New Directions for Software Metrics

Norman Fenton
Agena Ltd and Queen Mary University of London

Keynote Talk
CIO Symposium on Software Best Practices
Savoy Hotel
27 September 2006

Introduction

During the 1990s I wrote a book on software metrics that has sold over 30,000 copies worldwide. I'll have something more to say about this book shortly.

But certainly one of the best books on software metrics was written by Bob Grady of Hewlett Packard in 1987. Bob was responsible for what I believe was recognised as the first true company-wide metrics programme. His book described the techniques and experiences associated with that at HP.

A few years later I was at a meeting where Bob told an interesting story about that metrics programme. He said that one of the main objectives of the programme was to achieve process improvement by learning from metrics what process activities worked and what ones didn't. To do this they looked at those projects that in metrics terms were considered most successful.

These were the projects with especially low rates of customer-reported defects. The idea was to learn what processes characterised such successful projects. It turned out that what they learned from this was very different to what they had expected.

I am not going to tell you what it was they learnt until the end of my presentation; by then you may have worked it out for yourselves.

I will start by giving a lightening history of software metrics, explaining why almost all metrics activities can be traced to a couple of very simple technical objectives about resource and quality that are then wrapped up in standard statistical methods. The good news is that globally software metrics has provided some decent benchmarking information about software projects and locally has led in to managers having greater control over projects. The bad news is that traditional metrics and models don't provide the kind of quantitative information that software managers have a right to expect. I am talking about information like:

- For a problem of this size, and given these limited resources, how likely am I to achieve a product of suitable quality?

- How much can I scale down the resources if I am prepared to put up with a product of specified lesser quality?
- The model predicts that I need 4 people over 2 years to build a system of this kind of size. But I only have funding for 3 people over one year. If I cannot sacrifice quality, how good do the staff have to be to build the systems with the limited resources?

This is all about quantified risk assessment to help decision-making. Software metrics has never properly addressed this true objective. I will try to convince you that the true objectives can only be met by considering causal factors in the software process. I will show you a simple example of a causal model in action and show results achieved in practice that are very impressive compared with traditional approaches.

Software Metrics History

Since the early 1970's most serious Software metrics activities generally work in the following way (see Figure 1).

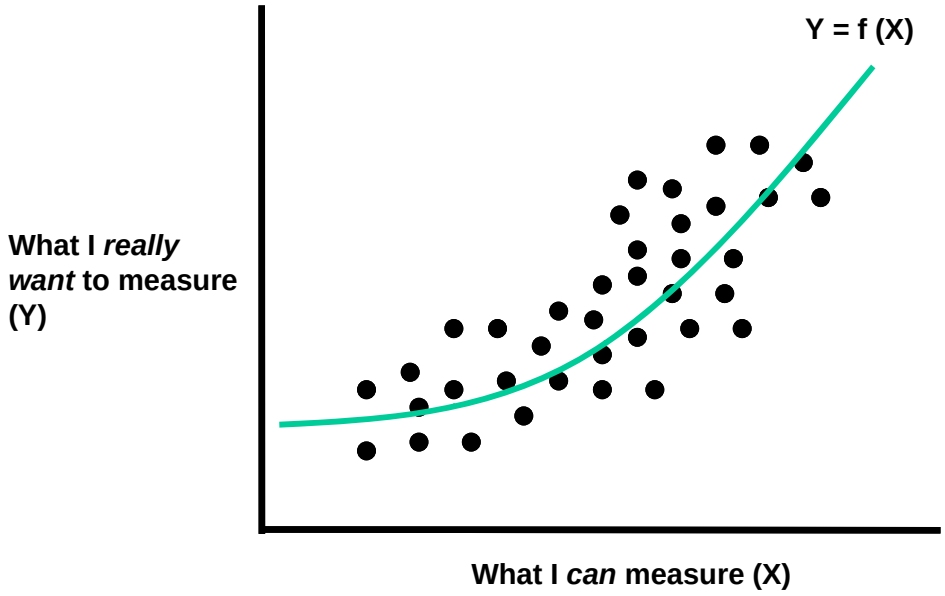

Figure 1 Typical metrics approach

There is something Y about a project that we really want to assess or predict e.g effort required to complete it or (as in the HP case) the quality delivered to end-users. The problem is that it to measure or predict Y we have to find one or more things X that we can measure relatively easily.

Think of the analogy of measuring room temperature – that's hard to measure directly so instead we measure the length of a mercury column, which is much easier. So we use X to infer what we really we want to measure or predict. For example, in software projects code size is a very commonly used X for effort and defects found in test is a very commonly used X for quality.

So, using past project data, we plot the two measures X and Y (where each dot represents a project). Then we apply statistical regression methods to find the best-fit function f. With this function f we can make predictions of Y in future projects once we've measured X.

So the defining method of software metrics is conceptually simple. And it turns out that just about every single technical development in the history software metrics driven by two types of Y values:

- Something like resource
- Something like quality

Hence most of software metrics has been driven by models that tend to look something like:

- 'productivity'=size/effort
  where size is often measured by LOC and effort by programmer month

- 'effort'=a*sizeb
  where effort again is some function of size. The form here is a classic one proposed by Boehm in his COCOMO model where size was a derivative of LOC. The values for a and b were calculated by the statistical regression approach that I previously indicated.

- 'quality'=defects/size
  Quality is again some function of size, so that the most common measure of quality is the very one that Bob Grady and HP used to determine which were the successful projects – those which had the lowest rate of defects reported by customers.

Almost as an aside to my talk, it is pretty clear from that brief review that size seems to matter a lot. For the models to be of practical use size has to be easily measurable. And the most widely recognised and easily measurable size measure is LOC (Lines of Code). But what this means is that LOC is used as a surrogate for different notions of size, notably complexity and difficulty. Unfortunately LOC, while being an excellent measure of programme length, is a rather poor measure of complexity or difficulty.

This and many other weaknesses of LOC as a size metric led to work on improved size metrics. The holy grail is a size metric to replace LOC that is not only language independent, but can capture notions of complexity and difficulty. A notable example here is the Function Point metric (FP). In principle this metric is ideal because it measures the functional complexity of the problem to be solved rather than the delivered solution. Hence, it is measurable at the start of the project. The drawback is that it is itself complex to measure.

The limitations of LOC also lead to a massive interest in finding improved complexity metrics. Here I mean complexity of the solution rather than the problem. Examples here are code structure metrics such as McCabe's cyclomatic complexity and metrics that can be extracted from designs rather than just code. There are, for example, now a whole host of metrics that measure object oriented design properties.

The Good News and The Bad News

The decent news about software metrics is that:

- Published empirical results and models have led to some very useful benchmarking standards
- A lot if IT companies have some kind of metrics program even if they may not regard them explicitly as such and where they do managers gain greater control over projects because they are better informed of what is going on
- Metrics has been a phenomenal success judged by academic/research output. There are thousands of published papers, dozens of text books, and several dedicated conferences. Software metrics is now accepted as a core component software engineering.
- Metrics are now a standard component of programmer toolkits. There are, for example free plugins for programming environments like Eclipse that automatically compute hundreds of different code and design based metrics.

