Research

Modelling fault-proneness statistically over a sequence of releases: a case study

Magnus C. Ohlsson[1,*,†], Anneliese Amschler Andrews[2] and Claes Wohlin[3]

[1]*Department of Communication Systems, Lund University, Box 118, SE–221 00 Lund, Sweden*
[2]*Computer Science Department, Colorado State University, Fort Collins CO 80523–1873, U.S.A.*
[3]*Department of Software Engineering & Computer Science, Blekinge Institute of Technology, Ronneby, Sweden*

Summary

Many of today's software systems evolve through a series of releases that add new functionality and features, in addition to the results of corrective maintenance. As the systems evolve over time it is necessary to keep track of and manage their problematic components. Our focus is to track system evolution and to react before the systems become difficult to maintain. To do the tracking, we use a method based on a selection of statistical techniques. In the case study we report here that had historical data available primarily on corrective maintenance, we apply the method to four releases of a system consisting of 130 components. In each release, components are classified as fault-prone if the number of defect reports written against them are above a certain threshold. The outcome from the case study shows stabilising principal components over the releases, and classification trees with lower thresholds in their decision nodes. Also, the variables used in the classification trees' decision nodes are related to changes in the same files. The discriminant functions use more variables than the classification trees and are more difficult to interpret. Box plots highlight the findings from the other analyses. The results show that for a context of corrective maintenance, principal components analysis together with classification trees are good descriptors for tracking software evolution. Copyright © 2001 by John Wiley & Sons, Ltd.

J. Softw. Maint. Evol: Res. Pract. 2001; **13**:000–000

No. of Figures: 14. No. of tables: 16. No. of References: 23.

KEY WORDS: software evolution; fault-prone components; fault prediction; code decay; principal components analysis; classification trees; corrective maintenance

*Correspondence to: Magnus C. Ohlsson, Department of Communication Systems, Lund University, Box 118, SE–221 00 Lund, Sweden.

[†]E-mail: Magnus_C.Ohlsson@telecom.lth.se

Contract/grant sponsor: Partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), grant 1K1P–99–06121.

1. INTRODUCTION

The software evolution considered in this paper occurs as software or a specific system gets changed as it goes through a number of releases and naturally inherits functionality and characteristics from previous releases. Tracking such software evolution is an important aspect of maintaining complex software systems. As new functionality and features are added over time, complexity may increase [1] and impact the maintainability of the system and its software components. Defects may also be injected as a result of bug fixes or other changes. It is hence important to track the evolution of a system and its software components. Further, it is also important not to violate the initial architecture. If violation happens, problems will arise and effort has to be focussed on problematic parts or components. A better solution is to evolve the architecture in a disciplined way, but this is not always done. Better tracking of the evolution of software systems requires models and methods to identify systems which are on their way to becoming difficult to maintain.

Tracking the faults detected during the evolution of these systems may serve two major objectives. First, the information can be used to direct the efforts when a new release is built. This could mean applying a more thorough development or maintenance process or assigning the most experienced personnel to the troublesome components. Second, the information can be used to determine when systems need to be reengincered. Components that are difficult to maintain are certainly candidates for reengineering efforts. Therefore the identification of fault-prone components makes it possible to target improvement efforts to identifiable, small portions of a system and thus makes maintenance less expensive and more efficient.

The basic idea for the research we report is to build models based on historical data from successive releases [2]. The data may include fault (defects in documents) and failure (problems during execution) data, and product and process measures when available. Also, fault surrogates such as relative complexity has been used to trace software evolution [3]. The intention of using historical data is to find trends and react early to warnings. Trends provide an opportunity to predict and plan focused maintenance activities.

2. BACKGROUND

Various approaches have been presented to classify components and to predict whether they will be fault-prone. Most methods are based on statistics, for example, principal components analysis [4] (hereafter ‘PCA’), Boolean discriminant analysis [5], Spearman rank correlation [6], optimised set reduction [7] and regression analysis [8]. Other work uses classification trees [9][10] and neural network techniques [11] to assess which descriptors are associated with fault-prone components.

The objective of our research is to identify suitable measures and to build models that emphasise software evolution over successive releases instead of focusing on releases as freestanding [12]. The intention is to identify relationships between measures and component behaviour in terms of primarily corrective maintenance (according to defect and failure reports) difficulty [13]. Based on these relationships, we propose to build models to provide understanding and visualisation of software evolution. We use the following model development process [6]:

- Build—create a prediction model based on certain measures.
- Validate—test if the model provides significant results.
- Use —apply the model to its intended domain.

Through measurement and data collection over releases, the models for emphasising software evolution provide an experience base, derived from the Experience Factory concept [14]. Based on historical data, we track problematic legacy components and emphasise software evolution. Knowing which parts of a system might need improvement makes planning and managing for the next release easier and more predictable. Our intention is to use PCA, classification trees, discriminant analysis and box plots as parts of the models. These methods are well documented and have shown to be useful. Also, comparison of how well they distinguish between fault-prone and non fault-prone components is another interesting aspect.

PCA [15] involves a mathematical procedure that transforms a number of correlated variables into some number of groups called principal components or factors. Factors are ranked based on the amount of variability in the data for which each factor accounts. The first of the extracted factors accounts for the highest proportion of the variability in the data. Each succeeding factor accounts for less. This helps to reduce complex data sets and to find relationships between variables.

A classification tree is a graphical representation of an algorithm to classify components (in our case as fault-prone or not fault-prone). In the tree, the nodes represent decisions and the leaves are the classification results. Based on the values for a certain component, the tree is traversed downward from decision node to decision node until a leaf is reached and the classification is completed. The quality of a classification model is assessed by its stability over successive releases and by its Type I and Type II misclassification rates [10]. Khoshgoftaar *et al.* [10] compare two types of classification tree models with regards to Type I and Type II misclassification rates and the models stability across releases. We define the null hypothesis, H_0 for that the component is fault-prone, and the alternative hypothesis, H_1 for that the component is not fault-prone. Type I misclassifications are components classified as non fault-prone when they actually are fault-prone and Type II misclassifications are the opposite.

Discriminant analysis is another multivariate statistical method for classification. The main purpose is to predict group membership, or to identify boundaries between objects, based on a linear combination of interval variables [15]. Discriminant analysis starts with a data set where group membership is known. The outcome is one or more discriminant functions (depending on the numbers of groups) that allows prediction of group membership. The quality aspects are the same as for the classification trees.

Box plots are good for depicting the dispersion and skewdness of samples, i.e., the distribution of a variable [16]. The box plot is a more basic statistical method and is constructed by indicating different percentiles graphically. Different approaches for how they should be created are available [17]. In our case they display the 10th, 25th, 50th, 75th, and 90th percentiles of a variable.

