M. C. Ohlsson, A. von Mayrhauser, B. McGuire and C. Wohlin, "Code Decay Analysis of Legacy Software through Successive Releases", Proceedings IEEE Aerospace Conference, 1999, Snowmass, Colorado, USA, 1999.

Published in the Proceedings of the IEEE Aerospace Conference, March 1999, Snowmass, Colorado,

Code Decay Analysis of Legacy Software through Successive Releases

Magnus C. Ohlsson, Anneliese von Mayrhauser,
Brian McGuire and Claes Wohlin

Abstract

Prediction of problematic software components is an important activity today for many organisations as they manage their legacy systems and the maintenance problems they cause. This means that there is a need for methods and models to identify troublesome components. We apply a model for classification of software components as green, yellow and red according to the number of times they required corrective maintenance over successive releases. Further, we apply a principal component and box plot analysis to investigate the causes for the code decay and try to characterise the releases. The case study includes eight releases and 130 software components. The outcome indicates a large number of healthy components as well as a small set of troublesome components requiring extensive repair repeatedly. The analysis characterises the releases and indicates that it is the relationship between components that causes many of the problems.

1. Introduction

As a system evolves over a series of releases, it is important to know which software components are stable versus those which show repeated need for corrective maintenance, and thus possible decay. Code has decayed when it becomes worse and worse in each successive release, and potentially unmaintainable some time in the future.

As new functionality and features are added over time, complexity may increase and impact the maintainability of the system and its software components. Defects may also be injected as a result of bug fixes. It is hence important to track the evolution of a system and its modules. We propose a method to identify software components, which are getting more and more difficult to maintain. This makes it possible to take action prior to having a component decay too much.

Identifying troublesome components versus healthy ones over successive releases has several uses. First, the information can be used to direct efforts when a new system release is developed. This could mean applying a more thorough development process or assigning the most experienced developers to the troublesome components. Second, this information can be used to tailor testing to parts of the system that are fragile, but also to assess the overall quality of the software. Third, 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 candidates for reengineering efforts and therefore 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 method uses fault and failure data, as well as product measures to identify troublesome and healthy components. The basic idea is to use maintenance history of the components for assessment. We view fault-proneness as an indication that something has been difficult to change or is going to be difficult to change.

Measures of maintenance activity also relate to fault-proneness: the number of revisions between two releases and the number of components affected for each repair or update activity. If a component needs many fixes between releases, there has been code churn. The software was hard to debug and "get right" and if many components were involved in a fix, the problem had a large impact.

The hypothesis is that these types 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.

Based on historical data, we classify components as green, yellow or red, depending on the amount of decay. Thresholds between green, yellow, and red components are developed based on an organisation's specific situation. 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 green, yellow and red 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 to warnings. Trends provide an opportunity to predict and plan focused maintenance activities. For this study, we propose models for identification of trends for components, i.e. whether they are on their way to becoming critical components from a maintenance perspective. The objective is to identify the components before they cause major problems.

In Section 2 we describe the models we are using and we discuss causes for code decay. Section 3 describes the environment and the collected data and how this data relates to code decay. The analysis of the collected data is presented in Section 4. Finally Section 5 presents the conclusions.

2. Models for Identification of Code Decay

2.1 Introduction

The objective of our research is to identify measures and to build models which identify particularly problematic legacy components before they cause major problems. The intention is to identify relationships between measures and component behaviour in terms of maintenance difficulty. Based on these relationships, we propose to build models to identify problematic components before they actually become problematic. The models consider and adapt to new releases. We use the following model development process [1]:

- 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.

Through measurement and data collection over releases, the models for identification of problematic legacy components provide an Experience Base, derived from the Experience Factory concept [2]. Based on historical data, we predict problematic legacy components. Knowing which parts of a system might need improvement makes planning and managing for the next release easier and more predictable.

Models trying to track [3] or to detect and predict fault-prone components have been presented [4][5] 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 [6] and finally used in a third project and refined based on the outcome [1][7]. 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 [8] or to build the model in one iteration (build, validate and use) and test it in the subsequent iteration [9]. These models for predicting and classifying fault-prone components provide important input to identify component decay [10][11].

2.2 Component Types

To enable identification of problematic legacy components, we use a model for classification of software components based on fault-proneness, which we view as a problem of maintaining software components. The components are classified according to a colour code, like a traffic light, depending on the amount of decay. The components should be classified as green, yellow or red.

Visualisation plays an important part in the interpretation of analysis results or complex relationships [12][13][14]. Even though our classification is a very simple method for visualising problematic spots, it helps understanding and many people can relate to these colour codes intuitively.

The amount of decay is judged based on the outcome of previous releases. The outcome criteria may be the number of faults, time to perform certain types of maintenance activities or the complexity of the component. Any of these attributes could make software increasingly difficult to understand and handle. The colouring scheme should be interpreted as follows [15]:

- 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.
- Yellow components (code decay) - As a component exceeds a 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.
- Red components (“mission impossible”) - A red component is difficult and costly to maintain. It is often the driving factor for maintenance schedules and cost. The red components have a tendency to appear on the critical path of a software project.

Ultimately, they will need reengineering. Yellow components, if allowed to decay further, may pass the upper limit and end up as red.