But there is some bad news:

- Much academic metrics research is inherently irrelevant to industrial needs
- Any software metrics 'programme' that depends on extensive data collection is probably doomed to failure
- Much industrial metrics activity is poorly motivated because it is a classic grudge purchase. People do it simply because a standard or a customer mandates it. And the reason is because, as I said in the introduction, software metrics has not delivered what managers and devleopers really want, which is as quantitative risk assessment and risk management.

Before I explain why I want to make it clear that I am not assigning blame to unnamed individuals. It is instructive look at my own book and see what it says about risk and decision-making in software projects. So let's look up in the index the word risk then the word decision. What do we find? There is absolutely nothing!

My only defence is one of ignorance – when I wrote the book I was wearing the same blinkers as most others metrics researchers.

So what *is* the problem with the traditional approach?

The problem with the regression-based models I have summarised is that, instead of predicting the future, all they really do is predict the past. It is a little bit like driving by looking through your rear-view mirror.

Let me give an example.

We are interested in predicting post-release faults as early as possible. There is a very strong belief that you can use use fault data collected during pre-release testing to make predictions about the number and location of faults that you might expect to get in operation (that is, post-release).

So let's look at what you might expect to happen in a large system made up of many modules (components). Let's suppose that you collect information about faults found in each component.

The number of pre-release faults is the total found during the various testing stages prior to release. The number of post-release faults are all those found by users after release.

The popular assumption is that there is a clear positive correlation. The more faults there are in a module pre-release, the more there are likely to be post-release.
In other words you might expect to see something like Figure 2 where each dot represents a module.

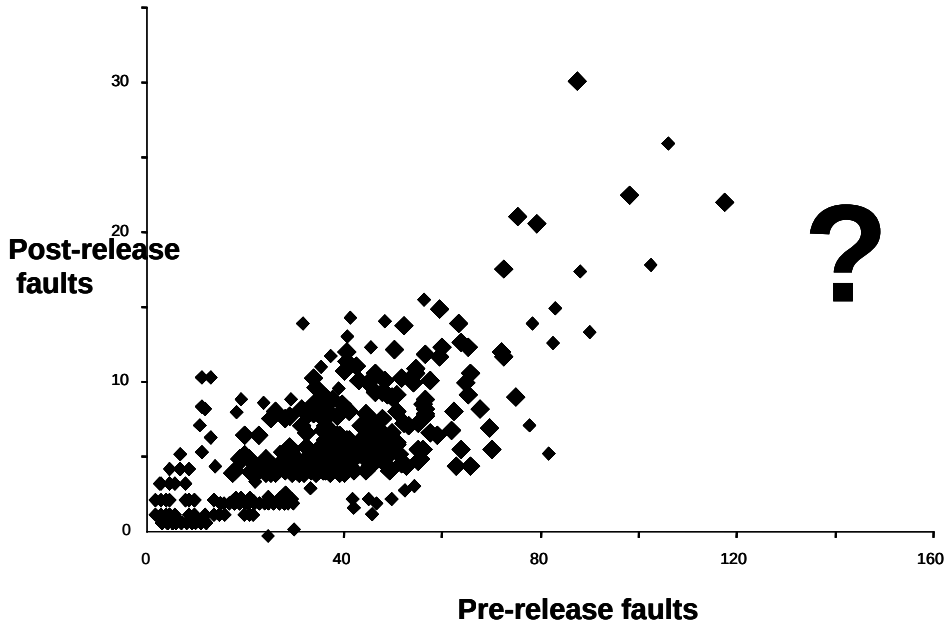

Figure 2 Relationship between pre and post release faults. Each dot represents a module

But is is a reasonable assumption? Let's think of an analogy. If you know that somebody has eaten a large lunch do you believe that they will also eat a large dinner later that day. You might argue people who eat large lunches are inherently big eaters or simply greedy and therefore such people are also likely to eat large dinners. On the other hand it seems

reasonable to assume that somebody who has eaten a large lunch will simply be too full up to eat a large dinner.

By the same account there may be good reasons why modules which are the most fault prone pre-release are not necessarily the most fault-prone post release. In fact, while the positive correlation shown here has been observed in some systems, it is by no means common, and Figure 3 is a plot of actual data taken from a major telephone switching system.

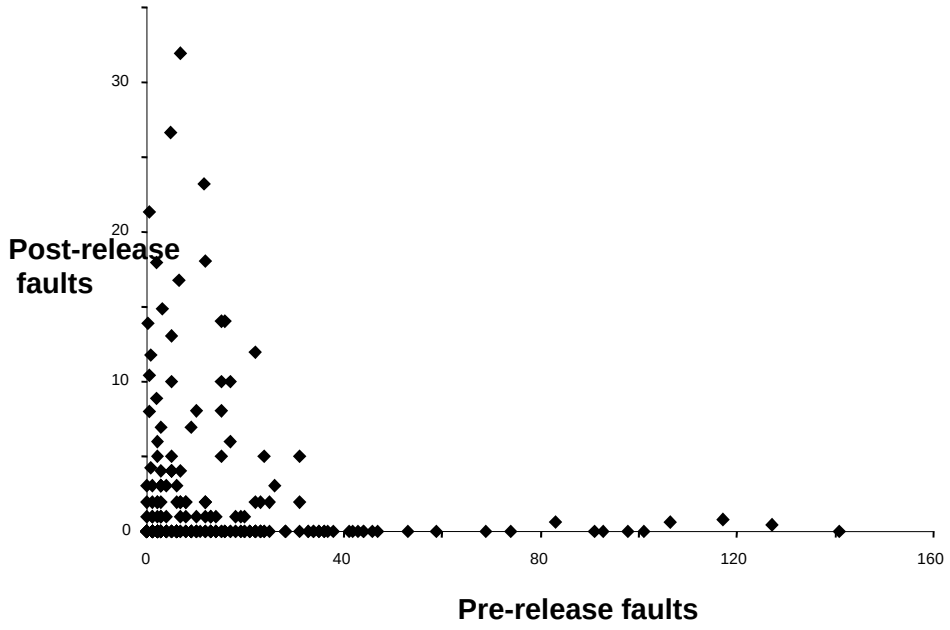

Figure 3 Real system plot of pre-release versus post-release faults

This was a system of several millions of LOC of software. Each dot represents a module sampled randomly. A module typically has around 2000 LOC. What you can see is that modules which are fault-prone pre-release tend to reveal zero faults post-release. Conversely, the problem modules post-release are those which had few faults pre-release.

This result is not a one-off. We have found similar behaviour in other complex software systems. The regression based approach here simply does not work because it fails to take account of some very obvious causal explanations for what is going on.

It turns out that some of the modules that had a high number of faults pre-release just happened to be very well tested, leading to a low number of post-release faults. Others were simply never run in operation. Conversely the modules with high number of post-release faults were never properly tested before operation. The amount of testing is therefore a very simple explanatory factor that must be incorporated into any predictive model of defects.

Need for a causal approach

What we really need to meet our true objective for Software metrics is an approach that incorporates:

- genuine cause and effect relationships;
- uncertainty
- multiple types of evidence
- expert judgement;
- incomplete information.

We also want all our assumptions to be visible and auditable.

