Critical Logic
Integrated Quality Management

Cause -Effect Modeling: The Case for Using Models to Improve Quality, Shorten Time-to-Market, and Reduce Costs

Prepared By: Critical Logic

WHITE PAPER

_Cause-Effect Modeling: The Case for Using Models to Improve Quality, Shorten Time-to-Market, and Reduce Costs

Introduction

In this whitepaper, we'll discuss the proven approach of Cause-Effect Modeling. By representing detailed functional requirements with mathematically complete models, Cause-Effect Modeling represents a fundamental change in the software development rules of the game.

> Efforts to implement true test-driven development have been stymied by the lack of solutions that allow for requirements to drive tests. Until we know that our tests cover all the business's needs, TDD will continue to prove an elusive goal.
>
> -Peter Becker
> Founder of Critical Logic

What Is Cause-Effect Modeling?

Cause-Effect Modeling is the process of building a mathematically rigorous representation of the intended functional behavior for proposed software. This isn't an entirely new approach; the problem of defining, measuring, and testing detailed functionality has been tackled by hardware engineers in the past, particularly chip testing engineers, who needed to be able to quickly confirm the correct functional behavior of every computer chip coming out of a fabrication plant. The algorithms used to design tests for chip logic can also apply to the functional logic of software.

How Is Cause-Effect Modeling Applied?

This previous work has been used by Critical Logic and its partners to develop a modeling tool that allows functional and test engineers – the line between the two is significantly blurred when models represent requirements as well as tests – to focus on building a graphical model of the intended behavior of the software, then rely on software to analyze the requirements and generate the actual test cases.

This approach represents a paradigm shift in the use of requirements and testing. If you make the model complete and correct, then the specifications and test cases – which guarantee full functional coverage – are produced automatically.

The modeling algorithms are based on chip-testing technology that essentially builds a series of cause-effect graphs. The purpose of the model is to:

- ***_Represent all of the software's functional requirements***
- ***_Identify ambiguities or other quality issues in the requirements prior to coding***
- ***_Generate specifications that can be used for coding and/or documentation purposes***
- ***_Design and generate the fewest number of test cases that exercise 100% of the defined functionality***
- ***_Provide a mechanism that simplifies the maintenance of requirements and their associated tests***

The Process

The essential process steps in Cause-Effect Modeling are straight-forward:

1. ***Collect information that describes the intended behavior of the system as observed at its interfaces (GUI, databases, inter-system transactions, etc.)***
2. ***Represent this functional behavior in a cause-effect logic diagram***
3. ***Process the model through automated algorithms that design the optimum number of test cases giving full functional coverage***
4. ***Revise the models and regenerate tests as necessary to support functional changes to the software***

Cause-Effect Modeling and QA

The effectiveness of Cause-Effect Modeling is derived from two components of the process:

1. Functional Requirements. First, the modeling process itself forces rigor in the representation of the functional logic. Ambiguous or poorly understood rules of behavior create disconnects in the model which then need to be resolved. The assumption is that if the modeler encounters these ambiguities, then it is likely that others participating in the development effort will also have the same issues, and the potential for these to translate into incorrect software output is high. In fact, repeated analysis has shown that 70-80% of all software defects originate from poor or misunderstood definitions of required functionality. Rigorous representation of functional logic substantially lowers or eliminates the occurrence of these defects.

Another way of looking at this is a requirement's ambiguity must eventually be resolved in order to deliver functioning code. Cause-Effect Modeling provides an approach that allows the functional modeler, working with the business stakeholders, to identify and resolve requirements ambiguity prior to coding. Without functional modeling, the coder will often be defining the requirements in order to get code delivered. Many times their educated guesses are correct; but all too often they are not, resulting in customer dissatisfaction and rework.

Great requirements obviously have substantial value, but the real benefit of effective Cause-Effect Modeling rests in the design of test cases. Cause-Effect Modeling means that IT now has the ability to "prove" to the business that all their requirements are being met because the tests cover all defined functionality.

2. Functional Testing. The rigorous representation of functionality allows the application of sophisticated mathematical and logic algorithms to design test cases. The need to test complex logic in a small number of test cases has existed in the hardware world for decades and became especially critical with the advent of computer chips. The result has been the creation of very effective test case design algorithms for verifying functional logic. These have been an important component for assuring the very high reliability of computer hardware. Unfortunately, software QA and testing groups have largely eschewed formal test design technology in favor of informal and largely undisciplined testing processes. Applying formal test design algorithms to test functional logic in software results in very high reliability software for less time and cost than traditional testing approaches.

