Chapter 1
Software Project and Quality Modelling Using Bayesian Networks

Norman Fenton
Queen Mary, University of London, United Kingdom

Peter Hearty
Queen Mary, University of London, United Kingdom

Martin Neil
Queen Mary, University of London, United Kingdom

Łukasz Radliński
Queen Mary, University of London, United Kingdom, and University of Szczecin, Poland

ABSTRACT

This chapter provides an introduction to the use of Bayesian Network (BN) models in Software Engineering. A short overview of the theory of BNs is included, together with an explanation of why BNs are ideally suited to dealing with the characteristics and shortcomings of typical software development environments. This theory is supplemented and illustrated using real world models that illustrate the advantages of BNs in dealing with uncertainty, causal reasoning and learning in the presence of limited data.

INTRODUCTION

Software project planning is notoriously unreliable. Attempts to predict the effort, cost and quality of software projects have foundered for many reasons. These include the amount of effort involved in collecting metrics, the lack of crucial data, the subjective nature of some of the variables involved and the complex interaction of the many variables which can affect a software project. In this chapter we introduce Bayesian Networks (BNs) and show how they can overcome these problems.

We cover sufficient BN theory to enable the reader to construct and use BN models using a suitable tool, such as AgenaRisk (Agena Ltd. 2008). From this readers will acquire an appreciation for the ease with which complex, yet intuitive, statistical models can be built. The statistical nature of BN models automatically enables them to deal with the

DOI: 10.4018/978-1-60566-758-4.ch001

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

uncertainty and risk that is inherent in all but the most trivial software projects.

Two distinctive types of model will be presented. The first group of models are primarily causal in nature. These take results from empirical software engineering, and using expert domain knowledge, construct a network of causal influences. Known evidence from a particular project is entered into these models in order to predict desired outcomes such as cost, effort or quality. Alternatively, desired outcomes can be entered and the models provide the range of inputs required to support those outcomes. In this way, the same models provide both decision support and trade off analysis.

The second group of models are primarily parameter learning models for use in iterative or agile environments. By parameter learning we mean that the model learns the uncertain values of the parameters as a project progresses and uses these to predict what might happen next. They take advantage of knowledge gained in one or more iterations of the software development process to inform predictions of later iterations. We will show how remarkably succinct such models can be and how quickly they can learn from their environment based on very little information.

BACKGROUND

Before we can describe BN software project models, it is worthwhile examining the problems that such models are trying to address and why it is that traditional approaches have proved so difficult. Then, by introducing the basics of BN theory, we will see how BN models address these shortcomings.

Cost and Quality Models

We can divide software process models into two broad categories: cost models and quality models. Cost models, as their name implies, aim to predict the cost of a software project. Since effort is normally one of the largest costs involved in a software project, we also take "cost models" to include effort prediction models. Similarly, since the "size" of a software project often has a direct bearing on the effort and cost involved, we also include project size models in this category. Quality models are concerned with predicting quality attributes such as mean time between failures, or defect counts.

Estimating the cost of software projects is notoriously hard. Molokken and Jorgensen (2003) performed a review of surveys of software effort estimation and found that the average cost overrun was of the order 30-40%. One of the most famous such surveys, the Standish Report (Standish Group International 1995) puts the mean cost overrun even higher, at 89%, although this report is not without its critics (Glass 2006). Software quality prediction, and in particular software defect prediction, has been no more successful. Fenton and Neil (1999) have described the reasons for this failure. We briefly reproduce these here since they apply equally to both cost and quality models.

1. Typical cost and quality models, such as COCOMO (Boehm 1981) and COQUALMO (Chulani & Boehm 1999) take one or two parameters which are fed into a simple algebraic formula and predict a fixed value for some desired cost or quality metric. Such parametric models therefore take no account of the inaccuracy in the measurement of their parameters, or the uncertainty surrounding their coefficients. They are therefore unable to attach any measure of risk to their predictions. Changes in parameters and coefficients can be simulated in an ad-hoc fashion to try to address this, but this is not widely used and does not arise as a natural component of the base model.
2. Parametric models cannot easily deal with missing or uncertain data. This is a major problem when constructing software process

models. Data can be missing because it simply wasn't recorded. It can also be missing because the project is in a very early stage of its lifecycle. Some of the problems appear quite prosaic, for example ambiguity arises because of difficulties in defining what constitutes a line of code. Do comments, empty lines and declarative statements count? Similarly there is uncertainty about what counts as a defect. What is regarded as a defect in one project might be regarded as a user change request in another. Do only post production software failures count as defects, or do defects uncovered during testing count towards the total?

3. Traditional models have difficulty incorporating subjective judgements, yet software development is replete with such judgements. The cost and quality of a software project clearly depend to a significant extent on the quality of the development team, yet such a metric is rarely available form that is easily measured and is more usually the subject of opinion than of measurement.
4. Parametric models typically depend on a previous metrics measurement programme. A consistent and comprehensive metrics programme requires a level of discipline and management commitment which can often evaporate as deadlines approach and corners are cut. Failure to adjust the coefficients in a parametric model to match local conditions can result in predictions which are significantly (often several hundred percent) different from actual values (Briand et. al. 1999; Kemerer 1987).
5. Metrics programmes may uncover a simple relationship between an input and an output metric, but they tell us nothing about how this relationship arises, and crucially, they do not tell us what we must do to improve performance. For example, if we wish to reduce the number of defects, are we better off investing in better test technology, in more training for the test team, or more experienced developers?

Many attempts have been made to find alternatives to simple parametric models. These include multivariate models (Neil 1992), classification and regression trees (Srinivasan & Fisher 1995), analogy based models (Shepperd & Schofield 1997; Briand et. al. 1999), artificial neural networks (Finnie & Wittig & Desharnais 1997) and systems dynamics models (Abdel-Hamid 1989). However no single one of these approaches addresses all of the problems outlined above. In the next section, we demonstrate how BNs can help overcome these disadvantages and add considerable value and insight to software process modelling.

Introduction to Bayesian Networks

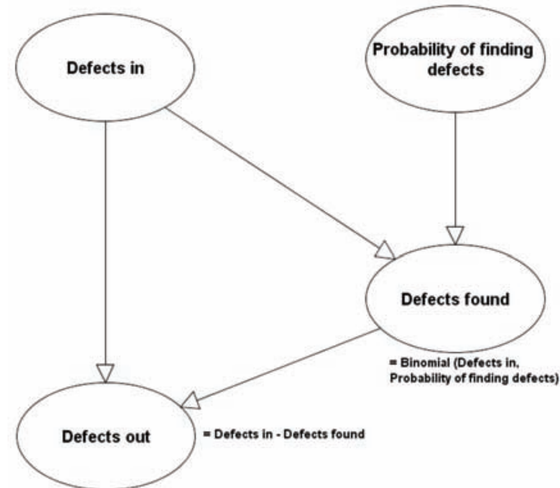
A Bayesian Network (BN), (Jensen, 2001), is a directed acyclic graph (such as the example shown in Figure 1), where the nodes represent random variables and the directed arcs define causal influences or statistical or functional relationships. Nodes without parents (such as the "Probability of finding defects" and "Defects In" nodes in Figure 1) are defined through their prior probability distributions. Nodes with parents are defined through Conditional Probability Distributions (CPDs). For some nodes, the CPDs are defined through deterministic functions of their parents (such as the "Defects Out" node in Figure 1), others (such as the "Defects Found" node in Figure 1) are defined as standard probability distribution functions. Conditional independence (CI) relationships are implicit in the directed acyclic graph: all nodes are conditionally independent of their ancestors given their parents. This general rule makes it unnecessary to list CI relationships explicitly.

Trivial as this example may seem, it actually incorporates a great many of the features that make BNs so powerful. within comparison with regression based models there are a number of

beneficial features and advantages:

1. One of the most obvious characteristics of BNs is that we are no longer dealing with simple point value models. Each of the random variables in Figure 1 defines a statistical distribution. The model simultaneously deals with a wide range of possible outcomes. In particular, its predictions, in this case limited to the "Defects out" node, is in the form of a marginal distribution which is typically unimodal, giving rise to a natural "most likely" median value and a quantitative measure of risk assessment in the form of the posterior marginal's variance.
2. We can run the model without entering any evidence. The model then uses the prior distributions which can be based on empirical studies of a wide range of software projects, such as those provided by Jones (1986; 1999; 2003), or by publicly available databases (ISBSG 2008). Nodes such as "Probability of finding defects" may be assigned priors in this way. The "Defects in" node can also be assigned a prior based on empirical studies. In more complex BNs it will often be assigned a distribution which is conditional on other causal factors, or on evidence gathered from the project.
3. The nodes in this model are all represented by numeric scales. BNs are not limited to this however. Any set of states which can be assigned a probability can be handled by a BN. We shall show shortly how ordinal measures can be used to include subjective judgments into BN models.
4. Unlike parametric models, where the underlying variation in software process measurement has been "averaged" out, this model contains all of the available information. It is therefore not limited to an "average" project, but can encompass the full range of values included in its priors. As we shall see later, this dramatically reduces the amount of information needed in order to adapt a BN model to local conditions and enables us to dispense with the metrics collection phase traditionally used to tune parametric models.
5. Unlike simple regression models the relationship between all the causal factors is explicitly stated. This means that, in addition to using the model to make predictions, we can set our desired outcome (for example to how many defects are acceptable in the released product) and the model will be able to tell us how many "Defects in" are the most likely explanation of this and what our probability of finding a defect must be. In larger models, where our initial conditions are defined by the available resources, such models can be used for management decision support as well as project management and planning.

