A. von Mayrhauser, M. C. Ohlsson and C. Wohlin, "Deriving Fault Architectures from Defect History", Journal of Software Maintenance, Vol 12, No. 5, pp. 287-304, 2000.

Research

Deriving Fault Architectures from Defect History

Anneliese von Mayrhauser[1],Magnus C. Ohlsson[2] and Claes Wohlin[2]

[1] *Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873, U. S. A.*

[2] *Dept. of Communication Systems, Lund Institute of Technology, Lund University, Box 118, S-221 00 Lund, Sweden.*

SUMMARY

As software systems evolve over a series of releases, it becomes important to know which components are stable compared to components that show repeated need for corrective maintenance. The latter is a sign of code decay. Code decay can be due to the deterioration of a single component. In this case it manifests itself in repeated and increasing problems that are local to the component. A second type of code decay is due to repeated problems that become increasingly difficult to fix and are related to interactions between components. That is, components are repeatedly fault-prone in their relationships with each other. The latter requires repairs in multiple components and is a sign of problems with the software architecture of the system. Software architecture problems are by far more costly to fix and thus it is very desirable to identify potential architectural problems early and to track them across multiple releases. To do this, we adapt a reverse architecting technique to defect reports of a series of releases. Fault relationships among system components are identified based on whether they are involved in the same defect report, and for how many defect reports this occurs. There are degrees of fault-coupling between components depending on how often these components are involved in a defect fix. After these fault-coupling relationships between components are extracted, they are abstracted to the subsystem level. We also identify a measure for fault cohesion (i.e. fault-proneness of components locally.) The resulting fault architecture figures show for each release what its most fault-prone relationships are. Comparing across releases makes it possible to see whether some relationships between components are repeatedly fault-prone, indicating an underlying systemic architecture problem. We illustrate our technique on a large commercial system consisting of over 800 KLOC of C, C++, and microcode. **Copyright ©2000 John Wiley & Sons, Ltd.**

J. Softw. Maint. Res. Pract. 2000;

No. of Figures: 7. No. of Tables: 3. No. of References: 25.

KEY WORDS: decay identification, architectural faults, release analysis, fault-prone components

* Correspondence to: Dr. Anneliese von Mayrhauser, Department of Computer Science, Colorado State University, Fort Collins, CO 80523-1873, USA.
†Email: avm@cs.colostate.edu

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, 3], and becomes a legacy system. As new functionality and features are added, there is a tendency towards increased complexity. This may impact the system's maintainability. This makes it important to track the evolution of a system and its components, particularly those components which are becoming increasingly difficult to maintain as changes are made over time. Increasing difficulty in further evolution and maintenance is a sign of code decay.

The identification of these components serves two objectives. First, this information can be used to direct efforts when a new system release is being developed. This could mean applying a more thorough development process, or assigning the most experienced developers to these difficult components. Secondly, the information can be used when determining which components need to be re-engineered at some point in the future. Components that are difficult to maintain are certainly candidates for re-engineering efforts.

Early decay identification is desirable so that steps can be taken to prevent further degradation. The question is how to identify this decay and what to do to stop it. A major source of information that is commonly available are defect reports. Defect reports are written when developers, testers, or users, encounter defects during system development, test, or use. Defect reports usually contain information about the nature of the defect and how it was fixed (i. e. the underlying fault(s) that had to be removed). Fault removal can necessitate changes in one or more components. If changes are local to a component, the component's defect is said to have cohesion. By contrast, if fault removal involves changes in multiple components, the defect shows coupling. This concept of looking at a defect as either local to a component or as coupling components with regards to defect removal is analogous to describing structure or architecture of software [4, 5]: Software architecture consists of a description of components and their relationships and interactions, both statically and behaviorally [5]. Problems and possible architectural decay can be spotted via defects related to relationships and interactions of the components. A software architecture decay analysis technique must identify and highlight problematic components (we propose to do this via defect cohesion measurement) and relationships (we propose to do this via defect coupling measurement).

Code decay can be due to the deterioration of a single component. In this case it manifests itself in repeated and increasing problems that are local to the component. A second type of code decay is due to repeated problems that become increasingly difficult to fix and are related to interactions between components. That is, components are repeatedly fault-prone in their relationships with each other. The latter requires repairs in multiple components

and is a sign of problems with the software architecture of the system. Software architecture problems are by far more costly to fix and thus it is very desirable to identify potential architectural problems early and to track them across multiple releases.

Decay analysis identifies both components and relationships between components that are problematic. To assess potential decay that is local to a component, we define a defect cohesion measure at the component level. To determine decay related to faulty relationships between components, we define a defect coupling measure between components. High measures in either indicate problems, but of a different sort. High defect cohesion measures identify components that are broken in their functionality, while high defect coupling measures highlight broken relationships between components.

There are choices in which cohesion and coupling measures to use. One cohesion measure is to use the number of defects written against a component. This technique was used by Ohlsson et al. [6] to identify the most fault-prone components across successive releases. This approach has also been used with simple coupling measures based on common code fixes as part of the same defect report [7]. We provide a variant of this cohesion measure to more clearly distinguish defect fixes that involve single versus multiple file changes in a component. Similarly, a variety of coupling measures can be identified. [7] used simple coupling measures based on common code fixes a part of the same defect report. [8] used a defect coupling measure that is sensitive to the number of files that had to be fixed in each component that was identified as fault-prone through a defect cohesion measure. These defect cohesion and coupling measures can be computed for all components and components relationships that contain faults. However, usually only the most fault-prone components and component relationships are of concern, since they represent the worst problems and the biggest potential for code decay. We use the word code decay to denote code that, in some way, is becoming worse and worse for each software release, raising concerns about lack of maintainability in the future.

With this in mind, the primary interest is to identify components and component relationships that exhibit faults the most often, i. e. with the highest defect cohesion and defect coupling measures. Thresholds need to be determined to distinguish whether components and component relationships are fault-prone or not.

In this paper, we investigate ways

- to identify components and relationships between components that are fault-prone. This can be done either through an existing, up-to-date software architecture document in conjunction with defect reports, or, in its absence, through reverse architecting techniques such as [9, 10, 11, 12, 13, 14, 15, 16]. This paper tries to deal with the latter situation: an obsolete or missing software architecture document and the need for some reverse architecture effort. Reverse architecting in this context refers to the effort of identifying a system's components and component relationships without the aid of an existing architecture document.