One goal of this study is to combine PCA, the use of classification trees, discriminant analysis and box plots to track software evolution. We use PCA for two reasons. First, to evaluate stability of principal components, second, to see how they relate to classification trees, i.e., so that can we see relationships between principal components, the trees' decision variables and the variables selected in the discriminant analysis. An important reason for using these methods are that they are fairly stable and not sensitive to outliers and that they have shown to be useful in other studies [4][10][18].

We produced the classification trees and discriminant functions for a number of releases. They classify components as fault-prone or not, based on the variables describing type and magnitude of the code changes. This would enable a maintainer to track the types of changes in release *n+1* and estimate whether the components are likely to be fault-prone, based on the classification tree developed in release *n*. It is also possible to develop the models for release *n −1* and use it in *n+1*, but since the components may be changed between the releases, it is necessary to update the models continuously to get accurate results. Therefore, it is more appropriate to develop trees using release *n* and apply them to release *n+1*.

For this approach to work, the classification trees and discriminant functions should be fairly stable across releases. This is why it is important to compare them across releases. When classification trees change from one release to the next (i.e., variables included in the tree and/or their threshold values change), then code decay may be indicated. We use the word code decay to denote code that, in some way, is becoming worse and worse for each software release, raising concerns about lack of maintainability in the future. For example, when smaller changes indicate fault-proneness increasing, then increasing problems and more complex relationships due to code decay may be present. This should be true for discriminant functions as well. Other reasons for changes across releases could indicate re-structuring or reengineering, but after this has occurred they should stabilise again. Finally, box plots should be used to investigate differences between fault-prone and non fault-prone components further by comparing measurement of variables in the classification trees, discriminant functions and from the PCA. This could give an opportunity to investigate further why some components cause a large portion of the problems.

This study should be considered as an exploratory study using a real case because of the limited number of releases, applications and organisations, and the case's emphasis on corrective maintenance. The objectives of this study can be summarised in the form of the following number of questions:

- How do PCA, classification trees and discriminant analysis relate to each other and which aspects of quality problems do they describe?
- Is it possible to use classification trees and discriminant analysis in combination with PCA to track decaying systems and identify the underlying characteristics of the decay?
- Are classification trees stable across releases?
- Are discriminant functions better than classification trees in tracking software evolution?
- Does box plots provide deeper insight into software evolution than only using PCA, classification trees and discriminant analysis?

In Section 3 we present the approach. The case study is reported in Section 4. Finally, Section 5 gives a summary.

3. APPROACH

The approach consists of a number of steps; we overview these in Figure 1. The steps are applied to the case study data in Section 4. The intention of these steps is to analyse the characteristics and stability of changes to the system and the system itself as it evolves. The outcome from one step should act as input to its successor. The steps are the following:

Figure 1 about here, please

Figure 1. Overview of the approach.

Step 1. **Determine variables to include in the analysis**. Based on the purpose of the models, different criteria exist for inclusion of variables. Things such as completeness (missing data points), availability (some defect reports in databases are more detailed than others) and validity (have the data been collected in a credible manner) of the data are issues of concern. For code decay, the most interesting variables to include are those related to the nature (e.g. corrective, adaptive, perfective and preventive) and size (e.g., LOC or check-ins and check-outs) of code changes. Metrics describing structural changes such cohesion and coupling are also of interest.

Step 2. **Identify fault-prone components**. Fault-prone components are identified using a model classifying components as green, yellow or red (GYR) [18]. This approach is used to determine decaying components over successive releases. To find a threshold for this study, components among the top 25 percent in terms of defect reports written against the component are extracted from the first release and the component with the lowest number of defects sets the threshold. This enables comparison over successive releases. Alternatively, values can be set depending on the base quality of the system, purpose of the analysis and the amount of available resources. For detailed discussion how to set the thresholds see [18]. The threshold is set to provide a manageable number of problematic components from each release for further analysis and still be able to compare the characteristics of the releases. Thus, choosing a threshold is a matter of subjective judgement.

Step 3. **Perform PCA**. Doing PCA on the variables of interest determines which groups of variables are linearly correlated and how this relationship changes across releases. For code decay analysis, analysis of structural changes may help maintainers identify and react to major changes in the components. Changes in the system's structure might be indicators of future problems regarding the ability to evolve. Interesting indicators may be increases or decreases in the number of factors, or when some variables switch between PCA factors in successive releases. We attempt to use PCA to reveal relationships among and characteristics of changes made to the system.

Step 4. **Create classification trees**. The next step is to create classification trees for each release. An algorithm creates a tree based on the most significant variables and relates them to the components' classification as fault-prone or not. The outcome may need adjustment such as pruning to reduce redundant information, i.e., removing redundant leaves such as two leaves from the same decision node with the same classification result, and redundant branches. We use classification trees to highlight and visualise the evolution of the system. A classification tree also provides information about the magnitude of different changes made to both fault-prone and non fault-prone components. For each release we create one tree where the algorithm is allowed to choose between all variables and one with variables with the highest loadings in each factor from the previous step, the PCA.

Step 5. **Perform discriminant analysis**. In this step, stepwise discriminant analysis is applied to include those components that significantly contribute to the model and to define discriminant functions. The discriminant functions are used to calculate the probabilities of group belongings. Therefore, one discriminant function is assigned to each class a component could belong to. The creation of the discriminant functions is an iterative procedure, which is repeated until no further variables can be added to the model. The same procedure as in the previous step is applied to select the variables to include. For each release we create discriminant functions where the algorithm is allowed to choose between all the variables and one with variables with the highest loadings in each factor from Step 3.

Step 6. **Stability analysis**. Applying the tree from release *n* to data from release *n+1* investigates whether the results are stable. This gives an indication of how predictable (and thus persistent) problems are. It provides information about the characteristics of the changes in the releases, i.e., how stable the system is.

Step 7. **Create box plots**. To further evaluate the results from the other steps, we create box plots. They aid the interpretation of the results by depicting in summary form the shapes of the distributions of the variables. This is especially useful for identifying outliers and factors that may affect the results, and thus allow a deeper analysis of system evolution. The box plots should be created for both fault-prone and non fault-prone components.

Step 8. **Analysis summary**. As a final step we analyse and compare the results from the different steps. The intention is also to draw conclusions from the results and to evaluate how well the different methods supported tracking software evolution, i.e., answer the questions defining the study objectives.

