save Defect count for Test Levels above and including Integration Testing. Unit Level can be better envisioned with KLOC (or Function Points) executed Per Day.

Mistake -> Setting targets on number of defects to be found per unit time and linking personnel appraisal, awards and recognitions to the number of defects found.

Mitigation -> Defect target reduces the efficiency of testing and diverts the focus to unessential aspects rather than improving quality from Customer's view point. Worst of all would be the build-up of low morale within the team. Though defects can indicate that the product quality is not up to the mark, but never would it indicate the performance of the team. A more positive approach would be looking for improvisation in the "test techniques" by using this base metric, "Defect".

Mistake -> Wrong assignment of severity and criticality of the defects

Mitigation -> Managers are most often misguided in setting priority to solve the problem. For example a typo in the label on a GUI would probably be classified as "Cosmetic" defect and priority is set to low. But it may be the scenario where this defect may cause "Serious" damage to the user. Hence imparting end-to-end functionality view of the application to all the team members shall positively help in assignments of priority of solving defects and thereby increasing the efficiency.

Metrics derived using: Cost and Time

Mistake -> Not considering Static processes like reviews into the testing cost & time estimation.

Mitigation -> Most managers do realize that static processes play an important role where nearly 25% of the defects could be found, but still hold reluctant in its inclusion. Difficulty magnifies when re-estimation is not done even after realization. Estimation mostly uses ratios, like Test Cost to the Total cost and Actual cost to the Budgeted Cost, obtained from historical data. Projects would be better if breathing time for recovery process is planned well ahead.

Mistake -> Team members failing to report findings or issues to the managers within time causing overhead in recalculation.

Mitigation -> Educating Testers on the test process and time

frame for reporting would prove beneficial.

General Metrics derivation:

Mistake -> Multiple people tracking the test progress stepping into others' shoes.

Mitigation -> Age old saying, Too many cooks spoil the broth, holds good in metric evaluation too. Assign roles judiciously to the team members and ensure that they adhere to it.

Mistake -> Discarding or changing metrics too often.

Mitigation -> Do not conclude on the pros and cons of a metric just from a pilot run. Take time to analyze and reap benefits from any metric.

Mistake -> Including names and assigning blames in the Test Metric report.

Mitigation -> Such pointers breed hostility in the team. Let the report indicate lag or achievement of the test process, but not that of team.

Mistake -> Ignoring minor data which might be critical for metrics calculation.

Mitigation -> ‚*How does a project get to be a year late? One day at a time.*' Said Frederick Brooks in his landmark book, The Mythical Man-Month. Careful review of the changes plays an important role. However there are some challenging tradeoffs. The purpose is not to micromanage the details of the tester's job, but to help the test manager and the tester understand which tasks the tester is doing well and which not.

"*Take Chances, Make Mistakes. That's how you grow*" approach may not suit this risk-averse era. Have an extra eye on the intricate details in any metric. Learn from others' mistakes rather than committing one. Last Line; do not forget to attain the ultimate goal of knowing project health in course of juggling between different metrics.

Biography

Mithun Kumar S R works with Siemens Information Systems Limited on its Magnetic Resonance - Positron Emission Tomography (MR-PET) scanners. He earlier worked with Tech Mahindra Limited for a major Telecom project. An ISTQB certified Test Manager, he regularly coaches certification aspirants. Mithun has his Bachelors degree in Mechanical Engineering and is currently pursuing Masters in Business Laws from NLSIU.

Measuring Software Quality With Metrics

– Why Static And Dynamic Analysis Should Walk Hand-In-Hand.

by Bruno Kinoshita

In April of this year, the company that I work with, started a project with a new client to continue the development of a Workflow system. What the client wanted was to implement two new features and fix a performance issue in the system. The Commercial Department decided to call us, the Quality Assurance Department, to make a pre-assessment on the existing source code of the system. This assessment would help the technical staff to prepare a proposal of development and testing.

We had two premises though, the assessment had to be delivered In Two Weeks And It Would Not Be Possible To Set Up A Similar Environment To The One The Client Had In Production. Because Of These Facts We Decided On Making Only A Static Analysis Of The System's Source Code.

Static Analysis Is A Technique For Code Inspection. You Analyze Code Without Actually Running It. A Static Analysis Can Use Many Metrics To Measure The Quality Of The Source Code Like Cyclomatic Complexity, Rules Compliance, Coverage, Number Of Classes, Afferent Coupling, Among Many Others.

For The Sake Of Simplicity And Time We Picked Up Only Three Metrics: Cyclomatic Complexity, Rules Compliance And Coverage. Our Team Considered That These Metrics Could Provide Us With A Deeper Understanding Of The System's Code Actual Situation.

Cyclomatic Complexity

Cyclomatic Complexity (Cc) Is A Metric That Can Measure The Paths Through A Method. What It Basically Does Is Count 'If', 'For', 'While' And Other Statements That Can Control The Flow In The Program. This Metric Is Useful To Know When A Method Is Becoming Too Complex Or To Measure How Many Tests Are Required To Obtain More Coverage.

Every Method Begins With The Cc Value Of 1. For Each Control Statement Found We Sum Up 1 To The Cc Value. It Is A Good Practice To Divide The Method In Smaller Methods When Its Complexity Is More Than 10.

E.G.: The Following Code Has A Cc Value Of 3.

```
1. public void proceedCheckout(Integer roomId, In-
teger clientId, Double value) {
2.      if ( isRoomAvailable( roomId) ) {
3.              if ( isClientActive( clientId ) ) {
4.                      // ...
5.              }
6.       }
7.       throw new RuntimeException("...");
8. }
```

Rules Compliance

There are many tools that are able to scan source code and look for potential problems like possible bugs, anti design patterns, duplicated code, and other software engineering bad practices. The violations can then be classified in different categories according to its warning level (minor, major, critical, and so on).

e.g.: In the following example a Rules Compliance tool can detect that there is a bug on line 2. Can you see it? Writing a test case with the right parameters could give us 100% coverage, however we would still have a bogus code.

```
1. public void checkRoomAvailability(Integer id) {
2.      if ( id.intValue() > 0 && id != null ) {
3.              // ...
4.       }
5.      throw new RuntimeException("...");
6. }
```

Coverage

Coverage is the more test-oriented metric of the three that we chose. It measures how much of the source code is covered by tests. These tests can be unit tests, automated tests, manual tests or any other type of test. Usually this metric is obtained by (a) running tests using a tool that calculates how much of the source code was tested or by (b) running the tests on an instrumented version of the system.

In a static analysis scenario the only way to measure coverage is comparing the code used whilst testing and the whole system's source code, and then calculating coverage percentual.

e.g.: The following code is the corrected version of previous topic's example

```
1. public void checkRoomAvailability(Integer id) {
2.      if ( id != null && id.intValue() > 0 ) {
3.              // ...
4.      }
5.      throw new RuntimeException("...");
6. }
```

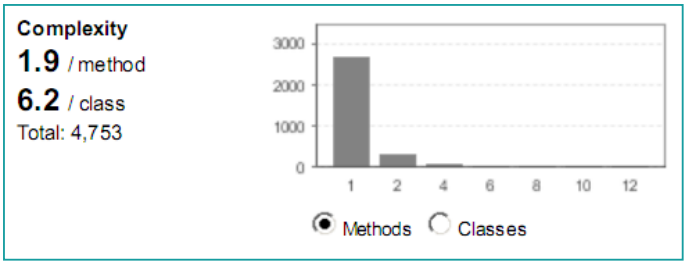
And the following unit test provides 100% coverage to this code.

```
1. public class RoomServicesTest extends TestCase
{
2.      public void testRoomAvailability()    {
3.              RoomServices t = new RoomSer-
vices();
4.                      t.checkRoomAvailability(new
Integer(10));
5.      }
6.      throw new RuntimeException("...");
7. }
```

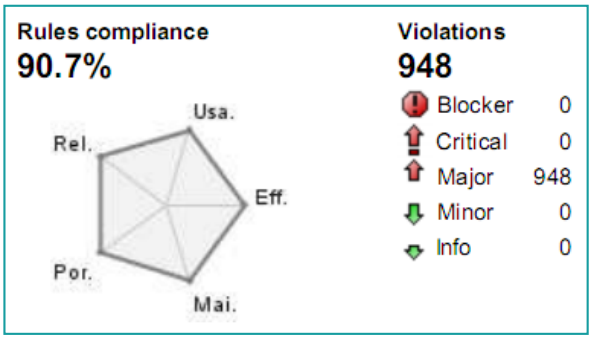
First Impression Of Static Analysis

"The first impression is the deepest" is an expression that denotes the importance of the first contact with a thing, a person or even with a source code. We had set up a test environment with the necessary tools to glean the metrics from the source code. These were the first results (and our first impression) of the Workflow System's source code.

Complexity:



Rules compliance:



Coverage:

This happened because of the numbers of getters and setters (with CC value of 1). These methods would have to be revised.

Because of the lack of time to choose an appropriate set of rules, we picked up only a few rules pointed by a group of specialists as fundamental (such as referencing a null object member). We didn't have any Blocker violation (the most critical violation type), a really good thing.

And lastly the worst point. The previous company didn't write any kind of test for the Workflow system. Any changes in the system had to be manually tested by the final user. We added planning tests as a requirement for the system.

Were Static Analysis Sufficient To Measure The Testing And Development Effort?

Although the static analysis assessment provided us some significant information, we missed an important point: Performance metrics, a type of dynamic analysis metric.

Once we started the development, things started to run down. As we implemented the new features in the system its performance started to degrade. Performance became an issue for developers and testers as both of them had to waste time waiting for the system to respond in some situations.

We had to deal with the issues on the project effort and schedule, putting more people to fix the performance issues before implementing the client's new features, and even paying extra hours for the programmers and testers.

We learned that there is no way of finding a relation between static analysis metrics (Cyclomatic Complexity, Rules Compliance and Coverage) and dynamic analysis metrics (System Performance, Unit Tests, Usability Tests, etc). The best way to have a good understanding on a new system's actual situation is with a good combination of static and dynamic metrics, good documentation, existing tests and a production-like environment.

Dynamic analysis would give us a better idea of what kind of issues we would find during the development and testing of new features in the system. Dynamic analysis includes executing tests on a running version of the application (instrumented or not) and measuring the performance, coverage and security.

Three months later we had to make a new quality assessment on this system before some new features were implemented. This time we already had an environment like the one the client had in production so we included dynamic analysis in this assessment.

Using performance metrics like response time and throughput, and profiling the system with a specialized tool showed us several issues in memory management in database access. This time we were able to measure our time better to develop and test what our client needed.

Biography

Bruno lives in Sao Paulo in Brazil. He is currently working as a consultant with Sysmap Solutions (www.sysmap.com.br).
Bruno has 7 years commercial IT experience, mainly in telecom services, with Claro, Tim and Vivo (mobile carriers from Brazil).
His experience covers java development, requirements analysis, systems architecture, test management and quality analysis.
Bruno holds SCJP, SCWCD, ITIL and Cobit certifications and is member of various Open Source projects such as Hudson and Crux.

Subscribe for the printed issue!