- to measure defect cohesion and defect coupling for the components and component relationships identified.

- to set thresholds when distinguishing between fault-prone and not fault-prone components and component relationships. The components and component relationships that are fault-prone form the part of the software architecture that is fault-prone. This is called the fault architecture.

Section 2 reports on existing work related to identifying (repeatedly) fault-prone components. It also summarizes existing classes of reverse architecting approaches. Few researchers have tried to combine the two [17, 18]. We preferred two steps rather than a combination, because we wanted to use the reverse architecting approach both for building a fault architecture and a reverse architecture. Section 3 details our approach. Section 4 reports on its application to a sizable embedded system across 4 releases. The results show identifiable persistent problems with a subset of the components and relationships between them, indicating systemic problems with the underlying architecture. Section 5 draws conclusions and points out further work.

2 Background

2.1 Tracking and Predicting Fault-prone Components

It is important to know which software components are stable versus those which repeatedly need corrective maintenance, because of decay. Decaying components become worse as they evolve over releases. Software may decay due to adding new functionality with increasing complexity as a result of poor documentation of the system. Over time decay can become very costly. Therefore it is necessary to track the evolution of systems and to analyze causes for decay.

Ash et al. [20] provide mechanisms to track fault-prone components across releases. Schneidewind [21], Khoshgoftaar et al. [22] provide methods to predict whether a component will be fault-prone. [6, 7] combine prediction of fault-prone components with analysis of decay indicators. It ranks components based on the number of defects in which a component plays a role. The ranks and changes in ranks are used to classify components as green, yellow and red (GYR) over a series of releases. Corrective maintenance measures are analysed via Principal Components Analysis (PCA) [23]. This helps to track changes in the components over successive releases. Box plots are also used to visualize the corrective maintenance measures and to identify how they differ between releases.

2.2 Reverse Architecture

Reverse architecting is a specific type of reverse engineering. According to [24], a reverse engineering approach should consist of the following:

1. Extraction: This phase extracts information from source code, documentation, and documented system history (e. g. defect reports, change management data).

2. Abstraction: This phase abstracts the extracted information based on the objectives of the reverse engineering activity. Abstraction should distill the possibly very large amount of extracted information into a manageable amount.

3. Presentation: This phase transforms abstracted data into a representation that is conducive to the user.

Objectives in why code is reverse architected drives what is extracted, how it is abstracted, and how it is presented. For example, if the objective is to reverse architect with the associated goal to re-engineer (let's say into an object oriented product), architecture extraction is likely based on identifying and abstracting implicit objects, abstract data types, and their instances. This is the case with [9, 12, 13, 15]. Alternatively, if it can be assumed that the code embodies certain architectural cliches, an associated reverse architecting approach would include their recognition. [11] describes an environment that uses recognizers that know about architectural cliches to produce different architectural views of the system.

Other ways to look at reverse architecting a system include using state machine information [18], or release history [17]. CAESAR [17] uses the release history for a system. It tries to capture logical dependencies instead of syntactic dependencies by analyzing common change patterns for components. This allows identification of dependencies that would not have been discovered through source code analysis. It requires data from many releases. This method could be seen as a combination of identification of problematic components and architectural recovery to identify architectural problems.

If we are interested in a high level fault architecture of the system, it is desirable not to extract too much information during phase 1, otherwise there is either too much information to abstract, or the information becomes overwhelming for large systems. This makes the simpler approaches more appealing. In this regard, we found Krikhaar's approach particularly attractive [14]. The approach consists of three steps:

1. defining and analyzing the import relation between files. [14] defines the import relation via #include statements in the source code. Each file is also assigned to a subsystem (in effect creating a part-of relation). The import relation at the subsystem level is then derived as follows: if two files in different subsystems have an import relationship, the two subsystems to which the files belong have one as well.

2. analyzing the part-of hierarchy in more general terms (such as clustering, levels of subsystems). This includes defining the part-of relations at each level. These usually will be defined differently for each level. It also includes further definition of possible import relations and their abstractions.

3. analyzing use relations at the code level. Examples include call-called by relationships, definition versus use of global or shared variables, constants and structures. Analogous to the other steps, Krikhaar [14] also determines the part-of relation and abstracts use relations to higher levels of abstraction.

Within this general framework, there are many options to adapt it to a specific reverse architecting objective [10]. For example, Bowman et al. [25] also fits into his framework: the reverse architecting starts with identifying components as clusters of files. Import relations between components are defined through common authorship of files in the components (ownership relation). Use relationships are defined as calls-called-by relationships (dependency relation) of functions in components. Bowman et al. [25] also includes an evaluation of how well ownership and dependency relationships model the conceptual relationship.

We developed our adaptation based on the need to represent defect relationships between components and the ability to focus on the most problematic parts of the architecture.

2.3 Fault Architecture

von Mayrhauser et al. [8] combined the concepts of fault-prone analysis and reverse architecting to determine and analyze fault-architecture of software across releases. Components were identified as fault-prone based on the number of defect reports. Component relationships were fault-prone depending on how often a defect repair involved changes in files belonging to multiple components. These measures were ultimately used to rank components and component relationships with respect to how fault-prone they are. The top 25% in either ranking were considered fault-prone.

Components are identified based on the physical architecture (the directory structure). The directory structure of the fault-prone components (parts that are not fault-prone are omitted) is defined as the Fault Component Directory Structure. The Fault Architecture at the component level shows all components and component relationships that are identified as fault-prone. The Fault Architecture at the subsystem and system level is derived through abstracting the component level fault architecture. Multiple release analysis tracks defect reports for components across releases (i. e. their defect history).

The measures used to identify fault-prone components and fault-prone component relationships are very simple. They do not take into account that defects may differ in how much change is required to repair the underlying fault(s). When identifying fault-prone component relationships, the method only identifies components as having fault-prone relationships if these relationships rank in the top 25%. This omits components in the fault architecture whose fault relationships rank lower, but who may have a large number of them.

It is unclear whether this simple approach is sufficient or whether a more sophisticated set of measures would better identify the nature of architectural problems. Further, it might be useful to investigate setting thresholds for fault-proneness as a function of the measures obtained, rather than as a function of rank.