Setting thresholds between, green, yellow, and red components needs careful consideration. The lower and upper limit should be determined based on historical data and updated when the quality of the components improves. Actual threshold values depend on, for example, company, application domain, system and customer requirements. Also, when many components indicate a high level of decay, this may indicate problems with the release as a whole, e.g. lack of resources, unsatisfactory process etc. [16].

2.3 Relationships

Maintenance problems may be related to strong coupling between components or to problems stemming from software architecture and its decay over time [17][18]. If the system was not designed properly initially, this will lead to problems later, especially when adding new functionality. The design also has to be updated, especially when new features are added. Not doing so can cause serious system decay.

If software components communicate with many other components, they are critical to the system in the following sense: a decaying component of this type is likely to have a large impact on the remainder of the software. A change in such a component likely affects other parts of the software, requiring multiple changes elsewhere. When the component decays, it is likely that each fix requires multiple corrections to “get it right” (because it may become increasingly difficult to assess the impact of each change made on the rest of the system). Thus, as a symptom of decay related to structural problems, we might see more code changes for components that interact with many other components.

Also, components that have a central position, for example, a shared data structure might show more corrective activities according to its coupling to other components as it decays. Even within components there might exist relationships and encapsulations that are misused in one way or another.

2.3.1 Size

Even though a component is large, it is not necessarily complex. The problem is that it might be difficult to have a good grasp of what it contains. A good understanding of the system is achieved by adding useful comments and by documenting relationships between components. For example, when only a few people understand the system and move to another organisation or department, many problems arise, because nobody has an understanding of the system as whole [19].

2.3.2 Measures

For a useful classification model, we need appropriate measures to trace the components. The measures should, if possible, cover product, process and resource measures. It is important that the measures are related to components and to the system as a whole. The focus is on quantitative measures. Possible aspects and measures are (see also [20][21][22][23]):

- Size measures – lines of code or other suitable measures of size like function points.
- Structural measures – relative and cyclomatic complexity, amount of modified code or interface characteristics. The amount of modified code may be viewed as a structural measure, because if changes are large the structure of the software will change.
- Relational measures – describing the interfaces between components, e.g. number of components that share the same data structure.
- Process measures – 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 – classification of different types of faults, e.g. using a fault taxonomy [24].

3. Case Study

3.1 Environment

This study is based on data from a large software/hardware vendor. The data we used for our case study comes from a large embedded mass storage system which includes about 800,000 lines of C code and 130 software components. Each component contains a number of files. We have studied the system over eight releases, though five of the releases were closely related sub-releases and therefore treated as one release. The releases were predominantly corrective releases.

The data is based on defect/fix reports or source change notices (SCN). This means that every report indicates a problem that had to be corrected. From the defect/fix reports a number of measures were collected for the components. The measures we have used are based on the information contained in the defect/fix reports. This could be argued but our concern has been to not increase the burden for the developers and instead to use existing data to calculate our measures. Part of the objective in this case study is thus to evaluate how useful currently existing data is for identifying problematic components. In the analysis of the results, we will also make recommendations on what data might be more useful.

In total we collected 28 different measures. Twelve of those are size and amount/type of change measures based on LOC. The others describe, for example, in how many defect reports multiple components were changed.

Since we cannot measure the quality of the architecture or measure the interfaces between components directly, we built the model based on which of the components were affected by some defect, i.e. impact on other components. This also provides insight into which of the components are the most critical.

It could also be argued that many of the measures should be normalised to the size of the components. The reason we have not done so is that we are interested in where it is necessary to spend effort. For this, normalisation is not desirable.

3.2 Description of Measures

3.2.1 Impact Measures

These measures are indications 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 indicating a fault-prone and 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 developers affect this measure in terms of poor programming habits. We do not consider this a problem since this also likely causes code decay and thus should be flagged.

- Sys_impact_c = Total number of changes to .c files in a release.
- Sys_impact_h = Total number of changes to .h files in a release.
- Sys_impact = Sys_impact_c + Sys_impact_h

3.2.2 Type of Impact

Here we distinguish between impact on .c files and impact on .h files. Changes to .h files imply that defects occurred in data shared between multiple files. The measures identify the proportion of such changes with data versus function impact. A single .h change impacts multiple .c files. As a result, the changes to .h files should be more indicative of code decay. By ordering the components by those numbers, an indication of which components rely more on shared data structures, will be given.

- Sys_impact_c% = Sys_impact_c/Sys_impact
- Sys_impact_h% = Sys_impact_h/Sys_impact

3.2.3 Changed Components

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

3.2.4 Effort

These measures provide information about the average number of files (.c and .h) that required changes for a single defect fix. This should be related to the amount of effort each defect/fix report required.

Having to change multiple files to fix a single defect indicates a component with strong dependencies among its files. The purpose of separate files is to minimise such dependencies, and their presence indicates that files have been altered time and again in ways that conflict with the initial design. Strong interconnection of files related to defect fixes is an indication of code decay because the complexity of the interconnections between files within the component has grown too much.

As before, the measures distinguish between .c and .h files. We also provide a combined measure as an overall effort surrogate. SCN_comp measures the number of defect fix reports for a component in a given release.

