C. Wohlin, M. Höst and M. C. Ohlsson, "Understanding the Sources of Software Defects: A Filtering Approach", Proceedings the 8th International Workshop on Program Comprehension, pp. 9-17, Limerick, Ireland, 2000.

Understanding the Sources of Software Defects: A Filtering Approach

Claes Wohlin, Martin Höst and Magnus C. Ohlsson
Dept. of Communication Systems
Lund University
Box 118, SE-221 00 Lund, Sweden
E-mail: (claes.wohlin, martin.host, magnus_c.ohlsson)@telecom.lth.se

Abstract

This paper presents a method proposal of how to use product measures and defect data to enable understanding and identification of design and programming constructs that contribute more than expected to the defect statistics. The paper describes a method that can be used to identify the most defect-prone design and programming constructs and the method proposal is illustrated on data collected from a large software project in the telecommunication domain. The example indicates that it is feasible, based on defect data and product measures, to identify the main sources of defects in terms of design and programming constructs. Potential actions to be taken include less usage of particular design and programming constructs, additional resources for verification of the constructs and further education into how to use the constructs.

Keywords

Defect understanding, prediction system, product measurements, root cause analysis, software metrics.

1. Introduction

One important aspect in the development of software systems is the understanding of the underlying reasons for defects. The understanding forms the basis for decision and actions in order to change the process, provide education or direct verification and validation activities. This is the basis for further improvement activities, such as evaluation of new methods, choice of pilot projects etc. The objective of this paper is to formulate a method to identify design and program constructs that in some sense are over-represented in terms of their relation to software defects. The understanding and identification of such constructs can then form the basis for informed decisions for further actions.

The application of root cause analysis in software engineering is described in, for example, [1, 2, 3]. The main objective for the persons who perform the root cause analysis is to try to extract the underlying reasons of the defects, and in some cases the analysis is performed through interviews in the organization. The experience-based method, suggested in this paper, proposes the use of simple metrics, primarily product metrics, in the analysis. The result of the evaluation can be an important input, together with defect reports, interviews, etc. to the process of extracting the underlying reasons. In a number of ways, this resembles the type of approach that is taken in [4]. There, the defects are first automatically analyzed according to filtering functions based on functions of interest, and then the results form a basis for human interpretation with the objective to improve the process. The work presented here has similarities with the first step described above, i.e., to automatically analyse defects in order to receive a result that can be interpreted by humans. This is also related to the data mining process, for example, described in [5], where the objective is to obtain knowledge from databases which are so large that it is not possible to do this manually.

The presented method is intended to be as easy to use as possible. The basic idea behind it should be easy to understand, and it is described as a number of simple steps that should be carried out. The basic idea of the method is to identify the design and programming constructs that are most related to the defects observed. A construct is defined as the logical abstraction of, for example, a specific programming notation (an if-statement is abstracted to a logical decision) and the construct is represented through a statement, for example, if-then-else in a programming language. When constructs related to defects have been identified, it is possible to take informed decisions of how to address the problems of software defects. The use of the

proposed method is illustrated through data collected from a large software system.

The method presented in this paper is an approach to experience-based understanding and identification of design and programming constructs. The paper is outlined as follows. The usage of product measurements is elaborated in Section 2, and the method proposal is presented in Section 3. To illustrate the method, an example, based on real product measures and defect data, is presented in Section 4. Finally, in Section 5, some conclusions are presented.

2. Product measures as a means of identification

2.1. Process and product relations

The general hypothesis is that a better process results in a better product. That is, there is a relationship between the process and the product. The actual quantitative relationship must be treated for each specific case, but methods are needed to enable control of the relationship. To support the relation from process to product (i.e., the upper arrow in Figure 1), methods for prediction of product quality attributes based on process characteristics are needed. This includes predictions of the implications of process change, for example, what is the effect on the product when the process is changed (presumably improved) in a specific way?