3 Approach

The approach consists of the following steps:

- Determine locally fault-prone components and their Fault Component Directory Structure.
- Determine components in fault-prone relationships and the Fault Architecture at the Component level.
- Abstract the Fault Architecture to the subsystem and system level.
- Perform Multi-release Analysis.

The basic strategy uses defect cohesion measures for components and defect coupling measures between components to assess how fault-prone components and component relationships are. If the objective is to concentrate on the most problematic parts of the software architecture, these measures are used with thresholds to identify

(a) the most fault-prone components only (setting a threshold based on the defect cohesion measure);

(b) the most fault-prone components relationships (setting thresholds based on two defect coupling measures).

3.1 Locally Fault-prone Components and the Fault Component Directory Structure

von Mayrhauser et al. [8] used the number of defect reports related to a component as a very simple measure of defect cohesion. This measure did not differentiate whether a defect in a component required modifying one file in a component, or dozens. If we assume that a defect is more complex to repair when its repair involves more of the component's files, we need a more sophisticated defect cohesion measure. We assume that defect reports include information about how many files in the component had to be changed to repair the fault(s) related to a given defect report. Further, we assume that for each change in a file, all other files that were changed to repair the defect had to be analyzed (to assess the impact of a change in one file on all the other changes). Graphically, this would lead to a fully connected graph (nodes are files, arcs are change relationships between files). This approach will sharply distinguish between single and multiple file changes related to a defect repair. We decided not to normalize this measure since in most systems components differ widely in size and how many files they contain. Further, we are not so much interested in the actual values of the cohesion measure, but in what relative ranking it determines between the components. This simplifies issues related to validation of the measures [19].

The defect cohesion for measuring the defect relationships between files of the same components is defined as follows:

$$Co_{<C>} = \begin{cases} \sum_{d_i=1}^{n} \frac{f_{d_i}(f_{d_i}-1)}{2} & for \quad f_{d_i} > 1 \\ 1 & for \quad f_{d_i} = 1. \end{cases} \qquad (1)$$

f_{d_i} is the number of files in component C that had to be changed to repair defect d_i; n is the number of defects. This provides an indication of local fault-proneness. This measure would include components as fault-prone that are only involved in a few defects, but where each defect repair required changing a large number of files. This provides a ranking of components with respect to how locally fault-prone they are.

For purposes of the case study, we classified a component as problematic when it is in the most fault-prone quartile in a release. Components are defined as collections of files in the same directory. Thus the directory structure of the software can be used as the "part-of" relationship. Fault-prone components are illustrated as leaves in this directory structure. We denote this as a *Fault Component Directory Structure*. Subsystems are defined through the directory structure.

3.2 Components in Fault-prone Relationships and the Fault Architecture at the Component Level

The next step is to develop the *Fault Architecture*. We adapted an existing reverse architecting technique [14] to identify the fault architecture of a system and to highlight both nature and magnitude of the architectural problem. Two or more components are related, if their files had to be changed in the same defect repair (i.e. in order to correct a defect, files in all these components needed to be changed). Defect relationships between components are measured with a defect coupling measure. For any two components C_1 and C_2, the relationship measure $Re_{<C_1,C_2>}$ is defined as:

$$Re_{<C_1,C_2>} = \sum_{i=1}^{n} C_{1_{d_i}} \times C_{2_{d_i}} \qquad C_1 \neq C_2 \tag{2}$$

where $C_{1_{d_i}}$ and $C_{2_{d_i}}$ stand for the number of files in component C_1 and C_2 that had to be fixed in defect d_i; n is the number of defects whose fixes necessitated changes in components C_1 and C_2.

In addition, we need to define how to set a threshold for fault-proneness with respect to relationships between components. There are two reasons, why a component C can be fault-prone with respect to relationships:

1. the defect coupling measure is high for a particular pair of components $< C, C_i >$.
2. none of the defect coupling measures are high, but there are a large number of them (the sum of the defect coupling measures is large).

The second situation prompted us to determine a threshold for fault-prone relationships between components based on the sum of the defect coupling measures for a component:

$$TR_C = \sum_{i=1}^{m} Re_{<C,C_i>} \qquad C \neq C_i \tag{3}$$

where m is the number of components other than C, $Re_{<C,C_i}>$ is the defect coupling measure between C and C_i.

The threshold for including a component in the fault architecture is set as 10% of the highest TR_C measure. The threshold for including a fault relationship arc in the architecture is set at 10% of the highest $Re_{<C,C_i>}$. This means that we are setting the threshold about one order of magnitude lower than the highest value for TR_C and $Re_{<C,C_i>}$. Components that have been flagged as fault-prone for the lowest level nodes of the *Fault Component Directory Structure*. Components and fault relationships that have been identified as fault-prone via the two defect coupling measures form the nodes and arcs of the lowest level of the *Fault Architecture*. The Fault Architecture Diagram may have nodes to represent components with a high TR_C measure, but low $Re_{<C,C_i>}$ measure. In this case, the node shows no fault-relationship arcs, to denote situation 2 above.

3.3 Fault Architecture at the Subsystem and System Level

The fault relationship can be abstracted to the subsystem and system level: Two subsystems are related, if they contain components that are related. This represents Krikhaar's "lift" operation [14]. Let `aa`, `bb` be the names of two subsystems of the system to be analyzed. Then the the defect coupling measure between these two subsystems `aa` and `bb` is the sum of coupling measures between components in `aa` and components in `bb`:

$$Rsub_{<aa,bb>} = \sum_{Rset} Re_{<C_{aa},C_{bb}>} \qquad (4)$$

where $Rset = \{(C_{aa}, C_{bb})|C_{aa} \in aa \cup C_{bb} \in bb\}$.

Similarly, this method can be applied when going from the subsystem to the system level. Changes in such patterns, or persistent fault relationships between components, across releases, is an indicator of systemic problems between components and thus architecture.

3.4 Multi-release Analysis

The result of this phase is a series of *Fault Architecture Diagrams*, one for each release. These results are also used to update the Fault Component Directory Structure as follows: components in bold face appear in the Fault Architecture Diagrams. These are components with fault-prone relationships to others in at least one release. Bold components are also annotated with the release identifiers in which they were considered relationship fault-prone. Non-bold components are internally fault-prone in at least one release, but do not show fault relationships with other components.