The really good news is that:

- Quantitative risk assessment and risk management meeting those requirements is possible. And I mean real intelligent analysis that enables you to consider trade-off between time, effort, scope and quality.
- You don't need a heavyweight metrics programme to make it work. Nor do you need a heavyweight or pre-defined software process. I know (because of my background in software metrics) that these assumptions are the kiss of death
- We've done most of the hard stuff for you. It's all about building causal models rather than regression models and populating the models with the best of what is known from empirical data.

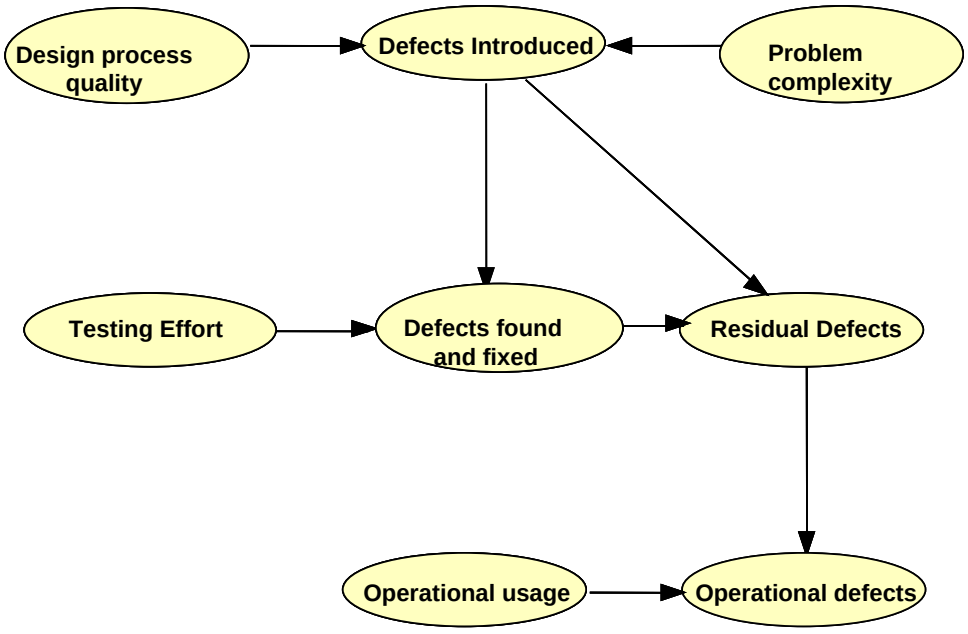

Figure 4 Causal Model (Bayesian net)

Figure 4 shows a simple example of a causal model (or Bayesian Net). Here the number of operational defects (those found by customers) in a software module is what we are really interested in predicting. We know this is clearly dependent on the number of residual defects. But it is also critically dependent on the amount of operational usage. If you do not use the system you will find no defects irrespective of the number there. The number of residual defects is determined by the number you introduce during development minus the number you successfully find and fix. Obviously defects found and fixed is dependent on the number introduced. The number introduced is influenced by

- problem complexity and
- design process quality

The better design the less defects, the less complex the less defects.
Finally, how many defects you find is influenced not just by the number there to find but also by the amount of testing effort..

So a causal model is simply a graph whose nodes represent variables (some of which are known and some of which are unknown) and whose arcs represent causal or influential relationships.

We call these Bayesian nets because associated with the nodes are probabilities that capture the uncertainty about the state of the node given the states of parent nodes. And there is a Bayesian calculation engine that updates the probability of all unknown variables whenever you enter known values. Unlike traditional statistical models these known values can be entered anywhere. This allows backwards as well as forwards reasoning.

Figure 5 shows this simple model in the AgenaRisk tool.

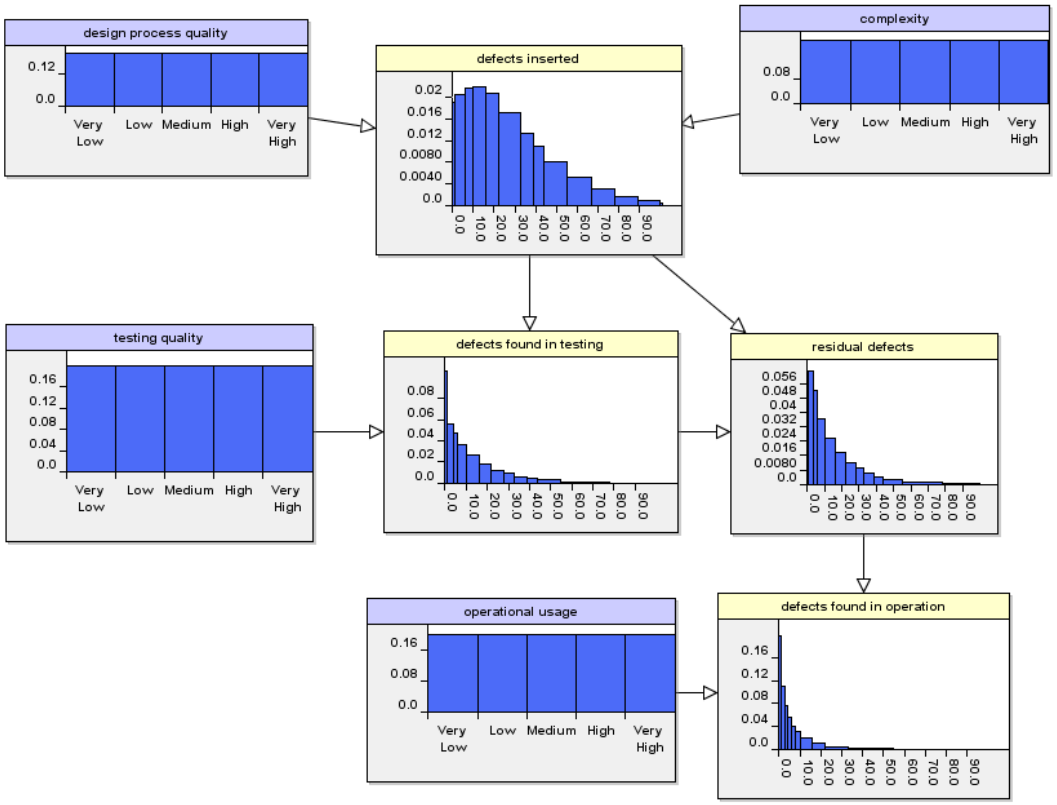

Figure 5 BN model in the AgenaRisk Tool

The probability distributions shown here are the so called marginal values – this represents our uncertainty before we enter any specific information about this component. What it means, for example, is that the component is just as likely to have very high complexity as very low, and that the number of defects found and fixed in testing is in a wide range where the median value is about 18-20. As we enter observations about the module the probability distributions update as shown in Figure 6.