- SCN_effort_c = Sys_impact_c/SCN_comp
- SCN_effort_h = Sys_impact_h/SCN_comp
- SCN_effort = Sys_impact/SCN_comp

3.2.5 Unique Files

These measures count how many of the components' files were touched in a release. It is an indication of how widespread the corrective actions within a component have been or how much the component was broken and needed to be fixed. As before, the measures distinguish between .c and .h files. We also provide a combined measure for an overall assessment of how much of a component needed fixing.

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

3.2.6 Average Number of Changes

This measure indicates how fault-prone and complex a component is. It returns 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 within all components. The measure gives each component a number which increases as the component requires more changes for defect corrections. This measure could be an indication of decay resulting from increased complexity of particular files or breakdown of encapsulation between files, i.e. internal coupling in components between files 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. Therefore more change was needed. The action of repeatedly changing a file shows 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 a sign of code decay.

- Avg_fix_c = Sys_impact_c/Unique _c
- Avg_fix_h = Sys_impact_h/Unique_h
- Avg_fix = Sys_impact/Unique

3.2.7 Size

This measure provides information about how much the components (their .c and .h files) have changed. It measures also where the largest amount of change has occurred, in .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 amounts in these measures are a sign of potential decay.

- Size_add_c = Number of LOC added accumulated for .c files.
- Size_add_h = Number of LOC added accumulated for .h files.
- Size_add = Number of LOC added accumulated for a component.
- Size_del_c = Number of LOC deleted accumulated for .c files.
- Size_del_h = Number of LOC deleted accumulated for .h files.
- Size_del = Number of LOC deleted accumulated for the component.
- Size_exec_add_c = Number of executable LOC added accumulated for .c files.
- Size_exec_add_h = Number of executable LOC added accumulated for .h files.
- Size_exec_add = Number of executable LOC added for a component.
- Size_exec_del_c = Number of executable LOC deleted accumulated for .c files.
- Size_exec_del_h = Number of executable LOC deleted accumulated for .h files.
- Size_exec_del = Number of executable LOC deleted for a component.

3.2.8 Coupling

This measure will associate a number within each component that denotes how often it was involved in defects that required corrections that extended beyond the current component. The higher the number is, the more the encapsulation between the component and the components it interacts with has broken down. Such strong dependencies are indications of relationship decay between the components or very poor design from the start.

In the absence of actual coupling measures associated with corrective maintenance reports, we need 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 when components come to rely too much on the particulars of the implementation of other components because it describes a break in the encapsulation components are supposed to have with respect to each other. Often these 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 exist in a single component but its fix has ramifications in the way other components can 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 all changed components.

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

3.3 Description of Releases

The releases differ from each other in terms of defect fixes and their purpose. Release n was a fairly large release with approximately twice as many changes in .c files as .h files. The amount of LOC added and deleted was significantly larger for the .c fixes. Release *n+1* had almost the same characteristics but with a larger amount of changes in the .h files.

Release *n+2* was a small release with few defects reported. Approximately twice as many .c files as .h files were changed but the number of defects was smaller (about a third) compared to the other releases. Finally, release *n+3* was a very large release but very .c intensive in its defect files. The changed LOC (added plus deleted) for the .c and the .h files are approximately seven times larger for the .c files than for the .h files.

4. Analysis

4.1 Code Churn

Our analysis proceeds in two steps. First we identify the fault-prone components in each release. The measure we use for this is the number of defect fix reports for a component, SCN_comp. This is what we refer to as code churn. As threshold we chose the top 25 percent. Thus all components whose number of defect fix reports is in the upper quartile are considered fault-prone.

Tables 1-4 show the number of defect fix reports for the 10 most fault-prone components and their ranks in the release. The 10 least fault-prone components all had zero defect fixes.

Problematic	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Reports	8	9	10	11	11	11	13	15	15	24
Rank	120	122	123	124	124	124	127	128	128	130

TABLE 1. Defect reports in release *n*.

Problematic	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Reports	10	12	12	12	13	14	17	19	22	37
Rank	116	122	122	122	125	126	127	128	129	130

TABLE 2. Defect reports in release *n+1*.

Problematic	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Reports	8	8	9	9	9	11	13	14	15	15
Rank	121	121	123	123	123	126	127	128	129	129

TABLE 3. Defect reports in release *n+2*.

Problematic	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Reports	12	13	13	18	20	22	23	26	29	43
Rank	121	122	122	124	125	126	127	128	129	130

TABLE 4. Defect reports in release *n+3*.

In step two we analyse how components change status according to this classification. Components which are fault-prone in two successive releases are considered red, those that are not fault-prone (normal) in either release are considered green. Components that change status (from fault-prone to normal or from normal to fault-prone) are classified as yellow.

Tables 5-7 show the results for all four releases. Table 5 shows that 20 components stayed fault-prone (red) from release *n* to *n+1*, 90 stayed normal (green) and 11 changed from normal to fault-prone (yellow). Nine components improved from fault-prone to normal (yellow).

Going from release *n+1* to *n+2* (Table 6), only ten components stayed fault-prone, 84 remained normal. A larger number both improved to normal (21 versus 9) and decayed to fault-prone (15 versus 11). This trend continues from release *n+2* to *n+3* (Table 7). More components become fault-prone (20 versus 15), fewer improved from fault-prone to normal (16 versus 21). Thus there is an indication of decay, code is becoming more difficult to improve.