To investigate further the nature of continued problems between components, we also aggregate these diagrams into a *Cumulative Release Diagram*. The nodes represent components that occur in at least one Fault Architecture Diagram. Two nodes are related (i.e. have an arc between them), if there is a fault relationship between corresponding nodes in at least one Fault Architecture Diagram. The arcs are annotated as follows: if Fault Architecture

Table 1: Number of Releases in which Components were Fault-Prone

Times fault-prone	0	1	2	3	4
Number of components	63	34	22	9	2

Diagrams for releases n and m show an arc between components C_i and C_j, then the Cumulative Release Diagram's arc between C_i and C_j is annotated with $T_{n,m}$ (for release transition n to m). This highlights repeatedly problematic relationships in the Fault Architecture.

4 Case Study

4.1 Environment

We applied this technique to a large embedded mass storage system of about 800 KLOC of C, C++, and microcode in 130 software components. Each component contains a number of files. We studied four releases. The data is based on defect/fix reports or source change notices (SCN). Every report indicates a problem that had to be corrected.

4.2 Identification of Fault-prone Components and the Fault Component Directory Structure

To extract locally fault-prone components we decided to rank all components based on the defect cohesion measure and then identify the top 25% of the ranked components as locally fault-prone. In case of ties in rank that would cause more than 25% of the components to be included, the smaller set was chosen. Table 1 shows to which degree components were repeatedly fault-prone. For example, 63 components were never identified as fault-prone while 2 were identified as fault-prone in all four releases. Components that were fault-prone in at least one release are included in the Fault Component Directory Structure (see Figure 1). We refer to a component as a collection of files in the same directory.

4.3 Analysis of Fault-prone Relationships

This analysis extracts the fault architecture of the system. Two fault-prone components have a fault relationship between them if they each contain files that had to be repaired during corrective maintenance for the same defect report. All data is stored in a database which contains records with information for every defect report, including the files that had to be changed. The component fault-relationships were extracted with SQL scripts.

The project was also analyzed for its fault relationships using the defect coupling measure $Re_{<C,C_i>}$. Table 2 summarizes the measures for all releases and for all components. The first column identifies the release for which data is reported. The next three columns state

Table 2: Defect Coupling for All Components: Release 1-4

	All		
	Components	Relations	$Re_{<C,C_i>}$
Re 1	75	1163	1-1449
Re 2	92	962	1-444
Re 3	29	74	1-200
Re 4	53	210	1-128

Table 3: Defect Coupling for Fault-Prone Components: Release 1-4

	Fault-prone				
	Components	Relations	$Re_{<C,C_i>}$	Components	TR_C
Re 1	19	26	145-1449	18	711-7107
Re 2	32	70	44-444	19	422-4219
Re 3	13	16	20-200	13	54-541
Re 4	27	38	13-128	15	58-577

how many components the release contained, how many non-zero defect coupling measures existed, and the range of values found for defect coupling. Table 3 shows this data for those considered fault-prone based on the 10% threshold. In addition to the defect coupling measure $Re_{<C,C_i>}$, we also show the cumulative defect coupling measure TR_C. The first column identifies the release for which data is reported. The next three columns identify how many of the components were fault-prone, how many fault-prone component relationships existed, and the range of values for defect coupling measures for the fault-prone relationships.

Based on these results we also added to the Fault Component Directory Structure in Figure 1, marking components with fault-prone relationships to other components in bold. Bold components are annotated with the release identifiers in which they were considered relationship fault-prone. For example, `sd52` and `sd53` were relationship fault-prone in Release 2 and Release 3. The components not marked bold are fault-prone components with internal problems instead of fault-prone relationships to other components.

Figure 2 shows the Fault Architecture Diagram for Release 1. Nodes represent components. Arcs between two vertices show that components are fault-prone in their relationship. The weights on the arcs indicate the defect coupling measure $Re_{<C,C_i>}$. Note that Figure 2 includes subdirectories at different levels of the directory structure. This was necessary when they included (changed) files. For example, `/A/c/cc/sd23` contains such files.

Figure 2 consists of two distinct fault architecture structures. Components `System/A/e` and `System/A/b/sd18` are at the center of one. The second involves many components in subsystem `System/A/f`. In this figure, two components, `System/A/c/cc/sd23` and `System/A/d/nf22`, show no defect relationship with others. They are included because both components have a high aggregate defect coupling measure TR_C. The numbers in parentheses next to the component identifiers indicate the value of TR_C. It means that these two components are