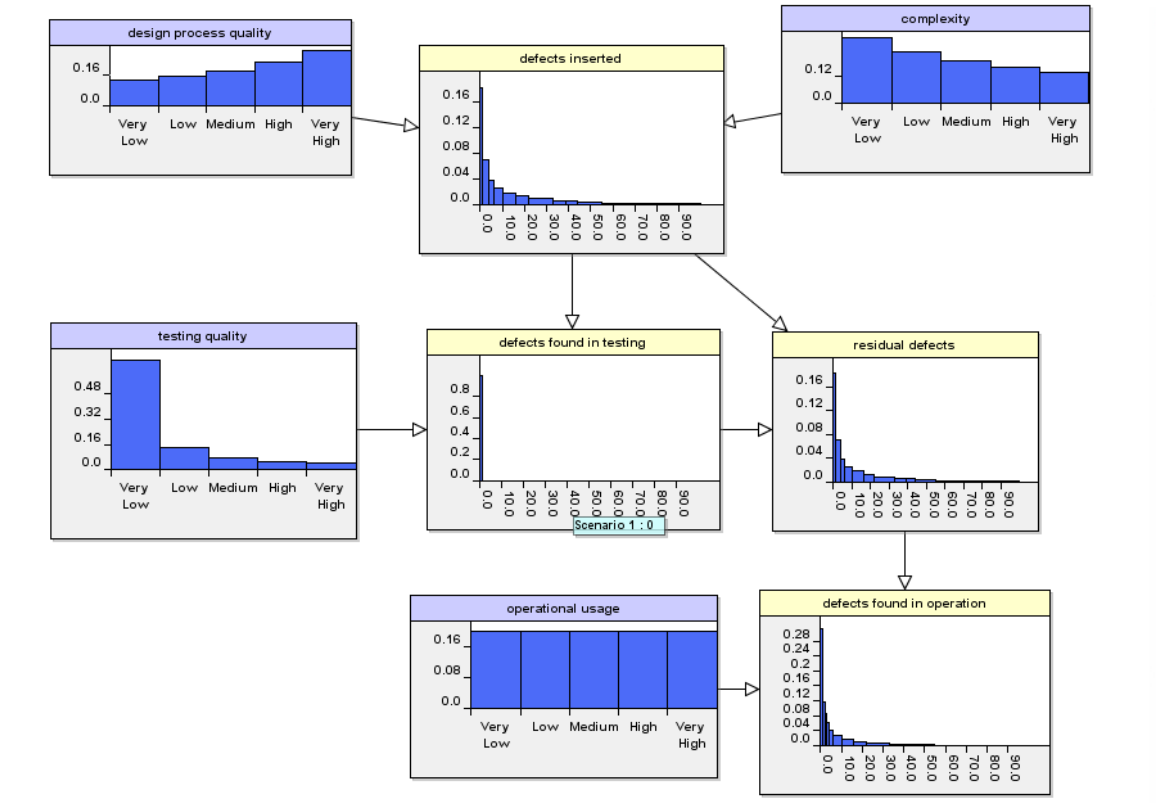

Figure 6 Zero defects found in testing

Here we have entered the observation that this module had 0 defects found and fixed in testing. Note that all the other distributions changed. The model is doing both forward inference to predict defects in operation and backwards inference about, say, design process quality. This low number does indeed lead to a belief that the post-release faults will drop, but actually the most likely explanation is very low testing and lower than average complexity.

If we find out that the complexity is actually high (Figure 7) then the expected number of operational defects increases and we become even more convinced of the inadequate testing.

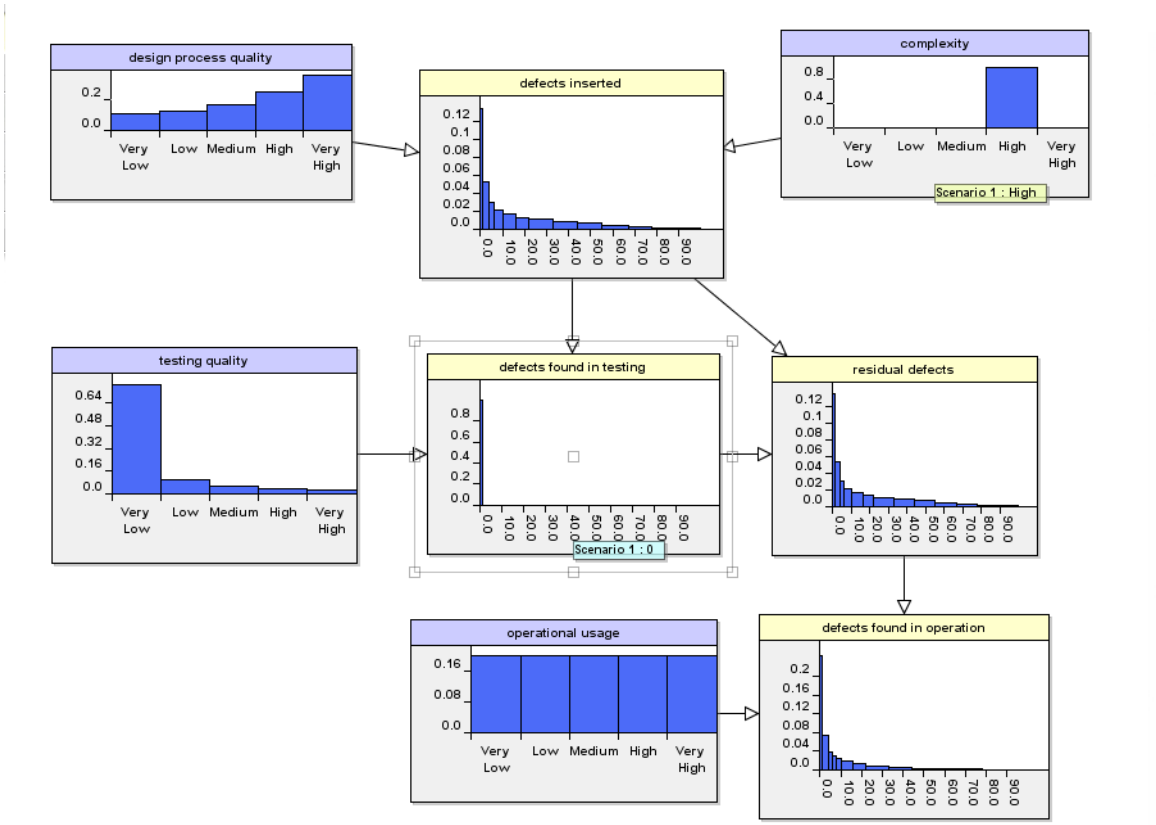

Figure 7 High complexity observed

So far we have made no observation about operational usage. If, in fact the operational usage is very high (Figure 8) then what we have done is replicate the apparently counter-intuitive empirical observations – a module with no defects found in testing has a high number of defects post-release.…

