M. C. Ohlsson and C. Wohlin, "Identification of Green, Yellow and Red Legacy Components", Proceedings International Conference on Software Maintenance, pp. 6-15, Washington, USA, 1998.

Identification of Green, Yellow and Red Legacy Components

Magnus C. Ohlsson and Claes Wohlin
Department of Communication Systems
Lund University, Box 118
S-221 00 Lund, Sweden
E-mail: (magnuso, claesw)@tts.lth.se

Abstract

Software systems are often getting older than expected, and it is a challenge to try to make sure that they grow old gracefully. This implies that methods are needed to ensure that system components are possible to maintain. In this paper, the need to investigate, classify and study software components is emphasized. A classification method is proposed. It is based on classifying the software components into green, yellow and red components. The classification scheme is complemented with a discussion of suitable models to identify problematic components. The scheme and the models are illustrated in a minor case study to highlight the opportunities. The long term objective of the work is to define methods, models and metrics which are suitable to use in order to identify software components which have to be taken care of through either tailored processes (e.g. additional focus on verification and validation) or reengineering. The case study indicates that the long term objective is realistic and worthwhile.

1. Introduction

As a system evolves and goes through a number of maintenance releases [1], it naturally inherits functionality and characteristics from previous releases [2] and [3], and therefore becomes what we refer to as a legacy system. As new functionality and features are added over and over again, the complexity may increase and impact the maintainability of the system and its components. It is hence important to track the evolution of a system and its components. The objective must be to enable us to identify components which are getting more and more difficult to maintain due to the changes made. We must be able to take actions prior to having a component which is almost impossible to change, or at least it takes a lot of time to update the component due to poor understanding of the component and its functionality.

Experiments have been presented [4] to provide insight into the area [5], and some solutions have been suggested to be able to address these issues [6] and [7]. Models trying to detect and predict fault-prone components have been presented with the objective of identifying the most fault-prone components within a specific project. The models have been created based on the outcome from one project, validated for a second project [8] and finally used in a third project and refined based on the outcome [9] and [10]. Another approach has been to take the outcome from one project and divide the data set into two parts and build the model based on one half and validate it for the other half [11] or build the model in one iteration and test it in the subsequent [12]. This type of models for predicting and classifying fault-prone components is one important input to identify component decay. We use the word decay to denote code which in some way is becoming worse and worse for each software release, and which potentially will be unmaintainable some time in the future.

The objective of this paper is to focus on the evolution of software components. We need models and methods to identify components which are on their way of becoming extremely difficult to change. The identification of these components may serve two major objectives. First, the information can be used to direct the efforts when a new system release should be developed. This could mean applying a more thorough development process or assigning the most experienced developers to these components. Secondly, the information can be used when determining which components need to be reengineered in the long run.

Components which are difficult to maintain are certainly the main candidates for reengineering efforts.

The basic idea is to use the historical data of the components. The data may include fault and failure data, and product and process measures when available. In our approach, we view fault-proneness as being an indication that something has been difficult to change or is going to be difficult to change. The product and process measures may be difficult to relate to change directly, hence we are trying to identify major changes in the components, for example, in terms of major changes in the structure of the components. The hypothesis is that these type of measures are indicators of potential maintenance problems. Thus, although we are interested in fault-proneness, it is with the long-term view that components which are fault-prone will be difficult to maintain in the future.

The data are then used for building models which can be used to identify components which perform worse than the rest. We would like to address code or component decay using models and historical data. Based on the historical data, we would like to create models to classify components as green, yellow or red, depending on the amount of decay. These component categories are further discussed in Section 2.2. We do not believe that it is possible to formulate general rules of what constitutes a green, yellow and red component respectively. The actual limits between the classes must be determined based on the specific situation, including, for example, requirements and system type.

The intention of using historical data is to find trends and react early on warnings. Also, it provides an opportunity to predict and plan the necessary activities. In this paper, we propose some possible models for identification of trends for components, i.e. are they on their way of becoming critical components from a maintenance perspective? The objective is that we should be able to identify the components before they cause any major problems in the projects (or different system releases).

The proposed models are illustrated for a data set containing two system releases. The objective is to show how the models can be used to identify the potentially problematic components.