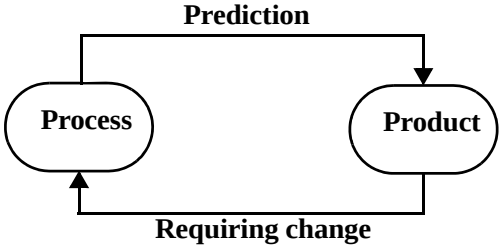

Figure 1. Process and product relationship.

The focus here is on supporting the relation from product to process (i.e., the lower arrow in Figure 1). Based on knowledge and experience of existing products, actions may be taken to change the way software is developed or the information can be used as a means for directing resources to certain parts of the system. In addition, the findings can be a basis for changing review checklists or other methods for defect detection. The intention is to provide a simple method to identify design and programming constructs that need special attention in some way.

The main source of information available is the product and its defect statistics. Hence, it would be favorable if it was possible to identify defect-prone constructs directly from simple product measures.

2.2. Usage of product measures in general

In general, product measures can be used for two different purposes, see Figure 2. First, it can be used to predict attributes or behavior of the current or a forthcoming product, and hence the product measures are fed into a prediction system. This system must be derived based on, for example, the relationships between different measures and defects found. Prediction systems are further discussed in [6] and the use of them to identify defect-prone components is discussed in, for example, [7, 8]. Second, the measures can be used to identify problems in the development process, i.e., to support the relationship from the product to the process in Figure 1. This also requires a prediction system. This type of system should be used to enhance the understanding of defect-prone methods, techniques, activities or, as in this case, specific design and programming constructs.

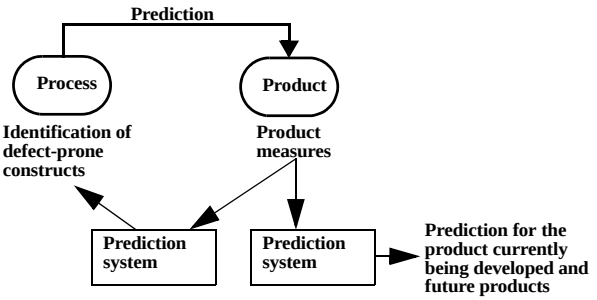

Figure 2. Two possible uses of product measures.

It should be noted that the intention is not to pinpoint the construct contributing to most defects, but to identify those constructs which, in some sense, are over-represented among the trouble reports.

3. Method description

3.1. Introduction

This section describes a method that can be used to apply the prediction system as described in Section 2. In particular, the method may be applied on documents from the design and documents from implementation. The method could be applied to other documents from other phases, but as it is described here, it focuses on design and programming constructs. The method can also be modified in order to use it for other purposes than lowering the number of defects.

The prediction method consists of a number of steps, which should be executed in sequence. The steps resemble the steps normally discussed in data mining [5], and they are:

1. Data collection: The primary data to collect are defect data and product data. This step provides the input to the further steps of the method.
2. Data understanding: Since a number of measures are collected it is important to obtain a basic understanding of how the measures are related. This can be obtained through a factor analysis.
3. Application of prediction filters: The application of the prediction system relies on the process of determining correlations between measures collected from product documentation, and defect data. Based on the understanding and correlation between defect data and other measures collected, constructs that are candidates for actions are identified.
4. Recommendation: Finally, based on the outcome of the previous steps, a recommendation for constructs to be considered for special treatment is formulated. The objective is not to be prescriptive, but to provide support for management in taking decisions.

These steps are further discussed below.

3.2. Data collection (step 1)

The first step in the method is data collection. Product data and defect data should be collected for every component included in the study. Preferably, product data are collected as part of system development and defect data are collected routinely from test and operation. It should be noted that the method as such does not depend upon a certain definition of defects. It is basically up to the user of the proposed approach to define what constitutes a defect, i.e. a defect is defined by the company standard and reporting routines.

Even if the data collection is not performed during the actual development in the project, it should not pass too long from the actual development until the data are collected. Actions taken, with the identification as a basis, should be timely and address the current problems.

The prediction system is based on the correlation between the number of defects and the number of appearances of statements related to different constructs in the components. Therefore, the number of appearances and the number of defects must be measured for the different product documents.