25%		Prediction (Release *n*)	
		Fault-prone	Normal
Outcome (Release *n+1*)	Fault-prone	20	11
	Normal	9	90

TABLE 5. Result from release *n* to *n+1*.

25%		Prediction (Release *n+1*)	
		Fault-prone	Normal
Outcome (Release *n+2*)	Fault-prone	10	15
	Normal	21	84

TABLE 6. Result from release *n+1* to *n+2*.

25%		Prediction (Release *n+2*)	
		Fault-prone	Normal
Outcome (Release *n+3*)	Fault-prone	9	20
	Normal	16	85

TABLE 7. Result from release *n+2* to *n+3*.

Times fault-prone	**0**	**1**	**2**	**3**	**4**
Number of components	68	29	17	13	3

TABLE 8. Times components were pinpointed as fault-prone.

On the positive side, results show a good number of components that are stable between successive releases (classified as green, lower right corner of table) and a few components that always are pinpointed as fault-prone between releases (classified as red, upper left corner of table). The ones that are classified as yellow are those components that should be further investigated to find out what the reasons have been for the shift in classification, i.e. why they became fault-prone or what activities improved them to “green” status.

One reason why some of the components were classified as yellow is that corrective maintenance for a specific release focused on some specific parts of the system. For example, in release n the focus was on part A of the system, release *n+1* fixed part B and finally in release *n+2*, part A was again the focus of corrective maintenance.

If a component is enhanced during one specific release, this might be a reason for a component to change state. Also, it might be that it is affected because of problems in other components and is therefore reported as changed. The latter is an architectural problem. If a component has been pinpointed as fault-prone it might have been corrected properly and is therefore not subject to being fault-prone any longer. The problematic components (from a classification view) are those that are classified as green and then show some indication of problems and then are classified as green again. For those components it is important to use historical data and find some threshold to be able to pinpoint them.

Table 8 shows to which degree components were repeatedly fault-prone. This shows that 68 components were healthy throughout all releases while 3 components where fault-prone in all four releases. Obviously, these components bear investigation further. They also require immediate attention.

Overall, the software we investigated is quite healthy, although it shows some problems that need to be looked at. Our results indicate:

- A large number of healthy components in the system.
- A small identifiable set of problematic components. Most of these include shared data structures The small set of red components also show that this analysis has the potential for focusing on maintenance and thus make it easier to plan for and do it less expensively.

- A limited number of components that changed state, indicating high predictability of red versus green components. In order to assess them further, we will try to identify what characteristics cause a fault-prone component to become fault-free (lower left corner of Tables 5-7) and what causes a fault-free component to become fault-prone (upper right corner of Tables 5-7).

4.2 Structure

To investigate the characteristics of green, yellow and red components across releases, we have applied a principal component analysis (PCA). This analysis method groups a number of correlated variables into a number of factors [25].

Changes in the number of factors and variables changing factors are indicators that the system is not stable. In our case the grouping of the factors are descriptive of the release, i.e. on which parts of the system were the maintenance activities focused and how do these system parts relate to each other.

We used a standard principal component analysis with a orthotran/varimax transformation to extract as high differences between the variables as possible [26]. We stopped extracting factors when 75 percent of the variance of the variables were explained. This reduces the number of factors to those that explain 75 percent of the variance in the data and excludes those that only have small impact.

We performed factor analysis on two groups of software components:

- Healthy components - components that stayed or became normal in two successive releases (second row in Tables 5-7).
- Problematic components - components that stayed or became fault-prone in two successive releases (first row in Tables 5-7).

Table 9 shows the results of this factor analysis for the healthy components and Table 10 for the problematic components. In the tables the dark gray correlation values identify the variables that contribute most to the factors identified by the column. The light gray values have a correlation less than 0.5 or have a negative value and the white values do not contribute to a factor.

4.3 Healthy Components

The principal component analysis for the healthy components shows two distinct groups of factors in release *n+1* and *n+3*, one related to the .c group and one related to the .h group. Though there exist some minor differences, most of the variables have very small correlations. The summary data of the .c and .h measures are grouped with the .c files which means that most of the changes made to the components involved .c files and were simple code corrections. Even though the Size_add_h measure is related to the .c group the difference between the factors is very small and the reason why it belongs to the .c group is that the code added to the .h files was slightly more than the other measures. Avg_fix_c and Avg_fix were also grouped with the .c factor because Avg_fix_c dominates Avg_fix.

Release *n+2* is very different. All variables except one are grouped into one main factor. The relative variance contribution is also very high for this main factor. This is because release *n+2* reported the smallest number of defects and most of the healthy components were not changed at all.

By contrast, PCA results for release *n+3* are similar to release *n+1*. Measures that are grouped with different factors are Avg_fix_c, Size_add_h, Avg_fix and Multi_rel. Three of those four show low factor correlation values for one of the two releases, only Size_add_h points to a significant difference between the two releases. This indicates more emphasis on adding shared data to .h files, increasing complexity of shared data. With fewer improved software components, this could indicate reasons for beginning decay, especially if problematic software components show the same classification of this factor.