The paper is organized as follows. In Section 2, we discuss models for identification of decaying software components. Several models are proposed as being candidate models for identification of decaying components. The proposed models are illustrated in a minor case study in Section 3. Finally, some conclusions are presented in Section 4.

2. Models for Identification of Code Decay

2.1. Introduction

The objective of our research is to identify measures and to build models which depict particularly problematic legacy components before they cause any major problems. The intention is to identify relationships between measures and component behaviour, in terms of maintenance difficulty. Based on these relationships, we would like to build prediction models to identify problematic components before the actually become really problematic.

It is important to note that we do not anticipate to derive a stable model. On the contrary, it is important to update the model as new releases are available. Thus, by doing this, the model takes, for example, changes into the development process into account. This implies the following process [10]:

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

Due to the fact that the components are included in a number of releases, this means that the model should be improved after each release, according to the outcome of the projects. Otherwise, the model will become invalid as the prerequisites change. By continuously measure and collect data for improvement of the models for identification of problematic legacy components, we infer the Experience Base, derived from the Experience Factory concept [13]. Based on historical data, we would like to predict and plan for the problematic legacy components and support the development organization during project execution.

2.2. Component Types

To enable identification of the problematic legacy components, we would like to introduce a model for classification of software components based on fault-proneness, which we view as a problem of maintaining the software components. The components are classified according to a colour code, like a traffic light, depending on the amount of decay (see Figure 1). The components should be classified as green, yellow or red. The amount of decay must be judged based on the outcome of previous releases, and the criteria may be number of faults, time to perform certain types of maintenance activities or that the structure of the component is becoming more and more difficult to understand and handle. The colouring scheme should be interpreted as follows:

- Green components (normal evolution)
  Green represents normal evolution and some amount

of fluctuation is normal. These components are easily updated, i.e. new functionality may be added and faults corrected without too much effort. Furthermore, we do not need system experts to maintain the components. The components should be traced from release to release to be able to find trends and when a component exceeds a certain limit (referred to as the lower limit), it becomes yellow.

- Yellow components (code decay)
  As a component exceeds the lower limit, it is classified as yellow, and particular attention has to be paid to this component to avoid future problems. Components in the yellow region are candidates for specific directed actions. These may include launching a more thorough development process or the component is identified as a candidate for reengineering. If the yellow components are not treated properly, the components may exceed the upper limit and become a red component.
- Red components ("mission impossible")
  A red component is difficult and costly to maintain. It is often the driving factor for schedules and cost. In other words, the red components have a tendency to end up on the critical path of a software project. In order to change the components, we need experts, and the components are often viewed as "mission impossible" tasks. The components are no longer candidates for reengineering; they need reengineering.

The classification scheme is illustrated in Figure 1.

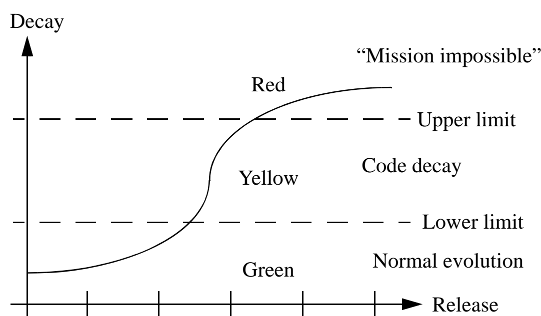

Figure 1. Growing amount of decay for a legacy component.

The lower and upper limit should be determined based on the historical data and continuously updated according to the fact that we improve the quality of the components. It is, of course, not possible to state generally where the limits are located. The limits are governed by our interpretations of green, yellow and red legacy components, and they are dependent on factors which must be determined for each case separately. The interpretation may depend on, for example, company, application domain, system and customer requirements.

It is necessary to be aware of that even if some components indicate a high level of decay, it may depend on the release as a whole, e.g. lack of resources, unsatisfactory process etc. For example, we may have found that components with more than a certain number of faults in testing should be classified as yellow, but when looking at the system as a whole we realize that the total number of faults is very high. This may indicate that the component as such is not the problem, we may have a problem with the release as a whole. Therefore, the whole system must be studied in conjunction with the individual components prior to finally identifying certain components as being of a specific class.