Figure 1. A simple Bayesian Network illustrating defect detection in a software project



When a variable is actually observed, this observation can be entered into the model. An observation reduces the marginal probability distribution for the observed variable to a unit probability for the observed state (or a small

interval containing the value in the continuous) and zero otherwise. The presence of an observation updates the CPD of its children and, through Bayes theorem (Equation 1), the distributions of its parents. In this way observations are propagated recursively through the model. BN models can therefore update their beliefs about probable causes and so learn from the evidence entered into the model. More information on BNs and suitable propagation algorithms can be found in Jensen (2001) and Lauritzen & Spiegelhalter (1988).

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \quad (1)$$

Dynamic Bayesian Networks

Dynamic Bayesian Networks (DBN) extend BNs by adding a temporal dimension to the model. Formally, a DBN is a temporal model representing a dynamic system which changes state usually over time (Murphy 2002). A DBN consists of a sequence of identical Bayesian Networks, Z_t, $t = 1,2,...,T$ where each Z_t represents a snapshot of the process being modelled at time t. We refer to each Z_t as a *timeslice*. For iterative software development environments this is a particularly apt approach.

The models presented here are all first order Markov. Informally this means that the future is independent of the past given the present $P(Z_t \mid Z_{1:t-1}) = P(Z_t \mid Z_{t-1})$ and in practice it means that we do not need to recompute the model afresh each time a new prediction is needed. The first order Markov property reduces the number of dependencies, making it computationally feasible to construct models with a larger numbers of timeslices. Consistent propagation is achieved using standard junction tree algorithms (Lauritzen & Spiegelhalter 1988). These algorithms provide exact (as opposed to approximate) propagation in discrete BNs and are generally regarded as among the most efficient BN propagation algorithms (Lepar & Shenoy 1998).

Indicator Nodes and Ranked Nodes

Two types of nodes deserve special mention - these are indicator nodes and ranked nodes. Indicator nodes are nodes with no children and a single parent. They are often used in circumstances where the parent is not directly observable but where some indicator of the parent's status can be measured easily (hence their name). When no evidence is entered into an indicator it has no effect on its parent. When evidence *is* entered into an indicator node it causes a change in the likely distribution of states in the parent.

Indicator nodes can also be used where a large number of causal factors all have a direct impact on a single child node. The number of entries in the CPD of the child grows exponentially with the number of parents. If there are more than a couple of such causal factors then the CPD of the child can become unmanageable. Thanks to Bayes Theorem (Equation 1) we can reverse the direction of the arrows and turn the causal factors into indicators. Since each indicator node only has a single parent, their CPDs become much easier to define.

Ranked nodes are nodes whose states are measured on an ordinal scale, often with either three or five states ranging from "Very Low" through to "Very High". They are used to elicit subjective opinions from experts so that they can be entered as observations into the model. In the tool used to build most of the models described here (Agena Ltd. 2008), the underlying ordinal scale is represented by an equi-partitioned continuous scale in the range [0, 1]. It is thus possible to easily combine ranked nodes and variables consisting of numeric states in the same distributions.

THE MODIST MODEL

Fenton and Neil's pioneering paper (Fenton & Neil 1999) inspired a number of research groups to apply BNs to software process modelling. Wooff, Goldstein, and Coolen (2002) have developed BNs modeling the software test process while Stamelos et al (2003) used COCOMO81 cost factors to build a BN model of software project productivity. Bibi and Stamelos (2004) have shown how BNs can be constructed to model IBM's Rational Unified Process. Fenton and Neil's own research group have also gone on to develop a series of BN models, culminating in the AID tool (Neil, Krause, & Fenton 2003), the MODIST models (Fenton et. al. 2004), and the trials of revised MODIST models at Philips (Neil & Fenton 2005; Fenton et. al 2007a; Fenton et. al 2007b). A similar model has been developed by Siemens (Wang et. al. 2006). Here we will discuss the MODIST models and its successor, the "Philips" model.

A greatly simplified version of the MODIST model, with many of the causal factors and indicator nodes removed, is shown in Figure 2. The model is most easily understood as a series of subnets, fragments of a whole BN, which capture specific aspects of the software development process. The subnets are:

- **Distributed communications and management**. Contains variables that capture the nature and scale of the distributed aspects of the project and the extent to which these are well managed.

- **Requirements and specification**. Contains variables relating to the extent to which the project is likely to produce accurate and clear requirements and specifications.
- **Process quality**. Contains variables relating to the quality of the development processes used in the project.
- **People quality**. Contains variables relating to the quality of people working on the project.
- **Functionality delivered**. Contains all relevant variables relating to the amount of new functionality delivered on the project, including the effort assigned to the project.
- **Quality delivered**. Contains all relevant variables relating to both the final quality of the system delivered and the extent to which it provides user satisfaction (note the clear distinction between the two).

The full BN model is too large to be fully described here but in the next section we discuss just one of the subnets, the "People quality" subnet and use it as an example to show subjective judgements can be entered into the model.

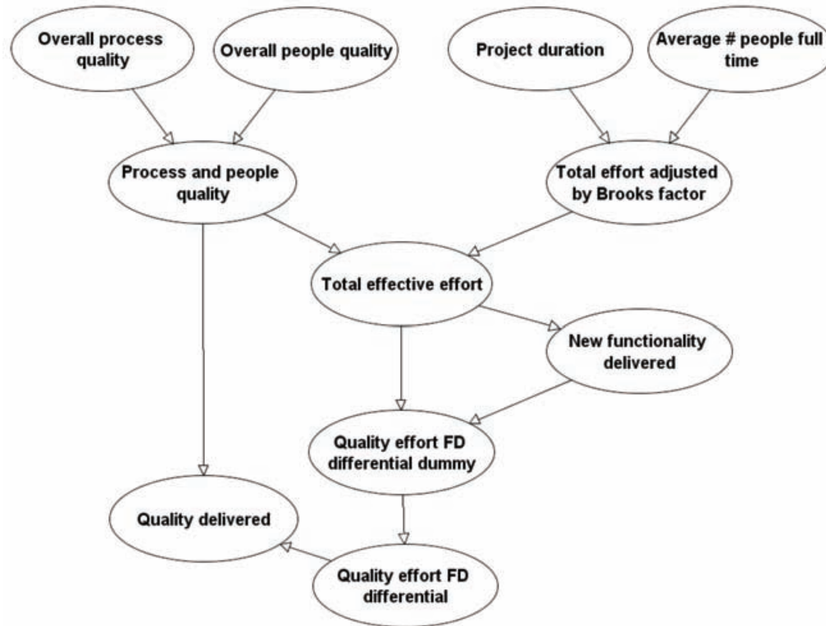
People Quality Subnet

Figure 3 shows the variables and causal connections in this subnet. A description of the main variables, including the model's rationale is given in Table 1. All of the variables shown in Figure 3 are ranked nodes, measured using a five point scale ranging from very low to very high. Observations are not normally entered directly on the variables described in Table 1. Instead we enter observation at primary causes (variables with no parents in the BN) and indicators.

In the 'people quality' subnet (Figure 3), indicator nodes are used to infer the staff quality. The default variables in our model for this are: staff turnover, staff experience, staff motivation, staff training and programming language experience. This can be varied to suit local process needs and is replicated in a similar way in the other subnets.

Although this approach is very flexible and does not insist on an organisation using specific metrics it does depend on some regularity in the way that development projects are estimated. This ensures that there are usable indicators for key attributes of quality and size, which our model can be adapted to use. It also makes it likely that

Figure 2. A simplified version of the MODIST model



the organisation's project managers will have accumulated the experience to be able to make stable judgments about the strength of these indicators. Some organisations may also accumulate data from past projects; there is no difficulty in principle in using such data in adapting the model and we hope to provide this capability in future versions of the toolset.

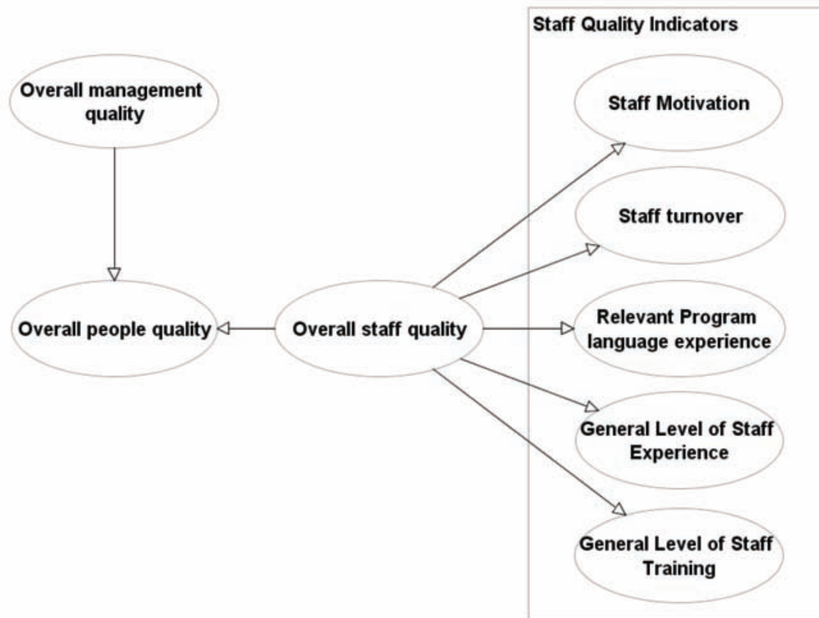
Using the Bayesian Net for Decision Support

What makes the Bayesian resource model so powerful, when compared with traditional software cost models, is that we can enter observations anywhere to perform not just predictions but also many types of trade-off analysis and risk assessment. So we can enter requirements for quality and functionality and let the model show us the distributions for effort and time. Alternatively we can specify the effort and time we have available and let the model predict the distributions for quality and functionality delivered (measured in function points).