Release	**n+1**		**n+2**		**n+3**	
Variable	**Factor 1**	**Factor 2**	**Factor 1**	**Factor 2**	**Factor 1**	**Factor 2**
Sys_impact_c	0.788	-0.116	0.942	0.158	0.909	0.24
Sys_impact_c%	0.472	-0.441	0.547	0.705	0.455	0.298
SCN_effort_c	0.835	0.018	0.942	0.158	0.834	0.306
Unique_c	0.812	-0.05	0.942	0.158	0.922	0.186
Avg_fix_c	0.49	-0.24	0.692	0.634	0.36	0.525
Size_add_c	0.875	0.19	0.815	-0.48	0.929	0.112
Size_del_c	0.895	0.201	0.758	0.286	0.923	0.238
Size_exec_add_c	0.887	0.163	0.932	-0.46	0.92	0.21
Size_exec_del_c	0.918	0.152	0.862	0.167	0.887	0.321
Sys_impact_h	0.212	0.926	0.964	-0.166	0.322	0.873
Sys_impact_h%	-0.052	0.692	0.798	0.059	0.083	0.879
SCN_effort_h	0.591	0.71	0.964	-0.166	0.218	0.791
Unique_h	0.218	0.915	0.964	-0.166	0.401	0.796
Avg_fix_h	0.167	0.609	0.876	0.055	0.349	0.85
Size_add_h	0.678	0.591	0.871	-0.424	0.076	0.797
Size_del_h	-0.041	0.812	0.951	-0.256	0.276	0.793
Size_exec_add_h	0.595	0.698	0.871	-0.416	0.118	0.845
Size_exec_del_h	-0.041	0.837	0.944	-0.316	0.262	0.866
Sys_impact	0.715	0.443	0.97	0.05	0.862	0.406
SCN_effort	0.848	0.29	0.97	0.05	0.753	0.484
Unique	0.699	0.537	0.97	0.05	0.89	0.333
Avg_fix	0.441	-0.138	0.692	0.634	0.327	0.598
Size_add	0.867	0.243	0.817	-0.479	0.92	0.167
Size_del	0.887	0.277	0.827	0.211	0.9	0.305
Size_exec_add	0.877	0.256	0.834	-0.458	0.898	0.28
Size_exec_del	0.891	0.311	0.914	0.075	0.853	0.406
SCN_comp	0.395	0.051	0.692	0.634	0.506	0.62
Multi_rel	0.316	0.111	0.722	0.425	0.361	0.44
Proportional var. contributions	**0.651**	**0.349**	**0.852**	**0.148**	**0.585**	**0.415**

TABLE 9. Healthy components.

Release	**n+1**		**n+2**		**n+3**	
Variable	**Factor 1**	**Factor 2**	**Factor 1**	**Factor 2**	**Factor 1**	**Factor 2**
Sys_impact_c	0.855	-0.151	0.964	-0.154	0.852	-0.324
Sys_impact_c%	0.294	-0.738	0.241	-0.795	0.223	-0.828
SCN_effort_c	0.803	-0.26	0.875	-0.388	0.527	-0.656
Unique_c	0.84	-0.192	0.812	-0.228	0.775	-0.356
Avg_fix_c	0.392	-0.572	0.735	-0.527	0.479	-0.691
Size_add_c	0.888	-0.012	0.961	-0.115	0.92	-0.117
Size_del_c	0.806	-0.0005	0.968	-0.165	0.945	-0.128
Size_exec_add_c	0.915	0.004	0.962	-0.127	0.924	-0.109
Size_exec_del_c	0.863	-0.023	0.971	-0.163	0.946	-0.134
Sys_impact_h	-0.049	0.99	0.088	0.945	0.011	0.949
Sys_impact_h%	-0.294	0.738	-0.241	0.795	-0.223	0.828
SCN_effort_h	0.069	0.848	-0.009	0.91	-0.116	0.775
Unique_h	-0.154	0.978	-0.018	0.824	0.016	0.934
Avg_fix_h	0.632	0.344	0.323	0.523	-0.15	0.536
Size_add_h	-0.036	0.938	0.077	0.904	0.036	0.765
Size_del_h	0.062	0.921	0.039	0.958	0.071	0.867
Size_exec_add_h	-0.049	0.944	0.101	0.926	0.06	0.908
Size_exec_del_h	0.088	0.898	0.032	0.956	0.046	0.924
Sys_impact	0.586	0.704	0.928	0.271	0.86	0.23
SCN_effort	0.803	0.167	0.859	0.234	0.263	0.471
Unique	0.32	0.888	0.627	0.453	0.538	0.696
Avg_fix	0.409	-0.164	0.805	0.186	0.493	-0.5
Size_add	0.648	0.693	0.963	0.083	0.927	0.036
Size_del	0.667	0.626	0.958	0.226	0.952	0.136
Size_exec_add	0.628	0.731	0.967	0.078	0.936	0.04
Size_exec_del	0.604	0.719	0.938	0.29	0.961	0.038
SCN_comp	0.025	0.909	0.508	0.445	0.712	0.057
Multi_rel	-0.175	0.899	-0.011	0.666	0.166	0.173
Proportional variance contributions	**0.41**	**0.59**	**0.594**	**0.406**	**0.539**	**0.461**

TABLE 10. Problematic components.