We applied the eight-step method described above to a large embedded mass storage system that stores and retrieve cartridge tapes. It contains about 800 KLOC of C code in 130 software components. Each component contains a number of files. We studied four releases. The data is based on defect fix reports or source change notices (SCN) most of which are reflective of corrective maintenance action. Every report indicates a problem that had to be corrected. Since the data come from defect reports we are not able to predict the outcome for each release, i.e., we cannot do early predictions based on design and code measures. Neither could we distinguish between different defect fix types. We were restricted to the data in the source change notices because we had no possibility of collecting measures from the design or code. Instead the model should be used to track changes even if it is possible to extend the usage of the model when some of the variables in the model can be predicted or collected in an early development phase. In this case study, components are defined as collections of files in the same directory. The directory structure is the physical architecture and our assumption (due to lack of information) is that it, to some extent, should reflect the functional architecture.

We performed the analysis using SYSTAT® for the creation of the classification trees, and SPSS® for the PCA and discriminant analysis. In creating the classification trees, SYSTAT® uses an algorithm, Gini index, which recursively partitions the set of components [19]. When creating a tree, the data are successively split with a threshold of an independent variable that maximally distinguishes between the dependent variable in the left and right branches. The splitting continues until the nodes are pure or the data are too sparse. We created the box plots with StatView®.

4. CASE STUDY

4.1. Step 1: determine variables to include in the analysis

4.1.1. Data availability

The variables to include in the analysis are based on the information contained in the defect database. Each defect report included a series of code change metrics that were automatically collected through measurement tools that compared the code before and after the defect fix. We chose these as the variables to include in the analysis because they were relevant, complete, credible and available. These metrics measured impact of change, how many files were involved and the magnitude of the change. The measures are described in the following subsections.

The measures are not normalised (according to number of files in each component). It could be argued that this should be done to better compare the effect of large changes on components with few files and vice versa. We wanted to reflect the actual characteristics (where most effort is spent) of the problematic components and decided to use the raw data instead. Unfortunately, real effort data was not available. Other possible measures can be found in [20].

4.1.2. Changed components measure

This measure indicates how many times a component had to be changed in a release. The higher the number, the more fixes had to be made to the component. It can also be an indication that it is a central or complex component.

- SCN_comp = Number of SCN's (defect fix reports) that involve a component.

4.1.3. Degree of interaction measure

This measure associates a number within each component that denotes how often it was involved in defects that required corrections extending beyond the current component, i.e., reported defects involving more than one component. The higher this number is, the smaller the degree of encapsulation and also the higher the degree of coupling or interaction between a component and others. Such strong dependencies are indications of relationship decay between the components, increasing interaction, or poor design from the start.

In the absence of the actual degree of interaction measures associated with corrective maintenance reports, we needed a surrogate measure. Our rationale for the measure proposed is based on the following: if a set of strongly coupled components decays, corrective actions to fix defects are likely to require changes to many of the related files.

Further, it is very undesirable for components to come to rely too much on the particulars of the implementation of other components because it represents a lack of encapsulation. In such cases, defects are related to shared data structures. The need to access more than one component (similar to the files within a component) indicates that a defect exists in more than one component or the defect exists in a single component but its fix has ramifications in the way other components interact with the particular component.

The measure is calculated as follows: for each defect fix report where more than one component was changed, increase the measure by one for each changed component. Thus for 1 fix report on one component but involving 3 other components, the measure is 4.

- Multi_rel = Number of times a component was changed together with other components.

4.1.4. Impact measures

These measures are indicators of how many times the files were changed for each component in each release. This is an indicator of how large the changes were in a component. The larger the number is, the more times the files had to be changed. This can indicate a fault-prone or complex component. We distinguish between .c and .h files to evaluate the role of shared data (.h) versus functions (.c). There might exist a risk that poor programming habits influence this measure, e.g, if the problems was not properly solved and a file had to be changed over and over again. But this could also be an indication of increasing complexity and code decay.

- Sys_impact_c = Total number of changes to .c files in a release per component.

- Sys_impact_h = Total number of changes to .h files in a release per component.

4.1.5. Unique files measures

These measures count how many of the components' files were touched in a release. It can be an indication of how widespread the corrective actions within a component have been, or of how much the component was broken and needed to be fixed. As before, the measures distinguish between .c and .h files.

- Unique_c = Number of unique .c files fixed in a component.
- Unique_h = Number of unique .h files fixed in a component.

4.1.6. Average number of changes measures

This measure indicates how fault-prone and complex a component is. It tells the number of changes to a component's files normalised by the number of unique changed files in the component. Since we do not have any actual values for the file sizes we assume that the average file sizes are similar for all components. This assumption can be misleading if there are large variations between the different files, because some of the files might be larger due to more functionality and therefore more effort was directed towards these files.

The measure gives each component a number that increases as the component requires more changes for defect corrections. This measure could be an indication of decay resulting from the increased complexity of particular files or the breakdown of encapsulation between files, i.e., the internal degree of interaction between files in components causes more than one file to be changed. Having to change a file multiple times indicates that a change was made and that the change probably did not work. The action of repeatedly changing a file indicates that maintenance may have made incorrect decisions in correcting the file or that the file had multiple problems. Another explanation could be that the complexity of the code in the file makes it difficult to comprehend. Either way, increasing numbers in these indicators are signs of code decay.

- Avg_fix_c = The quotient from dividing Sys_impact_c by Unique_c.
- Avg_fix_h = The quotient from dividing Sys_impact_h by Unique_h.

4.1.7. Size measures

These measures provide information about how much the components (their .c and .h files) have changed in their lines of code (LOC) as a result of doing the fixing work, i.e., the incremental size of the work. They measure also where the largest amount of change has occurred in the .c or .h files. The measures cover added LOC, deleted LOC, added executable LOC, and deleted executable LOC. These measures provide information about how stable the component and its files are and the amount of change from release to release. Increasing numbers in these measures might be a sign of potential decay. Normally.h files do not contain executable code, but in this case some .h files were observed to contain function prototypes. This is not considered a problem because the amount of shared data is much larger.

- Size_add_c = Cumulative number of LOC added for .c files.
- Size_add_h = Cumulative number of LOC added for .h files.
- Size_del_c = Cumulative number of LOC deleted for.c files.
- Size_del_h = Cumulative number of LOC deleted for.h files.

- Size_exec_add_c = Cumulative number of executable LOC added for .c files.
- Size_exec_add_h = Cumulative number of executable LOC added for .h files.
- Size_exec_del_c = Cumulative number of executable LOC deleted for. c files.
- Size_exec_del_h = Cumulative number of executable LOC deleted for .h files.