As an example of this, if we simply set "New functionality delivered" to 1000 function points, the model produces the predictions shown in Figure 4. This tells us that the most likely duration is 6 to 8 months (the modal value), although the mean and median values are considerably larger at 21 months and 17 months respectively. The standard deviation is very large, at 16 months, because we have not entered any of the other project attributes. The model similarly predicts that the number of people required will have a modal value of one or two, but large mean and median values of 21 and 16 respectively.

If we fix the number of people at 10 and set the "Process and people quality" to "Very High", we get a modal value for the duration of 2 to 4 months, and mean and median values of 10 and 5 months respectively with a standard deviation of 13 months, making the model far less uncertain in its predictions. These figures are consistent with typical parametric models. However, this ability to vary the project constraints in any desired way

Figure 3. The "People quality" subnet of the MODIST model



while producing consistent estimates of the risk, provides far greater insight into the interplay between variables and allows far greater flexibility than is possible with parametric models.

THE DEFECT MODEL

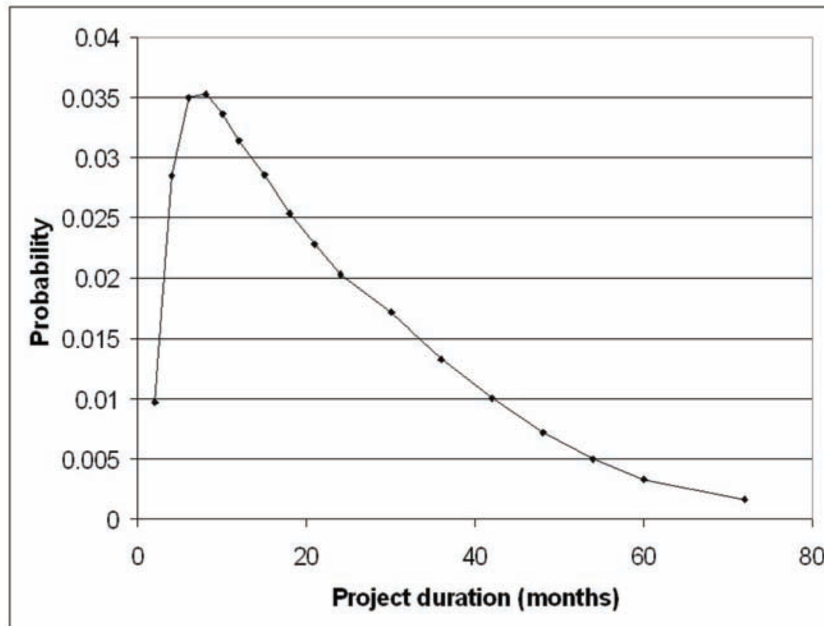
As with the MODIST model, the defect model (Neil & Fenton 2005) is too large to be shown here in full. The core of the model is shown in Figure 5.

Separate subnets governing: the scale of new functionality, specification and documentation quality, design and development quality, testing and rework quality, and features of the existing code base, all feed into this core defect model. Unlike the MODIST model, the defects model does not require all of its subnets to be included. The defects model is designed to model multiple phases of software development where not all aspects of the full software development process are present in each development phase. So there could be phases which involve requirements and specification, but no development or test. These give rise to specification errors only. Alternatively there can be phases which have development stages but no new requirements, giving rise to new

Table 1. Details of subnet for people quality

Variable Name	Description
Overall management quality	This is a synthetic node that combines 'communications management adequacy', 'subcontract management adequacy' and 'interaction management adequacy'. If any of these three is poor then generally the value of 'overall management quality' will be poor.
Overall staff quality	This is the quality of non-management staff working on the project.
Overall people quality	This is a synthetic node that combines 'overall management quality' and 'overall staff quality'.

Figure 4. Probability distribution of "Project duration" for 1000 FPs



code defects. The model can link these different phases together so that the defects which arise, but aren't fixed, in one phase, to be carried over to the next. In this way, the model can handle a very large and diverse range of possible software development processes.

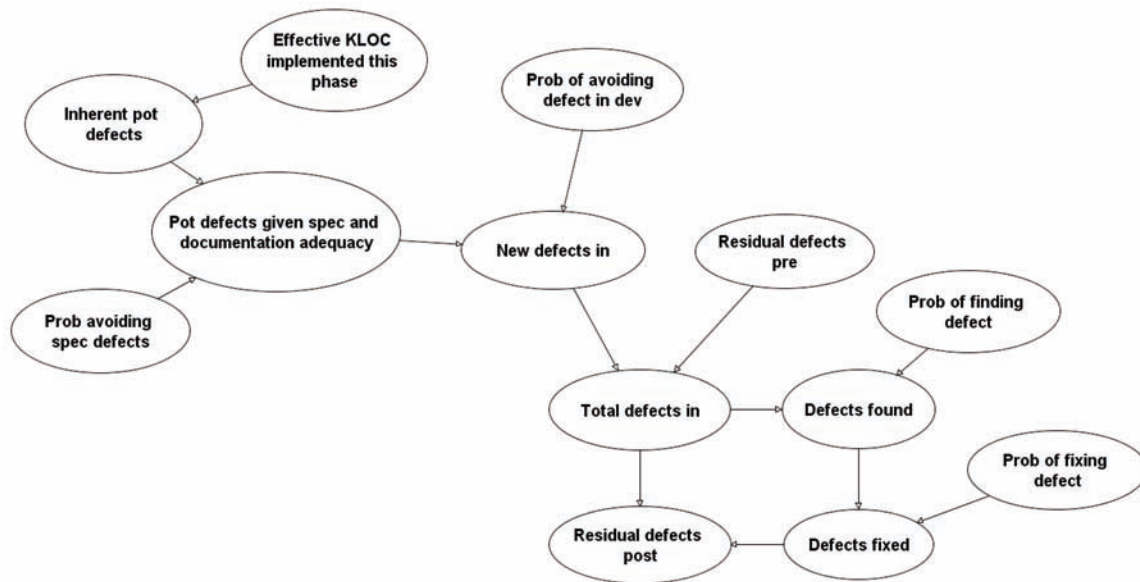
The model operates as follows. A piece of software of a given size, in this case measured inn thousands of lines of code (KLOC) gives rise to a certain number of defects. Some of these defects will be the result of inaccuracies in the specification and requirements. A mature software process, combined with close customer contact and experienced management can help to mitigate this. This gives rise to the "Probability of avoiding specification defects" node. Similarly, a good development team can reduce the number of defects introduced during development, leading to the "Probability of avoiding defects in development" node. The effort and experience of the test team affects the probability of finding the remaining defects, and the quality of the rework phase affects the ability to fix them.

The model shown in Figure 5 actually comes from a model validated at Philips (Neil & Fenton 2005; Fenton et. al 2007a; Fenton et. al 2007b). This differed from the original defects model in several important respects.

Project size was measured in KLOC. The original defects model used function points (FPs) as its size measure. However, FPs were not widely used at Philips and it seemed unlikely that they could be introduced purely to validate this model. Initialling the model was simply modified to deduce the number of FPs from the KLOC. This led to problems because it introduced a level of uncertainty in the FP measure which was not originally present. FPs also include a measure of problem complexity which is absent from the KLOC measure. Eventually it became clear that the "scale of new functionality" subnet had to be re-written to accommodate the switch from FPs to KLOC.

The second big change from the defects model was that many of the original indicator nodes were converted to causal factors. This was because the

Figure 5. The core of the defects model



development managers at Philips found it easier to understand the model when expressed in these terms. Managers frequently added evidence to both an indicator node *and* its parent, thus blocking the indicator evidence. However, turning indicator nodes into causal factors led to the classic problems that arise when nodes have multiple parents: how to construct CPDs involving large numbers of nodes, and how to provide tools so that such CPDs can be easily elicited from domain experts. These problems were solved by devising a comprehensive set of weighted mean functions (Fenton, Neil & Caballero 2007).

The model has been extensively trialed at Philips, with considerable success. The R^2 correlation between predicted and actual defect values is in excess of 93%. It is of course possible to construct regression models with similar, or even higher correlations, but only by including most of the data as training data, which implies an extensive metrics collection programme. The defects model's success was achieved using only empirical data from the literature, and observational data from the projects on which the model was being tested. Much of the raw data for these trials has been published and is available for use with the model (Fenton et. al 2007b).