Another advantage of building models for decay is that it provides a focus on the reasons for decay. The reasons for decay are implicitly part of the prediction models, hence providing important input to the design organisation so that they can try to avoid these problems in the future when they add functionality, correct programs or even reengineer them.

2.3. Measures

The key to prediction is the collection of proper measures capturing the aspects which influence the behaviour we want to predict. In this particular case, the prediction means identifying components which are on there way of becoming a red component. Therefore, it is necessary to decide, as early in the model building as possible, what to measure and what these measures should be used for.

According to our previously presented classification model, we need some useful measures to be able to trace the components. These measures should be possible to use to characterize the components, and in particular to describe the level of decay. We must be able to judge if the components are green, yellow or red components.

The measures should, if possible, cover product, process and resource measures. It is important that the measures are related to both components and the system as a whole. The main focus here is on quantitative measures. Some important aspects and measures to cover are (see also [14] and [15]):

- Size measures
  For example: Lines of code or number of if-statements. The if-statements are included as a size measure, since the number of statements contribute to the size of the software. Another possibility is to choose to use it as a structural measure. The actual placing of the different measures is best done after, for example, a principal component analysis, which groups different variables into factors or components.
- Structural measures
  For example: Cyclomatic complexity, amount of modified code or interface characteristics. The amount of modified code may be viewed as a struc-

tural measures since it indicates the amount of structural change in the software.

- Process measures
  For example: Number of occurrences of different events in different phases, or time to perform certain changes. The latter requires that we are able to define some "benchmark" changes.
- Fault data
  For example: Classification of different type of faults.

These measures could be combined to calculate interesting figures and norms. For example, it is possible to combine number of faults with LOC to find the fault density. This figure could be combined with amount of modified code in a component to see how the degree of modification affects the fault density.

Finally, one dimension we have left out is the qualitative measures, the subjective opinions. They can provide different aspects and clarify certain indistinctions and find answers to certain circumstances. The problem is that it is difficult and time consuming to collect this information and is therefore not within our scope.

2.4. Analysis

The basis for the analysis should be made through collection of measures applying a goal-oriented approach, hence we are now able to analyse the data using models which pin-points different types of components. The analysis forms the basis for taking the appropriate decisions regarding the components. There are many different kinds of analysis methods available. The simplest just creates a plot while the more complicated includes multivariate analysis. Within the software engineering area, some of the most used statistical methods are [14] and [16]: plots (box and scatter), correlation (Pearson, Spearman and Kendall), regression (linear, non-linear and multiple), principal component analysis (PCA) and ranking.

These methods are easy to understand and provide different opportunities for analysis. Some of these are often used in combination, for example, a correlation calculation is often done before a linear regression calculation and principal component analysis could be used together with ranking. Different statistical tests could also be applied to determine whether a relationship is statistically significant or not.

The statistical methods are used to build different types of prediction models using the data collected. Some possible models are:

- Faults models
  - Rank the components according to fault content, and study the pattern in order to identify the most fault-prone components over time.
  - Divide the components into different fault classes, and investigate if some components always are among the most fault-prone.
- Structural models
  - Investigation of stability of principal components, where the principal components are derived from different product measures.
- Size models
  - Major changes in size.

In our models we are interested in trends and this could be analysed in several different ways. One possibility is to plot the degree of decay for each release and graphically make a decision (see Figure 2).

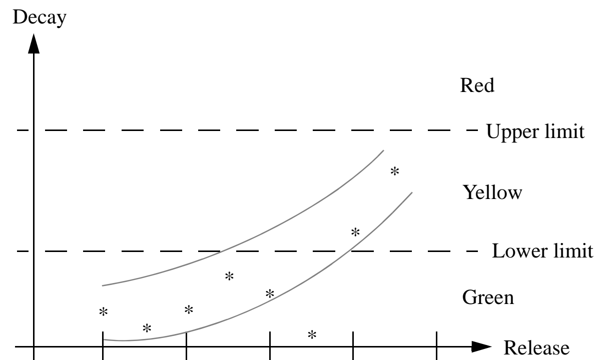

Figure 2. Trend for a number of different releases.

For this type of diagram, it is possible to apply regression analysis and predict the outcome for the following releases.