Table l about here, please

Table I. Number of components identified as fault-prone in each release and defects covered

4.2. Step 2: identify fault-prone components

Components are ranked based on the number of defect fix reports (SCN_comp) written against it. To extract the subset with the most fault-prone components in each release we used a threshold. The threshold in the case study is derived from the first release and used in the following releases. The top 25 percent of the ranked components are considered fault-prone. In case of ties in rank4 that would cause more than 25 percent of the components to be included, the smaller set of components was chosen. Thus in this case 25 percent of the 130 components equals 32 components, which yields a lowest ranking of 98. For Release 1, 40 components (instead of 29) would have been extracted if we had chosen the next lower ranking as the threshold, which was 91 (102 was used). The number of defect fix reports (SCN_comp) of the fault-prone components (top 25 percent) is used as the threshold for all releases. The top 25 percent in Release 1 has four or more defects written against the components. Therefore, four defect reports for a component is used as the threshold for considering a component as fault-prone.

Table 1 shows how many components were identified as fault-prone in each release. Table I also shows the number of defects reported for the fault-prone components together with the total number of defect reports for each release. The ratio between the defects covered by the fault-prone components and the total amount of defects in the releases is also shown.

Table II about here, please

Table II. PCA results from Release 1.

Table III about here, please

Table III. PCA results from Release 2.

Table IV about here, please

Table IV. PCA results from Release 3.

Table V about here, please

Table V. PCA results from Release 4.

4.3. Step 3: perform PCA

The main purpose of PCA is to simplify complex sets of data so it can be represented by a few underlying factors. In our case, we use it to determine which measures of defect correction work are grouped together and whether these groupings are stable across releases. The intention is to reveal the characteristics of the changes made to components due to some defect. We applied PCA with an orthotran/varimax transformation and only extracted factors with an eigenvalue greater than 1.0. The results of the PCA for the four releases are detailed in Tables II through V. We did not include SCN_comp because it is our dependent variable. Multi_rel has a high correlation with SCN_comp, i.e., most of the defect reports affected more than one component, and was therefore not included.

In Tables II through V the factors are arranged so that the first of the extracted factors accounts for the highest proportion of the variability in the data. Each succeeding factor accounts for less. The accumulated proportions of explained variance can be found at the bottom of the tables. Dark grey stipple areas in the factor columns indicate the variables contributing the highest loadings, i.e., correlations between variable and factor. Light grey stipple areas indicate variables with loadings less than the dark grey stippled ones but higher than 0.5.

The PCA for Release 1 produced four factors (see Table II). The first factor groups measures related to LOC changes in the .c files and Avg_fix_h. Sys_impact_c has a high loading too. A possible explanation is that adding and deleting LOC in .c files affects the number of changed files. Avg_fix_h is more difficult to explain and can be a statistical artefact due the derived nature of these variables (see Section 4.1.6). Unfortunately we cannot investigate this further because of the nature of our data. The second factor groups the variables related to number of changed .h files and LOC deleted in these files. Number of changed .c files is grouped together in factor three. In factor 4, the number of added LOC in .h are grouped. Also, Unique_h has a high loading in the second factor, which shows it dependencies to other .h related variables. One interpretation could be that code has been added to many unique .h files. Most of the groupings make sense but the variance separation between them indicates that the changes are very different in their nature.

Release 2 yields three factors. The grouping of the variables is more homogenous and intuitive than for Release 1. Factor 1 groups most of the .h related measures except for Avg_fix_h, which belongs to Factor 2 together with the LOC related .c measures. The reason for this is difficult to say. Factor 2 also includes the measure for number of unique .c files even though this one almost has the same loading in Factor 3. The third factor is related to number of changed files. Compared to Release 1, the results from Release 2 show factors which are more related to .c and .h measures separately with a few exceptions.

For the other two releases (see Tables IV and V) the .c and the .h files are grouped together respectively. There are only two factors. The reason for this might be that the releases are very homogeneous and no major problems have occurred. The only difference between the release is the order of the factors and that Avg_fix_h still has a relatively high loading related to .c files in Release 3.

Since the first factor explains most of the variance (of the separate factors) in the data set, this factor could be considered to describe the main characteristics of the changes in a particular release. Each succeeding factor describes additional changes. In Release 1, the major part of the changes was .c LOC related. The secondary impact was the deletion of LOC in .h files and the number of touched .h files. In Release 2 the first factor grouped most of the changes that were made to .h files, which could be an indication of major restructuring compared to Release 1 where the changes mostly were functional. As mentioned above, the two last releases group the .c and the .h files together respectively. In Release 3 most of the changes are related to .c files, which supports the impression of a release with mostly functional defect fixes while Release 4 grouped .h files in the first factor. It should be noted that in Release 4 the difference of explained variance between the two factors is very small.

Overall, the trend from Release 1 to Release 4 is that different .c and .h measures were grouped into a number of different factors in Release 1, but across the releases factors are stabilising by forming groups with .c and .h measures respectively. One reason could be that the type of defects are more homogenous and more related to certain types of changes.

Table VI about here, please

Table VI. Classification tree parameters.

4.4. Step 4: create classification trees

4.4.1. Selection of data

A classification tree can be seen as a visualisation of a decision algorithm where the nodes are decision points and the leaves are the results of the decisions. Each decision is symbolised by a threshold for a certain variable. When the condition in the node is true the left branch is taken otherwise the right one is taken. For example, in Figure 2, the first decision is whether a component has fewer than 6 changes to .c files. If this is true the second decision is whether the number of LOC added to .h files was less than 182. If this also was true the component should be classified as non-fault-prone; otherwise it should be classified as fault-prone. The proportional reduction in error is equal to the goodness of fit. It is also comparable with R^2 for a Pearson correlation analysis. The impurity is a measure of how 'clean' a node is and the closer the impurity is to zero, the better [19].

We have created our trees based on the raw data. It could be argued that we should use the factors and factor scores instead. Since the tree algorithm reduces the variables to those significantly discriminating between fault-prone and non-fault-prone there might be a risk that variables not contributing to the discrimination are weighted into the factors. This may have undesirable effects on the results and make the trees difficult to interpret. Also, the number of variables for the tree algorithm to choose between would be very small, i.e., in this case there would have been two to four. Given this, we created two types of trees: one where all the variables were available to the algorithm and one where the algorithm was allowed to choose between the variables with the highest factor loadings in each release.

The characteristics of a release can be defined according to how complex a tree is and the nodes' thresholds. It should be noted that the thresholds used in the trees decision nodes are not the same threshold used to classify components in Section 4.2. Looking across releases, it is possible to analyse changes in the decision variables, and to track the evolution of the system with regard to fault-proneness.

The parameters used when creating the trees can be found in Table 6. The number of splits decides maximum number of splits or decisions in the tree, while the minimum proportion is the smallest proportion reduction in error for the tree allowed at any split (i.e., how much it must improve the goodness of fit). The split minimum is the smallest value allowed at any node, and the minimum objects is the minimum count at any node. Finally, the split method is the method used to create the tree.