4.4 Problematic Components

Like the healthy components, PCA results for the problematic components show two factors, one is more related to .c measures, and the other to .h measures. The differences show in the factor values for the variables and in how the variables are grouped into factors. One difference is that we have several negative correlations between .c measures and the .h factor. One reason is that these variables (for example Sys_impact_c% and Avg_fix_c in release *n+3*) are almost constant because many of the components exclusively contain .c files. Therefore these variables are negatively correlated to the .h factor. We have to be aware that this affects some of the summarised variables if they depend heavily on the variables with negative correlation. Also, if we had not stopped extracting factors in the PCA at 75 percent these variables would probably have been grouped in a third factor.

In release *n+1* and *n+2* we can see that the relationship measure (Multi_rel) is related to the .h files. In release *n+3* the relationship measure does not contribute, because the release had a very high degree of changes in the .c files. We can also see that even though the summarised measures change from the .h group to the .c group, the measure describing relationships between components (coupling) still are related to the .h factor (although the correlation is small).

Finally, this analysis gives us a picture of the characteristics of the releases and highlights the problematic areas, for example, relationships between components and the change in amount of LOC changed in .c files, that need further investigation. It is difficult to say anything about trends for the components when only four releases are available and the characteristics are so different.

4.5 Box Plot Analysis

The PCA results pointed to differences between factor grouping of variables between releases and between the healthy and problematic components. We now investigate these differences further by comparing actual measurements of these variables. Figures 1-6 show box plots of these variables. A box plot shows the second and third quartiles (Q2 and Q3) of values as a box and the range of values, the lower and upper quartiles (Q1 and Q4), as vertical lines.

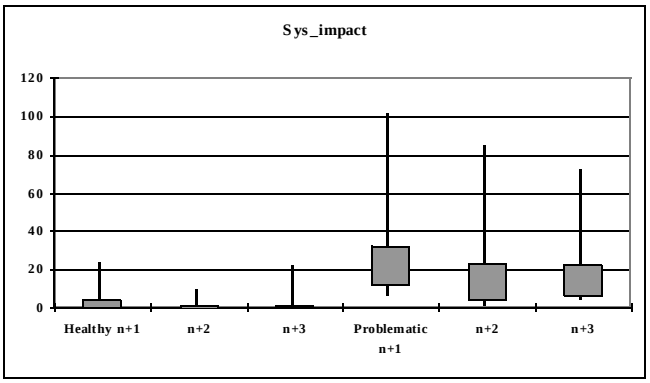

FIGURE 1. Number of changed files.

Figure 1 shows number of files that were changed in each release (Sys_impact). The healthy components behave different from the problematic ones. Release *n+1* was a large release and therefore there are slightly more files changed among the healthy ones. The number of files changed in the problematic components are slightly decreas-

ing in range in successive releases, the decrease is even less between the lower and upper quartile (the box). Problematic components clearly show more system impact.

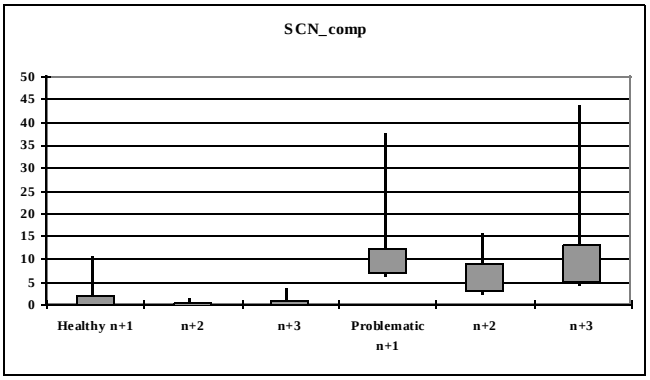

FIGURE 2. Number of defect reports.

Looking at the number of defect reports in Figure 2 for each release indicates an increasing trend for problematic components. As mentioned earlier, release *n+2* was a small release compared to the others and therefore the values for this release are small. Noticeable is the size of the box for the problematic components. For release *n+2* the box is even larger than *n+1*. Release *n+3* has the largest one. This is an indication of decay.

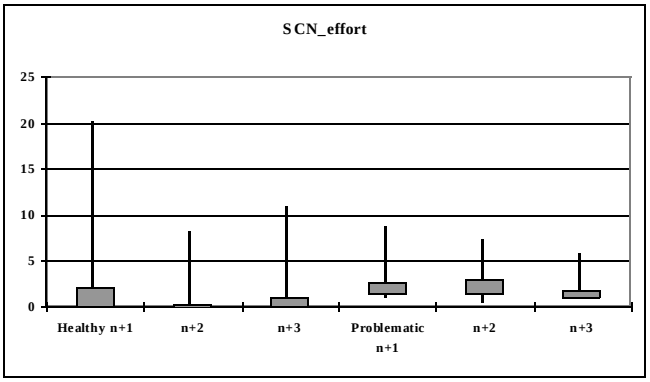

FIGURE 3. Number of files changed per defect report.

Figure 3 plots the SCN_effort. It describes the average number of files changed in a component per defect report. While the range of SCN_effort is larger for the healthy components, the boxes are lower than the problematic ones. SCN_effort for the problematic components show less variance and is higher when we consider the two middle quartiles (the box). If we only compare the problematic parts we can see that the values are slightly decreasing. This might be an indication that the problems are related to cer-

tain sets of components and files. Noticeable also is that the box for release *n+2* is slightly larger than the other two.