3. Case Study

3.1. Data Description

This case study is incorporated to illustrate some of the underlying ideas. Unfortunately, the data have been collected in retrospect. This implies that we have been unable to apply a goal-oriented approach for data collection. We have been forced to settle for the data which were available or could be made available. Furthermore, all data have been collected manually, hence restricting the data collection. The actual objective of the data collection was to increase the understanding and evaluate how to continue with data collection and how to use the data for systematic improvement. In particular, the data were primarily collected from design descriptions since the intention was to apply early predictions of quality problems. Code measures are only available for one of the releases. Thus, the data collection has not been targeted at the type of models discussed in this paper.

The study is based on an investigation of 28 software components for two releases of a large real time system in

the telecommunication domain. The system studied is one of the company's main products. The actual releases investigated are considered typical for releases of this particular system, although the choice of system and releases was based on availability.

The system consists of approximately 100 components. 28 of these components had full documentation required for this study, and were therefore included in this study. Other models were typically not updated to the actual implementation and could therefore not be used. The size of the components is between 270 and 1900 lines of code. The programming language used is a company internal language targeted at their specific hardware platform. For each of the components, the number of fault reports from test and operation were counted and several design and code measures were collected. These types of measures are frequently referred to as complexity measures, but primarily they describe different aspects of the software and no measure can really be considered to capture the actual complexity. In total, 19 different measures were collected, with 10 measures originating from the design documentation and 9 measures (including the number of lines of code) being collected from the resulting code. The code measures are however not used in this study, since they are only available for one of the releases. Please note, all measures have been scaled due to confidentiality. This also includes the number of faults reported.

The lack of documentation and information from a large number of the components, of course, implies that the conclusions from this study are difficult to generalize to all components in the system. But, on the other hand, the main objective here is to illustrate the approach for identification of green, yellow and red components, rather than actually using the models in practice. The latter is, of course, an objective in the future.

The design data collected were primarily a count of the number of symbols of different types. The language used during design is a predecessor to SDL, (Specification and Description Language) Figure 17, and it is basically a finite-state-machine with states and signals. A modified version of McCabe's Cyclomatic Complexity, Figure 18 was used as well. The modification was simply due to being able to handle signal sending and reception respectively. The design measures are listed in Table 1.

3.2. Analysis

3.2.1 Faults

The number of faults found in the components are presented in Table and Table . It varies between 0 and 32 faults in release *n* and between 0 and 30 in release *n+1*. In Table , the ranks for the components in release *n+1* could be found. The components are ranked according to the number of faults. The components in Table are already sorted according to their rankings. Ranking of components allows us to make comparisons and to make some simple predictions. Our hypothesis is that the same components always are among the most fault-prone in every release.

Measure	**Description**
SDL-pages	Number of design pages in terms of SDL diagrams.
Tasks	Number of task symbols in the SDL descriptions.
Outputs	Number of output symbols in the SDL descriptions.
Inputs	Number of input symbols in the SDL descriptions.
If	Number of if-boxes in the SDL descriptions.
States	Number of state symbols in the SDL descriptions.
McCabe	This measure is an adaptation of the original proposal by McCabe. The measure is adapted to enable measurement on design and in particular to include the signalling concept used in SDL.
External inputs	Number of unique external inputs to a component.
External outputs	Number of unique external outputs to a component.
Internal signals	Number of unique internal signals within a component.

Table 1. Design measures collected.

Component	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Faults	0	0	0	0	0	0	1	2	2	2	3	4	4	5
Rank	1	1	1	1	1	1	7	8	8	8	11	12	12	14
Component	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**	**AA**	**AB**
Faults	7	7	7	8	9	9	9	9	10	11	15	20	22	32
Rank	15	15	15	18	19	19	19	19	23	24	25	26	27	28

Table 2. Faults in release *n*.

Component	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Faults	8	18	0	0	3	4	7	3	8	0	9	2	12	14
Rank	15	25	1	1	9	11	14	9	15	1	17	8	21	23
Component	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**	**AA**	**AB**
Faults	0	0	10	11	1	0	22	30	23	11	15	6	5	12
Rank	1	1	18	19	7	1	26	28	27	19	24	13	12	21

Table 3. Faults in release *n+1*.

This means that we are be able to pin-point or identify the most fault-prone components in each release according to the ranking of the components. Let us assume that we would like to focus on the most fault-prone quartile.