THE PRODUCTIVITY MODEL

While the MODIST and defects models were successful in practical applications they contain some limitations. Overcoming or reducing these limitations became the motivation for the Productivity Model. This model provides unique features which were not available in previous models:

1. This model permits custom prior productivity rates and custom prior defect rates. Companies typically gather some data from their past projects. Productivity and defect rates are among the easiest to extract from project databases. Even if a company does not collect effort data, which may happen, it is often quite easy to estimate post hoc. The ability to enter such prior rates arose from criticism of past models, which work well

but only within their targeted domain. This Productivity Model can accommodate local user data by explicitly capturing prior rates provided by users. In cases where providing such custom productivity and defect rates is not possible, these rates can be estimated by our PDR model discussed later.

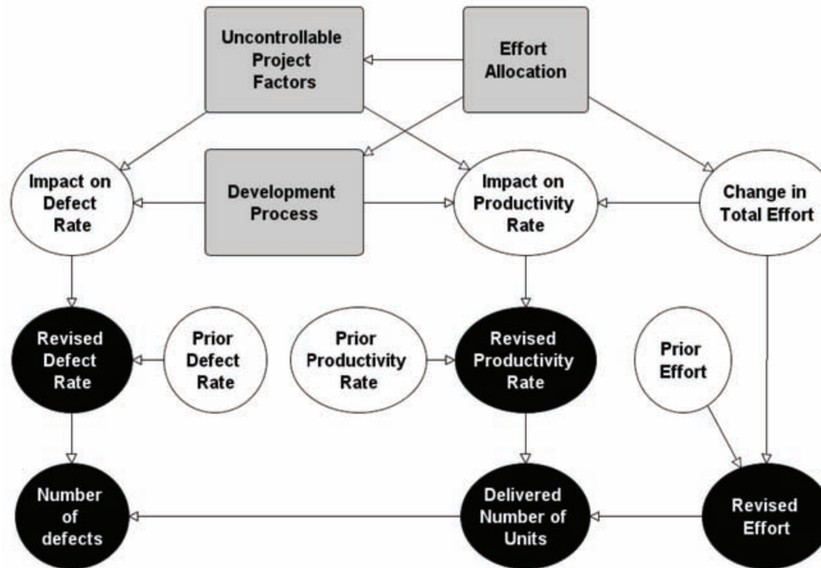
2. This model enables trade-off analysis between key project variables on a numeric scale. All the key project variables, namely: functionality, software quality and effort are expressed as numbers, not on a ranked scale, as effort and quality were in some of the past models.
3. This model enables customised units of measurement. Previous models captured key variables expressed in fixed units of measurement. For example, effort was captured only in person-months. This model can express key numeric variables in various units. Users can use one of a set of predefined units provided by the model, and can even introduce their own units.
4. The impact of qualitative factors can be easily changed by changing weights in node expressions. We provide a questionnaire which can help determine users' opinions on the relationships between various model variables. These opinions can be different for different software companies and may depend on the type of software developed, the processes used and various other factors. Proper model calibration ensures that the model reflects reality more accurately.
5. This model allows target values for numeric variables to be defined as intervals, not just as point values. For example, this model can answer the question: how can we achieve a defect rate between 0.05 and 0.08 defects/FP for a project of a specific size and with other possible constraints.
6. Numeric variables in this model are dynamically discretised. This means that setting intervals for numeric variables occurs automatically during model calculation. This not only frees us from spending time setting these intervals for each numeric variable, but as a main benefit, ensures that the discretisation is more accurate because it takes into account all observations entered into the model.

An early version of this model has been published in (Radliński et al. 2007). The structure of the most recent version of the model is illustrated on Figure 6. Ellipses represent key model variables while rounded rectangles are subnets containing detailed variables.

There are three steps which need to be followed before the Productivity Model can be used in performing analyses:

Step 1: Calibrate the model to the individual company. This step involves adding new detailed factors or removing those which are not relevant. This is achieved by setting new weights in weighted means: aggregated project, process and people factors. Regardless of whether the list of detailed factors has been changed or not, another task here is to define the impact of the aggregated factors on the adjusted defect and productivity rates. This is done by entering proper values for constants (multipliers) in expressions for the adjusted rates. The user has to answer questions like: if project factors change to the most/least desired state how would defect and productivity rates increase/decrease? We calibrated the model using the results of questionnaires obtained from employees working in software companies (mostly managers at different levels) and researchers with some commercial experience. We suggest recalibrating the model again with a similar questionnaire to ensure that the model reflects the terminology and development process used in a particular company.

Figure 6. The core of the Productivity Model



Step 2: Nominate a typical past project from the past project database. The user nominates a past project developed by this company which is closest to the project to be modelled. The effort, defect and productivity rates achieved in these projects should be entered as observations to the model (*prior defect rate*, *prior productivity rate* and *prior effort*). If there are many relevant past projects which could be picked, their distributions should be entered to the model.

Step 3: Estimate the difference in the current project. This step involves assessing how the current project is different from the project(s) nominated in step 2. The differences in detailed variables should then be entered into the model in subnets: *uncontrollable project factors*, *development process* and *effort allocation*.

Using Productivity Model in Decision Support

Let us suppose that a software company has to deliver software consisting of 500 function points but constrained to 2500 person-hours of effort. Suppose that in similar projects in the past the defect rate was typically 0.15 defects per function point, productivity rate was 0.2 function points per person-hour and the effort was 2000 person-hours. For a fair comparison in these examples we assume that all other factors included in the model are the same in all scenarios. With this information passed to the model, it predicts the number of defects delivered will be around 79 defects (all predicted values discussed here are median values of the probability distribution unless stated otherwise). Managers decide that this number of defects is unacceptably high and they wish to know how they can improve it to, say, just 40 defects. The model predicts that to achieve this quality target a better development process and more productive people are required. Also, allocating more effort on specification and testing at the cost of effort on coding is required. The lower effort on coding

is balanced by the better coding process and the ability of more productive people to deliver the assumed product size.

Now let us further assume that the company is not able to improve the process and people for this project. Thus we need to perform trade-off analysis between key project variables. We might remove a constraint for the product size. This would result in predicting lower product size containing lower total number of defects. But sacrificing product size would rarely be a good solution. We rather analyze how much more effort is required to achieve the target for the number of defects. The model now predicts that this company should spend around 7215 person-hours on this project to achieve the lower number of residual defects after release. The majority of the increased effort should be allocated to specification and testing activities.

As a second example of using the Productivity Model in decision support, let us assume that we analyze the impact of uncontrollable project factors on estimates provided by the model. Suppose that the size of the software to be developed is 1000 FPs. We leave the prior effort, productivity and defect rates at their default states. The model initially predicts *revised productivity rate* of 0.177 FP/person-hour and *revised defect rate* of 0.056 defects/FP. Now we further assume that *project complexity* is as high as possible – which means as high as has been observed by the company in its previous most complex projects. In this case the model predicts that we are likely to achieve a lower *revised productivity rate* (0.161 FP/person-hour) and that the developed software will be of lower quality (higher *revised defect rate* with median=0.062 defects/FP) compared to the scenario which assumes no change in *project complexity*.

In addition to the higher *project complexity*, suppose we also assume that there is the highest possible *deadline pressure*. In this case, increased *deadline pressure* causes the developers to work faster and thus become more productive (0.176 FP/person-hour). However, it also means they are less focused on delivering software of similar quality and thus their revised defect rate is expected to further increase (0.073 defects/FP).

Let us now assume that in addition to the previously entered known project factors they anticipate receiving *input documentation* of higher quality. We assume that the exact improvement in *input documentation* quality is unknown. It is only certain that it is of higher quality. We enter such information as *soft evidence*. Entering observations in this way means that we believe that the first four states from 'extra lower' to 'the same' are impossible while the last three from 'higher' to 'extra higher' are equally probable according to our knowledge. The model predicts that with increased quality of input documentation we should expect to be more productive (0.180 FP/person-month) and deliver better software (0.067 defects/FP).