Please fax this form to +49 (0)30 74 76 28 99, send an e-mail to info@testingexperience.com or subscribe at www.testingexperience-shop.com:

Billing Adress

Company: _____

VAT ID: _____

First Name: _____

Last Name: _____

Street: _____

Post Code: _____

City, State: _____

Country: _____

Phone/Fax: _____

E-mail: _____

Delivery Address (if differs from the one above)

Company: _____

First Name: _____

Last Name: _____

Street: _____

Post Code: _____

City, State: _____

Country: _____

Phone/Fax: _____

E-mail: _____

Remarks: _____

1 year subscription

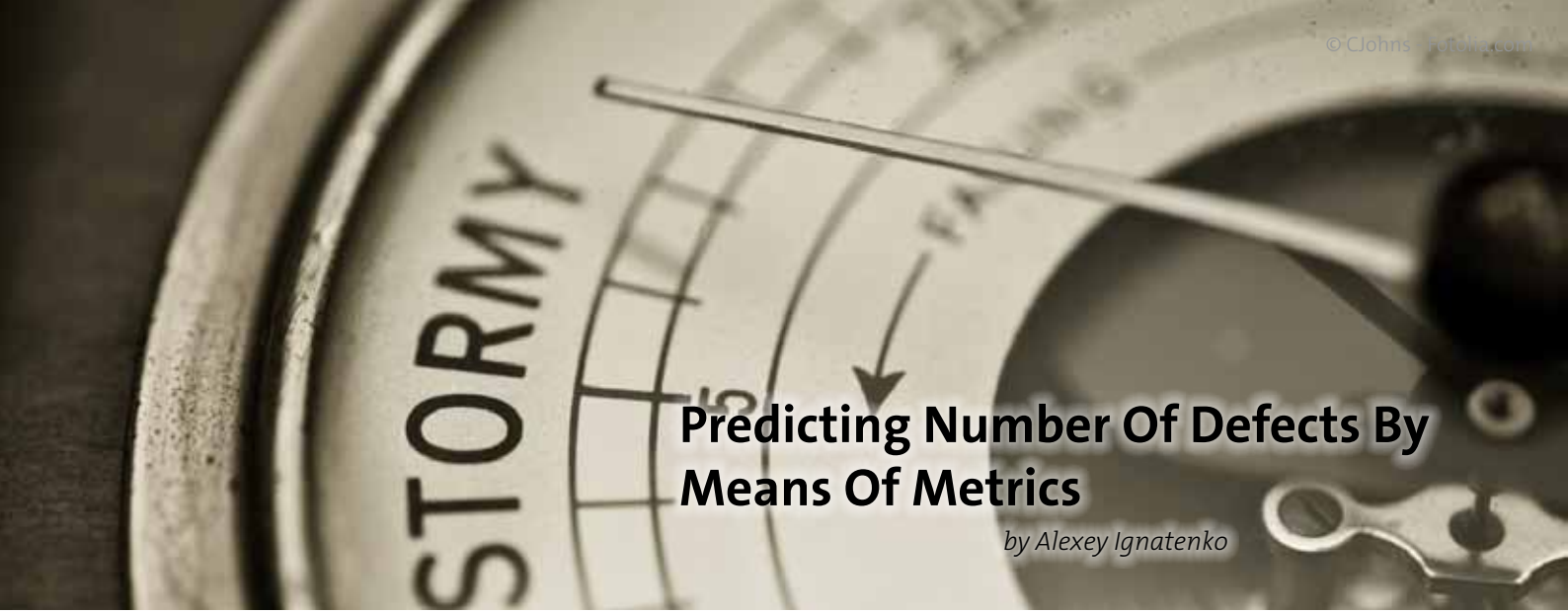
☐ **32,- €**
(plus VAT & shipping)

2 years subscription

☐ **60,- €**
(plus VAT & shipping)

Date

Signature, Company Stamp



Predicting Number Of Defects By Means Of Metrics

by Alexey Ignatenko

Most software projects develop in the same order of processes: requirements analysis, implementation, testing. In customer-centric software projects, testing process is carried out until all found defects are fixed. As soon as release date depends on this process, many projects cannot meet predetermined deadlines. That's why testing estimation is changing from being a percentage of development estimation to an independent methodology. Now, it is not very uncommon in the industry when a project manager calls a QA team leader, development team leader and asks:

1. How many defects will be found during the testing phase?
2. How much time will be needed to fix them all?

Without proper preparations it is either impossible to answer such questions or the answers to them lead to a wrongly counted release date.

It should be noted that the second question is much easier to answer because it is based on the answer to the first question multiplied by the average number of hours needed to fix one defect. In this article we will try to show how to answer the first question using the "degree of freedom" (DoF) metric.

DoF metric. The "degree of freedom" concept borrowed from physics where it means the minimum number of parameters necessary to describe a state of a system. In this article "degree of freedom" will mean the minimum number of cases needed to be verified in software by test team. The "minimum number" mentioned in this definition refers to equivalent partitioning of all possible cases that can potentially be verified. As soon as we don't expect more than one defect found in each equivalent class (otherwise we would split it into several classes), DoF equals to the maximum number of defects that can be found in software under test. In other words,

```
DoF = min{card(C)} = max{card(D)},    (1)
```

where `card(X)` – the number of elements in set `X` ("card" stands for cardinal number),

`C` – set of cases to be verified,

`D` – set of defects to be found.

Obviously, equality (1) can be satisfied only if the code has very poor quality. In all other cases there should be inequality: `DoF = max{card(D)} > card(D)`. To transform it back to equality it is needed to multiply DoF by some coefficient:

```
card(D) = (1-q)*DoF,    (2)
```

where `q (0 ≤ q ≤ 1)` will be called quality coefficient.

The bigger `q`, the better code's quality. Equality (1) can be satisfied only when q equals to zero (the worst quality of the code).

As follows from formula (2), to answer the question about how many defects will be found during testing it is needed to know two quantities – DoF metric and quality coefficient q. How to count both of them will be described in the following two paragraphs.

How to count DoF. As soon as `DoF = min{card(C)}`, counting DoF can be boiled down to counting test cases, but with some reservations. Broadly speaking, test case can check several things at a time, because this speeds up the testing process. For DoF it is not acceptable to count just the number of test cases, it is required to count each thing that is being verified by test case. DoF can be counted during requirements analysis. For each written requirement it is needed to count the minimum number of checks that must be performed to assure that the requirement is satisfied.

Let us consider simple examples that will help to understand the general approach to counting the degree of freedom. If it is needed to verify the influence of one input parameter on software feature, and all possible values of this parameter are split into five equivalent classes, this adds five points to the feature's DoF. Each independent parameter (meaning that it does not have influence on other parameters) adds the size of its equivalent partitioning to the feature's DoF. If there are groups of parameters that depend on each other, then equivalent class should be built for each group of dependent parameters, and the size of each class should be added to DoF as well. Likewise, any interaction with user interface should also be counted. Tooltips, buttons, lists, drop-down boxes, etc., their appearance and contents also add points to DoF in accordance to requirements and equivalent classes of checks built to verify these requirements.

How to count quality coefficient. Counting quality coefficient is quite straightforward provided we have properly gathered and supported historical data. Imagine project team had developed and/or released several features that were independent to some extent. Independence in this context means that these features were tested separately, so that it is possible to distinguish one feature from another by its testing artifacts. For each feature we need to know just two artifacts – DoF and the number of defects found during the testing phase. If we know them, we can com-

pute quality coefficient by means of least-squares method. Let's take a look at historical data from a real project in the table below (real features' names are omitted):

	Degree of freedom	Number of found defects
Feature 1	109	42
Feature 2	208	95
Feature 3	81	36
Feature 4	54	30

Table1 - Historical data about defects

By means of least-squares method we can obtain the following relationship between degree of freedom and the number of found defects: `card(D) = 0.44*DoF + 1.24`. As soon as intercept of this line is close to zero, relationship between DoF and card(D) can be considered unbiased: `card(D) = 0.44*DoF`. From this equation, quality coefficient `q = 1 - 0.44 = 0.56`.

With quality coefficient we can now predict the number of defects to be found in new features using formula (2) provided we counted DoF for these features.

Limitations and extensions. First limitation of this technique is that it is not applicable to very small pieces of software. If a new feature has a very small degree of freedom, it is unlikely to obtain accurate results when computing number of defects to be found in it.

Second limitation is that quality coefficient is the characteristic of particular developer, because each developer makes one mistake in every N lines of code and this number of lines N is different for each developer. This means that when computing quality coefficient, Table1 should contain features implemented by particular developer. Furthermore, when counting the number of defects to be found in a new feature, it is needed to use quality coefficient of the developer who will be implementing this new feature. Otherwise, obtained values will be inaccurate. In case we don't have enough historical data to create representative statistics for any given developer, quality coefficient computing technique can be extended by means of seniority multipliers. Let's assume that a middle developer makes half again as many defects as a senior developer, whereas a junior developer makes half again as many defects as a middle developer. Using these multipliers we can normalize data in Table1 by seniority level of the developer who will be implementing a new feature. For example, if Feature1 was implemented by a senior developer, but a new feature is going to be implemented by a middle developer, then the number of found defects in the third column of Table1 should by multiplied by 1.5 (42*1.5 = 63). And so on for each feature from Table1.

Third limitation is that features in Table1 should be similar to each other (by complexity, by programming language, by technology used, etc.). In other words they should be comparable. For example, features from Table1 are peer pieces of software at one layer interacting with another piece of software at subordinate layer. Or they are different workflows executed by one process management framework, etc. If we don't have enough historical data for similar features, quality coefficient computing technique can be extended by means of feature adjustment factors. Please refer to function points analysis for more information about adjustment factors. This is out of scope of this article.

Summary. The technique described in this article can be boiled down to the following six steps:

1. Establish metrics gathering process to accumulate historical data for a particular project team. For each released feature store its DoF and the number of defects found in it.
2. For any new feature calculate its DoF, and select a set of similar features from the historical database.
3. For selected set of similar features normalize their number of defects by seniority level (and/or using feature adjustment factors) if needed.
4. Using normalized data, compute the quality coefficient by means of least-squares method.
5. Estimate the number of defects to be found in a new feature by means of formula (2).
6. Once new feature is released, save it's DoF and the real number of defects into historical database. Compare estimated number of defects with real, review/refine seniority multipliers and/or feature adjustment factors if needed.

Biography

Education:
In 1998 entered National Technical University in Kharkov, Ukraine. In 2002 earned bachelor degree in computer science. In 2004 earned master degree in system analysis and research, after that started post-graduate course on progressive information technologies. In 2006 earned another master degree in intellectual property.

Work Experience:
Started career in winter 2004 as QA engineer at Mega Business Software Ltd., later acquired position of configuration manager there. In autumn 2004 started lecturing part-time at the department of automated management systems in National Technical University. In 2006 started to work for DataSoft company as a technical support engineer. In December 2007 was employed by DataArt Solutions Inc. as QA manager, later started to play roles of configuration manager and project manager. In the beginning of 2010 was employed by TEAM International as lead QA engineer where working now.
Was engaged in lots of projects including games, recognition systems, CRM systems, data providers, content management systems, incident management systems, etc. Had a wide range of responsibilities, such as requirements analysis, software testing, documents review, configuration management, customers support, quality assurance, etc.

Website metrics

– what are they and why do business owners and marketers need them?

by Peter Schouwenaars

Business strategy, in almost all industries, needs to make use of website metrics. That consumers can find information about your products or services increasingly easily on the web been a truism for years, as the so-called "internet age" has endured. What, then, do strategists, marketers, sales staff – in fact, everyone involved in making your business a success, need to understand to maximise consumer satisfaction?

The answer begins with an understanding of how consumers use the web and what impact this can make on your brand. It ends with an understanding of the breadth of information that can be gleaned, and a careful narrowing of these options so that they apply to and serve your business objectives. There are different types of analytics, different sorts of metrics, measuring different characteristics of your website. Establishing an integrated view and selecting the tools to empower your staff and not drown them in data that is the "holy grail" of metrics.