In this case we should identify seven components from release *n* with the highest ranks and use them as our prediction of the most fault-prone components in release *n+1*. It should be remembered that the prediction of fault-proneness is primarily used to identify components which are difficult to change and hence are on their way of becoming difficult to maintain.

25%		Prediction (Release *n*)	
		Fault-prone	**Normal**
Outcome (Release *n+1*)	**Fault-prone**	3	4
	Normal	4	17

Table 4. Result from identification of the 25 percent most fault-prone components.

The seven components from release *n* should then be compared with the seven components with the highest ranks in release *n+1* to see if they match. Due to the fact that four of the components in Table have the rank 19, we have chosen one of them at random. The result is presented in Table . Three components are correctly identified, but four components are incorrectly depicted as fault-prone.A potentially successfully way of using this model is to define green, yellow and red components as follows:

- Green - not in the fault-prone quartile for the two consecutive releases.
- Yellow - in the fault-prone quartile for one of the two releases.
- Red - in the fault-prone quartile for both releases.

In our study, these criteria would mean that components V, W and Y are classified as red components. If we have more than two releases, we could probably formulate better criteria to determine the colour of the components. With better we mean that we may take more than two releases into account, hence facilitating the ability to establish patterns in the behaviour of the models.

Another approach is to use the accumulated ranks to identify those components that need special attention. It should be noted that since the ranks are on an ordinal scale, addition it is not considered to be a meaningful operation. We have, however, chosen to use it as a model irrespectively of this, since we believe that a component, which is consistently among the components with most faults, is bound to continue to cause problems. Furthermore, it should be remembered that we add ranks for components in one system, hence the components being ranked are the same. We do not add ranks for components from different systems.

In Table 5, we have added the ranks from release *n* and *n+1*. This gives an indication of which of the components that are among those with most faults over both releases.

We may now use the accumulated rank to determine the lower and upper limits (see Figure 1) that differentiate between green, yellow and red components. For example, we may regard a value above 45 as being critical hence judging components with an accumulated rank of 45 or higher as being red components. A yellow component may be those that have a accumulated rank above 35. We may choose any threshold values we want to. It is not possible to provide any general rules. The limits must be determined for every specific case. In other words, the limits may depend on application and the subjective judgements which have provided us with a guideline, for example, the ones used above for illustration.

Another possibility would have been to utilise the difference between the ranks to identify those with large deviations between releases. For example, we may see that a component increases its rank from release to release, and hence it may be a component to watch in the future.

These analyses provide information focusing on the components and not on the system as a whole. Therefore, before making decisions about corrective actions the release as a whole has to be analysed.

3.2.2 Structure

Analysis of structural changes provides an ability to react on major changes in the components. Changes in the components' structure might be indicators of future problems regarding the ability to change the component. Interesting indications may be when the number of principal components increase or decrease, or when some variables switch from one group of variables to another. The basic idea behind using this approach is that a major change in the structure may be a potential source of future problems. It should, however, be remembered that we do not expect that this type of model gives us necessarily the optimal prediction in terms of problematic components. A major structural change may be that we have reengineered the components, hence (hopefully) made it better than before. The objective of the models must be to identify and pin-point the changes relative each other, then an expert must take a closer look at the components being depicted with this type of model or for that matter any other model. We do not expect to replace the expert, but to point the expert in the right direction.

The outcome of the analysis of the variables in our study is presented in Table . We used a normal principal component analysis with a orthotran/varimax transformation. There exist two distinct groups of variables, those related to communication (output, input, external input, external output and internal signal) and those related to size (SDL-page, task and if-statement). These two groups are stable across the releases even though their proportional variance contribution differs slightly. The other two variables are states and McCabe. These two have switched from one group to another. One reason might be that these variables are correlated with both communication and size and, depending on the release as a whole, they are placed in different principal components.

The change in the principal components indicates that some variables have changed differently than others. The indication means that focus has to be set on variables which differ from the common pattern. In this particular case, the variables state and McCabe have switched principal components. Hence, we should study the components which have changed most with regard to these variables. For example, we may choose to study the quartile with the largest increasing change in ranks, from release *n* to *n+1*, with regard to these two variables. Thus, we have eight components depicted with states (two components had the same ranking) and seven components pin-pointed with McCabe.