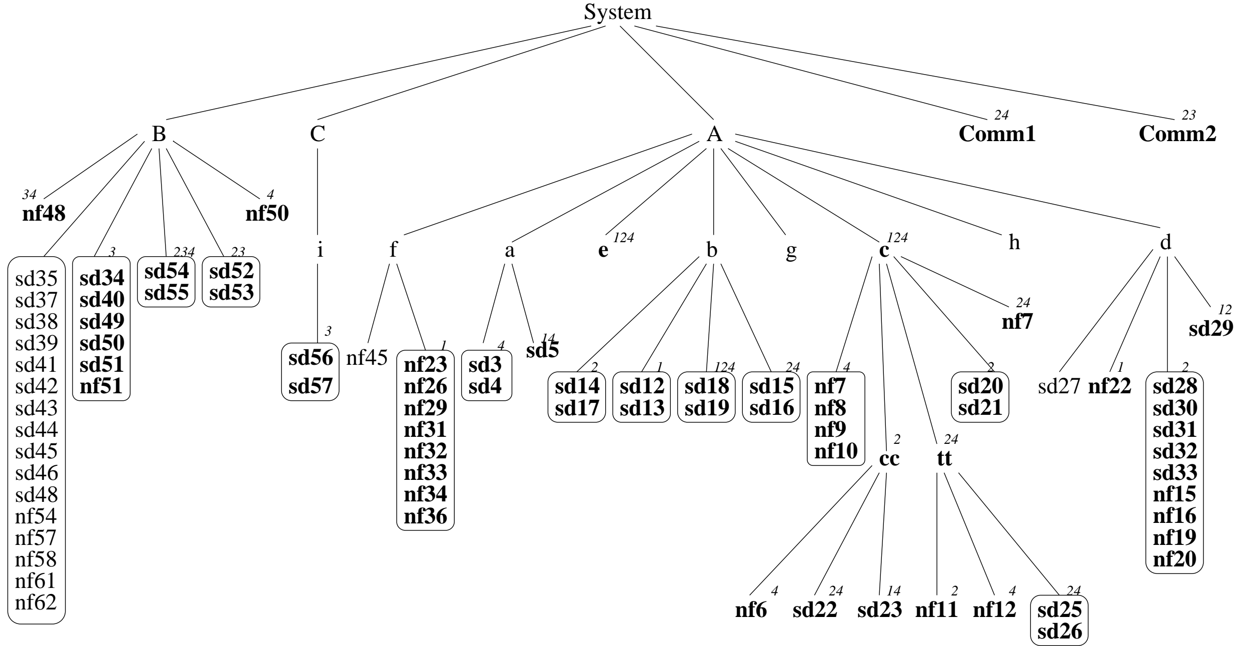

Figure 1: Fault Component Directory Structure

modified with many other components. Based on this figure, it is worthwhile to investigate the parts of the software architecture around `A/e` and `A/b/sd18` in particular, as well as the relationships between `A/f/nf/34` and components with which it has highly ranked fault relationships.

4.4 Fault Architecture Diagrams at the Subsystem Level

Figure 3 was constructed from the Component Level Diagram by lifting the component level relationships to the next higher level in the directory structure. For example, the components `A/b/sd12`, `A/b/sd18`, and `A/b/sd19` are aggregated to `A/b`. The fault relationship arcs of these components (to `A/b` and `A/e`) are aggregated as well. Thus the strength of the aggregated fault relationships between two subsystems is the sum of the defect coupling measures between components in either subsystem. If components within the same subsystem only have fault relationships with each other, the system level fault architecture shows a node (representing the subsystem) with an arc that starts and ends at the node. It is annotated with the sum of these fault relationships (sum of the defect coupling measures between components local to the subsystem). See, for example, subsystem `A/f`. The results from this operation are presented in Figure 3.

Figures 3-6 show the Fault Architecture Diagrams at the system level for all four releases. The thickest arcs indicate at least 500 problems involving both components, while medium thick arcs represent between 200 and 500 problems. Thin arcs indicate 50 and 200 problems, and thin dashed arcs represent less than 50 problems.

In Release 1, `System/A/f` shows large amounts of defect coupling between its components