First of all, how do consumers, prospects, competitors and the press interact with your brand on the web, and how does this guide marketing and business development professionals wanting useful metrics? Visits to your website are part of a patchwork of touch points that cover on and offline brand discovery. Offline, customers may browse in store before buying online after using the web to make price comparisons, or they may compare and browse online before making the trip to a physical retail outlet. Attitudes to completing transactions online differ within consumer groups, between age groups and depending on the product in question. This is the first reason why the engagement of metrics needs to be precisely fitted to your business objectives. If your customers prefer to buy in the retail store, your website probably ought to be geared (in usability, navigation, etc) towards an outcome other than a transaction – perhaps to completing an enquiry form, or simply to finding the right page (and staying on it for the appropriate amount of time to read) after few clicks (if that indicates ease of navigation in your context). Similar for a service company – you might be aiming at keeping the browsing party on your site for an appropriate amount of time to get them familiar with your services – but how many seconds browsing indicate interest, and how many confusion? Perhaps your website's main goal is support – in which case your chosen outcome might be the swift location of a phone number for your customer. The goals are precise, personal and specific. The behaviours that match these goals are unlikely to be transparently clear initially – less still the aspects of your site that make these behaviours happen. Website analysis tools can produce detailed, unwieldy, suffocating amounts of data, each type dealing with a different element. Ensuring that you receive the relevant information and find actionable insights for your board, marketing team or webmaster is key.

Most people's introduction to tracking of website-related data comes via a pure analytics tool that reports on website visitors, sources, crude source geographies and some behavioural trends like page abandonment. These tools, (from Google Analytics, Webtrends, Coremetrics), are the fundaments of starting to understand how interactions with your online presence happen. What they don't provide, however, is detailed performance-related insight. When a user abandons a site, is the reason something to do with visuals or usability, or does an object not render? Perhaps the button that would allow them to progress towards their goal falls below the fold on a certain browser/OS combination? These are site performance issues – the very things that performance monitoring tools pick up.

A holistic view of your web presence will encompass detailed performances metrics that, after devouring data, spit out a clear indication of precisely why your site goes wrong, slows down or otherwise fails to satisfy the end-user. This is part of an overall suite of proactive working practices - analysing your email marketing success or failure, assessing sources of traffic critically, linking online insights to offline behaviour, and consistently reviewing the goals that you've set for online activity, around which the usefulness of your metrics cling. To compete online, and tap into the latent knowledge waiting for the savvy business owner, it's clear that usage of some sort of metric is compelling. Why, then, do we select performance monitoring to focus on in this article? The answer is that its importance as part of this holistic view cannot be underestimated. Rocketing consumer expectations compel us to ensure high performance. Research shows that 40% of consumers will wait no more than three seconds for a web page to load before abandoning the site. That's a striking statistic – and one that sets a high bar for the performance of any website. Once we take this basic assumption on the part of consumers, and start contemplating whether we hit the mark considering

the variety of locations from which consumers access the site, the variety of hardware, the speeds of internet access, even the browser that they select from the increasing plethora...it suddenly becomes clear that a credible web presence is about far more than a website that looks great from where you're sitting right now. It's complicated – particularly because most of the nasty third party influences that detriment website speed lie in the space between your firewall and the customer. And that's before we start looking at mobile browsers, handsets, locations and networks. It's clear, if companies don't incorporate performance monitoring, customers will vote with their feet.

Our assumption throughout is this: websites and e-commerce applications are the engine rooms of modern business. Their importance, as part of a cross-channel tapestry, cannot be understated. Web applications that underpin operations, sales or simply operate as the facade of a business, its window on the world, directly or indirectly generate revenue. Badly performing websites damage the image of the company, may gobble resource as the call centre is flooded with complaints and effectively "slams the door" on the potential customer. So how do we stay within the magic three seconds website response rate, and blunt the potential damaging impact of third party issues on our website?

Sometimes simple tweaks to code infrastructure are all that's required, responding to events elsewhere in the website "food chain" that affect performance or fixing underlying fragilities in the website's architecture. Most in the market rely on an analytics tool, on the one hand, to give information consumer behaviour and expect details about website performance from the server side. This approach leaves some massive gaps in the knowledge gathered about the application. Knowing the steps taken by a user moving across a website sheds no light on how it was perceived by him. Investigating on the server-side the speed that the traffic is handled doesn't complete the picture either; as it doesn't tell the user how long it took to reach the server, or indeed whether the server was reached at all.

Brand and website owners need hard metrics about performance and user experience to improve their offerings - this is the data upon which decisions and corrections should be based, and helps improve repeatability of tests and test results. Software as a service solution that provide on-demand metrics and analytics for web applications, focused strongly on performance, are burgeoning in popularity.

Monitoring performance as well as functionality, from a user point of view, is best carried out when the tester is able to select equipment and geographical locations and check performance and functionality from this setup.

That ability to monitor performance and site availability from a dizzying array of geographies is highly valuable, allied with powerful, automated tools checking for faults in rendering across browsers, operating systems, screen resolution combinations and mobile devices. Accessing complex matrix of compatibility testing is also key as website performance and look is distinctly affected by the large and increasing number of potential combinations of hardware and software. Browser variation in particular is proliferating since IE stopped being bundled as the default browser with new Windows products and a "decision screen" put in place. As well as not loading in time, not loading correctly "slams the door" on the user – and if the "buy" button on a transactional website fails to render or is not immediately accessible without scrolling on a certain browser, then your conversion odds lengthen, with a clear potential impact on the bottom line.

Historic performance monitoring and metrics have been somewhat rudimentary: measuring how long it takes to connect to the server or download the content. Looking at these items critically – encompassing offsite redirects, root pages and the breakdown of each object on a page. The Holy Grail is identifying, moving or fixing the pesky item (the page object) that isn't rendering – or shift the all-important call-to-action button to ensure that it's above the fold. This is part of testing the complete user route, not the server side or analytics in isolation. Next to evaluating the elements that the user controls, the best tools also provide the necessary detail about the various pieces of third party content that impact on your application. Object tracking allows the collection of hard metrics on the content provided by, say, ad-servers or subcontracted elements of your application. This gives SLA peace of mind – as well as ensuring that you're upholding your commitments, there is a tool to verify that your partners are doing the same. Further, this tool offers the option of draw metric from real user visits as well as user behaviour deliberately generated for testing. Based on actual user visits, the tool draws information about the performance of the page. How fast did it load? Which elements hampered speed? What elements were loaded and visible at the point the user abandoned the site? Where fully automated user scripts can be set up to generate metrics concerning user traffic and other elements, completely controlled and giving you comparable and highly reliable measurements, then complete oversight is being approached. The combined results from those two approaches offer a unique insight in the performance of the application and the related user reaction.

Getting your own "house in order" via URL monitoring, transaction monitors and real-time tests, but more sustainable business improvement can be gleaned from mining the competitive intelligence that can be gleaned from the web. With server side monitoring and traditional analytics tools, it was not possible to glean hard data on how competitor sites performed. The "user route" approach of the new tools changes this as you interact with the site as if a member of the public. By simply setting up the benchmark monitoring facility, you can assess your site's performance compared to direct peers, and establish what weaknesses can be exploited in the competitor's offering. Just like "mystery shopping" by real-world retailers, sending moles into shops and identifying the good, the bad and the ugly of the store layout and associated customer experience, web monitoring can investigate the behaviour of your competitor's online store and allow you to refine, learn, glean and clinch that all-important sale. Website performance also affects search rank – again clear proof that tools and consequent tweaks of all types are part of a patchwork of active augmentations that bolster your online presence.

Acting on the findings of the metrics is key – today's performance monitoring tools can be set such that alerts are tuned to recognise availability failures. When these sound, they allow IT functions in businesses to identify and resolve issues before they impact the end-user. Developers and QA teams can use automated metrics and testing tools to minimise testing time and costs, particularly in regression rounds, whilst enjoying the confidence that the live phase of the web app's lifecycle will be fuss-free, with the produced item functioning as intended across all system combinations. The tool behind this cost-effective service and simple user interface is in fact a veritable powerhouse. Utilising the biggest and broadest performance network, the underlying principle is giving

the user confidence that they can monitor how the site appears to customers, wherever in the world those customers choose to access it.

On-demand tools of any kind should allow precise control over exactly the elements of website performance that matter to your business, allowing you to achieve delineated goals - whether those goals are increasing website "availability" percentages, encouraging longer stays on the site or driving customers to transaction completion. "Information overload" is a genuine concern – excessive exposure to metric data can paralyse marketers. These incredibly powerful tools put all this at your fingertips, but the right professional services are required to support the user in processing and analysing it effectively. That's where QA consultants come in – you might not have, and might not want, to dedicate resource to understanding the information and turning it into the actionable insights that will improve your bottom line. But without this analytical effort, the investment in metrics is wasted and an unintelligible string of data may be the output. So, which data do you need, and how do you access it?

Frankly, monitoring and metrics serve as safety nets for disaster as well as necessary informants of reflective, strategic thinking. Once you've set your baselines and defined what events will spark off monitoring "alarms", it's the lot of the company to decide how to respond to these alerts. Upon finding a site event that's defined as „at risk" or „critical", depending on your defined parameters, a status report digs more deeply into the source and symptoms to allow rapid response. These alerts might be triggered by response time failures, content matches, transaction failures, page object alerts, page inaccessibility and server unreachability. When the "siren" goes off, you can react before many of your customers do. The importance of this agility is paramount where the website is customer-facing in any way, transactional or informative. A bad website experience might be the first impression that your customer gets of your brand – or it might be the last as the savvy consumer votes with their mouse and visits a rival instead.

Biography

Peter Schouwenaars, software testing expert at Testronic Labs, has over a decade of experience providing independent testing services for consumer-oriented multimedia applications, websites and software. With years of experience managing accounts for international clients that span the entire European publishing market segment in education, websites, games and film, Peter is a source of excellent insight into QA processes of all kinds.

Testing iT

Hunting Bugs...Opening Business

Testing IT is a consultancy firm focusing on Software Testing Practices and Quality Assurance.

Testing IT Consulting

"...There is always a better way of doing things, and we will find it..."

Testing Processes Creation and Innovation, Equipping Testing Teams, Tool Integration, Mentorship, Assessments and Audits.

Testing IT University

"...Education and Experience is simply the soul of a Tester..."

ISTQB® Certification, other courses based on our Knowledge Base (TIBoK©), In-class and e-Learning Training... A different but effective way of training.

Testing IT Units

"...TEAM = Together Everyone Achieves More..."

To Be Responsible for the Testing Process Within the Life-Cycle of the Development.

Hunting Bugs... Opening Business

Information:

+52 55 5566-3535

http://www.testingit.com.mx

info@testingit.com.mx

mexico@hastqb.org

Paseo de la Reforma 107, int.102,
Col. Tabacalera, México, D.F., 06030

Selected Metrics in TPI Next model

by Magdalena Bełkot

Contents concluded in this article is based on "TPI® NEXT, Business Driven Test Process Improvement", by Sogeti, UTN Pubishers from the authors Gerrit de Vries, Ben Visser, Loek Wilhelmus, Alexander van Ewijk, Marcel van Oosterwijk and Bert Linker.

Assessing quality of testing process, important information can bring us a different kind of metrics. Metrics are used within the test process to estimate and coordinate the test process, to give motivation for the test process, to justify test advice and to compare systems or test processes.

TPI Next model introduce metrics depending on Maturity Level. Starting at an Initial level, a test process can develop form Controlled through Efficient towards Optimizing.

Initial level

The Initial level named as "Ad hoc activities" is the only Maturity level that does not contain any specific expectations and this Maturity level is automatically present in a test process. This level does not present any specific metrics. Comparison of projects or test processes is based on rather subjective observations.

Controlled level

At a Controlled level very important is record and store the required information. This level is expressed as "Doing the right things". The test process is effective in such a way that it enables earlier and better insight into the quality, allowing for timely corrective measures, less chance of delays caused by insufficient product quality. At this level metrics are defined and used to estimate and control the test project.

Below metrics appropriate for the Controlled level:

$$\text{1. Test execution rate} = \frac{\text{number of executed test cases}}{\text{number of specified test cases}}$$

This metric enables to monitor the test execution progress. It is important to keep this test phase as short as possible on the project.