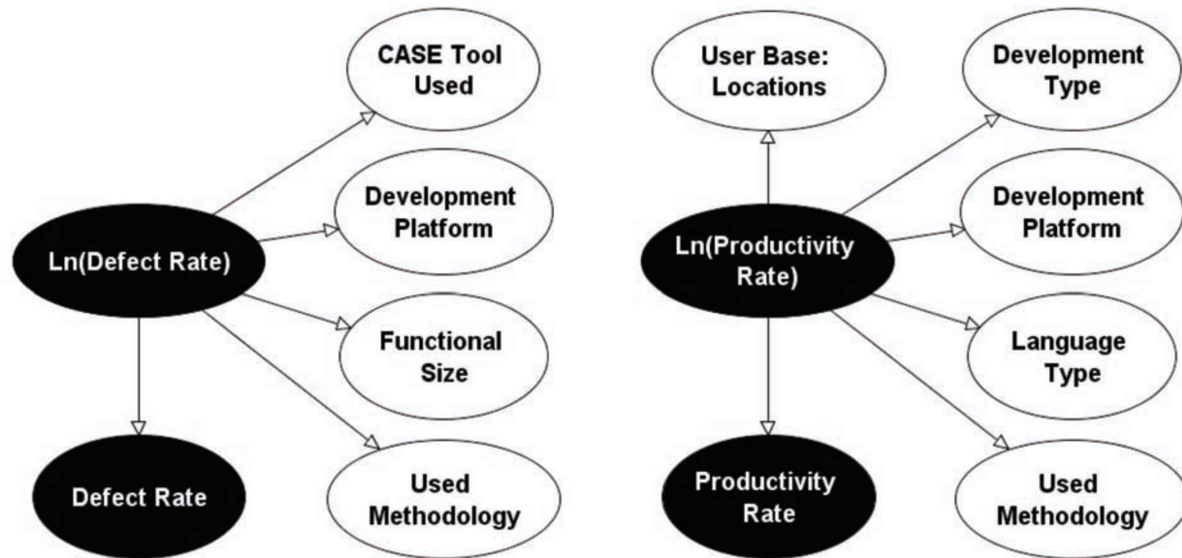
Modelling Prior Productivity and Defect Rates

The PDR model aims to estimate prior productivity and defect rates which are then passed as inputs to the Productivity Model. This model has a Naïve Bayesian Classifier structure (Figure 7). Observations are entered into qualitative factors describing the software to be developed (white ellipses). The model then estimates the log of productivity and defect rates. Finally, dependent variables (productivity and defect rates) are calculated on their original scale.

The links between these qualitative factors and dependant variables do not reflect any causal relationship. However, choosing this structure enables easier model creation since only pairwise relationships need to be analyzed (between each dependant variable and one predictor). As a result the model does not capture any relationships between predictors which need to be independent of each other in this type of model.

We identified the qualitative factors and

Figure 7. PDR Model



their impact on productivity and defect rates by analysing the ISBSG dataset R9 release (2005). While fitting various distributions for these dependent variables we observed that LogNormal distribution fits the data best. We adjusted the impact of qualitative factors on productivity and defect rate by other reported data, most notably (Jones, 2002 cited after Sassenburg 2006) and according to our expert-judgement. Details on the steps of empirical analysis, the structure of the model and various scenarios on model usage can be found in (Radliński et al., 2008a).

Here we demonstrate one example of the kind of analysis which this model can perform. We also pass the results obtained from the PDR model for productivity and defect rates to the Productivity Model. We analyze two scenarios for the same functional size (1000 FPs), the same target defect rate (0.1 defects/FP) and wish to get a prediction for development effort from the Productivity Model. In the first scenario we assume that the software project will be developed on a mainframe and with a 3GL programming language. The second scenario assumes that a project will be developed on multiple platforms using an application generator. We enter observations for the appropriate qualitative factors into variables in the PDR model. For fair comparison we further assume that in both scenarios the process and people quality and other factors from the Productivity Model have 'average' values.

After running both models we observe that the observations entered into the PDR model significantly change the predictions provided by the Productivity Model. Predicted revised effort is about 3.5 times higher in scenario 1 (10740 person-hours) than in scenario 2 (3074 person-hours). However, it would be wrong to conclude from this that we should only use application generators and multiple platforms, while avoiding 3GL languages and mainframe platforms. The predictors in the PDR model should be treated as uncontrollable factors which describe the inherent nature of the software project. This nature determines the best platform, language type to be used and other factors.

AGILE DEVELOPMENT ENVIRONMENTS

The models discussed so far mostly apply to large, traditional, waterfall style development environments. There has been a significant trend in recent years, especially in web based and other interactive applications, towards so-called "agile" development techniques (Agile Manifesto 2008). Agile methods eschew extensive specification and design phases in favour of rapid prototyping, iterative development, extensive customer feedback and a willingness to modify requirements in the face of changing business circumstances.

Agile development environments present two problems for traditional models. The first problem is that the lack of a formal requirements gathering phase makes it very difficult to quantify the size of the project. There is unlikely to be sufficient written documentation in an agile project to justify a rigorously determined function point count or any similar measure of problem size.

The second problem that traditional models face concerns data gathering and entry. The defects model described in an earlier section can require over 30 individual pieces of data in order to be fully specified. Although not all data is required in all phases of development, there will normally be a need to gather several separate numbers or subjective values. In the case of the subjective judgements, these must also be entered consistently across iterations. There is considerable redundancy here. If the node "Programmer capability" is set to "High" on the first iteration, then it is likely to remain high in subsequent iterations.

We can turn these problems into opportunities as follows. First, by restricting our models to whatever data is routinely collected by agile development processes. This means that we do not require agile projects to collect metrics, such as function points, which do not naturally arise in that environment. The looser definition of some of these metrics means that we somehow have to learn the exact interpretation of agile metrics on a given project. As we shall see, the iterative nature of agile projects makes this perfectly possible.

Second, instead of modelling all the causal factors which contribute to a project, we subsume their combined effect in our learning mechanism. We then model only *changes* in the development environment. This greatly reduces the amount of data which needs to be entered into models and completely eliminates the redundancy that would otherwise be present.

This combination of intelligent model adaptation, and learning using minimal data input, is only possible because of the empirically derived priors and the causal relationships elicited from domain experts. The BN models already represents a very wide range of possible development environments. Rather than "training" the model, data is needed to simply "nudge" the model towards a set of variable states that are consistent with the environment being modelled.

Here we present two models that take this approach. In the first we show how productivity, and consequently timescales, can be modelled in an Extreme Programming environment. In the second we present a learning defects model for use in iterative environments.

Extreme Programming Project Velocity Model

Extreme Programming (XP) is an agile development method that consists of a collection of values, principles and practices as outlined by Kent Beck, Ward Cunningham and Ron Jeffries (Beck 1999; Jeffries, Anderson & Hendrickson 2000). These include most notably: iterative development, pair programming, collective code ownership, frequent integration, onsite customer input, unit testing, and refactoring.

The basic unit of work in XP is the *User Story*. Developers assign the effort that they believe is required for them to design, code and test each user story. Once iteration i is complete, the estimates for the completed user stories are added together.

This is the *Project Velocity* V_i for iteration i. Assuming that the next iteration, $i + 1$, is the same length, the customer selects the highest priority uncompleted user stories whose estimated efforts sum to V_i. These user stories are then scheduled for iteration $i + 1$. The project velocity can therefore be thought of as the estimated productive effort per iteration.

The BN used to model project velocity is shown in Figure 8. To model the relationship between total effort E_i and the actual productive effort A_i, there is a single controlling factor that we call Process Effectiveness, e_i. This is a real number in the range [0,1]. A Process Effectiveness of one means that all available effort becomes part of the actual productive effort. The actual productive effort is converted into the estimated productive effort (or project velocity) V_i, via a bias b_i, which represents any consistent bias in the team's estimations.

The Process Effectiveness is, in turn, controlled by two further parameters: Effectiveness Limit, l_i, and Process Improvement, r_i. The Process Improvement is the amount by which the Process Effectiveness increases from one XP iteration to the next. To allow for failing projects, the Process Improvement can take on negative values.

Only E_i and V_i are ever entered into the model.

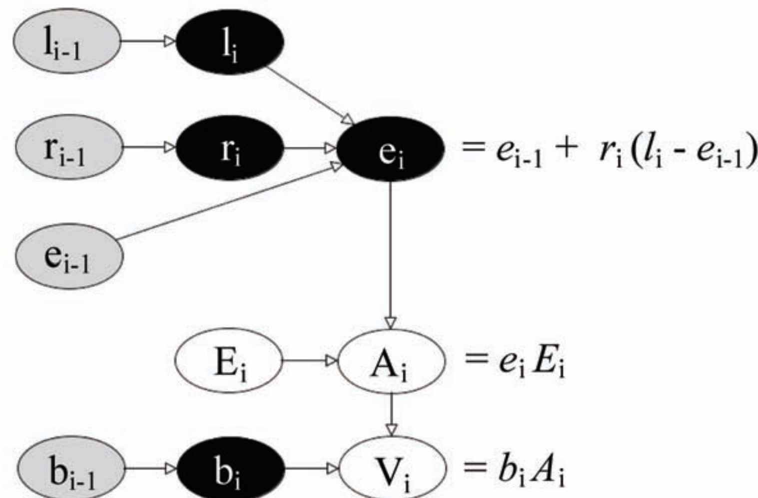
The model shown in Figure 8 is what is known as a 1.5 TBN (for 1.5 timeslice temporal BN). It shows the interface nodes from the previous timeslice and their directed arcs into the current timeslice. This is essentially the full model, it is not a cut down core such as the ones presented for the MODIST and defect models. Notice the tiny size of this mode compared to the others. This is due to the fact that it is only trying to predict one thing, V_i, and that is does so, not by taking into account all possible causal factors, but by learning their cumulative effect on the process control parameters: l_i and r_i.

The model works as follows. Initial estimates for the amount of available effort E_i are entered into the model for each iteration. At this stage the number of iterations is unknown, so a suitably large number must be chosen to be sure of covering the entire project. Using empirically derived industry priors for l_0, r_0, e_0 and b_0 the model then makes generic predictions for the behaviour of V_i. This enables the project management to see how many user stories are likely to be delivered in each iteration. The model correctly reproduces rapidly rising values for V_i over the initial iterations - a phenomenon that has been observed in multiple studies (Ahmed, Fraz, & Zahid 2003; Abrahamsson & Koskela, 2004; Williams, Shukla, & Anton, 2004).