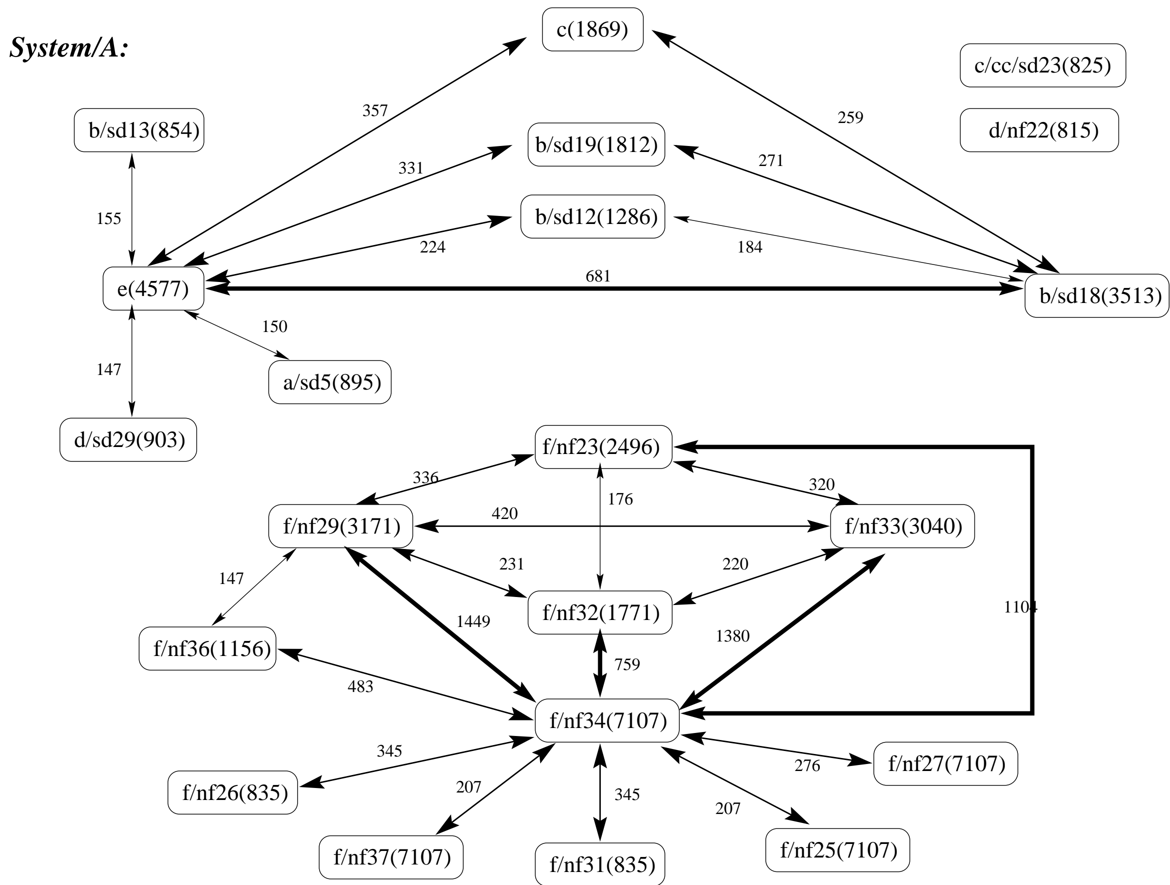

Figure 2: Fault Architecture at the Component Level for Release 1

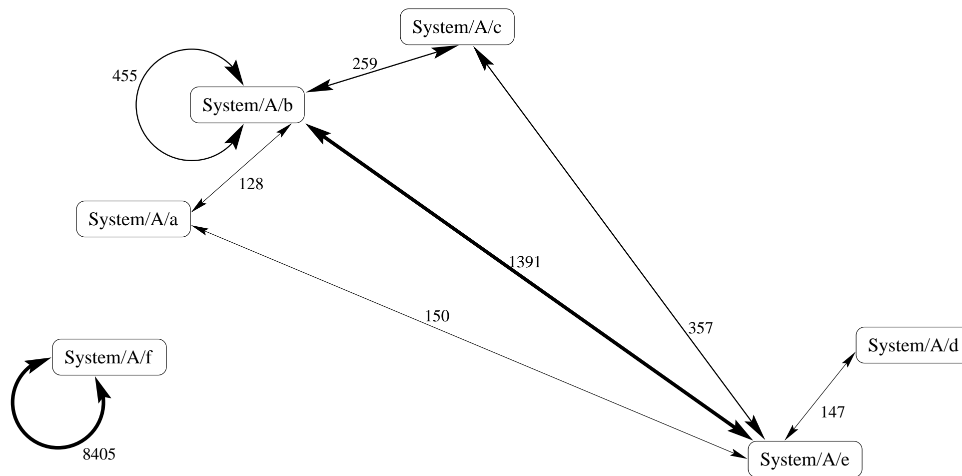

Figure 3: System Fault Architecture Release 1

(aggregate defect coupling of 8405). Although it seems a very problematic subsystem, it doesn't show any defect coupling with other subsystems. `System/a/e` and `System/A/b` are

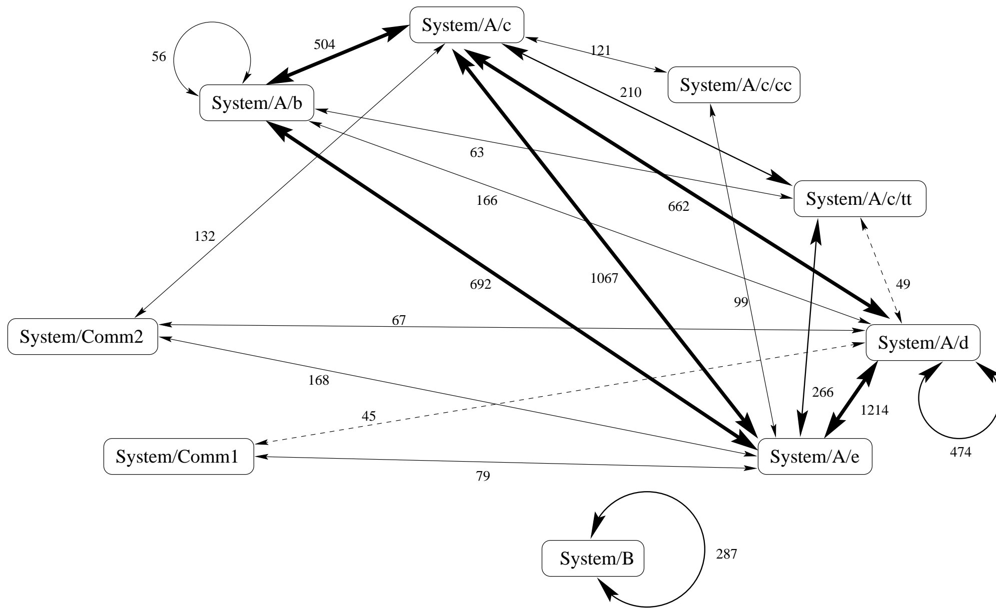

Figure 4: System Fault Architecture Release 2

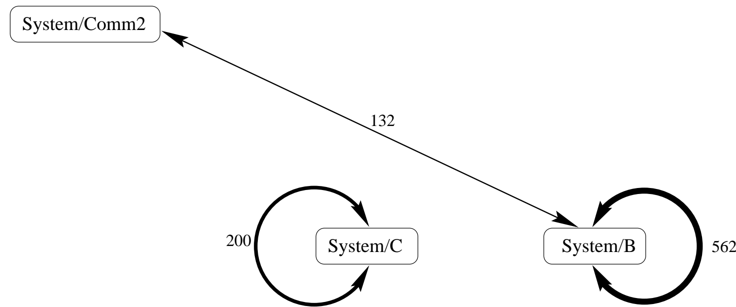

Figure 5: System Fault Architecture Release 3

the relationship fault-prone subsystems in Release 1.

In Release 2, the fault architecture is considerably more complex indicating more widespread problems. Nine subsystems out of thirteen are included as relationship fault-prone in Figure 4. `System/A/c`, `System/A/e` and `System/A/d` are relationship fault-prone in their interactions with at least six other subsystems. `System/A/b` is also problematic. It has two fault relationships with a fault coupling measure of over five hundred.

In Release 3, only three subsystems are relationship fault-prone. The new subsystem `System/C` is only relationship fault-prone between components local to `System/C`.

In Release 4, `System/A/e` is the center of the relationship problems while `System/A/c` has a high fault relationship with `System/A/c/cc`, `System/A/c/tt` and internally. The internal problems of `System/C` and `System/B` persist. Instead of fault relationships between `System/B` and `System/Comm2` in Release 3, a fault relationship between `System/B` and `System/Comm1`

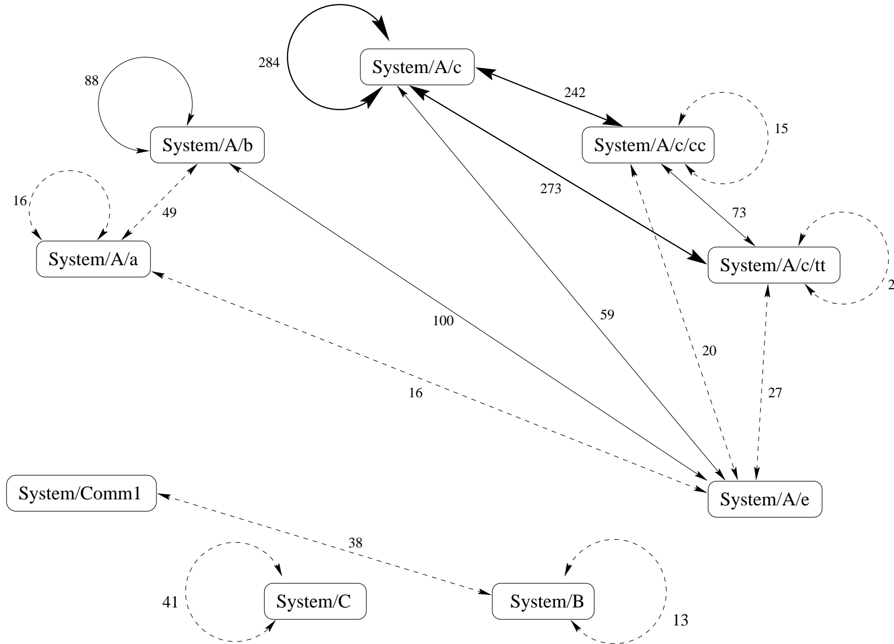

Figure 6: System Fault Architecture Release 4

appears.