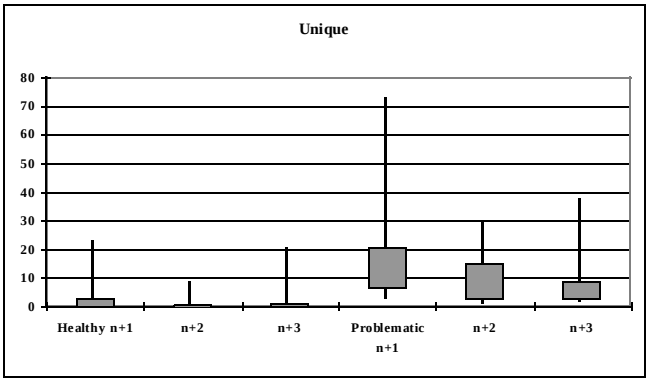

FIGURE 4. Number of unique files changed.

In Figure 4 we can see the decreasing number of unique files that are changed for each component. This is, as mentioned earlier, an indication that certain parts of the system are decaying and that problems are concentrated in specific parts of the system.

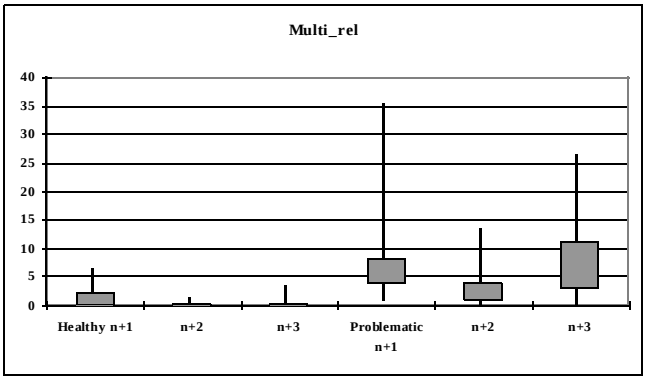

FIGURE 5. Number of defect reports with more than one component changed.

One of our conjectures was that the relationships between components cause decay and therefore also problems maintaining the system. Figure 5 shows the relationship measures of components that required corrective maintenance across releases. The healthy components have fairly low values. Values for release *n+1* where the highest, but still much smaller than for problematic components. Problematic components show much higher values. In release *n+1* the maximum value is very high but the box is not as large as in release *n+3*. This is definitely an indication of decay among the compo-

nents. We also have to keep in mind that release *n+3* included many changes in related .c files.

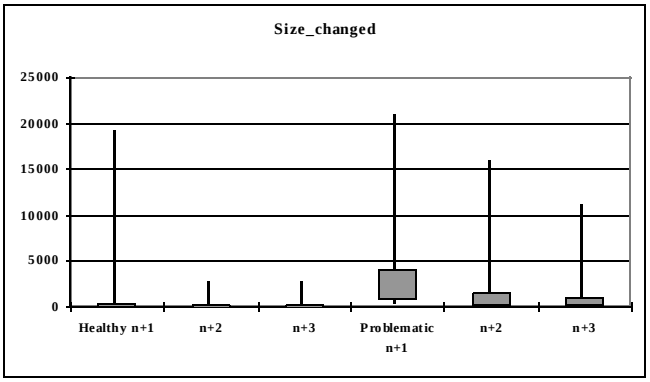

FIGURE 6. Changes in size.

Finally in Figure 6 the LOC changed (added plus deleted) are plotted. The healthy components are fairly stable with a very small box and low change values. By contrast, the problematic components show larger variance and larger boxes. More code has to be changed. Between release *n+1* and *n+3* the amount of code change decreases. Overall the results show for the problematic components that:

- Defect reports increase.
- More components needed to be changed to fix problems. Thus coupling and relationships between components play a major part in decay for this system.
- Size of 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 it is difficult to maintain. The source of decay is the interdependence between components.

5. Conclusions

Prediction of fault-prone components is a very important task to be able to direct effort and apply necessary corrective actions. But only looking at the defects from releases is not enough to assess fault-prone components.

We have classified components as green, yellow and red according to a certain threshold. In this case we chose to pinpoint 25 percent of the most fault-prone components, but other values are possible depending on the purpose and the amount of available resources. Healthy and problematic components are quite easy to predict. Harder to assess are the yellow ones, especially the ones that change from healthy to problematic. There is a need for more information to direct attention towards those components in a new release.

By applying PCA we tried to discover certain patterns among the healthy and the problematic components. Also, we focused on finding characteristics for the releases with the help of the PCA results. Among the healthy components we found groups of .c and .h related variables but no strong correlation to the relationship measure (Multi_rel).

One of the releases only touched a few components, which resulted in almost no separation between the variables for the healthy components.

The problematic components, on the other hand, showed some interesting reverse relationships between .c and .h variables. Also, the .h factors were mostly related to the relationship measures which indicates that shared data and coupling between components need repeated fixes. The box plots confirmed this as well.

Since we were limited in the kind of data collected, some of our results had to be based on surrogate measures, such as for component coupling. Ideally, we would prefer actual measures of connectivity between components. However, defect/fix reports are not likely to contain them. On the positive side, the study showed that a limited data set can provide useful information and help to find problematic areas (in this case component coupling). The results from the analysis have been verified by experts at the company. We do not think that too much effort should be spent on collecting other data, but in this case it would have been very useful to analyse connectivity measures.