For every component there is one design document and one code document. In the normal case both are included when the method is used, although it is not necessary. The measures that should be collected from every component are:

- Defect data, i.e. the number of defects that have occurred during test and operation for every component.
- Measure of size, i.e., for example, the number of lines or code or the number of pages in every document. One measure must be defined for design documents and one measure must be defined for code documents.
- Measures according to the different constructs of interest, i.e., the number of occurrences of statements representing each construct. Measures should be collected from both design documents and code documents.

The measures are further described and motivated in Section 3.4. The measures used in this particular study can with the right tools be counted automatically.

The constructs for which the number of occurrences is measured must be defined. One example of a construct that can be defined is a choice construct, which can be counted through if-statements, case-statements, etc. Further examples are given in Section 4.2. The constructs must be defined based on the nature of the documents from which they are counted.

When data have been collected, outliers must be identified and removed. Any actions taken should be decided based on components representing the normal execution of the process. Outlier components are components that represent the effects of more special circumstances such as external events, which are unlikely to ever appear again, or having a newly employed responsible for a component. These components should be removed and not used in the method proposed in this paper, since it will not identify problematic constructs in general. The latter is the main concern with the proposed method. The identified outlier components can, however, serve as an additional input to further root cause analysis where it can be analyzed why the component became an outlier.

Outlier analysis can be performed in a number of different ways. One approach is:

- Draw Box-plots [6, 9], for every collected measure.
- Identify potential outliers from the plots.
- Review every potential outlier document and decide if it is an outlier and should be removed or not. It must be decided what documents to remove. For example, if it is decided that a design document is an outlier and should be removed it must be decided if the code document from the same component also should be removed.

Problematic components will probably be identified through more than one measure.

3.3. Data understanding (step 2)

In this step, we should get a basic understanding of the collected data. Especially the product data are investigated. The different measures of the constructs are in many cases related to each other. The data can be analyzed by descriptive statistics such as plots, and the correlation between the different measures can be investigated.

One important aspect of this step is to investigate the relationships between the different constructs. Constructs that vary together are closely related and hence if constructs are related and one of them is identified to be over-represented in terms of defects, then it is probably also wise to consider the constructs that vary in a similar pattern. This can be performed through, for example, a factor analysis (see for example [10]). This analysis results in a number of factors representing underlying causes of the different measures. Related measures are measures that are highly correlated to the same factor. One factor analysis can be performed for the design constructs and one for the code constructs. If both the design document and the code document are used for all components, it is possible to perform one factor analysis for all constructs, both code constructs and design constructs, instead.

3.4. Application of prediction filters (step 3)

3.4.1. Introduction. The prediction system proposed includes a number of filters. The actual choice of the number of filters and which filters are up to the user of the suggested approach. The prediction system, as proposed here, consists of four different filters, which have been formulated based on subjective opinions about the expectations. Other filters could easily be formulated without changing the basic idea of the method. The usage of each filter results in a set of constructs that has been identified as interesting, see Figure 3.

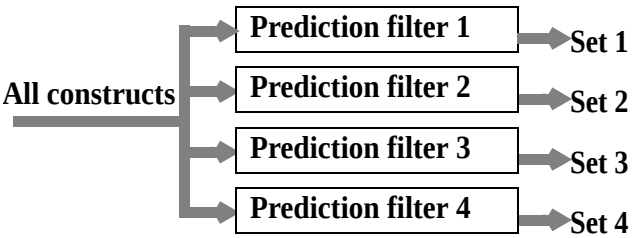

Figure 3. Four prediction filters that upon usage identify four different sets of interesting constructs.

Different sets of interesting constructs could be identified based on the definition of evaluation criteria. Therefore, the underlying motivation of the evaluation criteria should serve as basis for formulating the prediction filters. The four prediction filters are presented together with their underlying motivation in Section 3.4.2.

The usage of the different prediction filters will in most cases not result in exactly the same set of interesting constructs. All constructs in the union of the different interesting sets should be considered when trying to understand the source of software defects, but the constructs that are members of most sets are in some sense more interesting than other constructs.