2. Progress of test execution present number of executed test cases divided into:

- passed (successfully executed),
- failed (test case containing one or more defects),
- blocked (it is not possible to perform this test case).

These metrics give insight in the number of test cases that still need to be tested.

3. Test phase ratio – number of hours spent per test phase (preparation/specification and execution).

Collecting this metrics from previous projects support allocating appropriate resources for each planned test phase. For evaluated project bring information about realized plan to assumed.

4. Waiting time rate – idle time as percentage of the total test hours.

This metric show that testing processes is not only working hours spent on testing but also time of inaction. Waiting time rate help to plan realistic schedules for other projects.

Efficient level

The third level of maturity – **Efficient level** - is expressed as „Doing things the right way". On this level the different elements and aspects of the test process can be controlled and steered in such a way that cost and benefit are optimized. There is a balance between the effort necessary to gather, analyze and maintain the metrics and the actual result. Collecting metrics does not conflict with the progress and quality of the test process.

Some samples of metrics are:

1. Test basis stability – number of added, updated and removed test cases.

This metric gives an indication of the quality and stability of the test basis. The test process efficiency benefits if test case design

can be done first time right due to a stable test basis in sufficient quality.

2. Budget ratio – ratio between test costs and design and development costs.

The norm is: design : development : test = 2 : 5 : 3. This metric gives base to create a basic estimation for the required test effort within a project. If design of project takes 20 Man days, development 50 Man days we can expect that testing process will take 30 Man days. Having this measure we can propose a testing schedule.

3. Number of tests and retests indicates how many test and retests are necessary in order to test a part.

Optimizing level

Reaching **Optimizing level** means possibility to "continuously adapting to ever-changing circumstances". When the right activities are done in the right way, the focus should shift towards making sure it stays that way in future. Provided information on questions answered through metrics may result in new questions. The way metrics contribute to the information need is monitored. Changes in information need lead to new or optimization of metrics.

Some examples of metrics are:

$$DDP = \frac{\text{defects found per single test level}}{\text{the sum of defects found in all subsequent test level (including current level) + defects found in production}}$$

The Defect Detection Percentage (DDP) gives a measure of the effectiveness of the testing at a particular level in the testing cycle. It is calculated as a ratio.

For example, the DDP for level 1 of the testing process after the completion of level 2 is calculated as the number of defects found in level 1 divided by the number of defects found in level 1, level 2 and production. Suppose that 30 defects were found in level 1, 20 defects were found in level 2 and 10 defects in production. The DDP of level 1 after level 2 would be calculated as:

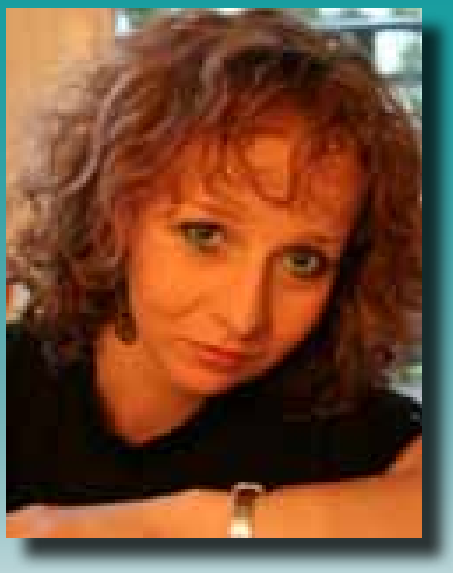
$$DDP = \frac{30}{30 + 20 + 10} = 50\%$$

This metrics can be used to optimize the test strategy and/or test case design.

Defect Introduction Rate - the rate at which defects are introduced, per lines of source code added or changed. It indicates the quality of submitted code, also needed for estimating defect density. By measuring the amount of code change in a package, we can use the defect introduction rate to estimate the number of new defects introduced into the code. It is usually calculated over a sufficiently long period (e.g. six months), so that defects existing at the start of the period (legacy defects) and defects remaining at the end can be ignored (or assumed to be the same). However, this measure can vary due to factors other than quality of submitted code, e.g. level of testing and level of legacy defects.

Conclusions

Every Test Manager has to know which metrics are the most appropriate. Using TPI Next model may be very helpful but a company can also choose metrics which are the most suitable for them.



Biography

Magdalena Bełkot is a Quality and Test Manager currently working at Allianz, where she has created a Test Department. Her experience gained in Ireland attending in testing new IT solution. She constantly improves her knowledge being part of many projects and attending in trainings and conferences. Privately she loves active relaxation like windsurfing, hiking and skying.

Coverage Criteria for Nondeterministic Systems

by David Faragó

This article describes how nondeterministic systems complicate the use of coverage criteria for functional testing. Firstly, it motivates that specifying the system under test (SUT) nondeterministically is often the only solution for complex systems, where the test harness cannot have complete control of the SUT. After showing how these systems are tested, the article explains the misdirection and difficulties of (code and specification) coverage metrics in this setting. It describes how the latest model-based testing (MBT) tools handle these difficulties and how coverage metrics can be extended.

Nondeterministic Specifications

For complex (e.g. multithreaded or even distributed) systems, lower levels, such as the operating system and network, are out of the tester's control and too complex to model and monitor. Therefore, the SUT (seemingly) behaves nondeterministically, i.e. reacts varyingly after applying a fixed stimulus. Prominent examples are race conditions and exceptions. The tester can cope with this by also using nondeterminism when specifying the SUT to cover all possible behaviors, for instance with the test `code if (USBException) {…} else {…}`.

Model-based Testing of Nondeterministic Systems

This article focuses on MBT with formal specifications, such as Labeled Transition Systems (LTS) or contracts, where nondeterminism can be used more concisely and also helps to model more efficiently by describing several behaviors without having to care which one occurs, e.g. what data is present in the underlying data base. Nondeterministic SUTs are modeled mainly by multiple outputs in one state. Alternatively, unobservable, so called internal transitions can be used.

MBT can automatically generate black box conformance tests from the reference behavior described in the formal specification. For this, MBT firstly traverses paths through the graph of the specification. These paths are considered as test cases: The SUT's inputs on the paths are the stimuli, i.e. drive the SUT, the outputs are the oracles, i.e. check that the SUT behaves conformly. If the test cases are too abstract for execution, they are translated to the technical level of the SUT in a second step. Thirdly, the tests are executed. Finally, the results are analyzed.

Older tools used *off-the-fly* MBT, which only performs the first two steps. Thus test execution and evaluation is done subsequently with classical tools. Since a priori test generation does not know which nondeterministic choices the SUT will take, all choices have to be considered. If many long branches have to be discarded because the SUT does not take them, off-the-fly is very inefficient. Newer tools support *on-the-fly* MBT, where all four steps are performed in lockstep. Hence the exploration and test generation can be guided by the observations made of the SUT for preceding stimuli.

Coverage Criteria for Nondeterministic Systems

In MBT, coverage metrics are not only used as termination criteria, but also to guide the traversal through the graph, such that the newly generated tests really help to reach the desired coverage. Since the formal specifications contain control flow, data and conditions, the same coverage criteria used on source code level can be applied, e.g. statement, transition, data-flow or MC/DC coverage. If the source code is present, it can also be used for coverage metrics. Which coverage criterion is best on the specification level varies, for instance MC/DC needs not be as powerful as is often expected (e.g. MUMCUT detects more fault classes, which is also the case for source code with many complex conditions).

Having nondeterministic SUTs, test segments must be executed repeatedly, such that the different behaviors can be probed. Since nondeterministic SUTs cannot be completely controlled by the tester, some behavior might (almost) never be reached. Hence coverage criteria must be considered with a grain of salt and a desired level of coverage might not at all be achievable.

Test purposes are automata that restrict the behavior that is explored and hence an alternative guidance technique. For some tools, they are as powerful as the functional specification and can also implement coverage criteria; other tools use specialized languages similar to regular expressions. Since the restrictions can be too strong, another verdict besides pass and fail indicates this, e.g. inconclusive.

Model-based Testing Tools for Nondeterministic Systems

This section introduces three powerful MBT tools, which all emerged from research. They all have their area of expertise and deficits.

TGV (Test Generation with Verification technology) from the French research consortium IRISA generates abstract test cases off-the-fly from test purposes and models that are variants of LTSs. To generate abstract test cases, TGV explores the model using depth-first search and performs abstraction and determinization. Tests are output in TTCN or graph formats of CADP and have the verdicts pass, fail or inconclusive.

Test selection can be guided by statement or transition coverage. Since these are often too weak, they can now be mixed with general test purposes. Additionally, more powerful coverage criteria are possible by using coverage directives, which are general expressions on specification variables whose reachable values must all be covered.

TGV is very effective when specific faults are targeted via test purposes. TGV has been used in several industrial case studies, e.g. on:

- a LOTOS specification of the cache coherency protocols for multiprocessor architectures of Bull (Polykid)
- an SDL specification of the SSCOP protocol of the ATM stack, standardized by ITU
- a UML specification of a component of the Transit Computerization Project for electronic exchange of information regarding goods in transit between EU countries.

TGV's weaknesses are the need to explicitly enumerate all required data values in the model and that it uses off-the-fly MBT.

TorX from the University of Twente performs primarily on-the-fly MBT; off-the-fly is possible via scripting. TorX explores the model randomly, which is effective for undirected but intensive testing. The random choices can be restricted using test purposes. They can be formulated as regular expressions or in LOTOS and be used for a coverage criterion based on trace distances, which heuristically limits the number of loop executions and outgoing transitions of a state. The generated abstract test cases have the verdicts {pass, fail} x {hit,miss}.

TorX has also been used in some industrial case studies:

- a communication protocol between video recorders and television sets for downloading channel presets
- the Lucent Technologies V5.1 access network protocol
- a mobile application from Interpay in the context of the Rekeningrijden project for automated fee charging for turnpike roads when a vehicle passes.

TorX chooses randomly amongst the (restricted) set of possible transitions, which causes weak guidance. More industrial case studies would give deeper insight to its practical capabilities.

Spec Explorer from Microsoft extends Visual Studio for modeling, visualizing and analyzing software behavior. The specification is written in C#. It focuses on off-the-fly testing and supports various unit testing formats. But an on-the-fly mode is also offered. It uses graph algorithms, game strategies and Markov Decision Processes to generate tests for optimizing the expected cost, reaching a given set of states efficiently, or for full state and transition coverage. Test segments are re-executed up to a fixed number of times for better coverage, but they need not be unwound explicitly in the automata. Spec Explorer also supports test purposes, formulated as regular expressions.

It is used broadly at Microsoft in projects of various sizes. In a large-scale project with over 300 test suites, an average productivity gain of 42% over manually-created test suites was measured, mainly because tests cases were created more quickly and requirement coverage was ensured.

Unfortunately, many of Spec Explorer's features are only available for off-the-fly testing. For on-the-fly testing, only random selection is possible in Spec Explorer. Hence the guidance in on-the-fly MBT is weak. Finally, nondeterminism has to be quantified, i.e. (non-rigid) probabilities, which the engineer often does not have, must be assigned to observable actions. The creators stated that adequate coverage and test sufficiency metrics must be developed.

Conclusion

Today's software development has become so complex that nondeterminism must be used. Research and industry has realized this and created testing tools that can generally handle nondeterministic SUTs. But the tools can still be improved by:

- Using better coverage criteria that incorporate nondeterminism, e.g. 1, 2, n, or all choices of nondeterministic branches must be covered. This could be handled easily with TGV's coverage directives.
- Using coverage criteria better: Since on-the-fly MBT currently chooses transitions randomly, coverage criteria can only vaguely help guidance. By using backtracking in the graph traversal, guidance can use coverage criteria much better, which results in fewer, shorter and more revealing tests (cf. the author's homepage for publications about this).

