Empirical Software Engineering, 8, 83–108, 2003.

An Empirically-Based Process for Software Architecture Evaluation

MIKAEL LINDVALL mlindvall@fc-md.umd.edu
Fraunhofer Center for Experimental Software Engineering, Maryland

ROSEANNE TESORIERO TVEDT roseanne.tesoriero@washcoll.edu
Fraunhofer Center for Experimental Software Engineering, Maryland and Washington College

PATRICIA COSTA pcosta@fc-md.umd.edu
Fraunhofer Center for Experimental Software Engineering, Maryland

Editor: Lionel Briand

Abstract. Software systems undergo constant change causing the architecture of the system to degenerate over time. Reversing system degeneration takes extra effort and delays the release of the next version. Improved architecture is intangible and does not translate into visible user features that can be marketed. Due to a lack of representative metrics, technical staff has problems arguing that stopping degeneration is indeed necessary and that the effort will result in an improved architecture that will pay off. We believe that architectural metrics would give technical staff better tools to demonstrate that the architecture has improved. This paper defines and uses a set of architectural metrics and outlines a process for analyzing architecture to support such an argument.

The paper reports on a case study from a project where we restructured the architecture of an existing client-server system written in Java while adding new functionality. The modules of the existing version of the system were "library-oriented" and had a disorganized communication structure. The new architecture is based on components and utilizes the mediator design pattern. The goal of the study is to evaluate the new architecture from a maintainability perspective. The paper describes our evaluation process, the metrics used, and provides some preliminary results. The architectural evaluation shows that the components of the system are only loosely coupled to each other and that an architectural improvement has occurred from a maintenance perspective. The process used to evaluate the architecture is general and can be reused in other contexts.

Keywords: Architectural evaluation, metrics, empirical study.

1. Introduction

Software systems constantly change. Users relentlessly require more functionality of a system and a successful system often has to be adapted to a changing environment.

As an effect of the constant change, the structure of the system becomes more and more complex; often it degenerates. The structure can only be preserved or simplified through extra effort through which the degeneration of the system is stopped and hopefully reversed (Lehman and Belady, 1985).

One reason for such increased complexity is the fact that change often is synonymous with growth. New features are added over time and each such new

feature requires more source code to be written. More source code means more modules and more modules means more interconnections among new and existing modules. These additions contribute to the complexity of the system and make it harder to understand and change.

It is clear that software will change over time, but it is a challenge to achieve successful software system evolution. The answer to smooth software evolution is related to the structure of the system. A system without an adaptable architecture will degenerate sooner than a system based on an architecture that takes change into account (Eick et al., 2001).

This paper reports on a case study from a project in which we reengineered a previous version of our experience management system (EMS) (Seaman et al., 1999). The goal of the project was to develop a new version incorporating a new architecture needed to handle a set of new requirements. The paper describes our evaluation process, the metrics used, and provides some preliminary results. The metrics helped us show that the previous architecture needed to improve and that the new architecture is indeed an improvement from a maintenance perspective. We believe the same analysis could be used by others and help them to argue their case better.

1.1. Background

One of Fraunhofer Center Maryland's (FC-MD) primary assets is our software system EMS, which manages experience supporting our "Experience Factory" approach for organizational learning (Basili et al., 1994a). In 1999, we found ourselves in an impossible situation. Both current and potential users of the system asked for more functionality, while it was clear that the current architecture of the system couldn't withstand additional changes. Each change to the system only made it harder to change again. The effort of adding any functionality widely exceeded what the effort "ought to be" to add that functionality and we realized that the system had decayed (Eick et al., 2001). When we understood that the system had become un-maintainable, we made the decision to restructure it. The plan was to design an architecture that would facilitate the addition of features in the future enabling us to further satisfy users of the system. Part of the development was going to be outsourced to a partner in Brazil. It was important to identify an architecture that allowed for geographically distributed development including independent design, implementation, and testing of modules.

While the overall architecture of EMS was, and still is, Internet-based and client-server, we converted our "library-oriented" modules (described in Section 2) into components and based the architecture of the client on components and the mediator design pattern (Gamma et al., 1994) (also described later in this section). We expected this planned design to enable independent development and to accommodate future addition of components. In order to assess whether this was the case, we defined a process based on the concept of software architecture evaluation.

1.2. Evaluation of Software Architecture

Software architecture evaluation (SAE) can be conducted at different points in time during the software life cycle and with different goals in mind. Here we distinguish between early SAE and late SAE. Early SAE is used to evaluate one or more software architecture (SA) candidates that are not yet implemented. Late SAE, which we used in this study, is used to evaluate the SA of an implemented version of a system compared to the SA of a previous version.

Early SAE, as described by Abowd et al. (1996) and Avritzer and Weyuker (1998), can be based on the description of the SA and other sources of information, for example interviews with the creators of the SA. Early SAE facilitates:

- Better understanding of the SA.
- Verification that all requirements are accounted for.
- Indication that the system will have the desired qualities or quality attributes (e.g. performance, reliability, and maintainability).
- Identification of problems with the architecture.