> … if you can never come close to completeness in testing, then you would be unwise to think of testing as 'quality assurance'
>
> -Richard Bender, article in Software Quality Magazine

This is very different from traditional test automation and management. Automating test cases that provide poor functional coverage does not remove any additional defects. It is the issue of test case design that keeps current testing practices at the level of keyboard pounding. On a typical project, the tester evaluates what results need to be validated and attempts to choose from many possible combinations of inputs that lead to those results. Test cases are derived from this analysis and executed. However, in the current state of the practice, this process is largely subjective and at the discretion of the test analyst.

The subjective nature of test case design is repeatedly shown in the following ways:

_Different test analysts testing the same functionality come up with different tests and cover different (though overlapping) functionality.

_In selecting testers for a team, the emphasis remains almost on individual traits (experience with the application, test "karma") rather than the process for designing tests.

_When these tests are evaluated against quantitative coverage criteria, functional coverage is well below 60% and code execution coverage is even less.

_Most importantly, defects routinely go undetected because the test cases were not designed to fully exercise the system's functionality.

A Better Approach

Model-based testing carries with it important implications:

_The functional modeler derives the model from the same business requirements that traditionally were used by the coder. The difference is that the functional modelers can start working almost as soon as the first set of overall Use Cases or User Stories are created. In this way, the functional modeler enhances the requirements discovery process by using the graphical tools in the modeling software to quickly expose inconsistencies and incompleteness in the requirements. This means the specifications are dramatically improved with a result that there is at least a 50% reduction in the number of defects found in code as it is delivered to the QA team for testing.

_Since the models are built as the requirements are completed, test case design can be accomplished in parallel with or even ahead of coding. Project managers can more accurately size the development effort since the requirements and specifications are much more stable. Coders use the model-based specifications and test cases as a guide for unit level testing and to verify correct interpretation of the specifications. Other test issues, such as data setup and test environment configuration, can also start much earlier in the development lifecycle.

_In Agile or short-period iterative development methodologies, test-driven development and having tests before coding is done is an important critical success factor. The reality is that getting tests ready this early has proven to be extremely difficult, leading to a testing bottleneck that extends the process to the point where the development cycles start looking more like a waterfall approach. Since functional models are built as part of the requirements process, model-generated tests may be the only practical means of meeting this important demand, short of applying significant numbers of QA staff to the problem. For example, Agile suggests a 1:3 ratio of QA staff to developers. With Cause-Effect Modeling, that ratio is reduced to around 1:5 or 1:6.

_With model-based testing, the impact of a change in requirements or specifications is immediately visible. Since specifications, test cases, and other quality products are automatically derived from the model, changing the model changes the test scripts.

_Test effectiveness is consistent and complete. Functional coverage is always 90-100%, and other test coverage measurements – such as code coverage – are also very high. These results are consistent from test engineer to test engineer and independent of their experience with the application or business area.

Measuring Actual Results

Do these anticipated benefits manifest themselves in real projects? The short answer is yes. However, exposing and quantifying the results becomes tricky when you consider that IT development efforts are seldom held accountable to standards and metrics. Whereas process-oriented endeavors such as manufacturing and large-scale service delivery (e.g. call centers, health or insurance claims processing) have continuous metrics to monitor performance, software development organizations seldom track and manage to clear performance metrics. Consequently, it is much harder to reliably assess the impact of a process change or a new tool in software development compared to when a new piece of equipment is installed in a factory production line. While IT organizations demand proof that model-based testing will improve software quality, few are willing to collect and track the metrics that would routinely supply definitive numbers.

Instead, verifying the value of Cause-Effect Modeling and test design requires reliance on individual opportunities to track data for specific projects and assess the results. This opportunistic approach yields important results that clearly show the benefits, even though the inconsistency of metrics from project to project makes generalizations harder to prove. Therefore, the remainder of this paper looks at specific results from individual projects where quantitative or qualitative data has been collected and conclusions arrived at about the effectiveness of Cause-Effect Modeling and model-based testing.

Defect Avoidance

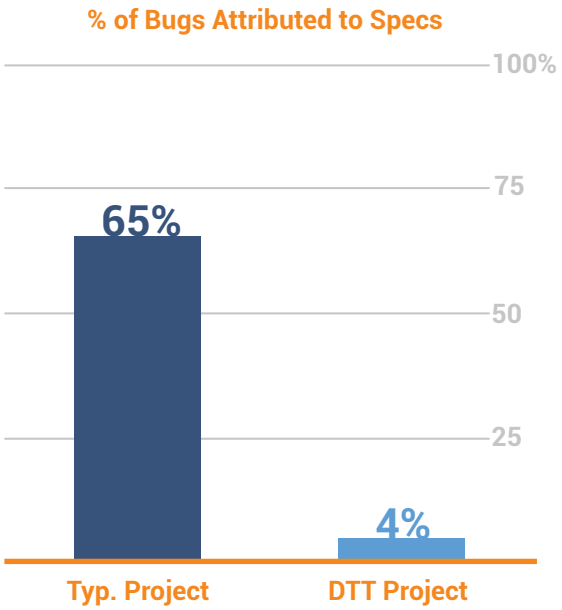
One of the strengths of Cause-Effect Modeling is that the modeling process itself highlights errors and ambiguities in the specification of the functionality. Such ambiguities lead to misunderstandings on the part of developers which, in turn, translate into coding defects. Therefore, modeling should reduce the number of initial functional defects introduced into code. Two Critical Logic client projects have done analysis in this area, both confirming the substantial reduction in defects.