3.4.2. Four prediction filters. The proposed prediction filters are:

- Prediction filter 1: identifies the construct with the maximum correlation between the number of occurrences and the number of defects. Informal criterion of filter: max(corr(#defects, #occurrences))
- Prediction filter 2: identifies the construct with the maximum correlation between the normalized number of occurrences and the normalized number of defects. Here, the normalization is done with respect to the size of the component where the number of occurrences is counted. Informal criterion of filter: max(corr(#defects/size, #occurrences/size))
- Prediction filter 3: identifies the constructs where the correlation between the number of occurrences and the number of defects is greater than the correlation between the number of occurrences and the size of the components. Informal criterion of filter: corr(#defects, #occurrences) > corr(size, #occurrences)
- Prediction filter 4: identifies the constructs where the correlation between the number of occurrences and the number of defects is greater than the correlation between the size of the components and the number of defects. Informal criterion of filter: corr(#defects, #occurrences) > corr(#defects, size)

The usage of the four prediction filters results in four sets of interesting constructs as illustrated in Figure 3.

The first set, I_1, consists of only one construct, the construct with the highest correlation between the number of occurrences of the statement and the number of defects. This is a straightforward and intuitive choice of construct, but if the sizes of the components are too related to the number of defects this may bias the result since the occurrences of the constructs are correlated with the size. Prediction filter 2 instead uses the normalized values of the number of occurrences and the number of defects instead of the non-normalized values. This results in the second set, I_2.

Instead of identifying the construct satisfying a maximum correlation, as prediction filters 1 and 2 do, all constructs satisfying a correlation greater than some threshold value could be identified.

Filters 3 and 4 have been defined since the size of the components always can be measured, and because it in

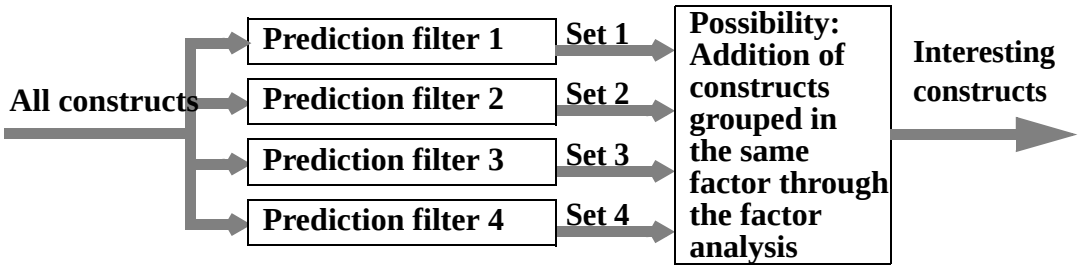

Figure 4. Constructs which are related to the same factors as the already identified constructs may also be considered interesting.

most cases has a significant impact on the number of defects in the component. The third set, I_3, consists of the constructs that are more correlated to the number of defects than they are correlated to the size. There is a strong correlation between the size and the number of appearances, because every statement adds to the size of the component it is part of. This, in its turn, means that constructs that appear in set I_3 can be suspected to be very defect-prone, and they are a natural basis for actions. The fourth set, I_4, consists of constructs that are more correlated to the number of defects than they are with the size.

The underlying motivation of the four proposed prediction filters is informally summarized in Table 1.

Table 1. Motivation underlying the definition of the four prediction filters.

Prediction filter	Motivation
Prediction filter 1	Identifies the construct that contributes the most to the number of defects.
Prediction filter 2	Identifies the construct whose relative occurrence contributes the most to the relative number of defects.
Prediction filter 3	Identifies the constructs that contribute more to the number of defects than they contribute to the size.
Prediction filter 4	Identifies the constructs that contribute more to the number of defects than the size does.

It has not been formally proven that the proposed prediction filters are the best possible filters that can be formulated. The underlying motivation is, however, made explicit to motivate the definitions of the prediction filters proposed.

The prediction system presented here is based on simple direct measures such as size. The size is in many cases a measure that is simple and straightforward to define. It is, of course, possible to include more complicated indirect measures, such as complexity and coupling. If this is done it is, however, important to carefully define how to measure these aspects.