Biography

David Faragó is doing research at the Karlsruhe Institute of Technology, Germany, for the last three years. Within the last year, he's on a research project on MBT for nondeterministic SUTs (see http://lfm.iti.uni-karlsruhe.de/farago.php), funded by the Deutsche Forschungsgemeinschaft.
He has two years of industrial experience as test engineer at WIBU-SYSTEMS AG, where he also had to cope with the nondeterministic behavior of the Universal Serial Bus.
He holds a Master's degree (Dipl.-Inform.) in computer science from the University of Karlsruhe and is currently conducting his Ph.D. He welcomes your feedback at david.farago@kit.edu

Next issue December 2010

Topic Open Source Tools

A Systematic Way to Construct Meaningful Testing Metrics

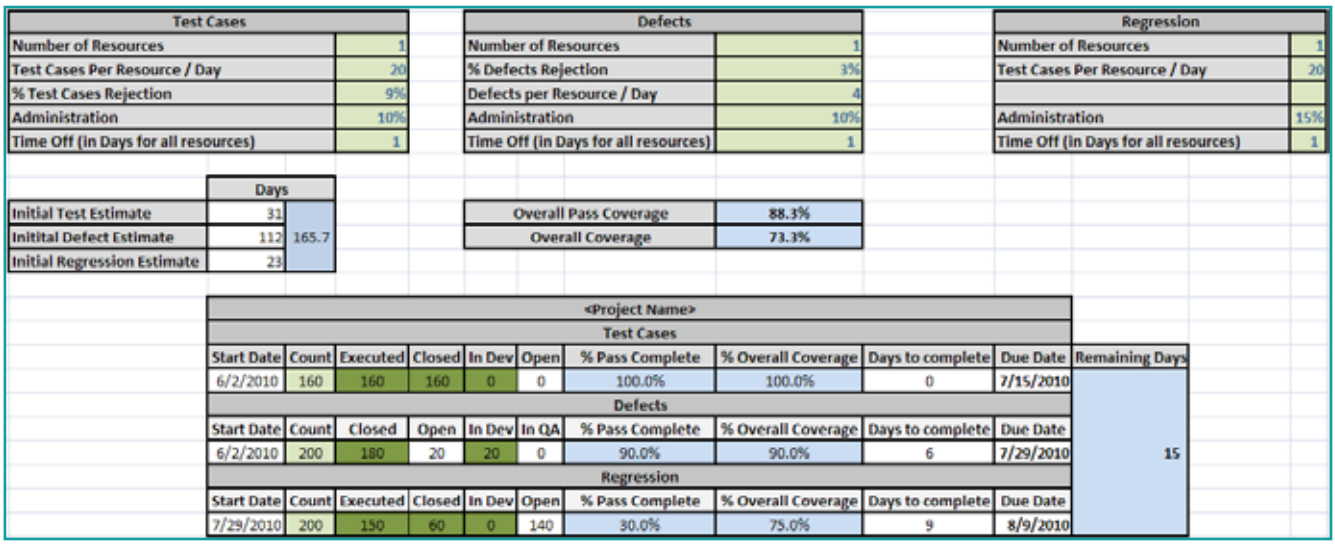
by Juan Pablo Chellew

Metrics, are they the crystal ball of software testing? In my twenty plus years of testing experience, I have not seen a metric that produces one hundred percent accurate or predictable results for software testing projects. During many years of research, designs and experiments, I have developed a predictable set of metrics that accomplishes my estimations and status report needs. Before I continue, I have to make it clear that metrics are only one of the many tools used in the daily planning activities throughout the testing cycles, and they provide invaluable results during the estimation and progress reporting activities. I am not saying that this method is an exact science, and you will need to keep in perspective that the results and accuracy of the metrics are as good as the quality of the data you work with. The more historical and the more accurate the data, the better the estimates and statuses you will be able to extract from these metrics.

Throughout my testing endeavors, I have constantly been asked to estimate the testing effort and have been questioned regularly about testing progress (Where are we? How is the quality? Are we behind or on schedule?). I never felt extremely confident with

<Project Name|>						Friday, July 09, 2010		
	#				#		Hrs.	
Number of Requirements	10	System Complexity	1	Total Test Cases	30	Expected Number Of Defect	0.9	
						Time to Close All Defects	0.45	Days 0
Functional								
Constant	Hrs.	Constants	Hrs.	Constant	Hrs.	Conctasnt	#	
Time to Close one Defect	0.5	Time to Execute One Test Case	1.0	Time to Create One Automated Script	2.5	Number of Manual Resources	1	
Av. Number of Test Cases/Req.	3.0	Time to Rewrite One Test Case	0.5	Time to Verify Script	1	Number of Automated Resources	1	
Av. Percentile of of Defects/Test Cases.	3%	Application Training Hours	0.0	Time to Update Automated Script	0.5	Number of Builds Expected	3	
Time to Wrtie a Test Case	2.0	Write Test Plan	16	Time to Execute an Automated Script	0.5	Time To Execute Smoke Test (hrs)	2	
Administration Time	10%			Avg. % of Requirement Changes	2%	Project Closure	1%	
Browser Testing								
Constants	#	Constants	Hrs.					
Number of Pages	0	Time to Test a Page	0		Hrs.			
Number of Links/Page	0	Time to Test a Link	0	Expected Number Of Defects	0			
Number of Browsers	0			Time to Close All Defects	0	Days	0	
OS Testing					Hrs.			
Constants	#	Constants	Hrs.	Expected Number Of Defects	0.0	Percent of Test cses to Automatye	80%	
Number of OS	0	Time to Test one OS	0	Time to Close All Defects	0.0			
				Total time to Test OS	0.0			
Tasks	Hrs.							
System		Days		Automation	Hrs	Days		
Application Training	0.0	0.0		Write Automated Test Plan	16	2.0		
Write System Test Plan	16.0	2.0		Write Automated Scripts	67	8.4		
Write Manual System Test Cases	60.0	7.5		Automated Test Execution	60.5	7.6		
Create Automated Test Case Scripts & Scenarios	83.0	10.4		Admin Work	14.35	1.8		
Execute **System** Test Scenarios	30.0	3.8		Total Automation Time	158	19.7		
Execute **Browser** Test Scenarios	0.0	0.0						
Execute **OS** Test Scenarios	0.0	0.0						
Test Cases Rework	0.6	0.1			Hrs		Days	
Administrative	17.4	2.2	26	Pre Execution Cycle Time	159.0	In Days	19.9	
Execute Smoke Tests	6.0	0.8						
Cycle 2	1.4	0.2		Cycle I Testing Time	48.0	In Days	6.0	
Cycle 3 (Regression)	12.0	1.5		Cycle II Testing Time	1.4	In Days	0.2	
Automation Test Execution	74.9	9.4		Cycle III Testing Time	12.0	In Days	1.5	
Project Closure	2.3	0.3	28					
				Cycle I, II and III Testing Time	61.4	In Days	7.7	
Actual Testing Hours								
	Hours	Days			Hrs	Actual Project Hours	Days	
Grand Total of QA Effort	303.5	38			380.5	Testing Duration	48	

Figure 1 – First testing metric



Test Cases	
Number of Resources	1
Test Cases Per Resource / Day	20
% Test Cases Rejection	9%
Administration	10%
Time Off (in Days for all resources)	1

Defects	
Number of Resources	1
% Defects Rejection	3%
Defects per Resource / Day	4
Administration	10%
Time Off (in Days for all resources)	1

Regression	
Number of Resources	1
Test Cases Per Resource / Day	20
Administration	15%
Time Off (in Days for all resources)	1

	Days	
Initial Test Estimate	31	
Initital Defect Estimate	112	165.7
Initial Regression Estimate	23	

Overall Pass Coverage	88.3%
Overall Coverage	73.3%

<Project Name>

Test Cases

Start Date	Count	Executed	Closed	In Dev	Open	% Pass Complete	% Overall Coverage	Days to complete	Due Date	Remaining Days
6/2/2010	160	160	160	0	0	100.0%	100.0%	0	7/15/2010	

Defects

Start Date	Count	Closed	Open	In Dev	In QA	% Pass Complete	% Overall Coverage	Days to complete	Due Date	
6/2/2010	200	180	20	20	0	90.0%	90.0%	6	7/29/2010	15

Regression

Start Date	Count	Executed	Closed	In Dev	Open	% Pass Complete	% Overall Coverage	Days to complete	Due Date	
7/29/2010	200	150	60	0	140	30.0%	75.0%	9	8/9/2010	

Figure (4) Reporting Metric

my responses; I always got a bad feeling about whether we were going to meet the proposed deadlines. I have come across plenty of information on software metrics that relates to complexity, process efficiency, process improvements and performance. On metrics, however, there is very little information that helps you to estimate, manage and control your testing efforts.

The lack of historical data and difficulties of accounting for all the activities that are needed to accurately estimate testing activities, including the unknowns, made this activity defective. Initially, I would look at the complexity and the number of enhancements for a particular release. Based on my experience I then came up with a "guesstimate" (sometimes I would get lucky and meet the deadlines, but that was the exception). Sometimes I would ask the team for their estimate input and then padded them with an additional 30%. The main idea of this activity was intended to make the testers accountable for their deliverables and make them feel closer to the decision-making process and part of the overall team. Since this process backfired most of the time, I found myself on a vicious cycle or having to repeatedly request additional time. The estimates were simply not close enough to reality. For this reason I later started using a widely used estimation process, where 40% of the development time was assigned to the testing effort. It did not take me too long to realize that this process was also not accurate, as the testing effort was based on a development estimate.

As the pressure increased to produce better and more accurate estimates, I began to create a list of activities and tasks performed during the testing cycles, and I included activities such as test plans, test cases, design reviews, test execution time, documenting defects, non-productive time (time off, meetings, etc.), ratios for requirements to test cases, defect rejection ratios, defect turn-around time (from developers), number of defects closed per tester, average number of defects found per requirement, training time, number of expected builds, the average number of test cases executed per tester, and the average number of defects found per test case. Basically, I just included every activity a tester would perform in the spreadsheet, including automation and performance tasks. The list grew until I had well over 100 items. When I was finished putting the tasks in the list, I incorporated the calculations for ratios, averages and actual numbers that I extracted from the defect management system. These ratios and averages were calculated using historical data taken from previous projects and from my defect management system, hence my historical data. Next I constructed a metric with my spreadsheet that captured all these items and built up the calculations to give me an estimate of the testing effort. At one point with the help of a Six Sigma "Black Belt", we ran the spreadsheet into a Six Sigma tool called Crystal Ball that tells you the accuracy of your calculations based on a number of tries and the inputs. The result was 75 to 85% on tens of thousands of runs (execution of the calculations). In Six Sigma, you look for 99.998% accuracy, but for me 85% accuracy was more than sufficient.

For the data points that were not readily available from previous projects or in the defect management system, I included a "guesstimate" and created either a process to capture the missing data or included a new field in the defect management system to capture the data for future use in my statistics and estimates. When I finally completed the metric and gathering of the data, I realized that I had constructed the "mother of all metrics". As I started using the metric (see figure 1 below), I realized that my estimating formulas were so complex that the maintenance would be extremely hard and time consuming. Even though my estimates were very accurate, the time it took to keep the metrics up to date was not worth it. To make my life a bit easier, I decided to simplify my initial metric and I began by prioritizing the tasks list based on the following factors: by importance or relevance to testing, criticality, and tasks which are time consuming. Through this process I ended up with ten key activities and tasks that I felt must be taken into account during the construction of the metric. I broke down my data into three basic but essential categories: test cases, defects and regression testing. For each of these categories I was able to identify the key activities and influences in the testing process. I then assigned to each category a numeric value extracted from my historical data and constructed the duration and percentage formulas to depict the estimates (values in the light green cells, see figure 2). As the testing progressed, I later updated only the dark green cells to produce progress percentages and remaining days to complete the testing (cells in light blue, see figure 2).