Acknowledgements

This work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), grant 1K1P-97-09673, and by a scholarship from the ISS’90 foundation. We would also like to thank the two employees who helped us collect and interpret the data.

References

[1] N. Ohlsson, M. Helander and C. Wohlin, “Quality Improvement by Identification of Fault-prone Modules Using Software Design Metrics”, Proceedings of the International Conference on Software Quality, ICSQ’96, pp. 1-13, 1996, Ottawa, Canada.

[2] 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.

[3] D. Ash, J. Alderete, P.W. Oman and B. Lowther, “Using Software Models to Track Code Health”, Proceedings of the International Conference on Software Maintenance, ICSM'94, pp. 154-160, September 1994, Victoria, British Colombia, Canada.

[4] N.F. Schneidewind, “Software Metrics Model for Quality Control”, Proceedings of the International Symposium on Software Metrics, Metrics'97, pp. 127-136, November 1997, Albuquerque, New Mexico.

[5] T.M. Khoshgoftaar and R.M. Szabo, “Improving Code Churn Predictions During the System Test and Maintenance Phases”, Proceedings of the International Conference on Software Maintenance, ICSM'94, pp. 58-66, September 1994, Victoria, British Colombia, Canada.

[6] T.M. Khoshgoftaar, E.B. Allen, N. Goel, A. Nandi and K.S. Kalaichelvan, “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.

[7] 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.

[8] 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.

[9] T.M. Khoshgoftaar, E.B. Allen, R. Halstead and G.P. Tiro, “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.

[10] M. Jorgensen, “Experience With the Accuracy of Software Maintenance Task Effort Prediction Models”, IEEE Transactions on Software Engineering, 21(8), pp. 674-681, 1995.

[11] T.M. Khoshgoftaar and E.B. Allen, “Classification of Fault-Prone Software Modules: Prior Probabilities, Costs and Model Evaluation”, Journal of Empirical Software Engineering, 3(3), pp. 275-297, September 1998.

[12] M.J. Baker and S.G. Eick, “Visualizing Software Systems”, Proceedings of International Conference on Software Engineering, ICSE'94, pp. 59-67, May 1994, Sorrento, Italy.

[13] K.B. Gallagher, “Visual Impact Analysis”, Proceedings of International Conference about Software Maintenance, ICSM'96, pp. 52-58, November 1996, Monterey, California.

[14] H. Gall, M. Jazayeri, R.R. Klosch and G. Trausmuth, “Software Evolution Observations Based on Product Release History”, Proceedings of International Conference on Software Maintenance, ICSM'97, pp. 160-166, October 1997, Bari, Italy.

[15] M.C. Ohlsson and C. Wohlin, “Identification of Green, Yellow and Red Legacy Components”, Proceeding of International Conference on Software Maintenance, ICSM’98, pp. 6-15, November 1998, Bethesda, Washington D.C.

[16] L.C. Briand, V.R. Basili, Y-M. Kim and D.R. Squier, “A Change Analysis Process to Characterize Software Maintenance Projects”, Proceedings of the International Conference on Software Maintenance, ICSM'94, pp. 38-49, September 1994, Victoria, British Colombia, Canada.

[17] I.D. Baxter and C.W. Pidgeon, “Software Change Trough Design Maintenance”, Proceedings of the International Conference on Software Maintenance, ICSM'97, pp. 250-259, October 1997, Bari, Italy.

[18] R. Takahashi and Y. Nakamura, “The Effect of Interface Complexity on Program Error Density”, Proceedings of International Conference about Software Maintenance, ICSM'96, pp. 77-86, November 1996, Monterey, California.

[19] A. von Mayrhauser and A.M. Vans, “Comprehension Processes During Large Scale Maintenance”, Proceedings of International Conference on Software Engineering, ICSE'94, pp. 39-48, May 1994, Sorrento, Italy.

[20] T. M. Pigoski and L.E. Nelson, “Software Maintenance Metrics: A Case Study”, Proceedings of the International Conference on Software Maintenance, ICSM'94, pp. 392-401, September 1994, Victoria, British Colombia, Canada.

[21] P. Oman and J. Hagemeister, “Metrics for Assessing a Software System's Maintainability”, Proceedings of the International Conference on Software Maintenance, ICSM'92, pp. 337-344, November 1992, Orlando, Florida.

[22] J. Tian and M.V. Zelkowitz, “Complexity Measure Evaluation and Selection”, IEEE Transactions on Software Engineering, 21(8), pp. 641-650, 1995.

[23] N.F. Schneidewind, “Measuring and Evaluating Maintenance Process Using Reliability, Risk and Test Metrics”, Proceedings of International Conference on Software Maintenance, ICSM'97, pp. 232-239, October 1997, Bari, Italy.

[24] W.S. Humphrey, “A Discipline for Software Engineering”, Addison-Wesley Publishing Company, 1995.

[25] R.L. Gorsuch, “Factor Analysis”, 2:nd edition, Laurence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[26] J.O. Ramsay and B.W. Silverman, “Functional Data Analysis”, Springer-Verlag, New York, 1997.