At each project iteration measured values for V_i become available and are entered into the model. This causes the learned parameters l_i, r_i and b_i to update, modifying future predictions. Using data from a real XP project (Williams, Shukla, & Anton, 2004), we were able to enter observations for V_1 and V_2, and verify the model's predictions of the remaining V_i. Initially this generates improved predictions for early iterations, but significantly worse predictions for later iterations. It turns out that there was a significant change in the environment half way through the project, when much better customer contact and feedback became available. This was added to the model by adding an "Onsite customer" indicator node to l_i. The effect of this indicator node had to be calibrated independently using a training model which was a slightly modified version of the project velocity model. Once the effect of the changed environment had been taken into account (Figure 9), the model was able to produce very good predictions for all future values of V_i (Hearty et. al. in press).

The model is not limited to productivity predictions. By adding a node which sums the V_i distributions, we can create predictions of delivered functionality, s_i, after each iteration.. Taking the median values of the s_i distributions and comparing them to the actual functionality delivered, we can determine the magnitude of relative errors (MRE) for each s_i. The mean val-

Figure 8. Project Velocity model



ues of the MREs give a good overall measure of the accuracy of the model. The mean MRE for s_i before learning was 0.51, an error of over 50%. After learning the mean MRE for s_i reduces to 0.026 - an extraordinary level of accuracy for a software process model.

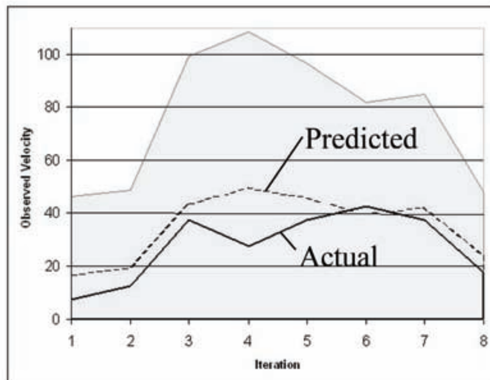
One of the great advantages of a BN model is its ability to deliver natural assessments of risk. If we take the cumulative probability distribution of an s_i node, then this allows us to read off the probability that any given amount of functionality will be delivered. An example for s_8 is shown in Figure 10. In this case we are trying to read off the probability of delivering up to 200 Ideal Engineering Days (IEDs) of completed functionality. (An IED is a day spent by a developer doing detailed design, writing code, or testing code and is assumed to be directly proportional to a fixed amount of delivered functionality.) Before learning, the model predicted that there was a 25% chance of delivering 200 IEDs or less. i.e. There was a 75% of delivering more than 200 IEDs. After learning the model reduced this to a 35% chance of delivering 200 IEDs or more - the model was initially too optimistic.

XP is the most common agile development method. Another common methodology is Scrum (Takeuchi & Nonaka 1986; Schwaber & Beedle 2002; Sutherland 2004). This approach uses *burndown charts* to plan a project. A burndown chart starts with a fixed amount of functionality that must be delivered. This reduces with each iteration as more and more functionality is completed. The slope of the burndown chart gives the project velocity, while its intercept with the horizontal time axis gives the projected completion date. We can model a burndown chart very easily using a modified version of the XP model. Instead of letting s_i represent the cumulative functionality delivered, we define it to be the amount of functionality remaining. We can then set s_i as evidence values, rather than V_i as in XP, and learn how the burndown chart must be altered as each iteration completes. An example is shown in Figure 11.

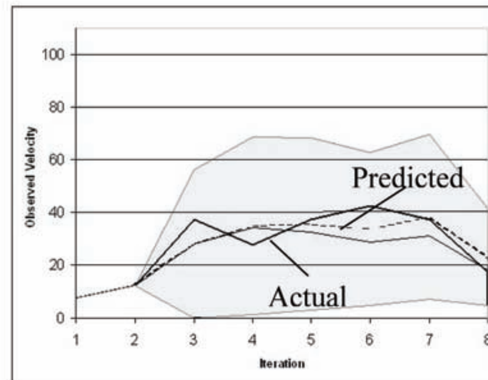
LEARNING MODEL FOR ITERATIVE TESTING AND FIXING DEFECTS

The models discussed earlier contain fixed relationships between variables. The model discussed here (see Figure 12) is a learning model in the sense

Figure 9. Project velocity predictions before and after learning



Predicted and actual median velocity values before any actual values have been entered into the model. The solid grey area shows the predicted values +/- 2 sd.



After v1 and v2 have been entered, and improved customer access has been added, the predictions improve significantly. The uncertainty surrounding the model's predictions also decrease. The solid grey line shows the predictions after v1 and v2, but before customer access has been added.

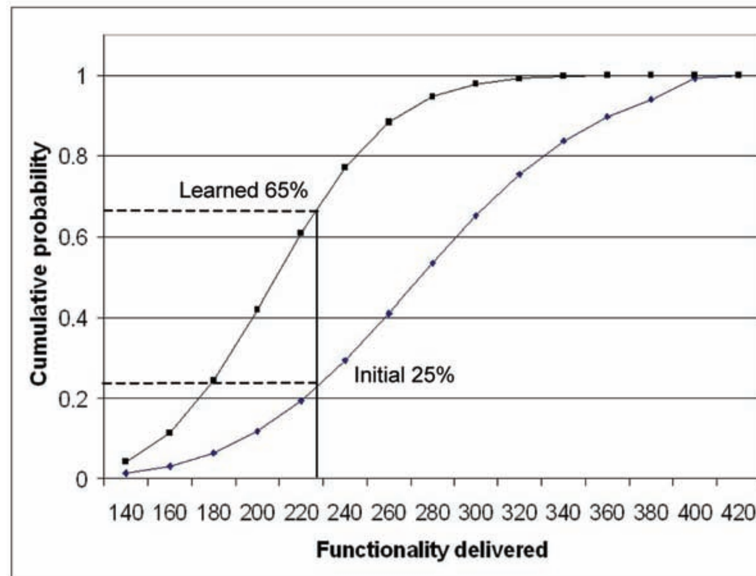
that it learns the impact of particular predictors on dependent variables using a set of past data. The aim for this model is to predict the number of defects found and fixed in an iterative testing and fixing process. This iterative process assumes that all functionality has been developed prior to testing. The testing and fixing process is divided into series of iterations lasting a variable amount of time. More information on earlier versions of this model and its assumptions can be found in (Radliński et al. 2008b).

Links in this model reflect causal relationships. Testing process factors (effort, process and people quality and other factors) impact on *testing effectiveness* which, after utilizing the number of *residual defects* at a given point of time, determine the number of *defects found* during a specific testing iteration. The number of *open defects* from the past iterations plus the number of *defects found* in the current iteration form an upper limit for the number of *defects fixed* in the current iteration. The second limit comes from the values of fixing process factors in a form of *potentially fixed defects* reflecting how many defects can be fixed given specific process data.

In contrast to traditional software reliability growth models this model does not assume that fixes need to be perfect. This means that it explicitly incorporates the number of *defects inserted* as a result of an imperfect fixing process. The model assumes that the number of *defects inserted* depends on the number of *defects fixed* during a specific iteration and on the fixing process quality. When many defects have been fixed than it is probable that a proportionate number of defects have also been introduced. When the fixing process is poor then the number of *defects inserted* will be relatively high.

The model works in the following way. A user enters data about the past testing and fixing iterations as well as the defects found and fixed in these iterations. The model uses the entered data to estimate its parameters – *multipliers* for each process factor and the number of *residual defects*. The values of these *multipliers* reflect the importance of each process factor in terms of its impact on the number of defects or defects fixed. Then the user enters planned values for process factors in future iterations – e.g. how much effort is planned to be allocated to testing and fixing defects. The model uses these process factors, learnt *multipliers* and learnt *residual defects* to predict

Figure 10. Cumulative probability distribution of functionality delivered in iteration 8



the number of *defects found* and the number of *defects fixed* in future iterations. At this point the user can analyze various scenarios which differ in effort, process and people quality, or other known factors possible in future iterations.