Figure 2 about here, please

Figure 2. Release 1 classification tree based on all variables.

Figure 3 about here, please

Figure 3. Release 2 classification tree based on all variables.

Figure 4 about here, please

Figure 4. Release 3 classification tree based on all variables.

Figure 5 about here, please

Figure 5. Release 4 classification tree based on all variables.

4.4.2. Trees based on all available variables

In Release 1 (see Figure 2), three variables are included in the tree. They are Sys_impact_c, Size_add_h and Avg_fix_c. It can be noted that if 6 or more changes to .c files occurred and the average fix rate to .c files was larger than 1.1, the component should be classified as fault-prone. This should be compared with the path, where few changes to .c files and 182 or more LOC added to .h files also classified a component as fault-prone. It seems like either there are changes in many .c files or large changes to the .h files among the fault-prone components. Finally, if the ratio between number of changed .c files and the uniqueness of those is larger than 1.1, the components are classified as fault-prone.

Figure 3 shows the classification tree for Release 2. The tree has the same complexity as the one from Release 1. The root has the same threshold as before and the threshold for Avg_fix_c has increased slightly. Size_del_h has replaced Size_add_h with a higher threshold. Based on this, it could be argued that the changes in this release are more extensive than in Release 1.

Figure 4 shows the classification tree for Release 3. This tree has one decision node less compared to Releases 1 and 2. It only contains two variables, Avg_fix_h and Size_del_c. Neither had been included in earlier trees. This could be an effect of the small number of defects reported in this release (this release was a small release without any major system changes). This affects the threshold because many of the components were not changed at all or only included small changes and thus the fault-prone components have low values for the change measures. Noticeable is that Avg_fix_h has high loadings in both factors in the PCA.

Finally, the classification tree for Release 4 consists of three variables (see Figure 5). Again Sys_impact_c is the root. Avg_fix_c shows up again but Sys_impact_h is new. The threshold for Sys_impact_c is a little bit lower than in Releases 1 and 2. The interesting part of this tree is the increasing value of Avg_fix_c that indicates a larger number of changes to each unique file.

If we exclude Release 3, the overall results show that Sys_impact_c is the root decision node even though the threshold decreases as the system evolves, from six to four changes to .c files. It is noticeable that the number of times a .c file have been touched on average increases from 1.10 to 1.38 for the fault-prone components. This is an indication of increasing problems among a subset of .c files. For the components with fewer changes to .c files, they are considered fault-prone based on their value for some .h related variables. In Release 1 it was Size_add_h, in Release 2 it was Size_del_h and in Release 4 it was Sys_impact_h. This makes it difficult to draw any general conclusion about the trend or direction of change in the system overall for Releases 1 through 4. Instead these decision nodes could reflect the characteristics of just the changes made to each release. It should be noted that it is not certain that the root decision variable comes from first PCA factor. On the other hand, the decision variables in the trees represent most of the PCA factors.

The overall results show that the trees are fairly stable as long as the releases have similar characteristics. Some decision thresholds may vary and some of the variables in the decision nodes may also change. If the characteristics are different the trees will look different too. This could be seen in the tree for Release 3, which was a smaller release, where the variables and thresholds used are completely different compared to the other releases.

Figure 6 about here, please

Figure 6. Release 1 classification tree based on selected variables.

Figure 7 about here, please

Figure 7. Release 2 classification tree based on selected variables.

Figure 8 about here, please

Figure 8. Release 3 classification tree based on selected variables.

Figure 9 about here, please

Figure 9. Release 4 classification tree based on selected variables.

4.4.3. Trees based on selected variables

The second variant of the created trees only allowed the tree algorithm to select among variables with the highest loadings in the PCA. For example, in Release 1 this approach resulted in the following four variables: Size_del_c, Size_exec_del_h, Avg_fix_c and Size_add_h. In Release 1 (see Figure 6) Avg_fix_c and Size_add_h are the same as in Figure 3 and with the same thresholds, but Avg_fix_c is the root. Size_del_c is new and indicates a low impurity value for the fault-prone components identified, even though the impurity in the opposite branch is higher. The fit of the tree (proportional reduction in error) is lower than in the all-variables tree, i.e., 0.727 compared to 0.846.

In Release 2 all selected variables were included in the tree (see Figure 7). The Avg_fix_c value indicates more changes to each unique .c file. Again the goodness of fit is not as good as in the previous all-variables tree (compare with Figure 4).

The tree created for Release 3 is the most differentiated of the trees because it uses Size_del_h in two decision nodes (see Figure 8). The tree algorithm excluded one variable, Size_exec_del_c and created a tree with the best goodness of fit among the trees based on the PCA selected variables. Noticeable is that the variables only are related to LOC deletion.

Finally, the fourth tree (see Figure 9) used both of the two variables that were available to the tree algorithm. Therefore the Size_exec_del_h is used twice with different thresholds.

Even though the goodness of fit for the last four trees are not as good as for the first set, where the tree algorithm was able to choose between all available variables, the goodness of fit values are still high. The shape is almost the same, except for Release 3, where it is a little bit more complex. But there is always a risk that these trees do not reflect the actual characteristics since they are forced in one way or another and therefore in this case study are open to question from an exploratory perspective.

Table VII about here, please

Table VII. Discriminant analysis parameters

Table VIII about here, please

Table VIII. Variables included in the discriminant analysis

4.5. Step 5: perform discriminant analysis

An alternative to classifications trees is discriminant analysis, since it can also be used to classify components as fault-prone or non-fault-prone. Discriminant functions are used to calculate the probabilities of group belongings. Therefore, there is one discriminant function for each class a component could be assigned to. Numerous approaches exist, but for the results to be comparable with the classification trees we have chosen to use a forward stepwise parametric approach. The inclusion and creation of the discriminant functions is an iterative procedure which is repeated until no more variable can be added or removed from the model with an improvement in discrimination power. The variables used as input are reduced based on a F-test with the inclusion criteria of 0.05 significance and 0.1 as exclusion criteria (these values are commonly used in many papers). An overview of the parameters can be found in Table VII.

In Table VIII the results from the creation of discriminant functions can be found. The hyphens (-) in the table indicate variables not included in the discriminant functions, while the dots (•) indicate variables included in the discriminant functions. The dots within parenthesis are the variables included when the selected variables from the PCA were used. The actual coefficients are not included since they are difficult to interpret according to the understanding of the evolution, because they are also affected by the constant (α) that affects the classification. The reason is that discriminant analysis is used to calculate probabilities of group belongings. The coefficients and the constants are maximised for this purpose, which means that the same result could be achieved with a positive constant and negative coefficients or vice versa and therefore differ between the releases. This makes interpretation of the coefficients difficult to explain briefly and clearly.