The total number of components pin-pointed is 14, i.e. only one component is the same for the two variables. This may be too many to study in detail, but the main objective here is to illustrate how the changes in principal components can be used to understand fault-proneness. In other words, you may choose to study only 10% of the components for each variable. In our particular case, the two variables pin-point half of the components which may be too many. Another approach would be to only focus on components, which are pin-pointed by both measures (McCabe and States).

Component	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**
Accumulated Rank	16	26	2	2	10	12	21	17	23	9	28	20	33	37
Component	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**	**AA**	**AB**
Accumulated Rank	16	16	33	37	26	20	45	47	50	43	49	39	39	49

Table 5. Accumulated ranks for the components.

	Release *n*			Release *n+1*	
Variable	**Factor 1**	**Factor 2**		**Factor 1**	**Factor 2**
Outputs	.863	.381		.38	.889
Inputs	.806	.504		.346	.852
External inputs	.962	.218		.402	.833
External outputs	.914	.317		.402	.841
Internal signals	.532	.262		.008	.503
States	.836	.206		.846	.108
McCabe	.481	.86		.626	.656
SDL-pages	.587	.771		.918	.364
Tasks	.128	.974		.891	.324
If	.352	.898		.848	.333
Proportional variance contributions	**.564**	**.436**		**.506**	**.494**

Table 6. Results from analysis of principal components.

If focusing on the components with the largest changes in ranking for McCabe, we would actually identify five out of the seven components with the largest increasing fault-proneness for release *n+1*. A similar study of the state variable identifies three out of the eight components with the largest increasing fault-proneness. It is interesting to note that two of the components identified by the state variable were not identified by the McCabe variable, hence by combining the state and McCabe variables we are able to identify all the seven most fault-prone components.

Thus, changes in principal components may be one fruitful way of identifying fault-prone components. We must, however, learn how to interpret the information we obtain. Is the McCabe measure, when it has changed principal component, a good measure to identify fault-proneness, and in the long run maintenance problems in general? Is it a good approach to only focus on the components depicted by all measures which have changed principal components? One thing is clear and that is that further studies are needed if we want to be able to predict components on their way to become red components.

3.2.3 Size

Information about the degree of modification and information about major changes in size are two important measures in the identification of problematic components. These two are often closely related. When a major change in size occurs, the degree of modification also increases. Therefore, these could be treated together. If the degree of modification is related to the number of faults, it may be possible to identify causes which have increased the number of faults. Thus, it is essential to understand the relations between fault-proneness, degree of modification and long-term maintenance problems.

Even if a component has a remarkably large amount of faults in a release, it does not necessarily mean that it is fault-prone in general. The reason might be that it has a quite large degree of modification in this release. The problems arise when we have a large amount of faults and a small degree of modification. This should be interpreted as even when a small part is changed, the effects are wide spread in the component and cause many faults, due to the complexity of the component.

Information about the degree of modification is not available for the components in these two releases. In our study, the best size measure, is the number of SDL-pages for each component. Unfortunately, we have not been able to find any strong correlation between changes in size and fault density. The size measures have to be further investigated, and in particular the degree of modification must be studied in conjunction with the number of faults.

4. Conclusions

Maintenance of software components often means that the complexity of the components increases. The increase

in complexity means that the components become harder to maintain. The code is decaying. To avoid these problems, preventive actions have to be applied to the affected components. One problem is the identification of components, and in particular the ability to identify the components before they cause major problems. A classification scheme for components has been proposed which divides the components into green, yellow and red categories.

The underlying hypothesis in our work is that fault-proneness and structural changes are indicators of maintenance problems, and that components which are pin-pointed by the proposed models are potential candidates for reengineering or other actions to raise the quality and in particular the maintainability of the components.

In our case study, we have used data from two releases to illustrate the identification of fault-prone components by using historical data. The data consists of a number of different design metrics and the number of faults for each component. By applying different analysis methods, we have shown that it is possible to classify components as green, yellow or red depending on the outcome of these analyses. Simple, but yet useful, methods as ranking of components based on different criteria, have provided satisfying results. For example, the accumulated ranks provided important information regarding the components in the long run.