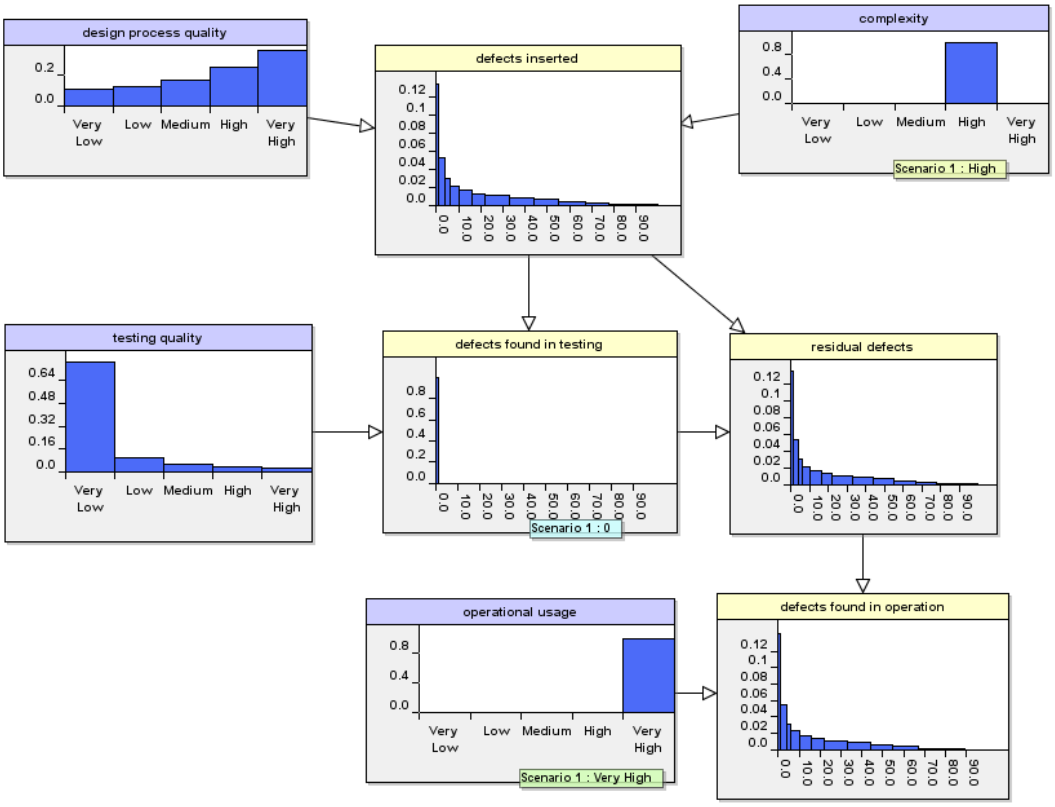

Figure 8 Very high operational usage

But suppose we find out that the test quality was very high (

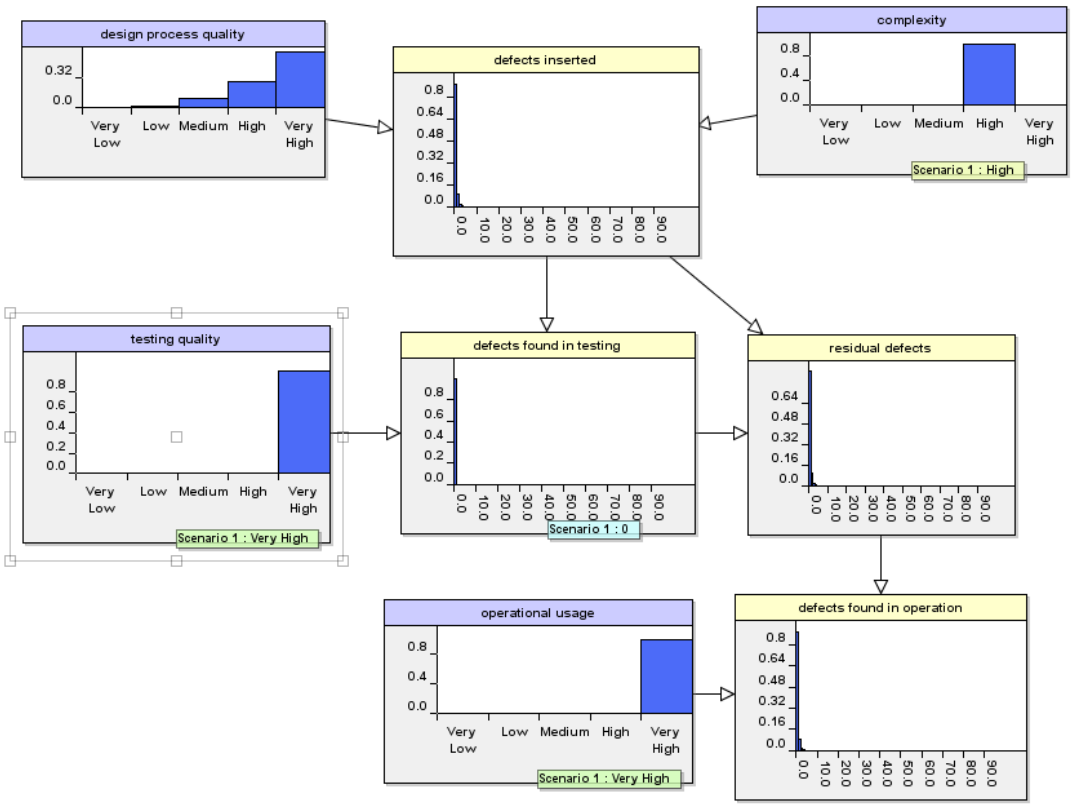

Figure 9 Testing quality very high

Then we completely revise out beliefs. We are now pretty convinced that the module will be fault free in operation. Note also that the 'explanation' is that the design process is likely to be very high quality.

Now let us restart the model. Suppose this time we observe a much higher than average number of defects in testing – 35 (Figure 10).