3.4.3. Relation to the Data Understanding step. In the Data Understanding step of the method a basic understanding is obtained about the measures that are collected. The Data Understanding step results in a number of groups (through the factor analysis) of constructs that are related. When constructs are identified through the prediction system, the other constructs that are related to the same factors may also be considered interesting. This is shown in Figure 4.

It would be possible to identify factors from the data understanding step instead of constructs. There are, however, a number of reasons why we do not recommend this:

- In many cases it is not possible to find high level concepts that correspond to every factor. This makes it hard to understand and identify actions based on the factor when it is identified by the prediction system.
- The presented method does not require that a factor analysis be performed. Data understanding can be performed differently.
- All constructs do not have to be included in any factor resulting from a performed factor analysis.
- In some cases a factor analysis can result in only a few factors.

3.5. Recommendation (step 4)

In the previous steps, a number of constructs that are over-represented in the defect data have been identified. The information obtained from the prediction system can be used as an input to the decision process. The results are, however, only recommendations of items that may be worth studying further, and their cost-effectiveness must be investigated before taking the final decision. The recommendation step comprises the traditional work of root cause analysis where the underlying reasons of defects are determined. The identified constructs can be the input to this work, but they may also be combined with interviews and experience, in order to determine which actions to recommend.

If a particular construct is found to result in many defects in comparison to the number of uses of this con-

struct, then it is necessary to investigate and examine the reason for this. It is in most cases not the actual use of the construct that is critical; it is errors made in connection with its use. This means that it is not just to say: -"Do not use construct A". To really understand, it is necessary to seek the cause of the defect based on the construct. The underlying problem may be that a certain construct is used when certain types of logical problems occur, and the reason for the defect is that these types of logical problems are inherently difficult and hence defect-prone.

While the prediction system is applied on data from both the design step and the coding step, it may be possible to determine that the constructs identified from implementation may be a result of the identified design constructs. This must be investigated before making any changes regarding the programming constructs, since a change related to a design construct may change the use of different programming constructs. This kind of effect cannot be identified if the prediction system is only applied on data from one development step. In many cases it is therefore better to apply the prediction system on data from both the design step and the coding step, as described in Section 4.

Another source of problem that can be identified may be transformations when, for example, a specific design construct is transformed into a programming language construct. A transformation may not be straightforward due to poor matching between the design method and the programming language or a transformation does include a too high degree of innovation. Thus, the transformations are the cause of the problem and not the constructs as such. This means that it is essential to use the prediction system with care, i.e. not in a prescriptive way but rather as a guide to understand the sources of software defects.

Based on the result from the prediction, three major types of actions can be taken to improve the situation:

- Use less
  Change the use of the construct to decrease the problems identified. For example, if it has been identified that the use of "goto"-statements results in many defects, then it may be possible to introduce structured programming techniques that reduce the use of unstructured constructs such as "goto".
- Use more correct
  Another possibility is to focus on verification techniques, i.e. based on the identified problems the verification techniques are improved. For example, the inspections can be improved by introducing better checklists in order to reduce the number of defects using the problematic construct.
- Use better
  Finally, the reason for the problems may be that the people do not have adequate education to handle the problematic construct. Thus, the solution may be to provide better education of the personnel.

These three opportunities are always available when the sources of software defects have been identified and better understood. The actual action(s) must be determined based on company internal experiences.

4. Illustration of the prediction method

4.1. Introduction

It must be noted that the objective is not to highlight the specific data used. The data are primarily used for illustration purposes. The data have been collected in retrospect from a telecommunication software project, which developed switching software for a public exchange. The total size of the studied code is approximately 30 KLOC, which is divided into 28 components. A potential problem with a retrospect study is the accuracy of some of the data, since the documents have been interpreted after the project finished by a person who did not participate in the project.

Data have been collected from design documents and from the software code. Defect data from the operational phase have been collected from the maintenance department of the operator of the software. The defect data come from several installations of the software. Since data have been collected both from the design and the code, the application of the method can be illustrated for both design metrics and code metrics.