Principal component analysis has also shown potential to be useful for finding structural changes that might have affected the number of faults. In our case, two variables shifted components. Further investigation indicated that large changes in complexity affected the number of faults in a component. This kind of analysis is useful to be able to pin-point components which may need special attention.

The case study only used data from two releases. To further investigate the usefulness of these models and to be able to see long term trends, it is necessary to use data from more releases. The presented case study is a pre-study to a larger study based on component data for several releases of a large software system.

The case study has provided valuable insight into some of the problems and it will form the basis for the continued studies.

5. Acknowledgement

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

6. References

1. D. Gefen and S. L. Schneberger, "The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications", Proceedings of the International Conference on Software Maintenance, ICSM'96, pp. 134-141, November 1996, Monterey, California.

2. N. H. Minsky, "Controlling the Evolution of Large Scale Software Systems", Proceedings of the International Conference on Software Maintenance, ICSM'85, pp. 50-58, November 1985, Washington, USA.

3. M. M. Lehman and L. A. Belady, "Program Evolution: Processes of Software Change", Academic Press, Austin, 1985.

4. A. Karr, A. Porter and L. Votta, "A Testbed for Understanding Code Decay", Progress Report from The International Software Engineering Research Network meeting, ISERN'96, August 1996, Sydney, Australia.

5. D. A. Gustavsson, A. Melton and C. S. Hsieh, "An Analysis of Software Changes During Maintenance and Enhancement", Proceedings of the International Conference on Software Maintenance, ICSM'85, pp. 92- 95, November 1985, Washington, USA.

6. V. Basili, L. Briand, S. Condon, Y. M. Kim, W. L. Melo and J. D. Vallet, "Understanding and Predicting the Process of Software Maintenance Releases", Proceedings of the 18th International Conference on Software Engineering, ICSE'96, pp. 464-474, March 1996, Berlin, Germany.

7. D. L. Parnas, "Software Aging", Proceedings of the 16th International Conference on Software Engineering, ICSE'94, pp. 279-287, May 1994, Sorrento, Italy.

8. T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi and J. McMullan, "Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System", Proceedings of the International Symposium on Software Reliability Engineering, ISSRE'96, pp. 364-371, October-November 1996, White Plains, New York, USA.

9. N. Ohlsson and H. Alberg, "Predicting Fault-prone Software Modules in Telephone Switches", IEEE Transactions on Software Engineering, 22(12), pp. 886-894, December 1996.

10. N. Ohlsson, M. Helander and C. Wohlin, "Quality Improvement by Identification of Fault-prone Modules Using Software Design Metrics", Proceedings of the 6th International Conference on Software Quality, ICSQ'96, pp. 1-13, 1996, Ottawa, Canada.

11. T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, N. Goel, J. P. Hudepohl and J. Mayrand, "Detection of Fault-Prone Program Modules in a Very Large Telecommunications System", Proceedings of the International Symposium on Software Reliability Engineering, ISSRE'95, pp. 24-33, October 1995, Toulouse, France.

12. T. M. Khoshgoftaar, E. B. Allen, R. Halstead and G. P. Trio, "Detection of Fault-prone Software Modules During a Spiral Life Cycle", Proceedings of the International Conference on Software Maintenance, ICSM'96, pp. 69-76, November 1996, Monterey, California.

13. V. Basili, G. Caldiera and D. Rombach, "Experience Factory", in Encyclopedia of Software Engineering, Vol. 1, edited by J.J. Marciniak, pp. 469-476, John Wiley & Sons, New York, 1994.

14. N. Fenton, and S. L. Pfleeger, "Software Metrics: A Rigorous & Practical Approach", 2nd edition, International Thomson Computer Press, Cambridge, 1996.

15. T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", Proceedings of the International Conference on Software Maintenance, ICSM'95, pp. 295-303, October 1995, Opio, France.

16. T. M. Khoshgoftaar and D. L. Lanning, "Are the Principal Components of Software Complexity Stable Across Software Products?", Proceedings of the International Symposium on Software Metrics, Metrics'94, pp. 61-72. October 1994, London, United Kingdom.

17. ITU, "Recommendation Z.100: SDL - Specification and Description Language", 1988.

18. T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, 4(2), pp. 308-320, 1976.