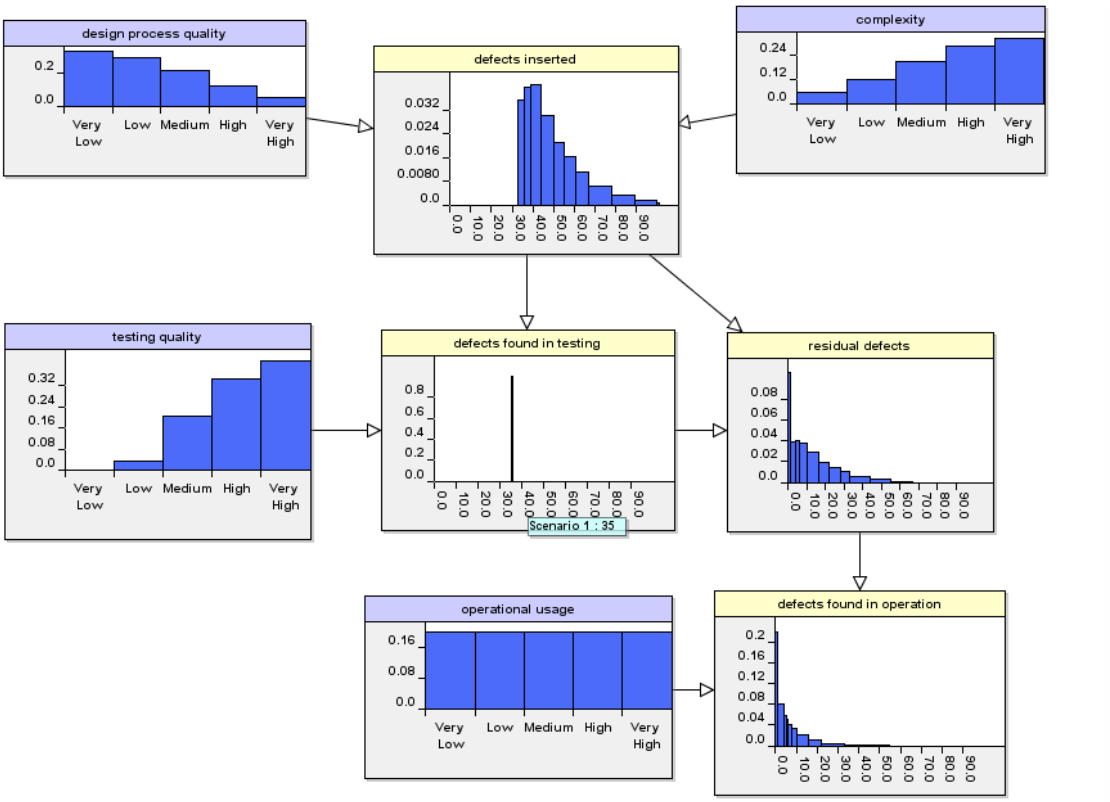

Figure 10 35 defects found in testing

The model does predict a slightly higher number of post-release faults, but more significantly are the changes to the distributions of testing quality, complexity etc. In fact if we enter specific observations as shown in Figure 11 we find that the prediction for number of operational defects is now very low. So again we have reproduced the converse counter intuitive empirical result of a module with high number of defects pre-release having a low number in operation. What is crucial to note here is that the model is actually predicting such a counter-intuitive result.

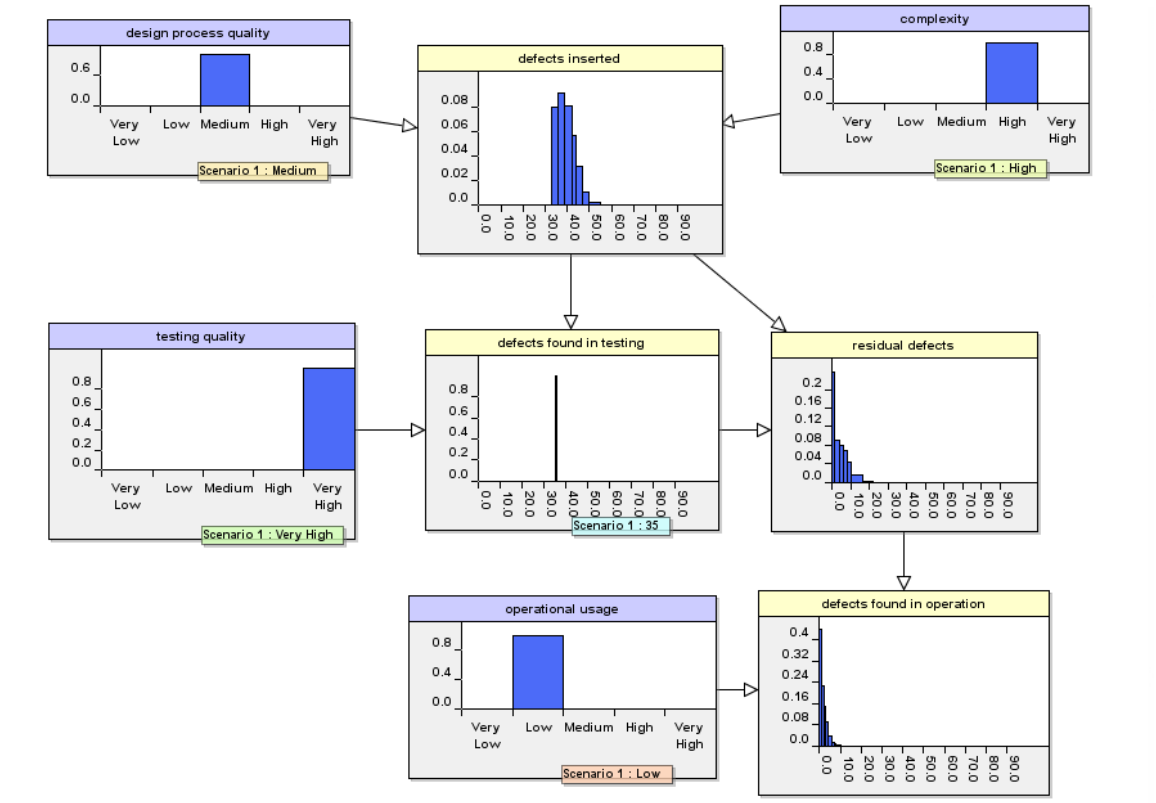

Figure 11 Some observations entered

Finally we reset the model again and this time use the model to argue backwards. Suppose that before development we know that this is a critical module that might have a requirement for 0 defects in operation. The model with this 'observation' is shown in Figure 12

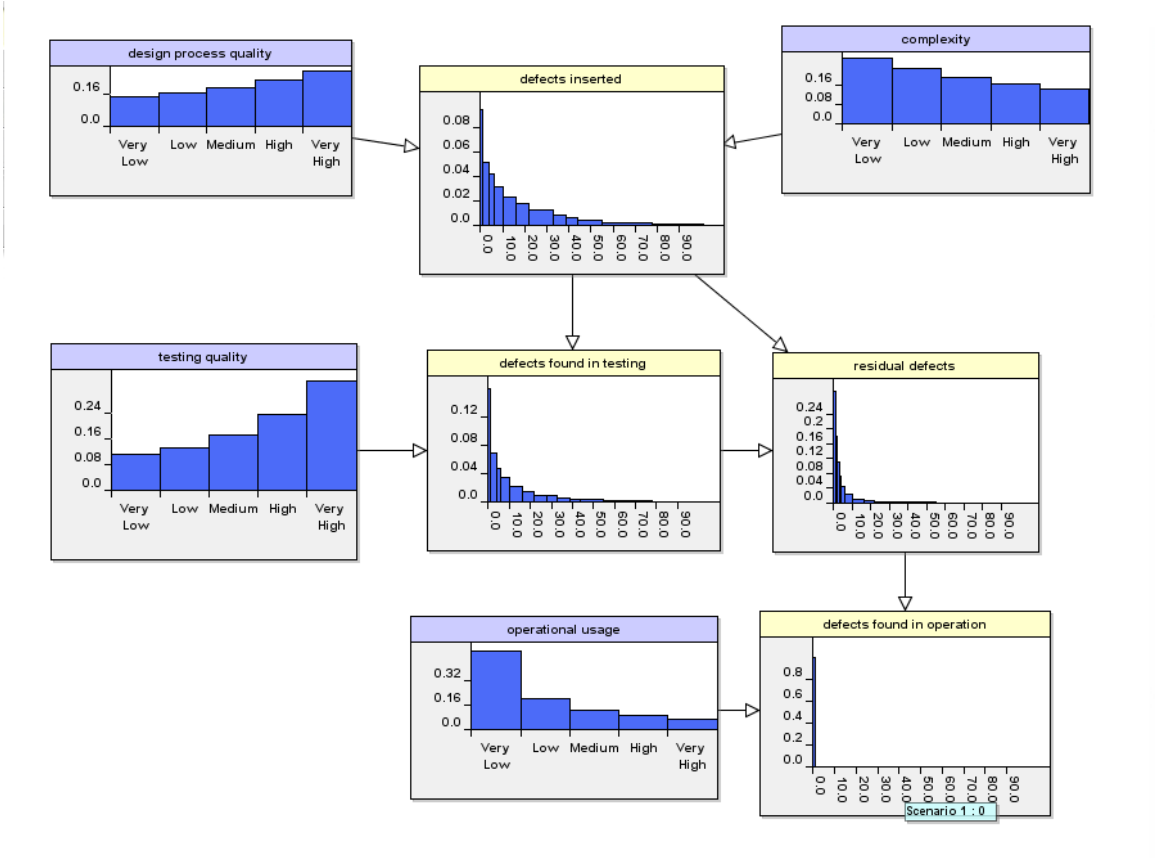

Figure 12 Zero defects in operation

The model looks for explanations for such a state of affairs. The most obvious way to achieve such a result is simply to not use the module much. But suppose we know it will be subject to very high usage as shown in Figure 13.