There are plenty of metrics that you can capture throughout your software testing development cycle (STDC). In fact, there are so many possibilities that you can easily get inundated and lost in your data. Managing your data and deciding what type of metrics you need to capture and maintain during your daily project activities is the key for successful estimations. If you do not have a well-established testing process, then you will need to start here.

QA Estimations for <Project Name>

QA Owner

Defects

Task	Days
Total Number of Defects	70
Expected Returns	5
Testing Days	3
Administration Time	0.3
Total Days to Test	3.4
Actual Days to close	3

Defects

Statistics	
Defect Return Rate (30%)	7%
Defects per Resource	4
QA Resources	6
Administration Time	10%

Average of Returns 10%

Enhancements

Task	Days
Number of Requirements	17
Number of Test Scenarios	51
Expected Returns	2
Test Cases Generation	5.7
Testing Days	4.4
Regression Testing	3.1
Administration Time	2.6
Total Days to Test	15.8
Actual Days to close	4

Enhancements

Statistics	
Defect Return Rate (30%)	12%
Test Scenarios per Requirement	3
Test Cases Executed per Day	3
Regression testing %	70%
Enhancements Executed per Resource	4
QA Resources	3
Administration Time	20%

Final Regression	Days
Regression Testing	5
Final Certification	4

Total Testing Days (Duration)	28

Figure (3) Basic Estimation Metric

In order to capture good-quality data for your metrics, you have to establish a standardized process that will allow you to capture data points. A standard and repeatable process allows you to capture normalized data points that can be used in your metrics system. A good place to start capturing your data points are your defect tracking system and your test case management system. From these tools you can extract data points, such as test case creation and execution time, defect ratios, defect verification time, testing time spent per requirement, averages for defect found per requirements, new defects found for a release, test case execution ratios and non-productive time, to name a few. If you are not yet capturing these types of data, I would strongly recommend starting this process as it will make your testing life much easier.

Currently, for quick or preliminary testing estimates, I use a subset of the metrics previously discussed (see figure 3), where the historical data integrity is maintained and kept up to date. I only use it to estimate the testing effort at the beginning of a project, when only raw data is available.

In summary, these tools are as accurate as the quality of the data we collect, and as in any software company the unknowns and variables will always present challenges to our estimates. If these challenges can somehow be represented in our metrics, then we can start to rely more effectively on what they are telling us and make the necessary changes and corrections to avoid slippages in delivery. Once you start tracking your data points throughout your releases, you can then start building confidence into your metrics, and the more historical data you collect, the more accurate your metrics will become.

As a good practice, always review your process and metrics data as often as possible, comparing your actual data with your estimates and making the necessary changes. Then re-evaluate and update your formulas, so your predictions become "certain".

Biography

Aristotle once said, "Quality is not an act, it is a habit." I believe it is critical to view Quality Assurance (QA) as a method of practice and habit, day in and day out. Juan Pablo Chellew has been leading Quality Assurance departments for the past 17 years. Pablo is a seasoned QA professional with more than 17 years of industry experience in software testing methodologies including white-box and black-box testing, performance, usability and process improvements. Pablo has worked in the Telecom, Billing, DB, eCommerce and Retail industries. For the past 6 years he has concentrated his efforts and research in the areas of lean development and metric-driven testing, and has attended training for Six Sigma (Green Belt) and ASQ (Certified Manager of Quality/Organizational Excellence).

Software Testing Automation Program ROI Metrics

by Parul Singhal

A lot has been written on the subject of software testing automation...evaluation criteria for projects to be chosen for automation, improving testing through automation, frameworks for automation, what leads to failure of automation programs, technicalities involved in automation testing, automation being a long term investment, and so on. However, very little information is available on what to measure to find out if the test automation program is indeed worth all the effort and cost spent!

I intend to fill this gap through this article. The article proposes and explains three powerful metrics to plan and evaluate the value of a software testing automation program.

As far as adoption of software testing automation is concerned, there are broadly two classes of organizations: (1) organizations that already have an automation program and (2) organizations that are planning for automation. For organizations in the former class, these metrics will help determine answers to common questions such as, "Which projects are having a smooth automation run?", "Which ones are going through bumps?", "Which ones are cost savers and which ones are not?", "What kind of benefits can be expected two/three/five years down the line?" and a lot more. For the latter, these metrics would help in planning the baselines for a successful automation program and also in maximizing the returns on software testing automation.

The metrics proposed are the following:

1. **ROI_{effort} metric** helps to determine if the effort spent running the automation program is worth the effort saved as a result of automation.
2. **Utilization Optimization Factor** metric helps evaluate if the automation tool used for automation program is optimally utilized.
3. **ROI_{cost} metric** helps evaluate if the cost spent running the automation program is worth the cost saved as a result of automation.

ROI_{effort}:

ROI_{effort} = (Effort saved as a result of test execution using automated scripts in hours) / (Effort involved in (Automation Scripts Development + Automated Script Execution + Automation Scripts Maintenance + Automation Training)) in hours

The numerator is calculated as follows:

(Effort for manual test execution of an application or project - Effort for automated test execution of an application or project)

The difference between the effort involved in test execution using manual scripts and using automated scripts is the benefit in terms of the effort saved as a result of test automation. The effort involved in automated test execution includes the human effort in hours spent in executing, monitoring and analyzing the automated scripts. This could also include time spent in preparing test bed or performing any other test execution activities that are not automated.

The investment that is required to get this benefit is the time spent in setting up the automation framework, designing and developing automation scripts and training the resources on automation, not to forget maintenance of the automation scripts, which will come into play later as the application under test automation undergoes changes.

Thus, ROI_{effort} metric will help evaluate if the effort saved as a result of test automation is worth the effort spent in running the test automation program.

$ROI_{effort} = 1$ indicates: There is no human effort saved as a result of test automation.

$ROI_{effort} > 1$ indicates: All other factors remaining constant, it is worth automating the application testing.

$ROI_{effort} < 1$ indicates: Effort saved as a result of test automation is not high enough to justify effort spent on automation implementation. Management needs to have a closer look to decide if they would like to pursue automation to achieve other automation benefits such as increased consistency in regression testing, leading

to higher confidence level in test results, etc.

In the short term, ROI_{effort} cannot be expected to be high. ROIeffort would be less than 1 during the setup period i.e. when the automation framework is under development and/or the automated scripts have not been run, but should eventually turn 1 or greater than 1 once the automated scripts are put to use. How soon or how late would depend on a number of factors like how much of the application is decided to be automated, how frequently testing cycles are scheduled, etc. As the frequency of automation scripts execution increases, the ROI_{effort} can be expected to increase. Viability of any functional or regression automation program would ultimately rest on a high value ROIeffort metric in the long term.

While ROI_{effort} for a particular project is an important metric for a particular duration, Cumulative ROI_{effort} metric provides the payback period in terms of effort saved. It can be used for an individual project or the automation program encompassing all the projects on a whole.

Calculating ROI_{effort} with the help of an example: Project ABC Data:

Table 1

Cycle No.	Year of Test cycle	Manual Test Execution Effort (hrs)	Automation Script Development Effort (hrs)	Automation Script Execution Effort (hrs)	Automation Script Maintenance Effort (hrs)	Automation Training Effort (hrs)	Effort saved as a result of automated scripts execution (hrs)
0 (Development Cycle)		40	90			25	0
1	2010			10			30
2	2010			10			30
3	2010			10			30
4	2011			10	20		30
5	2011			10			30
6	2011			10			30
7	2012			10	20		30
8	2012			10			30
9	2012			10			30

Cycle No.

Cycle #0 signifies the automated test development cycle i.e. when automation scripts were being designed and developed. During this cycle, automated scripts were not executed.

Cycle #1 signifies the first test execution cycle after automation scripts were developed. This is the first time when test execution was performed using automated scripts.

Year of Test Cycle

The above table depicts that there are three test execution cycles planned to be executed in a year. There are nine cycles planned to run over a period of three years.

Automation Script Development Effort

This would be a one time effort. Unless some new functionality or module is to be automated for the given project or application, the automation script development effort would be incurred only in Cycle #0. Later on, any effort spent in updating the test scripts would fall under the Automation Maintenance head.

Automation Script Maintenance Effort

There would be no maintenance costs upfront. They may be required only at a later stage if the project under automation evolves or changes.

For project ABC, we are assuming that every year, there is a release with enhancements or change orders, which requires maintenance effort of 20 hours for the automated scripts.

Automation Training Effort

25 hours have been assigned for training effort. This training effort is required to train only the project team responsible for execution. This will involve handing over the automated test scripts created by centralized automation team to the project testing team for execution.

Effort Saved as a result of execution using automated scripts

The effort saved as a result of using automation scripts remains constant for a given project.

It is defined as, Manual Test Execution Effort - Automated Test Execution Effort

For Project ABC, it can be calculated as: (40-10) = 30 hours

Project ABC ROI_{effort} Metric:

Table 2

Year	Cycle	ROI_{effort} for a year	Cumulative ROI_{effort}
Year #1	Cycle #0	ROI_{effort} for year #1 = (90/145) = 0.62	Cumulative ROI_{effort} for year# 1 = (90/145) = 0.62
	Cycle #1		
	Cycle #2		
	Cycle #3		
Year #2	Cycle #4	ROI_{effort} for year #2 = (90/50) = 1.8	Cumulative ROI_{effort} for year# 2 = (180/195) = 0.92
	Cycle #5		
	Cycle #6		
Year #3	Cycle #7	ROI_{effort} for year #3 = (90/50) = 1.8	Cumulative ROI_{effort} for year# 3 = (270/245) = 1.1
	Cycle #8		
	Cycle #9		

What does the metric signify for Project ABC?

1. ROI_{effort} is less than 1 for year #1 which is the setup period.
2. Cumulative ROI_{effort} is increasing every year.
3. ROI_{effort} is greater than 1 for year #3 i.e. returns start accumulating third year onwards.
4. It is worthwhile to automate the project.

Thus, automating testing is useful for project ABC in terms of effort saved.

Utilization Optimization Factor

UOF metric helps monitor the time usage of automation tool available.

The cost of license of an automation tool significantly influences the cost of the automation program. To ensure that automation is cost effective, it is important that usage of the tool is monitored and its optimal utilization is planned.

Having this metric on dashboard ensures redistribution of automation tool from projects which are not doing an optimal utilization to projects which have an under-allocation (shortage of availability) of the tool.

For this metric, we need to do the following:

1. Define an Organization Wide Baseline Average Utilization Hours Standard for an automation tool for a period such as a year.
2. Record the automation tool usage time for each project encompassing activities like script development, script maintenance, script execution and training on tool.

The Utilization Optimization factor for the project is calculated using the formula: (Actual Utilized Hours of the tool for the project/ Organization Wide Baseline Average Utilization Hours Standard for the tool).

This formula can be expanded to:

(Effort spent in (Automation Script Development + Automation Script Maintenance + Automation Training) + CPU Time Taken in Automated Execution)) / (No. of licenses * Average Utilization Hours for the automation tool for a time period)

CPU Time Taken in execution of automated scripts is the time taken by the scripts to run. This is the time that keeps the automation tool busy during execution.

Example:

Organizational Data:

Let's say, average utilization hours for tool for a year is defined as 2100 hours (assuming tool is put to use for 22 days a month for 8 hours daily).

Project Data:

Let's say, the tool was actually utilized for 3000 hours for a project for performing various activities like script development, maintenance, execution and automation training etc.

The UOF for the project would vary depending on the number of licenses the project used.

Table 3

No. Of Licenses	UOF
1	= 3000/(1* 2100) = 1.43
2	= 3000/(2* 2100) = 0.71
3	= 3000/(3* 2100) = 0.48

Hence, if the project has managed to do 3000 hours of work on the automation tool with one license, it has a UOF of 1.43 as against a UOF of 0.48 if the project has used three licenses.