4.5 Multi-release Analysis

Figure 7 shows the Cumulative Release Diagram. It illustrates persistent problems. This diagram aggregates relationships between releases. The arc annotations in the diagram describe which fault relationships persisted across releases. For example, `System/B` had internal problems carried from Release 2 to Release 3, and from Release 3 to Release 4 and is therefore annotated with T23 and T34. Key fault relationship "drivers" are subsystems `System/A/b` and `System/A/e`. They have fault relationships with most other subsystems in the upper portion of the diagram. The problems internal to `System/B` and `System/C` also are persistent. Moreover, Figure 7 also shows a persistent fault relationship problem between `System/A/c` and `System/A/b`, `System/A/c/cc`, `System/A/c/tt`.

Three releases show a large number of fault relationships between subsystems `System/A/b` and `System/A/e`. Persistent problems for Release 1 and Release 2 involve subsystems `System/A/b`, `System/A/c`, `System/A/d`, and `System/A/e`. Even in the fourth release there are problems in the relationships between subsystems `System/A/b`, `System/A/c`, and `System/A/e`. We can also see some problems resurface in `System/A/a`. This indicates a systemic problem that is not going away, and is a strong indicator of architectural problems in the relationship between these components. It could also be accompanied by code decay due to repeated problem fixes. A positive indication is the decreasing number of problems between `System/A/a` and `System/A/e`. T14 represents an interesting phenomenon. Arcs annotated with T14 reflect fault relationships that were present in Release 1 and Release 4, but not in Release 2 and Release 3. One interpretation could be that the underlying architectural problems were

never fully solved in Release 1 and, with addition of new features, reappeared in Release 4. We recommend to look at `System/A`, especially `System/A/b` and `System/A/e` in conjunction with their fault-related components which seem to be key to the biggest problems relative to multiple releases.

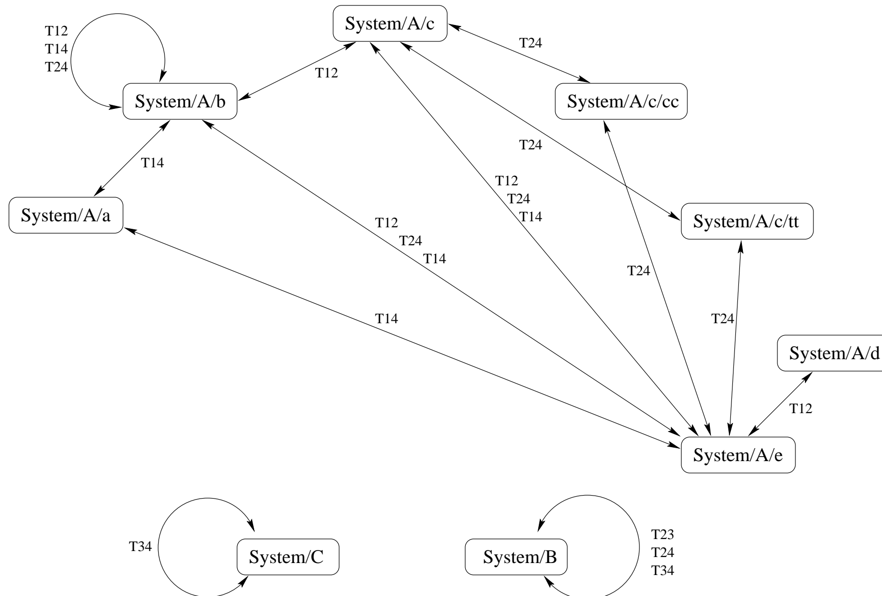

Figure 7: Cumulative Release Diagram

5 Conclusions and Further Work

This paper developed measures and a method to build fault architectures from defect reports. A case study illustrates the method. Defect reports are easily available and could therefore be used to identify the problematic parts of a system. We defined two measures that rank the most fault-prone relationships between components and subsystems in a number of releases. We created a Fault Component Directory Structure and investigated the fault-prone relationships between components. Fault Architecture Diagrams show fault-prone relationships at the component and subsystem levels. Finally, a Cumulative Release Diagram is created to track problematic relationships across releases. We were able to identify the most problematic components and subsystems, both locally and in their fault relationships. In addition, cross release analysis allowed investigation of trends and identification of those components and subsystems that show repeated problems. This identifies prime candidates for remedial action.

The measures presented in this paper represent a refinement of the approach originally discussed in [8]. We were able to clearly identify for every release what the most problematic component relationships are. The most problematic stayed that way over more than one release or reappeared. Even with improvement efforts in successive releases, the core problems in the architecture, while mitigated, never disappeared completely. Lesser problems, as for example the difficulties with the `System/Comm1` and `System/Comm2` subsystems in Figure 4,

may be due to premature release, rather than more deeply rooted architectural problems. In the latter case, they will disappear from the fault architecture. We see the key advantage to providing a fault architecture in drawing attention to the most pressing problematic component relationships. This identifies which relationships should be scrutinized whether they require corrective maintenance or re-architecting. The most central problems seem to revolve around interactions related to `System/A/b` and `System/A/e`. These subsystems have relationships to many other components in `System/A` and should therefore be analyzed in more depth.

None of these observations would have been possible by merely counting defects for each component. The value of the technique presented here lies in

- showing the magnitude and nature of problems (i. e. whether they are local to components or not).
- identifying and visualizing whether problem persist or get worse between releases.
- helping to focus on the most problematic as targets for improvement activities.

In the case study we were able to identify key parts of the system that warrant improvement.

This method differs from [8] in how defect reports are treated. [8] only counted defect reports, here we developed a cohesion measure that models the magnitude of change necessary for defect removal. We propose to use the latter if there are large differences in how defects are repaired.