Middleware Services at Wells Fargo Bank

In this project, the client was building a complete set of middleware methods to support various Web-based and telephone banking applications. Typically, each implementation would add or modify 5-10 business methods to the middleware inventory. Methods were fairly complex in terms of business rules and processes for analyzing data requested from the systems of record. Wells Fargo evaluated the total number of defects logged against 20 methods that were coded prior to the use of Cause-Effect Modeling as opposed to the number of defects logged against approximately 30 methods that were tested under the Cause-Effect Modeling and test technology. The conclusion of management was the number of defects was reduced by 47%. Further, the number of defects escaping into post-functional testing or production was close to zero.

Business Credit Application

In this project, careful metrics were kept on the source of defects. All functionality was subjected to functional modeling. By tracking the source of all defects, the client could determine if ambiguities in the functional specifications were being caught prior to coding or whether they were translating into defects in code, adding to the rework. As a baseline for comparison, an industry average was applied. Multiple studies have shown that approximately 60-70% of defects logged against software actually originate in the functional specifications. Therefore, a substantial reduction in spec-based defects would indicate that the Cause-Effect Modeling process is having an effect.



During the testing of the application, 2148 defects were logged. However, as the accompanying chart shows, the number of defects attributed to specification problems was less than 4%. Cause-Effect Modeling clearly had a major impact on ensuring that functionality was well understood.

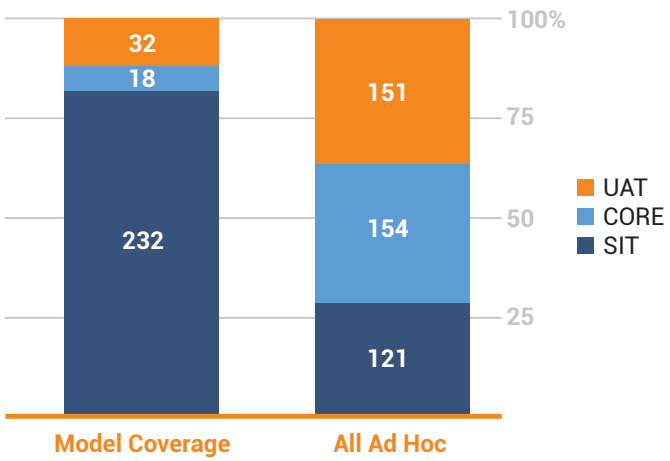
Defect Escapes

The key measure of any testing approach is prevention of defect escapes into production. Functional modeling and the automated test designs derived from it are valuable in that the resulting tests are quite thorough in their coverage of the application functionality. The following are representative examples:

Commercial Banking Application

This banking application involved complex business and presentation logic defined in 34 use case documents. Functional test cases for 24 of these use cases were generated from functional models using Critical Logic's functional modeling software (DTT). Functional test cases for the remaining 10 use cases were manually written by experienced test engineers. The application went through a three-phase test cycle where the functional tests were applied in the System Integration Test (SIT) phase, and then a select group of business experts (CORE) applied their tests. Finally, a separate group of test engineers performed a final comprehensive UAT (User Acceptance Test) phase using their own set of test cases.

Counts were kept of defects and the test phase in which each was found. If Cause-Effect Modeling and test design is effective, these tests should find most, if not all, of the functional defects – leaving relatively few to be detected in later test phases. Earlier detection means less rework. The results of the analysis are presented in the following chart to the right.