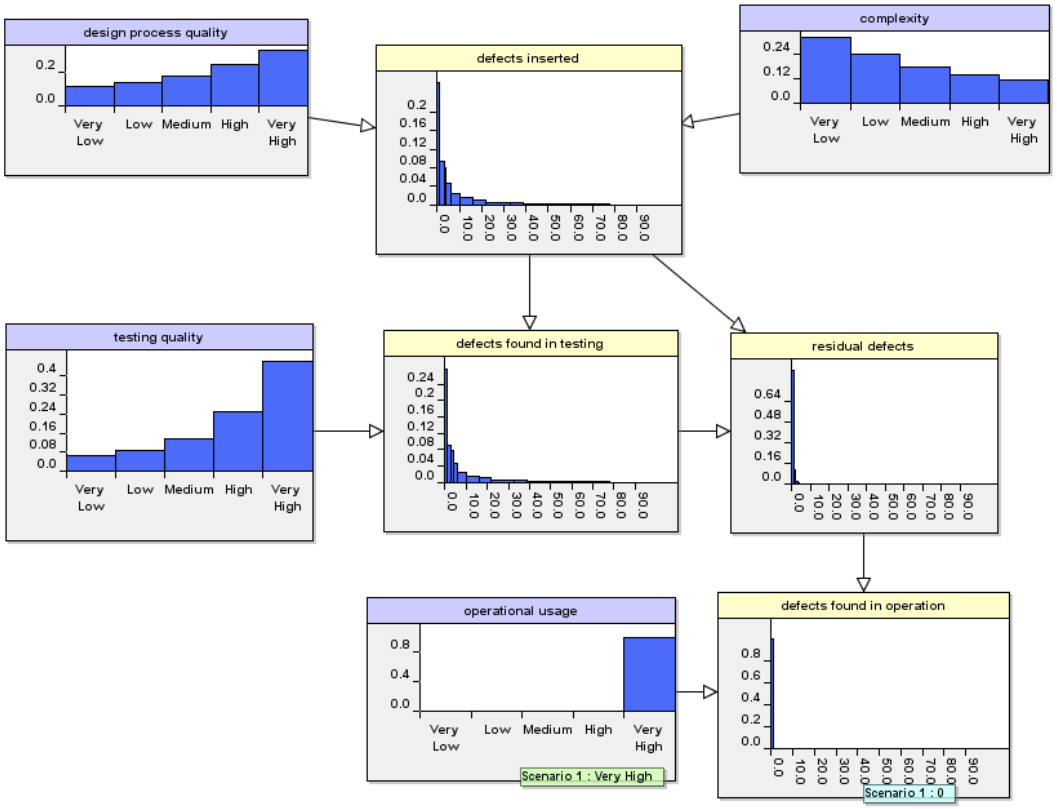

Figure 13 Very high operational usage

Then the model adjusts the beliefs about the other uncertain variables. A combination of lower than average complexity, higher than average design quality and much higher than average testing. But suppose we cannot assume our testing is anything other than average (Figure 14).

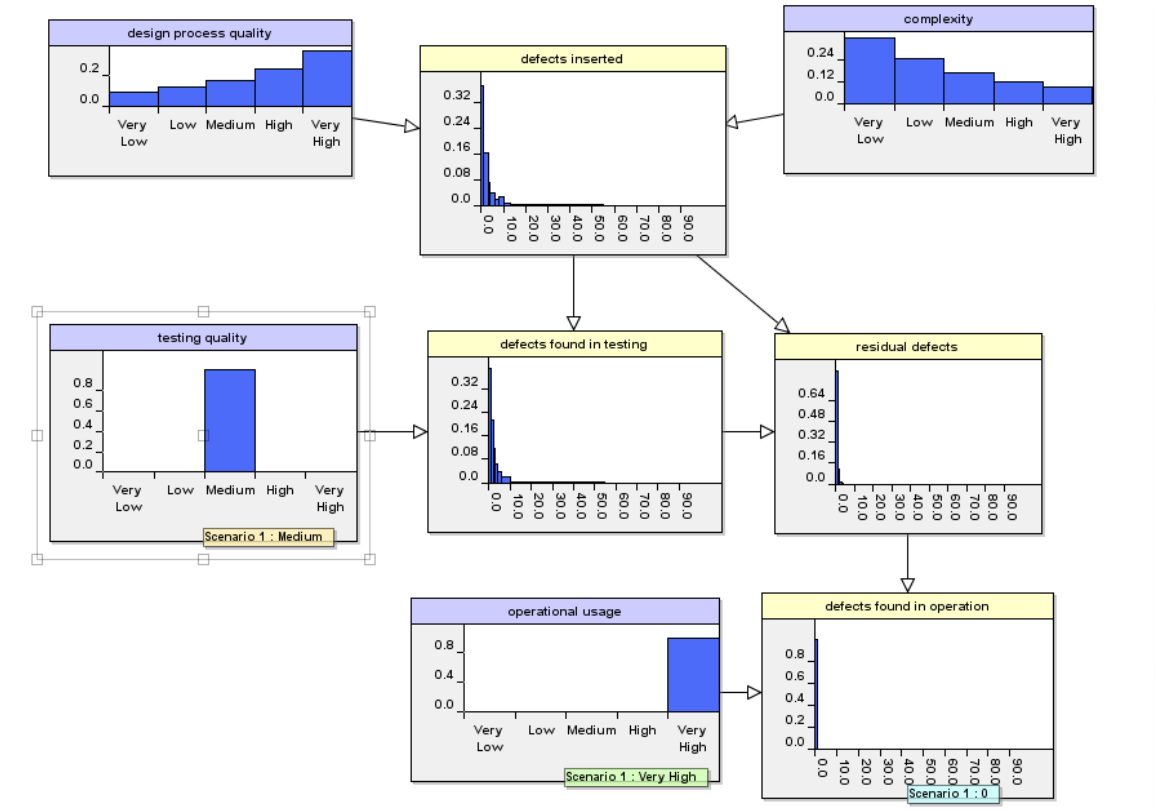

Figure 14 Average testing quality

Then better design quality and lower complexity are needed. But suppose the complexity is very high (Figure 15).