Compared to the trees in Section 4.4, the variables and the number included differ. For example, the tree from Release 1 included Sys_impact_c, Size_add_h and Avg_fix_c while the discriminant analysis used Sys_impact_c, Size_add_h, Unique_c, Size_add_c, Sys_impact_h, Unique_h and Size_exec_add_h (but not Avg_fix_c).

The variables included when only a limited number were available (based on the PCA) are similar to the variables included in the trees. The reason is that the information is sparser and to be able to achieve the best goodness of fit, almost the same variables are used. In this case we use the squared canonical correlation (R^2), which is the multiple correlation between the predictors and the discriminant functions (for more information see [21]). Overall, the goodness of fit values for the discriminant functions is not as good as for the classification trees even though they include more variables, i.e., the functions are based on more information. It could be argued that classification trees better describe the characteristics of the changes. The evaluation in the next step should provide more information before any conclusions are drawn.

Table IX about here, please

Table IX. Tress results from Release 1 to Release 2.

Table X about here, please

Table X. Tree results from Release 2 to Release 3.

Table XI about here, please

Table XI. Tree results from Release 3 to Release 4.

Table XII about here, please

Table XII. Tree results from Release 2 to Release 4.

4.6. Step 6: stability analysis

To further investigate the stability of the classification trees we apply the classification tree built in Release n to the data from Release *n+l*. This provides more knowledge about how stable the characteristics of the changes to the system are, and it provides a deeper analysis of the stability of the classification trees. This stability analysis can also be carried out for the discriminant functions, and is reported later in this Subsection.

The results from applying stability analysis to the classification trees can be found in Table IX to Table XI. The overall misclassification percentages are calculated as (Type I errors + Type II errors) divided by (total number of components). The miscalculation percentages within parentheses are the values from the trees based on selected PCA variables. The results show few Type I and Type II errors. One exception can be found in Table XI where the Type I error is as high as 13. An explanation is that the tree only contained two decision variables and therefore could not discriminate with fine enough granularity. If the tree had been more complex and included more variables, the tree might have corrected the errors further down in the classification hierarchy.

The most accurate explanation is that Release 2 is mainly a defect fixing release while the others are enhancement releases. This classification was done based on discussion with different persons in the organisation. To obtain decent predictability it is therefore important to consider the nature of the releases. It makes little sense to estimate the fault-proneness of a defect fixing release based on data for an enhancement release. Thus it makes more sense to estimate fault-proneness for Release 2 from Release 1 and for Release 4 from Release 2. The reason why we carried out this analysis was to highlight the discrepancies. The results estimating Release 4 from Release 2 can be found in Table XII. The result is a much smaller Type II error and a slightly smaller Type I error which is more comparable to the other results.

Even though there are few errors in Table X we have to keep in mind that Release 3 only had 15 fault-prone components, and therefore these values can be a little misleading. Overall the results are satisfying and show that the results from using the trees are stable even though the trees are slightly different.

The results from the trees based on selected variables are reasonably equivalent. They classify almost the same numbers of non-fault-prone components correctly even though the Type I errors sometimes are much higher.

In Tables XIII though XVI we present the results from applying stability analysis to the discriminant functions for the different releases. The results are very similar compared to the trees. The largest differences can be found when the discriminant functions based on the selected variables from the PCA are used. For example, the misclassification percentage in parenthesis in Table XIII reflects the fact that the Type I error is low while the Type II error is very high. Another observation for both Table XIII and Table XVI, is that when the Type I error decreases, the Type II error increases as a result of classifying too many components as fault-prone.

Table XIII about here, please

Table XIII. Discriminant results from Release 1 to Release 2.

Table XIV about here, please

Table XIV. Discriminant results from Release 2 to Release 3.

Table XV about here, please

Table XV. Discriminant results from Release 3 to Release 4.

Table XVI about here, please

Table XVI. Discriminant results from Release 2 to Release 4.

4.7. Step 7: create box plots

The results from PCA and the two classification approaches point to differences between factor grouping of variables between releases and for variables included in the classification approaches. To investigate these differences further we compare the actual measurements or distributions of these variables. Figures 10 through 14 show box plots of five variables for each of the four releases. A box plot shows the 25th and 75th quartiles as a box with the 50th (the median) as a line in the box. The 10th and 90th percentiles are shown as vertical lines and the small circles are outliers [22]. For each release there are two box plots, one for non-fault-prone components and one for fault-prone components (NO and YES in the figures).

Before we do any comparisons, we have to keep in mind that Release 3 only included 15 fault-prone components while Release 2 included 39, and Release 1 and 4 each contained 29. This affects the data distribution for fault-prone components, i.e., small changes affects the box plots more when there are few data points and the distribution often becomes more widespread.

The four diagrams in Figure 10 shows the number of .c files (Sys_impact_c) that were changed in each release. The boxes for the non-fault-prone components are decreasing slightly over the releases. Looking at the fault-prone components, the number of files changed in the problematic components is slightly decreasing in range in successive releases and the median is stable around 10 and the largest changes occurred in Release 2. Comparing the non fault-prone components and the fault-prone ones (NO and YES box plots), problematic components clearly show more system impact.

Avg_fix_c describes the number of changed .c files normalised according how many of those that were unique (Sys_impact_c/Unique_c). Values closer to one indicate that few changes were made to each .c file. Figure 11 shows that the distribution of the non-fault-prone components is almost identical to the distribution for the fault-prone components (except for Release 3), indicating between zero to one change to the Unique_c files. For the fault-prone components the values are increasing from Release 1 to Release 4. This could be an indication of a few complex and problematic files that cause many of the changes. The distinct differences in the distributions between non-fault-prone and fault-prone components is an indicator for inclusion of this variable in the classification trees and discriminant functions.

Figure 12 presents Size_add_c. Some outliers are not shown because they were very extreme and created distorted box plots. It could have been possible to redo the other steps in the analysis excluding the outliers but then the real characteristics of the releases would not been revealed. A trend for both non-fault-prone and fault-prone components is that the number of added LOC decreases. One explanation could be that the system becomes more stable according to functionality and no new code is added. Instead the changes in the code are defects fixes. The non-fault-prone components include very small changes but the difference between non-fault-prone and fault-prone is probably not enough to discriminate between them.

We can find similar trends for Size_del_h in Figure 13 as for Size_add_c, with a decreasing amount of deleted LOC in the .h files. This is probably the result of a stabilising system. In Release 2 and Release 3, the boxes are quite large even though the median line is around 100 LOC. Release 2 was one of the larger releases and had a lot of problematic code that could have been removed from the components and later rewritten. Release 3 was a defect fixing release with the intention to remove defects and enhance the system.