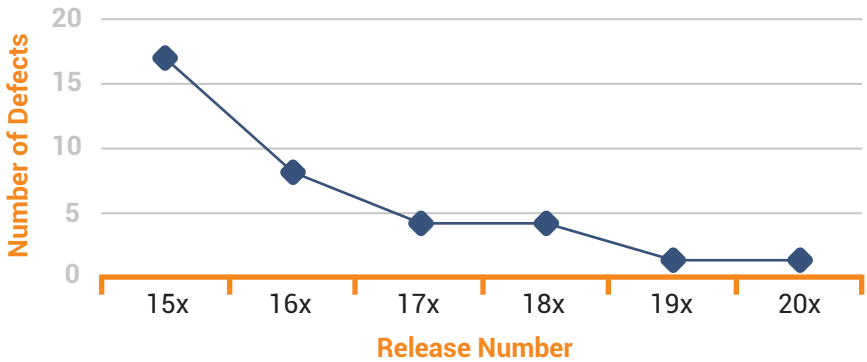
As the numbers make clear, the Cause-Effect Modeling tests were highly effective in identifying defects when compared to traditional ad-hoc approaches to test case design. In this project, the same test engineers did both the model-based test design and the ad-hoc test design, suggesting that it is the Cause-Effect Modeling that provides the value, not differences in test teams. As the numbers clearly show, ad-hoc test approaches unfortunately justify the need for multiple test phases using different teams, imposing higher project costs and later defect discovery.

Industrial MRP

An essential capability for Oracle (formerly known as Sun Microsystems) is the ability to define, price, and order complex hardware and software configurations to meet a customer's needs. To support its worldwide sales staff, Oracle created a "configuration support" application which allows sales representatives to build hardware configurations, generate customer quotes, and pass the approved sale on to manufacturing for fulfillment.

The "Configurator" application is complex. There are more than 60,000 business rules describing all options for integrating and pricing Oracle products. The fast pace of Oracle technology development means that each month, 25% of the business rules change as new products are introduced and current products changed or deleted. With this level of complexity and change, testing presents a huge challenge. ANY defect is costly and visible to the customer. At the same time, frequent, thoroughly tested releases are required to keep pace with product releases. There is no room for error and no time to go back and get it right later.

Initially, Oracle was experiencing excessive defect escapes into production, including errors of a severity that made configuration and sale of certain products impossible. Cause-Effect Modeling and test case design was instituted on a product-by-product basis over several releases and the number of defect escapes carefully tracked.



As the chart clearly indicates, production defect escapes immediately begin dropping and eventually reach zero or near-zero rates for ongoing releases.

Cost Avoidance and Time-to-Market Improvement

Substantial elimination of defect escapes is highly desirable, but it must be looked at in the context of the cost and time required to obtain these results. If a 90% reduction in defect escapes is only possible by tripling the test effort, then the ROI may be questionable. Fortunately, in all of the examples cited above, the delivery costs and time were reduced.

For the Business Credit application, reducing the number of requirements defects to 4% also reduced the project time by approximately 5 weeks or a 15% time and cost savings over the entire life of the project. Most of this gain was simply a result of avoided rework time and associated costs.

In the commercial banking application, recognition that the model-based tests were successful in locating most if not all functional defects allowed the UAT and CORE team to reduce and redirect their test efforts with the result that each full test cycle was shortened from 4 weeks to 2 weeks, or a 50% decrease in testing time.

The MRP example leveraged the model-based technology further. In this case, the models were used to create automated XML test scripts directly from the functional models. All test cases are executed in an automated environment, eliminating the need for laborious manual execution. Using this approach, every release experiences a full regression test (over 25,000 test cases), all run in a matter of 2-3 days. Full regression would be impossible if it had to be done manually. Moreover, once the models were built and all test maintenance is done simply by modification of the models, the total number of QA resources required for each release shrank by 33%. Another objective was time-to-market improvement. The original process required close to three months from start to finish. Cause-Effect Modeling and test automation has reduced the cycle time by 2/3 so that now the process requires only one month to complete.

While reductions in test time and staffing are desirable, the biggest focus must be placed on rework avoidance. 40-50% of total project costs are typically expended in rework, specifically in fixing defects after they have already made their way into the code and are discovered during test or production. Even if there is no reduction in the staffing or time required to model, generate, and execute tests, the ability to avoid defects or to find them in the earliest test phases means substantial rework elimination.

Summary and Conclusion

IT organizations rarely keep ongoing metrics about projects and processes. This makes it difficult to evaluate the effectiveness of new tools or process change. Such is especially the case for quality metrics, since by the time the project gets to the testing phase (where quality is definitively determined), any number of project factors will be cited to explain the achieved quality. Measuring the impact of significant testing and quality changes thus requires special effort to collect and interpret the data. For Functional Modeling, the expected improvements are in the avoidance of functional defects prior to coding and the detection of all significant remaining functional defects in the application during functional testing.

The projects cited in this paper were carefully measured to determine the impact of Cause-Effect Modeling and test design on defect rates. The evidence is clear that this approach does make a major contribution to the reduction in defects for delivered applications. Moreover, the ability of this technology to identify functional ambiguities in specifications and thus reduce the number of defects introduced into code means that rework costs will be substantially reduced as well.

To learn more, contact Critical Logic at www.Critical-Logic.com