4.2. Data collection

In the study, 11 design measures and 10 code measures were collected for each component. The 11 design measures consist of 2 different measures of size and 9 measures of occurrences of different statements (constructs c_{d1} - c_{d9}). The 10 code measures consist of one measure of size (length) and 9 measures of occurrences of different statements (constructs c_{c1} - c_{c9}). To go into detail concerning the design measures would lead too far, but briefly it can be stated that they have been collected from a notation resembling SDL [11]. The design measures are summarized in Table 2, but every construct is not fully described. The 9 code measures are summarized in Table 3. The programming language is application specific and company internal.

Table 2. Summary of design measures.

Counted quantity	Remark
Number of symbols	size
Number of pages	size
Tasks, including procedure calls and print commands	construct c_{d1}
Sending of signal	construct c_{d2}
Receiving signal	construct c_{d3}
Choice	construct c_{d4}
State	construct c_{d5}
Out-connector	construct c_{d6}
In-connector	construct c_{d7}
Receiving external signal	construct c_{d8}
Sending external signal	construct c_{d9}

Table 3. Summary of code measures.

Counted quantity	Remark
Lines of code	size
'var' statements (declared variables)	construct c_{c1}
'retrieve' statements (advanced signalling statement)	construct c_{c2}
'enter' statements (receiving a signal)	construct c_{c3}
'send' statements (sending of a signal)	construct c_{c4}
'goto' statements (used in a more structured way as 'break' instructions, for example, breaking a repetition statement)	construct c_{c5}
'branch on' statements (conditional branching)	construct c_{c6}
'to' statements (explicit addressing of signals)	construct c_{c7}
'if' statements	construct c_{c8}
'goto' statements (unstructured goto, which simply jumps to somewhere else in the program)	construct c_{c9}

Data are initially collected for 28 components. When the approach described in Section 3.2 for identification of outliers is used, 8 potential outliers are identified. That is, Box-plots have been drawn for the measure of defects, the measures of size, and the 18 measures according to the constructs. Some components were identified for more than one measure, but some were identified for only one measure. In this example a policy has been taken to remove every component that can be suspected to be an outlier. This is because the study is performed in retrospect and it is impossible to obtain the documents or contact the people that worked with the components. In this example, both the design and code documents were removed when a component was removed, even if the component was identified as an outlier only for one of the documents. Again, this is because the study is performed in retrospect and it is impossible to know if the anomalies are due to the specific document or the component as such.

Therefore, all 8 components are removed. This may seem as many, but it should be remembered that it was enough that a component turned up as an outlier in one out of 20 box plots to be regarded as an outlier. Moreover, the objective is to understand and identify the general behavior. Further, the example is included in order to explain the method, the result is not the most important. In a real case it is, however, important to carefully decide which documents to remove.

4.3. Data understanding

The collected data are analyzed through a factor analysis (see for example [10]). One factor analysis is performed for both the design constructs and for the code constructs. The result (factors with eigenvalue > 1 are used and orthogonal/varimax rotation is applied) is shown in Table 4.

Table 4. Results of the factor analysis for design and code constructs.

construct	F_1	F_2	F_3	F_4	F_5
c_{d1}	0.147	0.096	**0.875**	-0.092	0.318
c_{d2}	**0.906**	0.038	0.313	-0.143	-0.026
c_{d3}	**0.879**	0.153	0.209	-0.103	0.145
c_{d4}	0.404	0.133	**0.856**	-0.066	0.045
c_{d5}	0.127	-0.239	-0.137	**0.876**	-0.041
c_{d6}	-0.011	0.357	0.237	0.193	**0.824**
c_{d7}	0.196	0.176	0.156	-0.052	**0.932**
c_{d8}	**0.927**	0.272	-0.085	0.021	0.106
c_{d9}	**0.92**	0.261	0.098	-0.094	0.121
c_{c1}	**0.774**	0.437	0.131	0.171	0.002
c_{c2}	0.310	-0.257	0.003	-0.520	-0.160
c_{c3}	**0.890**	0.370	0.020	0.130	0.072
c_{c4}	**0.842**	0.104	0.393	-0.119	0.095
c_{c5}	0.274	**0.880**	0.299	0.048	0.068
c_{c6}	0.246	**0.788**	-0.110	-0.079	0.471
c_{c7}	0.211	**0.847**	-0.175	-0.114	0.284
c_{c8}	0.267	**0.904**	0.241	0.038	0.112
c_{c9}	0.274	**0.933**	0.123	0.030	0.107