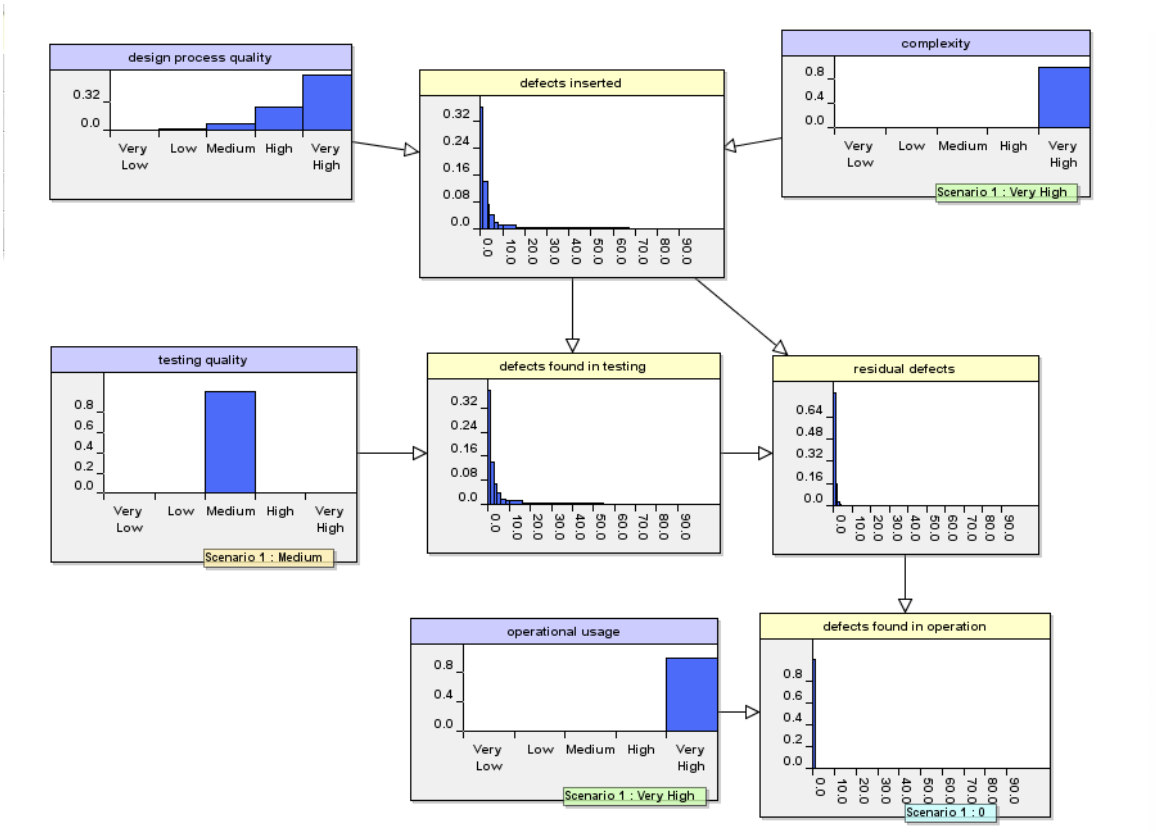

Figure 15 Very high complexity

Then we are left with a very skewed distribution for design process quality. What the model is saying is that, if these are the true requirements for the module, then you are very unlikely to achieve them unless you have a very good design process.

Evaluation

That model was a very simplified version of a model we developed in collaboration with Philips, QinetiQ and IAI. Philips did extensive trials involving 30-odd projects of software for consumer electronics devices (TVs etc). The results of the evaluation are shown in Figure 16.

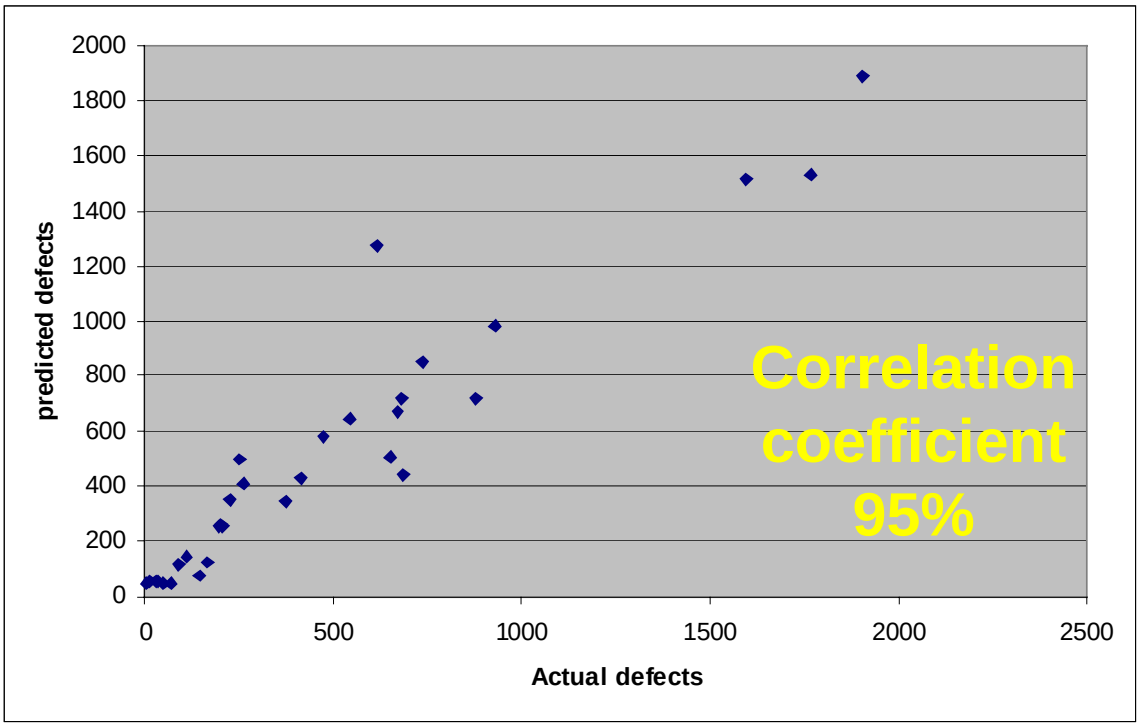

Figure 16 Correlation of actual versus predicted defects

The headline figure shown here was the comparison of the model predicted defects with actual defects. As a baseline this was compared with the best statistical regression models using the same data. Those models achieved typically less than 80%.

Given the nature of the application such improvements in accuracy are crucial – meaning improved decisions about, e.g., when to stop testing. But these issues are almost insignificant compared to the greater power that such an approach provides for the software manager. A model like that really does meet the objectives of quantified risk assessment and decision making.

Conclusions

- Data-mining and metrics is great for establishing an empirical basis
- And we could not have built the models without all of this work.
- But none of this address the true needs of quantitative risk assessment.

The true needs are addressed by using the kind of causal models that I have demonstrated. These models enable us to answer the kind of questions that software project managers and developers need to answer during the lifetime of their projects.

Reasoning of this kind can help avoid software project failures because it helps managers do proper quantified risk assessment early and make more rational decisions based on

software metrics. And the great news is that you can use the technology now in tools such as AgenaRisk (www.agenarisk.com).

Finally back to Bob Grady: What did they really learn from the projects with the lowest rate of customer-recorded defects? They were mostly so bad that they never got used. A causal model would have revealed that.