Finally, Figure 14 shows the degree of interaction measure (see Section 4.1.3) of components that required corrective maintenance across releases. The non-fault-prone components have fairly low values. Values for Release 1 were the highest, but still much smaller than for fault-prone components, which show much higher values. In Release 2 the maximum value (outliers) is high but the box is not as large as in Release 4. This is definitely an indication of decay among the components in Release 4. We also have to keep in mind that Release 4 included many changes in related .c files.

Overall the results show for the problematic components that:

- The number of changes to unique files increases.
- More components needed to be changed to fix problems. Thus the degree of interaction and relationships between components play a major part in decay for this system.
- The size of a change is slightly decreasing. Thus it is not any large obvious omission that is the problem, but subtle problems that involve multiple components.

Together this indicates that there are parts of the system that are decaying and that the system is difficult to maintain. The source of decay is most likely the interdependence between components.

4.8. Step 8: analysis summary

This section is intended to summarise the findings of the case study and to highlight interesting relationships between software evolution and the methods used. While summarising, we try to highlight the characteristics of the system and to answer the five research questions posed in Section 2.

Based on the analysis in the previous sections, the answers to the research questions are as follows:

- PCA, classification trees and discriminant analysis have different approaches. They have the ability to describe different aspects of system behaviour with respect to fault-proneness. PCA describes the overall nature of the changes to components and the characteristics of release, classification trees describe the nature of changes associated with fault-proneness, while discriminant analysis can be used for prediction but does not provide insight into the problems.
- Together PCA and classification trees could provide a tool for dealing with fault-prone components and software evolution. Discriminant analysis was not so appropriate.
- The classification trees are fairly stable across similar releases according to variables in the decision nodes and thresholds. However, some parts of the tree vary according to the characteristic of the releases. The results from applying a tree from release *n* to *n+1* provides few misclassifications for similar releases.
- Classification trees are better suited for tracking software evolution since they provide visualisation and could be more easily interpreted compared to discriminant analysis.
- Box plots are useful to get deeper insight into problems and some of the results were highlighted by the box plots. It was also easier to get an overview of the problems' magnitude.

In more detail, the PCA results show a decreasing number of factors over successive release, from four factors in Release 1 to two factors in Release 3 and Release 4. In the first release the factors are mainly related to .c size variables, .h files changed, .h LOC deleted, .c files changed, and .h LOC added. As the system evolves, the grouping of .c-related variables and .h variables becomes more distinct. This is probably the result of a stabilising system.

By looking at the classification trees across the four releases, we can see some trends. The root decision node in all trees except Release 3, is Sys_impact_c. The threshold varies a little but this variable seems to be the one that affects the classification of most of the fault-prone components. Included in all trees (except Release 3) is Avg_fix_c. The trend is that variables like Size_add_h is replaced by Size_del_h, Size_del_c and Sys_impact_h as the system evolves. The tree in Release 3 is one decision node smaller than the other trees and is based on changes to .h files and LOC deleted in .c files. One reason for the variations between the releases can be their characteristics. For example, Release 3 seems to be a smaller defect-fixing release stabilising the system, while Release 1 and Release 2 are larger, reflecting enhancements, according to discussions with involved people. Overall, Sys_impact_c seems to discriminate roughly between the fault-prone and non-fault-prone components, even though every release includes different variables for further refinement.

The results from the discriminant analysis show that more variables were included compared to the classification trees. The trend was going from seven variables in Release 1 to three in Release 3 but then to six in Release 4. The overlap in variables between the discriminant functions and the classification trees was most extensive in Release 4 where all three variables from the tree were used by the discriminant functions. None of the variables were used in all four releases but some of them were used three times. These were Unique_c, Sys_impact_h and Size_add_h. None of these were frequently used by the classification trees.

Comparing the variables included in the classification trees with the results from the PCA, we can see that the trees mostly include variables from all factors of the PCA even though they may not be the ones with the highest factor loadings. The exception is Release I where only variables from factors three and four are included, i.e., of the four factors shown, the two factors with the lowest eigenvalues. In Release 1 and Release 2, Sys_impact_c has a high loading in two factors, which indicate a large variability, and is therefore probably included. The discriminant functions included one or more variables from each release, which might be obvious when they used almost twice as many variables as there were factors.

The classification trees and discriminant functions based on a subset of variables from the PCA are a little bit different. The tree in Release 1 is similar for both cases, i.e., two out of three variables are the same with the same thresholds. The problem is that they are more or less forced and therefore do not reflect the underlying characteristics. This can be seen when we apply a classification tree or discriminant functions from release *n* to data from release *n+1*. Even though the differences sometimes might be small, they exist. The results from the application are promising because the misclassification rates are relatively low for both classification trees and discrimuiant analysis even though they are a little higher for the discriminant analysis. Release 3 was a smaller release and the results applying its tree and discriminant functions to Release 4 were not as good. Using tree or discriminant functions from Release 2, instead of those from Release 3, to predict Release 4 provided better results. This indicates that the trees are stable across similar releases and could therefore be used to guide improvement efforts.

The box plots confirmed some of the results by depicting the distribution of the measures. For example, the representation of the distribution for Sys_impact_c and Avg_fix_c provides understanding of why they were used frequently in the classification trees. Another interesting aspect provided by the box plots was the increasing number of defect fix reports where more than one component was changed among the fault-prone components. Together with the increasing Avg_fix_c, we can conclude that there exist a number of components that are associated with the majority of the defect reports.

5. CONCLUSION

Prediction of fault-prone components is a very important task in being able to direct effort and apply necessary corrective actions. But only looking at the defects is not enough to assess fault-prone components. The important action is to track software evolution. This paper reported an approach for tracking software evolution and to evaluate the approach, provided a case study of a C-implemented software system about which corrective maintenance historical data were available.

- Firstly, the purpose was to investigate the usefulness of the different statistical methods used in the approach. The approach included principal components analysis (PCA) to reveal the characteristics of the changes made to the components due to some defect, classification trees and discriminant analysis to visualise and to evaluate the stability for the prediction purpose, and box plots to highlight and depict distributions of different measures.
- Secondly, the intention was to compare which of two statistical methods provided the best information from a software evolution perspective, i.e., classification trees or discriminant analysis. The information used in the study was extracted from defect reports spanning four releases, and classified components as fault-prone or not depending on whether the number of defect reports written against them was above a certain threshold.

The approach was successful at tracking software evolution and highlighting underlying problems. Most suited to describe different aspects of a system were PCA and classification trees even though there are no real relationships between them. Classification trees are better suited to describe software evolution than discriminant analysis due to their easier graphic representation. The visualisation of the system via classification trees makes it easy to get an overview and find areas for further investigation. Together with box plots, a useable model can be created to track software evolution.