We perform model validation using a semi-randomly generated dataset. We set point values for prior *residual defects* and values of process factor multipliers. Values for process factors were generated randomly and then manually adjusted in some iterations to more realistically reflect the testing and fixing process. The effectiveness

Figure 11. Scrum burndown chart, before and after learning

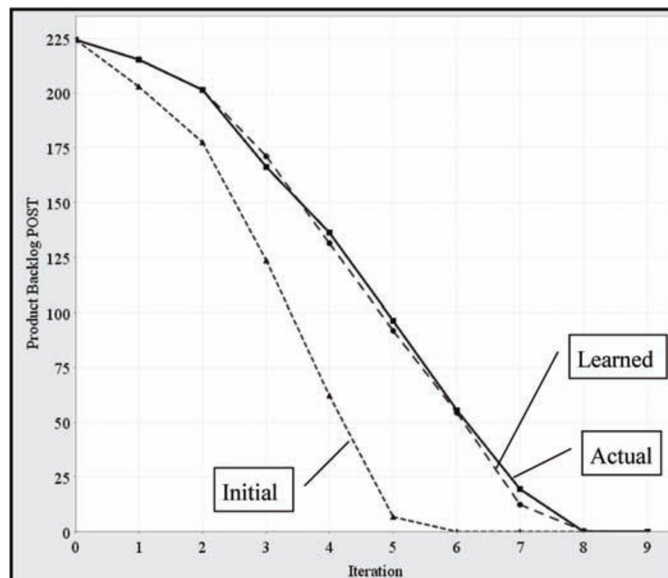
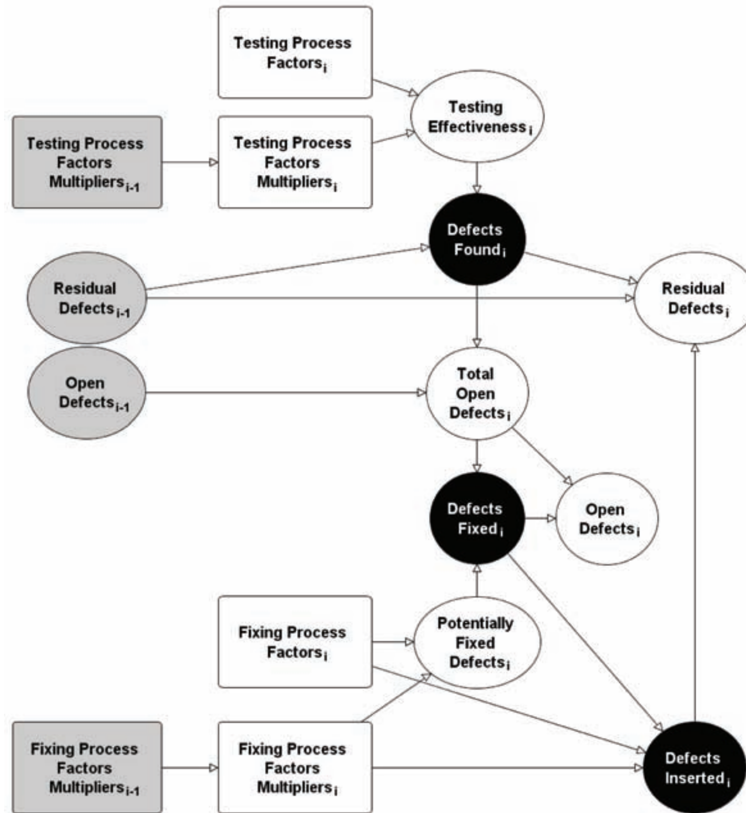


Figure 12. The structure of the LMITFD



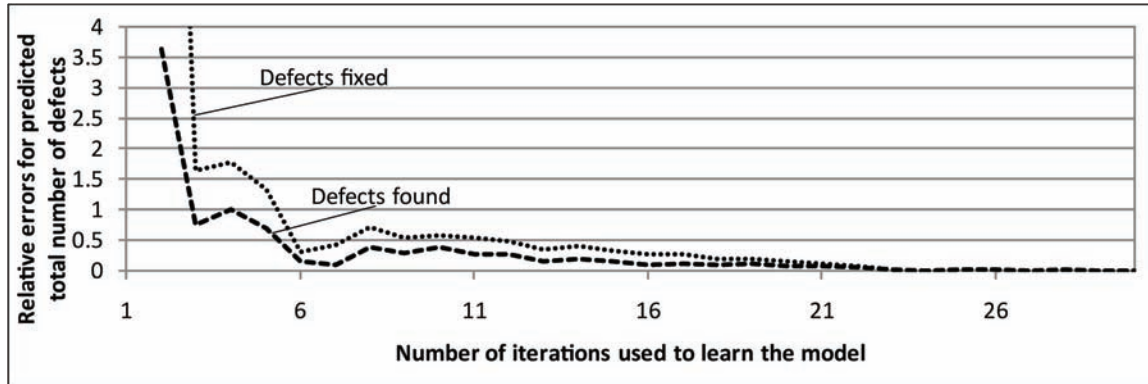
of the testing and fixing process increases over time as the process becomes more mature and the software under test becomes better understood. We estimated values for defects found, fixed and inserted using the values of process factors and the relationships as incorporated by the model. We used these generated datasets in the model validation stage and treated them as if they were observed values. In the validation stage we tested how fast the model can learn the number of *residual defects* and the values of process factors' *multipliers*.

We tested the model using 30 testing and fixing iterations. First we used a single learning iteration with 29 iterations where values of defects found and fixed were predicted. This was followed by 2 iterations for learning and 28 for prediction, and so on. Figure 13 illustrates the values of relative errors in predicted total number of *defects found* and *defects fixed* as estimated after a different number of learning iterations. This relative error is defined as illustrated on Equation 2.

$$relative\,error = \frac{\left|total\,predicted - total\,actual\right|}{total\,actual} \qquad (2)$$

These results confirm that after only 5 learning iterations, the model predicts the total number of defects found to within a 0.16 relative error of the actual value and predicts total number of defects fixed to within a 0.30 relative error of the actual value. Predictions then become less accurate because of higher fluctuations in actual number of *defects found* in the dataset. But from 8 learning

Figure 13. Relative errors in predictions for total number of defects found and defects fixed depending on number of iterations used to learn the model



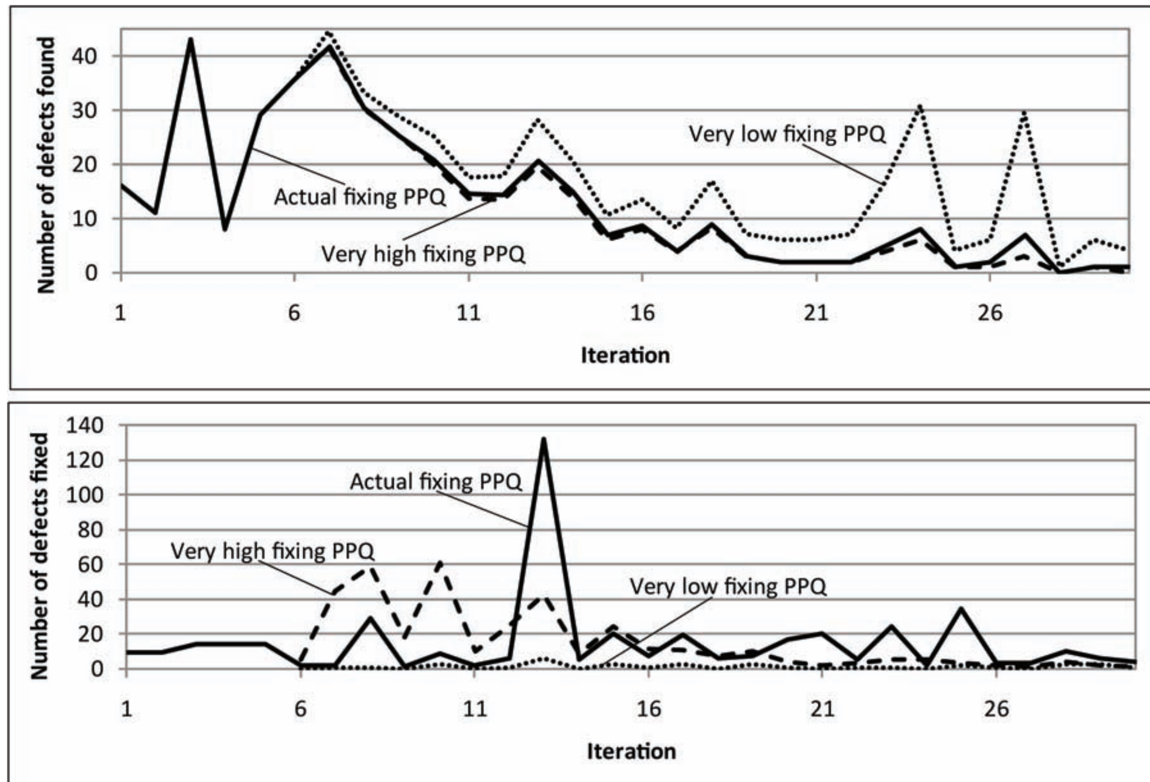
iterations the accuracy again increases (relative error decreases). These results show that the model is capable of learning its parameters after very few iterations and then generates predictions with reasonable accuracy.

One of the most important advantages of such models is the high potential for performing various types of what-if analyses. After the model learns its parameters, we can set as observations different values for future process factors to see how they influence predicted number of defects found. For example let us analyze model predictions in two scenarios when *fixing process and people quality* is 'very low' or 'very high'. Values of other process factors are the actual values in iterations used for prediction. We use 5 learning iterations and the remaining 25 for prediction. Figure 14 illustrates the model's predictions for this case. We can observe that, as we could expect, with lower *fixing process and people quality*, fewer defects are likely to be fixed in future iterations. These differences are very high, confirming that the model updated its *multipliers* and *residual defects* to reflect the fact that qualitative process factors have a strong influence on *defects found* and *defects fixed*.

We can see that, although we have only modified our observations of *fixing process and people quality*, the predicted values of *defects found* are also different in these two scenarios. The reason is the following. Lower *fixing process and people quality* with *fixing effort* unchanged leads to increased number of *defects inserted* as a result of imperfect fixing. Higher number of *defects inserted* causes more *residual defects* in subsequent iterations. Higher number of *residual defects* in turn causes there to be more *defects found* – the more *residual defects* the easier it is to find them.