In the table the large (a threshold value of 0.750 is chosen for this example) loadings, i.e. the correlations between a construct and a factor, are written in bold. Five different factors can be observed:

- F_1 is most related to c_{d2}, c_{d3}, c_{d8}, c_{d9}, c_{c1}, c_{c3}, and c_{c4}. These constructs mainly deal with sending and receiving signals. In addition to the signal constructs the constructs also include the var construct (c_{c1}), but mainly the factor is related to signalling constructs.
- F_2 is most related to c_{c5}, c_{c6}, c_{c7}, c_{c8}, and c_{c9}. These constructs deal with various programming aspects and it is not as easy to describe this factor as it is to describe Factor F_1.
- F_3 is most related to c_{d1} and c_{d4}. These two constructs deal with choices and an additional construct dealing with various constructs such as procedure calls and printing.
- F_4 is most related to c_{d5}. This construct deals with states.
- F_5 is most related to c_{d6} and c_{d7}. These constructs deal with connectors.

4.4. Application of prediction filters

4.4.1. Results from the design step. As a measure of size two different measures have been collected and can be considered:

- Number of pages of design,
- Number of graphical symbols.

Here the first alternative is chosen. The correlation between the number of defects and the size is 0.475. In Figure 5, the correlations needed to determine the interesting sets are plotted.

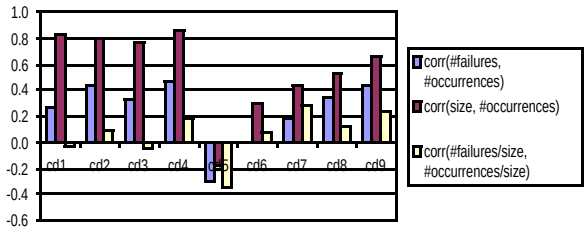

Figure 5. Correlation needed to determine the interesting sets of design constructs.

Based on the correlations, the four sets of interesting constructs can be determined as:

- $I_1 = \{c_{d4}\}$ (the highest of the left bars)
- $I_2 = \{c_{d7}\}$ (the highest of the right bars)
- I_3 is empty (the left bar is lower than the middle bar for all constructs)
- I_4 is empty (the left bar is lower than 0.475 for all constructs)

Two constructs that seem to significantly contribute to the number of defects are c_{d4} and c_{d7}.

4.4.2. Results from the implementation step. The correlation between the number of defects and the size (length) of the components is 0.482. In Figure 6, the correlations needed to determine the sets of interesting constructs are plotted.

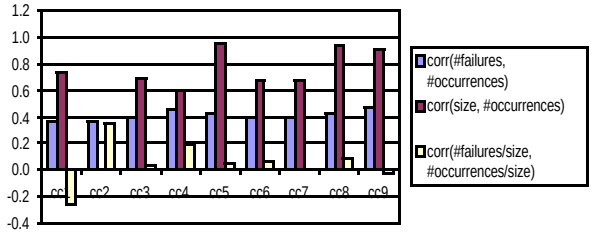

Figure 6. Correlations needed to determine the interesting sets of code constructs.

Based on the correlations, the four sets of interesting constructs can be determined as:

- $I_1 = \{c_{c9}\}$
- $I_2 = \{c_{c2}\}$
- $I_3 = \{c_{c2}\}$
- I_4 is empty

The interesting constructs that are identified from the filters are c_{c2} and c_{c9}. It is interesting to notice that c_{c2} is not included in any factor. As it is discussed in Section 3.4.3, if filtering was done with the factors instead of the constructs no factor would be related to this construct.