Thus, lesser the number of licenses, higher is the UOF.

An organization can use this metric to find out which projects use the tool more efficiently than others.

UOF = 1 indicates: Utilization of the automation tool as per organizational standard

UOF > 1 indicates: Automation tool availability to the project is less than the requirement

UOF < 1 indicates: Poor utilization of the automation tool

The UOF for a particular project may be lower than another due to a number of factors like:

1. The automated scripts are slow to execute. Thus, keeping the automation tool busy for a longer duration.
2. The development and maintenance of automated scripts is taking too long due to reasons like complexity or non-stability of project.
3. The automation tool is utilized for lesser number of hours, say 6 hours per day.

A low UOF may call for some actions like decreasing the number of licenses used by a project, selecting projects better suited for automation, etc.

A constant UOF for a few cycles say UOF = 1 may call for stretching the previously set goals. Say, increasing the expected average utilization hours for the tool. This could happen by increasing shared use of the tool across project teams, working in shifts, gaining maturity in terms of automation framework and script efficiency, developing scripts that can run 24X7 without manual intervention, etc.

ROI_{cost}

ROI_{cost} metric helps evaluate if the cost spent running the automation program is worth the cost saved as a result of automation.

Formula: ((Effort saved as a result of execution using automated scripts) * (Chargeability rate of Test Engineers per hour))/ ((Average Cost of usage of automation license per hour) * (No. of hours license is used))

The following are needed to calculate it:

Organizational Data:

1. Average Chargeability rate of Test Engineers: This is the per hour rate.
2. Average Cost of usage of automation license per hour for the organization: This can be derived using the formula: (Cost of one license for automation tool for a period say one year * Total No. of licenses available with the organization) / (Total Number of Hours automation tool is used during the corresponding period by all the projects or applications of the organization)

Project Specific Data:

1. Effort saved as a result of execution using automated scripts for the specific project: This is the same as was calculated for determining ROI_{effort}. This will be available in hours.
2. Number of hours license is used: This is the total number of hours that the automation tool has been put to use for a specific project. This includes the time for script development, maintenance, execution, training etc.

The numerator i.e. (Effort saved as a result of execution using automated scripts * Chargeability rate of Test Engineers) would provide the amount of money saved in terms of human resources as a result of automation. Cost of other resources like hardware is not included here, since these costs are usually indirect costs and too small to be considered.

ROI_{cost} = 1 indicates: The cost of manual and automated test execution is the same. Based on automation program, objectives this may/may not be an acceptable state.

ROI_{cost} > 1 indicates: Money saved as a result of human effort reduction is greater than the cost of automation. Hence, all other factors remaining constant, it is worth automating the application or project.

ROI_{cost} < 1 indicates: Automation is not cost effective.

Organization Data:

Cost of one License for a year: $1000

No. of Licenses: 3

No. of hours automation tool is used in a year by all the projects of an organization: 2500

Thus, Average Cost of usage of automation license per hour for the organization = ((1000 *3)/2500) = $1.2

Average Chargeability Rate of Test Engineers per hour: $50

Project ABC Data:

Table 4

Year	Automation Script Execution Effort (hrs)	No. of hours license was used for script development, maintenance and training (Data from Table 1)	No. of hours license is used for the project	Effort saved for the period (hrs) (Data from Table 1)	ROI_{cost}
1	30 (10 * 3 cycles in a year)	115	= (30+115) = 145	90	=(90*50)/(1.2*115) = 32.6
2	30 (10 * 3 cycles in a year)	20	= (30+20) = 50	90	=(90*50)/(1.2*50) = 75
3	30 (10 * 3 cycles in a year)	20	= (30+20) = 50	90	=(90*50)/(1.2*50) = 75

ROI_{cost} should be calculated over a long duration in order to get meaningful results.

This is because the denominator i.e. the cost of usage of automation license per hour (which is a very important factor in determining the ROI cost) is guided at the organization level and not at the project level. Thus, this cost will be low if overall usage of license is high in the organization. However, if the automation is done for a very few projects, the cost of usage of license per hour will turn out to be substantially high, affecting the results.

As with any other metric, this set of metrics too demands a logical and careful analysis before conclusions are drawn, results are interpreted, and actions are planned.

Some pointers around application of these metrics are:

1. Use these metrics to do automation planning and set-up the goals in advance. The metrics can be used to :
   - Find out how many average utilization hours of the tool should be aimed at and how much to automate to get a UOF = 1.
   - Plan your automation investments keeping in mind the payback period in terms of effort saved and costs
   - Give project specific goals to align with organizational automation goals.
2. Develop your own baselines. Use them to evaluate your organization's projects. Industry benchmarks are usually not available, even if available they are usually not applicable to organization's specific needs.
3. Use these metrics to monitor the performance of each automation project.
4. Use these metrics to monitor the performance of the overall automation program.
5. Use these metrics from time to time to see that the project-mix chosen for automation is correct.
6. Use these metrics along with other general metrics like Automation Penetration %, Automation Progress %, etc. for monitoring automation activities.

Biography

Parul Singhal is working as Senior Test Analyst at Hewitt Associates, India. She has over eight years of experience in software testing. She has worked in various domains like Health Insurance, Software modeling, Benefits and Human Resource.
She has been leading testing teams on projects involved with web testing, mainframes, business intelligence tools and client server applications. She has been intensively involved with defining and improvising testing metrics for her Project teams. She holds a Masters degree in Computer Applications (MCA). She can be reached at parul.singhal@gmail.com.

Do not use testing metrics for the wrong reasons

by Todd E. Sheppard

As testers or quality assurance professionals we are encouraged (and compensated) to constantly find fault, to improve processes, and to ensure quality. Measurement is inherent in what we do. We report on how many defects were found in a release, how well the requirements were written, how many times the development team had to rebuild the software, or a host of other metrics.

When the time comes to defining which ones of the many potentially valuable metrics will be captured for any given project, it is difficult because there are more metrics which could be tracked than is truly practical. The general focus of this article will be to look at a few of the many commonly used metrics and investigate how they may be misused. Some of the commonly used metrics include: Number of defects identified per lines of code delivered, mean time to repair defects, mean cost to discover a defect, total cost of quality activities, amount of rework, test time or cost planned versus actual cost, production defects verses test defects, hours to discover a defect, and many others.

All of these common metrics can be good, but they can also be very bad when used for the wrong reasons. The goal is to remind the reader to think of the type of behavior which is being encouraged based on which metrics are captured, how the data is utilized, and how the results are communicated. Just as we do when testing, it is important to remember to look to the root cause, when analyzing the metrics you collect.

There are a number of ways which collected metrics could turn into a hindrance rather than a benefit. It is fairly common for test metric information to be diligently recorded over the course of application development and testing, placed in a very pleasant looking graph for senior management and then placed on a shelf to collect dust. Unless actionable items or process changes are defined and acted on, it is pretty safe to say that the next release of the software will probably have a report which looks identical. When this happens there is no headway made on reducing total defects, and the quality of the product and the team members will not improve, because no effort was placed into improving. Management will wonder why nothing ever changes.

To not let that happen there needs to be a serious review of the metrics which were collected. The data needs to be analyzed and people or teams need very specific assignments regarding what changes will be made to improve. In almost every case the item to improve should be a process not the person. Most people do not do bad work on purpose; there are countless external factors which contribute to bad software. These factors range from unrealistic deadlines, lack of knowledge of a particular language, bad requirements, no user buy-in, all the way to personal issues impacting the workplace.

As a Quality Assurance Analyst (QA) it is our job to encourage utilization of all of the metrics which have been collected. While you perform the analysis of the metrics following a particular release remember to address the problem in the process, not beat the people over the head. For example: if there is too much rework on the development side (it seems that every time a defect is returned for retest it fails) then perhaps there needs to be enforcement of a code peer review process. While the development group is certain to grumble about more peer reviews, executing a peer review is much preferred to pulling all of the developers into an impromptu staff meeting and telling (or yelling) at them to "reduce the rework count or else".

We should all be well past the point where the QA can sit there quietly and only be a "software tester". In a world where everyone is expected to do more with less people and, by the way, we expect better quality than ever before because one little software flaw costs us customers, it is time for the QA to step up and truly implement change. While we are used to the role of validating the quality of a system, the role really needs to expand to influencing all other team members to continuously refine their processes. In doing this there are many hurdles which may need to be crossed. One of the primary hurdles being that often the QA does not have any real "authority" over the other teams involved. Frequently the QA reports up to the same manager as the development team. In that situation the development team usually has control because they have the ear of management, the QA is a necessary evil. In the case where the management chains are different and the QA group reports to a different management structure (generally larger corporations), it can be difficult to influence change across divisions. In order to be successful in the future the QA must learn to employ a fair amount of management tact. Gone is the time of software testers, we now need to be true quality partners and

implementers of change regardless of the hurdles thrown in our way.

The measuring stick which we use to determine where to focus change: Testing Metrics. The problem is that with each metric we capture, we need to make sure it does not negatively influence behavior. We are all professionals and as such it should be reasonable to expect that people will behave in a professional manner. Unfortunately human nature is often at odds with the professional manner we are striving for. If there is a fear of repercussion based on a test metric bad behavior can subconsciously be encouraged even in the most well intentioned individuals. As an example, consider reporting on "Number of Total Defects Discovered". On the surface this metric should not raise too much controversy. In reality if QA's believe they are being measured, celebrated, or even worse their raise is based on having that number high, there may be a tendency to open trivial defects or the QA may be encouraged to open multiple defects when one would suffice to communicate the issue.

The following is a quick examination of a few of the more common metrics and a look into the good and bad influences they may have.

Ratio of System Test defects discovered to defects which are discovered following system test

The general goal of this measurement is to determine how effective the system test team is by identifying the ratio of defects found during system testing to those which "escaped" from the system test group to be discovered downstream. The escaped defects would include any found in User Acceptance testing, during Beta testing, and in production for some warranty period. This is certainly a valid ratio to track, but like all of the others some care needs to be taken. This ratio can probably be refined a little bit. One way is to not include low priority defects identified downstream, this removes some of the "I don't like that screen color" type of defects. Like every metric there's a positive and negative way to address this metric. Threatening the testers with their jobs if the ratio does not improve, is one method. The other option would be to spend the time to review all of the escaped defects and then determine how they escaped. Use this knowledge to expand your regression scripts. The team could also work to identify the general type of defects which escaped and create templates to make sure they are covered. For example, if several issues were discovered in production related to date fields, then write a comprehensive script template to test every new date field introduced.

Total Number of defects reported

This one creates a myriad of trouble. It can encourage testers to open trivial defects or to open multiple defects when one will suffice. It encourages the development team to argue away as many defects as they can. In the end you would like everyone to just report it as it is, but when the fear of repercussion hangs in the air, even those with the normally best intentions can be tainted to distort the truth. This metric is also very difficult to compare between projects or releases because it does not take into account the scope of the changes. If fifty defects are opened on a simple screen change that is far more troublesome than fifty defects against a built from scratch brand new system. Caution needs to be used when comparing metrics from system to system as the number of variables is enormous. This leads us to look at different metrics. Total defects per lines of code or defects per development hours do present a slight improvement. While these still have some of the same problems they are much more scalable. The team may again want to consider removing low priority defects. Further refinement by platform and/or type of effort (new verses enhancement) can also help. Another method may be to monitor the number of currently open defects. Once the initial spike of new defects in system testing occurs, verify that the open defect count trends downward.

Time to repair defects or amount of rework

This tracks the time once a defect has been accepted to how long until it is ready to be retested or how many times a defect is reopened. This has the potential to encourage a developer to set the ticket to retest prematurely to hope that it gets closed or to negotiate with the tester "let's close that one and open a new one which is similar, but describes the new half fixed problem". The goal is not to open tickets; it is to deploy a quality system. Often tickets will remain open for a long time by plan; if this metric is to be tracked perhaps refine it to only critical defects.