The results from the case study show a stabilising system with a subset of components contributing most of the problems. One of the limitations of this study is the number of releases. Our study included four release, which should be compared to other studies that suggest a system may take some six or seven releases for its evolution to stabilise [23]. Another limitation is the limited data set, primarily from corrective maintenance, used in the study. For example, product measures were not available and we had to make some assumptions about file sizes, both of which could affect the interpretations of the results.

The number of factors extracted by the PCA decreased to two as the system evolved and .c- and .h-related variables are grouped respectively. This is one indication of less complex relationships between the different variables and therefore also of the characteristics of the defects. The classification trees show an decreasing amount of changes to .c files but with an increasing number of changes to each unique file. The variable describing the number of changes to .h files is included in the tree for the last release.

The box plots verified the previous results and also indicated an increasing amount of defects when more than one component was changed. Together this shows that a subset of problematic components are highly coupled with regards to defect repair. This could be an effect of architectural erosion and should be an incitement to focus on these components before maintenance absorb too many resources.

From the results just reviewed, the approach was useful for tracking software evolution both of the software system and of its problematic components. It also was useful in revealing underlying problems in the software system.

Acknowledgement

We thank The Swedish National Board for Industrial and Technical Development (NUTEK) for its support. We also thank the two employees at the case study site who helped us collect and interpret the data, and we thank the *Journal's* referees for their suggestions.

References

1. Parnas DL. Software aging. *Proceedings International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1994; 279–287.
2. von Mayrhauser A, Wang J, Ohlsson MC, Wohlin C. Deriving a fault architecture from defect history. *Proceedings 10th International Symposium on Software Reliability Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1999; 295–303.
3. Munson J, Elbaum S. Code churn: a measure for estimating the impact of code change. *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press: Los Alamitos CA, 1998; 24–31.
4. Khoshgoftaar TM, Allen EB, Kalaichelvan KS, Goel N. Early quality prediction: a case study in telecommunications. *IEEE Software* 1996; **13**(1):65–71.
5. Schneidewind NF. Software metrics model for integrating quality control and prediction. *Proceedings 8th International Symposium on Software Reliability Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1997; 402–415.
6. Ohlsson N, Relander M, Wohlin C. Quality improvement by identification of fault-prone

modules using software design metric. *Proceedings International Conference on Software Quality*. IEEE Computer Society Press: Los Alamitos CA, 1996; 1–13.

7. Briand LC, Basili VR, Hetmanski CJ. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering* 1993; **19**(11):1028–1044.
8. Zhao M, Wohlin C, Ohlsson N, Xie M. A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology* 1998; **40**(14):801–809.
9. Khoshgoftaar TM, Allen EB, Jones WD, Hudepohl JP. Assessing uncertain predictions of software quality. *Proceedings International Software Metrics Symposium.* IEEE Computer Society Press: Los Alamitos CA, 1999; 159–168.
10. Khoshgoftaar TM, Allen EB, Jones WD, Hudepohl JP. Classification tree models of software quality over multiple releases. *Proceedings 10th International Symposium on Software Reliability Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1999; 116–125.
11. Khoshgoftaar TM, Lanning DL. A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software* 1995; **29**(1):85–91.
12. Lehman MM, Belady LA. *Program Evolution: The Process of Software Change*. Academic Press: New York NY, 1985; 560 pp.
13. IEEE. *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12–1990).* Institute of Electrical and Electronics Engineers, Inc.: New York NY, 1991; 84 pp.
14. Basili V, Caldiera G and Rombach D. Experience factory. Marciniak JJ (ed.). *Encyclopedia of Software Engineering,* Vol. 1. John Wiley & Sons, Inc.: New York NY, 1994; 469–476.
15. Kachigan SK. *Multivariate Statistical Analysis: A Conceptual Introduction.* Radius Press: New York NY, 1991; 236–260.
16. Fenton NE, Pfleeger SL. *Software Metrics: A Rigorous and Practical Approach.* Thomson Computer Press: London, 1996; 201–204.
17. Montgomery DC. *Design and Analysis of Experiments.* John Wiley & Sons, Inc.: New York NY, 1997; 720 pp.
18. Ohlsson MC, Wohlin C. Identification of green, yellow and red legacy components. *Proceeding International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1998; 6–15.
19. Hand DJ. *Construction and Assessment of Classification Rules.* John Wiley & Sons, Inc.: New York NY, 1997; 232 pp.
20. Ramil JF, Lehman MM. Metrics of software evolution as effort predictors—a case study. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2000; 163–172.
21. Tabachnick BG, Fidell LS. *Using Multivariate Statistics*. Harper Collins College Publishers, Inc.: New York NY, 1996; 880 pp.
22. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering—An Introduction*. Kluwer Academic Publishers: Norwell MA, 1999; 88–89.
23. Turski WM. A reference model for the smooth growth of software systems. *IEEE Transactions on Software Engineering* 1996; **22**(8):599–600.

Biographies

Magnus C. Ohlsson is a doctoral student at the Department of Communication Systemsat Lund University in Sweden. His main research areas are the use of experience to build prediction models and models for process improvement. Most of the work has been carried out within effort prediction and maintenance. Magnus is a co-author of *Experimentation in Software Engineering—An Introduction* [22]. He received his Master of Science in Software Engineering from University of Karlskrona/Ronneby in 1996, and his Licentiate in Engineering from the Department of Communication Systems at Lund University in 1999. E-mail: Magnus_C.Ohlsson@telecom.lth.se

Anneliese Amschler Andrews is a Professor of Computer Science at Colorado State University and Director of the Colorado Advanced Software Institute. She has published on software testing, software metrics, software maintenance, reliability and performance modelling, and been active in the IEEE Computer Society. Anneliese received a Dipl.-Inf. degree in Informatik (1976) from the Technical University in Karlsruhe and the A.M. (1978) and Ph.D. (1979) in Computer Science from Duke University. E-mail: aaa@cs.colostate.edu

Claes Wohlin is a Professor of Software Engineering at the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in Sweden. Prior to this, he has held professor chairs in software engineering at Lund University and Linköping University. His research interests include empirical methods in software engineering, software metrics, software quality and systematic improvement in software engineering. Claes is the principal author of the book *Experimentation in Software Engineering—An Introduction* [22]. He is co-editor of the journal *Information and Software Technology* published by Elsevier Science, and is the program chair for the 2001 IEEE International Symposium on Software Metrics. He has a Ph.D. in Communication Systems from Lund University. E-mail: Claes.Wohlin@ipd.hk-r.se