Further root cause analysis would benefit from counting other indicators, but that depends on their availability. In [7] the same system was analyzed using Principal Components Analysis on detailed measures related to code changes in (shared) files. They derived measures for impact of change to components and to related components. The most fault-prone components show more decay. Our fault architecture concentrates more specifically on the relationships between fault-prone components in terms of the magnitude of the problems in which they are involved. The fault architecture could also be used in conjunction with the analysis in [7] to further investigate architectural problems. The fault architecture identifies which components and component relationships should be analyzed further through Principal Components Analysis or box plot trends.

We would also like to apply other techniques like [17] to the defect analysis reports and compare the results. Adapting a reverse architecture technique like [14] has the advantage that it can be used to identify both the existing module architecture as well as its fault related parts.

Acknowledgement

We gratefully acknowledge J. Wang's help in providing some of the data analysis.

References

[1] Gefen D, Schneberger SL. The non-homogeneous maintenance periods: a case study of software modifications. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1996; 134–141.

[2] Minsky NH. Controlling the evolution of large scale software systems. In *Proceedings International Conference on Software Maintenance–1985*. IEEE Computer Society Press: Los Alamitos CA, 1985:50–58.

[3] Lehman MM, Belady LA. *Program Evolution: Process of Software Change*. Academic Press, Austin TX, 1985; 552 pp.

[4] Allen R, Garlan D. Formalizing Architectural Connection. In *Proceedings International Conference on Software Engineering–1994*. IEEE Computer Society Press: Los Alamitos CA, 1994; 71-80.

[5] Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall: Upper Saddle River NJ, 1996; 242 pp.

[6] Ohlsson MC, Wohlin C. Identification of green, yellow and red legacy components. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1998; 6–15.

[7] Ohlsson MC, von Mayrhauser A, McGuire B, Wohlin C. Code decay analysis of legacy software through successive releases. In *Proceedings IEEE Aerospace Conference–1999*. IEEE Press, Piscataway NJ.

[8] von Mayrhauser A, Wang J, Ohlsson MC, Wohlin C. Deriving a fault architecture from defect history. In *Proceedings International Symposium on Software Reliability Engineering–1999*. IEEE Computer Society Press: Los Alamitos CA, 1999; pp. 295–303.

[9] Canfora G, Cimitile A, Munro M, Taylor CJ. Extracting abstract data types from C programs: a case study. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1993; 200–209.

[10] Feijs L, Krikhaar R, van Ommering R. A relational approach to software architecture analysis. *Software Practice and Experience 1998*; **28**(4): 371–400.

[11] Fiutem R, Tonella P, Antoniol G, Merlo E. A cliche-based environment to support architectural reverse engineering. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1996;319–328.

[12] Girard JF, Koschke R. Finding components in a hierarchy of modules: a step towards architectural understanding. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1997; 58–65.

[13] Harris DR, Yeh AS, Reubenstein HB. Recognizers for extracting architectural features from source code. In *Proceedings Second Working Conference on Reverse Engineering (WCRE '95)*. IEEE Computer Society Press: Los Alamitos CA, 1995; 252–261.

[14] Krikhaar RL. Reverse architecting approach for complex systems. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1997;1–11.

[15] Yeh AS, Harris DR, Reubenstein HB. Recovering abstract datatypes and object instances from a conventional procedural language. In *Proceedings Second Working Conference on Reverse Engineering (WCRE '95)*. IEEE Computer Society Press: Los Alamitos CA, 1995; 227–236.

[16] Younger EJ, Luo Z, Bennett KH, Bull TM. Reverse engineering concurrent programs using formal modeling and analysis. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1996; 255–264.

[17] Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1998; 190–198.

[18] Gall H, Jazayeri M, Kloesch R, Lugmayr W, Trausmuth G. Architecture recovery in ARES. In *Proceedings Second International Software Architecture Workshop (ISAW-2)*. ACM Press: New York NY, 1996; 111–115.

[19] Kitchenham B, Pfleeger SL, Fenton N. Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering 1995*; **21**(12):929–944.

[20] Ash D, Alderete J, Oman PW, Lowther B. Using Software Models to Track Code Health. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1994; 154-160.

[21] Schneidewind NF. Software metrics model for quality control. In *Proceedings International Symposium of Software Metrics–1997*. IEEE Computer Society Press: Los Alamitos CA, 1997; 127–136.

[22] Khoshgoftaar TM, Szabo RM. Improving code churn predictions during the system test and maintenance phases. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1994; 58–66.

[23] Gorsuch RL. *Factor Analysis, Second Edition*. Laurence Erlbaum Associates: Hillsdale NJ, 1983; 442 pp.

[24] Tilley SR, Wong K, Storey MAD, Muller H. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering 1994*; **4**(4):501-520.

[25] Bowman IT, R.C. Holt RC. Software Architecture Recovery Using Conway's Law. In *Proceedings CASCON'98*. November-December 1998, Mississauga, Ontario, Canada; 123-133.

AUTHORS' BIOGRAPHIES

Anneliese von Mayrhauser received a Dipl.-Inf. degree in Informatik (1976) from the Technical University in Karlsruhe and the A. M. (1978) and Ph.D. (1979) in Computer Science from Duke University. She is currently a Professor of Computer Science at Colorado State University and Director of the Colorado Advanced Software Institute. Dr. von Mayrhauser has published on software testing, software metrics, software maintenance, reliability and performance modeling.

Magnus C. Ohlsson is a doctoral student at the Department of Communication Systems, Lund University. He received his Master of Science in Software Engineering from University of Karlskrona/Ronneby in 1996, and his Licentiate in Engineering from the Department of Communication Systems, Lund University, in 1999. His main research areas are the use of experience to build prediction models and models for process improvement. Most of the work has been carried out within effort prediction and maintenance. Magnus C. Ohlsson is a co-author of "Experimentation in Software Engineering An Introduction".

Claes Wohlin is a professor of software engineering at the Department of Communication Systems, Lund University. Prior to this, he was a professor of software engineering at Linkping University. He has a Ph.D. in Communication Systems from Lund University. He is the founder and director of the research group in software engineering at the Department. His research interests include empirical methods in software engineering, software metrics, software quality and systematic improvement in software engineering. Claes Wohlin is the principal author of "Experimentation in Software Engineering An Introduction". He is on the editorial board of the Journal of Information and Software Technology. Dr. Wohlin is the program chair for the International Symposium on Software Metrics 2001.