4.4.3. Summary. In Section 4.4.1 and Section 4.4.2 the following constructs were identified: c_{d4}, c_{d7}, c_{c2}, and c_{c9}. c_{c2} is not highly related to any of the identified factors, but the other constructs are related to F_3, F_5, and F_2. If all constructs related to these factors are considered interesting, the following constructs have been identified: c_{d1}, c_{d4}, c_{d6}, c_{d7}, c_{c2}, c_{c5}, c_{c6}, c_{c7}, c_{c8}, and c_{c9}. These constructs deal with

- Jumps, i.e., connectors, goto, and branch (c_{d6}, c_{d7}, c_{c5}, c_{c6}, and c_{c9}).
- Choices (c_{d4} and c_{c8}).
- Various additional constructs dealing with, for example, explicit addressing of signals, retrieving signals, and procedure calls. c_{c2} is one of these constructs. This construct is not often used and it does not appear in every component. Lack of experience of it may be a reason to that errors are made when this construct is used.

4.5. Recommendation

The use of the prediction system identifies a number of design and programming constructs. These are mainly con-

cerned with jumps (c_{d6}, c_{d7}, c_{c5}, c_{c6}, and c_{c9}), choices, and various other constructs. Unfortunately, it is not possible to say more since the data were collected in retrospect and primarily used here for illustration purposes, and not really to propose any specific action. It has, however, been shown that it is possible to identify and understand sources of software defects in terms of design and programming constructs.

5. Summary

Software defects are a costly fact. Thus, methods, preferably simple methods, allowing for identification and understanding of potential sources of defects are needed. A simple method for identification of design and programming constructs that contribute more than expected to software defects has been proposed. The use of the method has been illustrated in a study. The objective is that the method should be generally applicable although it has been used here to identify critical design and programming constructs.

The study has illustrated that it is possible to extract valuable information from existing products in order to identify understand sources of software defects. The relationship between the software process and the product is poorly understood, in particular in quantitative terms. Hence, the objective of this work is to improve this situation by supporting the feedback from product measures to decisions and actions related to the software process.

Acknowledgment

This work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), grant 1K1P-97-09673.

References

[1] D N. Card, “Learning from Our Mistakes with Defect Causal Analysis”, *IEEE Software*, pp. 56-63, January-February 1998.

[2] G. Damele, G. Bazzana, F. Andreis, S. Aquilio, S. Arnoldi and E. Pessi, “Process Improvement through Root Cause Analysis”, *Proceedings Third International Conference on Achieving Quality in Software*, pp. 35-47, 1996.

[3] J. Kajihara, G. Amamiya and T. Saya, “Learning from Bugs”, *IEEE Software*, pp. 46-54, September 1993.

[4] I. Bhandari, M. Halliday, E. Tarver, D. Brown, J. Chaar and R. Chillarege, “A Case Study of Software Process Improvement During Development”, *IEEE Transactions on Software Engineering*, Vol. 19, No. 12, pp. 1157-1170, 1993.

[5] U. Fayyad, G. Piatetsky-Shapiro and P. Smyth, “The KDD Process for Extracting Useful Knowledge from Volumes of Data”, *Communications of the ACM*, Vol. 39, No. 11, pp. 27-34, 1996.

[6] N. Fenton and S. L. Pfleeger, *Software Metrics - A Rigorous & Practical Approach*, Second Edition, International Thomson Computer Press, London, UK, 1996.

[7] T. Khoshgoftaar, A. S. Pandya, and D. L. Lanning, “Application of Neural Networks for Predicting Program Faults”, *Annals of Software Engineering*, Vol. 1, No. 1, pp. 141-154, 1995.

[8] M. C. Ohlsson and C. Wohlin, “Identification of Green, Yellow and Red Legacy Components”, *Proceedings International Conference on Software Maintenance*, pp. 6-15, 1998.

[9] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering – An Introduction*, Kluwer Academic Publishers, 1999.

[10] S. K. Kachigan, *Multivariate Statistical Analysis: A Conceptual Introduction*, Radius Press, New York, second edition, 1991.

[11] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specifications*, Prentice Hall, UK, 1991.