Another issue which may be surprising in the beginning is the fact that predicted number of *defects fixed* in the late iterations is lower both with 'very low' and 'very high' *fixing process and people quality* compared with the scenario with the original *fixing process and people quality*. But there is also an explanation for such predictions. With 'very low' *fixing process and people quality* it is simply not possible to fix more defects without assigning significantly more *fixing effort*. On the other hand, with 'very high' *fixing process and people quality* defects which were found earlier were also fixed earlier. So in these late iterations there are fewer defects still to be fixed.

Figure 14. Predictions from LMITFD with very high and very low fixing process and people quality after 5 iterations used to learn the model



REFERENCES

Abdel-Hamid, T. (1989). The dynamics of software projects staffing: A system dynamics based simulation approach. *IEEE Transactions on Software Engineering*, *15*(2), 109–119. doi:10.1109/32.21738

Abrahamsson, P., & Koskela, J. (2004). Extreme programming: A survey of empirical data from a controlled case study. In *Proceedings 2004 International Symposium on Empirical Software Engineering, 2004*. (pp. 73-82). Washington, DC: IEEE Computer Society.

Agena Ltd. (2008). *Bayesian Network and simulation software for risk analysis and decision support*. Retrieved July 9, 2008, from http://www.agena.co.uk/

Agile Manifesto. (2008). *Manifesto for agile software development*. Retrieved July 18, 2008, from http://www.agilemanifesto.org/

Ahmed, A., Fraz, M. M., & Zahid, F. A. (2003). Some results of experimentation with extreme programming paradigm. In *7th International Multi Topic Conference, INMIC 2003,* (pp. 387-390).

Beck, K. (1999). *Extreme programming explained: Embrace change*. Reading, MA: Addison-Wesley Professional.

Bibi, S., & Stamelos, I. (2004). Software process modeling with Bayesian belief networks. In *10th International Software Metrics Symposium Chicago*.

Boehm, B. (1981). *Software engineering economics*. Englewood Cliffs, NJ: Prentice-Hall.

Briand, L. C., El Emam, K., Surmann, D., Wieczorek, I., & Maxwell, K. D. (1999). An assessment and comparison of common software cost estimation modeling techniques. In *21st International Conference on Software Engineering, ICSE* 1999, (pp. 313-322).

Chulani, S., & Boehm, B. (1999). *Modeling software defect introduction and removal: COQUALMO (COnstructive QUALity MOdel)*, (Tech. Rep. USC-CSE-99-510). University of Southern California, Center for Software Engineering, Los Angeles, CA.

Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P., & Mishra, R. (2007a, January). Predicting software defects in varying development lifecycles using Bayesian nets. *Information and Software Technology*, *49*(1), 32–43. doi:10.1016/j.infsof.2006.09.001

Fenton, N., Neil, M., Marsh, W., Hearty, P., Radlinski, L., & Krause, P. (2007b). Project data incorporating qualitative facts for improved software defect prediction. In *Proceedings of the Third international Workshop on Predictor Models in Software Engineering*, International Conference on Software Engineering (May 20 - 26, 2007). Washington, DC: IEEE Computer Society.

Fenton, N. E., Marsh, W., Neil, M., Cates, P., Forey, S., & Tailor, T. (2004). Making resource decisions for software projects. In *Proceedings of 26th International Conference on Software Engineering* (ICSE 2004), Edinburgh, United Kingdom, May 2004, IEEE Computer Society, (pp. 397-406).

Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, *25*(5), 675–689. doi:10.1109/32.815326

Fenton, N. E., Neil, M., & Caballero, J. G. (2007). Using ranked nodes to model qualitative judgments in Bayesian Networks. *IEEE Transactions on Knowledge and Data Engineering*, *19*(10), 1420–1432. doi:10.1109/TKDE.2007.1073

Finnie, G. R., Wittig, G. E., & Desharnais, J. M. (1997). A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models. *Journal of Systems and Software*, *39*(3), 281–289. doi:10.1016/S0164-1212(97)00055-1

Glass, R. L. (2006). The Standish report: Does it really describe a software crisis? *Communications of the ACM*, *49*(8), 15–16. doi:10.1145/1145287.1145301

Hearty, P., Fenton, N., Marquez, D., & Neil, M. (in press). Predicting project velocity in XP using a learning dynamic Bayesian Network model. *IEEE Transactions on Software Engineering*.

ISBSG. (2005). *Estimating, Benchmarking & Research Suite Release 9*. Hawthorn, Australia: International Software Benchmarking Standards Group.

Jeffries, R., Anderson, A., & Hendrickson, C. (2000). *Extreme programming installed*. Reading, MA: Addison-Wesley Professional.

Jensen, F. (2001). *Bayesian Networks and decision graphs*, New York: Springer-Verlag.

Jones, C. (1986) *Programmer productivity*. New York: McGraw Hill.

Jones, C. (1999). Software sizing. *IEE Review*, *45*(4), 165–167. doi:10.1049/ir:19990406

Jones, C. (2002). *Software quality in 2002: A survey of the state of the art*. Software Productivity Research.

Jones, C. (2003). Variations in software development practices. *IEEE Software*, *20*(6), 22–27. doi:10.1109/MS.2003.1241362

Kemerer, C. F. (1987). An empirical validation of software cost estimation models. *Communications of the ACM*, *30*(5), 416–429. doi:10.1145/22899.22906

Lauritzen, S. L., & Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems (with discussion). *Journal of the Royal Statistical Society. Series B. Methodological*, *50*(2), 157–224.

Lepar, V., & Shenoy, P. P. (1998). A comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer architectures for computing marginals of probability distributions. In G. Cooper & S. Moral (Ed.), *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence,* (pp. 328-337). San Francisco: Morgan Kaufmann.

Molokken, K., & Jorgensen, M. (2003). A review of software surveys on software effort estimation. In *2003 International Symposium on Empirical Software Engineering* (pp. 223-230). Washington, DC: IEEE press.

Murphy, K. P. (2002). *Dynamic Bayesian Networks: Representation, inference and learning*. PhD thesis, UC Berkeley, Berkeley, CA.

Neil, M. (1992). *Statistical modelling of software metrics*. Ph.D. dissertation, South Bank University, London.

Neil, M., & Fenton, P. (2005). Improved software defect prediction. In *10th European Software Engineering Process Group Conference*, London.

Neil, M., Krause, P., & Fenton, N. E. (2003). Software quality prediction using Bayesian Networks. In T. M. Khoshgoftaar, (Ed.) *Software Engineering with Computational Intelligence.* Amsterdam: Kluwer.

Radliński, Ł., Fenton, N., & Marquez, D. (in press, 2008a). Estimating productivity and defect rates based on environmental factors. In *Information Systems Architecture and Technology*. Wrocław, Poland: Oficyna Wydawnicza Politechniki Wrocławskiej.

Radliński, Ł., Fenton, N., & Neil, M. (in press, 2008b). A Learning Bayesian Net for Predicting Number of Software Defects Found in a Sequence of Testing. *Polish Journal of Environmental Studies*.

Radliński, Ł., Fenton, N., Neil, M., & Marquez, D. (2007). Improved decision-making for software managers using Bayesian Networks. In *Proceedings of 11th IASTED International Conference Software Engineering and Applications (SEA)*, Cambridge, MA, (pp. 13-19).

Sassenburg, J. A. (2006). *Design of a methodology to support software release decisions (Do the numbers really matter?)*, PhD Thesis, University of Groningen.

Schwaber, K., & Beedle, M. (2002). *Agile software development with SCRUM*. Upper Saddle River, NJ: Prentice Hall.

Shepperd, M., & Schofield, C. (1997). Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, *23*(11), 736–743. doi:10.1109/32.637387

Srinivasan, K., & Fisher, D. (1995). Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering*, *21*(2). doi:10.1109/32.345828

Stamelos, I., Angelis, L., Dimou, P., & Sakellaris, E. (2003). On the use of Bayesian Belief Networks for the prediction of software productivity. *Information and Software Technology*, *45*(1), 51–60. doi:10.1016/S0950-5849(02)00163-5

Standish Group International. (1995). *The Chaos Report*. Retrieved July 9, 2008, from net.educause.edu/ir/library/pdf/NCP08083B.pdf

Sutherland, J. (2004). *Agile development: Lessons learned from the first scrum.* Retrieved July 9, 2008, from http://jeffsutherland.com/Scrum/FirstScrum2004.pdf.

Takeuchi, H., & Nonaka, I. (1986). The new new product development game. *Harvard Business Review, Jan-Feb*.

Wang, H., Peng, F., Zhang, C., & Pietschker, A. (2006). Software project level estimation model framework based on Bayesian Belief Networks. In *Sixth International Conference on Quality Software*.

Williams, L., Shukla, A., & Anton, A. I. (2004). An initial exploration of the relationship between pair programming and Brooks' law. In *Agile Development Conference, 2004*, (pp. 11-20), Agile Development Conference.

Wooff, D. A., Goldstein, M., & Coolen, F. P. A. (2002). Bayesian graphical models for software testing. *IEEE Transactions on Software Engineering*, *28*(5), 510–525. doi:10.1109/TSE.2002.1000453