Time or cost to discover a defect

This is a measure for the QA to determine the average time for a tester to find a defect or the associated cost thereof. This can fall into the same category as total defects, by increasing defect count this number looks "better" from the individual tester perspective, because they are finding defects in no time at all. In reality if the QA is doing a good job ensuring quality throughout the entire process this number should get higher and higher as it becomes increasingly difficult to find defects in system testing. The better overall job a QA is doing will result in an uneventful system test.

Tester Performance

Another example of a potentially harmful metric is the ranking of testers based on test "quality". The quality being based on the number of defects opened, root cause, defect severity, and total test time. All in all it is a not a horrible measure, it does present a way to look at which testers appear to be high performers. The trouble is that encouraging competition between testers is not likely to create the proper motivation or atmosphere. It is good to compare these type of metrics from release to release as a measuring stick for the project team as a whole (application team, business analysts, and quality assurance team), but putting testers in competition with one another is probably not the best use of time. There are certainly times (compensation changes or performance evaluations) to compare testers, but this should not be something which is celebrated after each test event. So while the management team may find it beneficial to track this information, it generally should not be broadcast. There are too many variables, was someone assigned a more problematic area of the application, did the tester sacrifice deep quality testing to quickly cover more areas, does one tester have more robust knowledge of the system, or did one tester start earlier in the cycle finding all of the critical defects up front(scoring them big quality points)? The testing organization needs to work together as a team, not try to hide defects from each other or inflate defect counts so they can be the #1 tester. This type of activity may be joked about in the lab, but to institutionalize it should not be encouraged.

There is no doubt that where focus is placed impacts behavior, as such there is generally a need to track many metrics to obtain a good perspective of how your quality effort is going. It may become necessary to carefully plan what metrics are tracked such that where one could encourage a bad behavior, that behavior would be counteracted with another metric. For example, if tracking total defects, perhaps also track the percentage of "valid"

defects opened to minimize superfluous defects. In the end what metrics are tracked will not be an issue at all if the information is used correctly. The goal should be to identify root cause and attempt to address it. Build rapport with your team and the organization that you are trying to improve quality, not threaten employment, and this should relieve some of their fears. Follow through so that in the end, you react the way you said you will, by addressing the issues with the process. While doing so you must also try to modify behavior of any other manager who is misusing the metrics you collect.

So what are good metrics to collect? What about a customer survey? Even that can be tainted because it is measuring relationships (which are extremely important), but not system or process quality. The answer of course to what are good metrics is: it depends. It depends on the situation and what problems need to be addressed. There are few metrics which would be considered inherently bad to track as long as they are handled correctly. What is sure is that it is imperative to describe "why" you are tracking them, "what" the goal is when implementing change, and "how" the collected data will be used. The key message is: communicate, communicate, and then communicate. Bad communication results in fear, it is human nature. Lots of fear is already built up when we know we are being tracked. Discussions need to occur to assure the team that the goal is to improve quality, not destroy people. Care also needs to be placed in what audience you are speaking to. The next generation of quality assurance analysts (Gen Y) does not like to be told how bad they are doing. This is not to suggest that you coddle them, performance issues should be addressed, but published metrics are not the place to monitor performance issues. Today's workforce is more about cooperation. The days of brutal management are fading; the next generation will not tolerate it. While you may say good riddance as they are walking out the door, if you remain on this path one day you will look up and discover "oh my, the last tester just retired". If metrics are publicized to destroy people then the low performers will justify the ratings to themselves and will have no real motivation to improve, they will move on or worse they may actually stay and start to care even less. One approach to counteract this would be to pair low and high "ranked" people together for improvement, experienced Quality Assurance Analyst's and new hires have a great deal to teach each other if given the opportunity. The path to improving quality may include suggestions for personal behavior modification, but this should be communicated in a face to face development plan, calmly, and will probably have little to do with the metrics which were captured. The plan with metrics needs to be: track the metrics, identify the root cause, refine the process, and repeat. Once the team and project quality increase, evaluate the use of different or additional metrics and start over. The collecting of metrics is not a short term event; real change can only be accomplished by measuring, reviewing, instilling change, over and over. Use metrics to determine how well testing is going, use them to compare trends from release to release, but do not use them to reward or punish. They are a measuring stick not a baseball bat.

Biography

Todd E. Sheppard has been working as an IT professional for the past 17 years. Todd holds a Master of Engineering degree from the University of Louisville, Kentucky USA. He has worked as a System Integrator, a Software Developer, in Training and Deployment, a Release Manager, and a Quality Assurance Analyst. Todd has been focused solely on software testing and quality assurance for the past 10 years. He is currently employed as a Sr. Quality Assurance Analyst at UPS.

Masthead

EDITOR
Díaz & Hilterscheid
Unternehmensberatung GmbH
Kurfürstendamm 179
10707 Berlin, Germany

Phone: +49 (0)30 74 76 28-0 Fax: +49 (0)30 74 76 28-99 E-Mail: info@diazhilterscheid.de

Díaz & Hilterscheid is a member of "Verband der Zeitschriftenverleger Berlin-Brandenburg e.V."

EDITORIAL
José Díaz

LAYOUT & DESIGN
Katrin Schülke

WEBSITE
www.testingexperience.com

ARTICLES & AUTHORS
editorial@testingexperience.com

350.000 readers

ADVERTISEMENTS
sales@testingexperience.com

SUBSCRIBE
www.testingexperience.com/subscribe.php

PRICE
online version: free of charge -> www.testingexperience.com
print version: 8,00 € (plus shipping) -> www.testingexperience-shop.com

ISSN 1866-5705

Index of Advertisers

HANDBUCH

TESTEN
IN DER FINANZWELT

Das Qualitätsmanagement und die Software-Qualitätssicherung nehmen in Projekten der Finanzwelt einen sehr hohen Stellenwert ein, insbesondere vor dem Hintergrund der Komplexität der Produkte und Märkte, der regulatorischen Anforderungen, sowie daraus resultierender anspruchsvoller, vernetzter Prozesse und Systeme. Das vorliegende QS-Handbuch zum Testen in der Finanzwelt soll

- *Testmanagern, Testanalysten und Testern sowie*
- *Projektmanagern, Qualitätsmanagern und IT-Managern*

einen grundlegenden Einblick in die Software-Qualitätssicherung (Methoden & Verfahren) sowie entsprechende Literaturverweise bieten aber auch eine „Anleithilfe" für die konkrete Umsetzung in der Finanzwelt sein. Dabei ist es unabhängig davon, ob der Leser aus dem Fachbereich oder aus der IT-Abteilung stammt. Dies geschieht vor allem mit Praxisbezug in den Ausführungen, der auf jahrelangen Erfahrungen des Autorenteams in der Finanzbranche beruht. Mit dem QSHandbuch sollen insbesondere folgende Ziele erreicht werden:

1. *Sensibilisierung für den ganzheitlichen Software- Qualitätssicherungsansatz*
2. *Vermittlung der Grundlagen und Methoden des Testens sowie deren Quellen unter Würdigung der besonderen Anforderungen in Kreditinstituten im Rahmen des Selbststudiums*
3. *Bereitstellung von Vorbereitungsinformationen für das Training „Testing for Finance!"*
4. *Angebot der Wissensvertiefung anhand von Fallstudien*
5. *Einblick in spezielle Testverfahren und benachbarte Themen des Qualitätsmanagements*

Herausgegeben von Norbert Bochynek und José M. Díaz Delgado

Die Autoren
Björn Lemke, Heiko Köppen, Jenny Siotka, Jobst Regul, Lisa Crispin, Lucia Garrido, Manu Cohen-Yashar, Mieke Gevers, Oliver Rupnow, Vipul Kocher

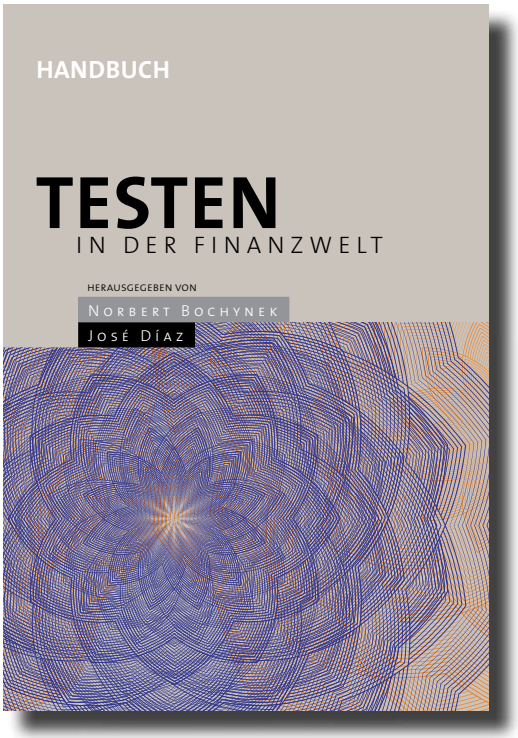
Gebundene Ausgabe: 431 Seiten

ISBN 978-3-00-028082-5

1. Auflage 2010 (Größe: 24 x 16,5 x 2,3 cm)

48,00 € (inkl. Mwst.)

www.diazhilterscheid.de



Training with a View

Díaz Hilterscheid

"A casual lecture style by Mr. Lieblang, and dry, incisive comments in-between. My attention was correspondingly high. With this preparation the exam was easy."
Mirko Gossler, T-Systems Multimedia Solutions GmbH

"Thanks for the entertaining introduction to a complex topic and the thorough preparation for the certification. Who would have thought that ravens and cockroaches can be so important in software testing"
Gerlinde Suling, Siemens AG

06.09.10-10.09.10	Certified Tester Advanced Level – TECHNICAL TEST ANALYST	Berlin
13.09.10-16.09.10	Certified Tester Foundation Level	Berlin
20.09.10–24.09.10	Certified Tester Advanced Level - TECHNICAL TEST ANALYST	Wien
04.10.10-07.10.10	Certified Tester Foundation Level	Berlin
11.10.10-13.10.10	Certified Tester Foundation Level – Kompaktkurs	Hannover
11.10.10-15.10.10	Certified Tester Advanced Level – TESTMANAGER	München
18.10.10-19.10.10	Testen für Entwickler	Berlin
25.10.10-29.10.10	Certified Tester Advanced Level – TEST ANALYST	Stuttgart
25.10.10–25.10.10	**Anforderungsmanagement !! NEW !!**	Berlin
26.10.10-28.10.10	Certified Professional for Requirements Engineering – Foundation Level	Berlin
02.11.10-05.11.10	Certified Tester Foundation Level	München
02.11.10–04.11.10	**ISEB Intermediate Certificate in Software Testing !! NEW !!**	Berlin
08.11.10-12.11.10	Certified Tester Advanced Level – TEST ANALYST	Berlin
15.11.10-18.11.10	Certified Tester Foundation Level	Berlin
22.11.10-26.11.10	Certified Tester Advanced Level – TESTMANAGER	Frankfurt
22.11.10–23.11.10	**Testmetriken im Testmanagement !! NEW !!**	Berlin
29.11.10-30.11.10	Testen für Entwickler	Berlin
01.12.10-03.12.10	Certified Professional for Requirements Engineering – Foundation Level	Berlin
06.12.10–08.12.10	**ISEB Intermediate Certificate in Software Testing !! NEW !!**	Berlin
07.12.10-09.12.10	Certified Tester Foundation Level – Kompaktkurs	Düsseldorf/Köln
13.12.10-17.12.10	Certified Tester Advanced Level – TESTMANAGER	Berlin

also onsite training worldwide in German, English, Spanish, French at

http://training.diazhilterscheid.com/

training@diazhilterscheid.com

- subject to modifications -