The methodology proposed by Yacoub and Ammar (2002) is an example of the third bullet in that it analyzes architectures from a risk perspective early in the life-cycle. Bengtsson and Bosch's architecture level prediction of software maintenance is another example of the same kind of analysis (Bengtsson and Bosch, 1999). The approach suggested by de Bruijn and van Vliet (2001) is based on the generation of a basic architecture that is immediately evaluated and problems identified. A new architecture is generated and again evaluated. This iterative approach continues until the architecture fulfills the software requirements. The architecture-level modifiability analysis (ALMA) model by Lassing et al. (2002) emphasizes that the goal should govern how the evaluation will be carried out. ALMA distinguishes between three different goals:

- Risk assessment; finding types of changes for which the system is inflexible.
- Maintenance cost prediction; estimating the cost of maintenance effort for the system in a given period.
- Software architecture comparison; comparing two or more candidate software architectures to find the most appropriate one.

Late SAE (e.g. Murphy et al.'s reflexion models (Murphy et al., 1995)) can utilize data measured on the actual software implementation. These metrics can be used to reconstruct the actual SA in order to compare it to the planned SA that was used in the early SAE.

An analysis of the actual SA of a previous version of a system can be used to increase understanding of the system in order to improve it. Based on such an analysis, the late SAE approach can be used to compare and evaluate the new actual SA with the planned SA and the previous actual SA. The analysis can be used to evaluate whether the new actual SA fulfills the planned SA and whether it better fulfills the defined goals and evaluation criteria than does the previous actual SA. An early example of such an approach is Schwanke's re-engineering tool (Schwanke, 1991). The tool uses a concept of similarity based on Parnas' information hiding principle, but is not specifically designed for object-oriented systems. The tool supports two services: clustering, which identifies similar procedures that should be grouped in a module, and a maverick analysis, which identifies procedures that appear to be in the wrong module. While the tool can provide advice on a system's ideal modularization and can be tuned to include the architect's preferences, the advice is limited to the similarity concept, and cannot be used to check conformance to architectures that do not follow the similarity concept as a design rule.

An example of evaluation approaches for object-oriented systems include the Pattern-Lint tool (Sefika et al., 1996), which analyzes a system's design and determines its compliance to its high-level design models. The tool combines static and dynamic analysis in order to identify deviations from the intended design. (Fiutem and Antoniol, 1998) use a traceability approach to carry out the design to code matching process. The approach points out code areas that do not match the design and helps the user detect inconsistencies between the two. A related, but more code-oriented approach is Meyer et al.'s CCEL (1993), which defines design, implementation and stylistic constraints. The constraint checker produces a list of violations that help the user identify problematic areas of the code.

For both early and late SAE, there are two basic categories of architectural evaluation: those that generate qualitative questions to ask of a software architecture and to those that use quantitative measurements to be taken from an architecture (Abowd et al., 1996).

The different techniques for acquiring data for the qualitative questions are scenarios, questionnaires, and checklists. Scenarios seem to be the most utilized form for acquiring data (Kazman et al., 1996). A scenario is ''a specified sequence of steps involving the use or modification of the system and provides a means to characterize how well a particular architecture responds to the demands placed on it by those scenarios'' (Abowd et al., 1996). Measuring techniques are objective ways of acquiring information about a SA as metrics can be precisely defined and, many times, automatically collected. The development of architectural metrics is relatively new and not yet empirically well established. The work by Shereshevsky et al. (2001) outlines, for example, a set of metrics for software architectures based on coupling and cohesion of architectural components that are theoretically sound, but still untested. Antoniol et al. (1998) is an example of an approach in which metrics are used to identify design patterns (Gamma et al., 1994). What in general makes metrics hard is not the collection of the data, but with the selection of metrics and the interpretation of the data. Interpretation of data from measurements must be compared to pre-defined criteria that are derived from a goal. Antoniol et al. (1998)

use, for example, a conservative interpretation of metrics, meaning that if a pattern exists in the code it will be reported, but false patterns may also be reported. In the same way as the goal can vary from project to project, the interpretation of the data can also vary from project to project. One good way of defining a measurement program is to use the goal-question-metric (GQM) approach (Basili et al., 1994b), which helps derive metrics from questions based on the goals of the measurement program.

1.3. Design Patterns

Design patterns address the problem of maintainability. They are descriptions of communication objects and classes that are used to solve an object-oriented problem or issue. The goal of patterns is to create solutions that can help software developers resolve recurring problems.

One of the 23 design patterns in Gamma et al. (1994) is the mediator pattern. The mediator pattern encapsulates how a set of objects interact, providing loose coupling among the objects as the objects do not reference each other explicitly. The mediator coordinates the interactions between the objects (colleagues). The control flow logic is put in the mediator instead of distributing it to the colleagues. When a colleague needs a service from another colleague, it contacts the mediator instead of contacting the object directly.

The advantages of using this pattern are Gamma et al. (1994):

- The control flow behavior is localized to the mediator. To change this behavior, it is necessary to subclass only the mediator. The colleagues can be reused as is.
- There is no coupling between the colleagues, they don't know about each other.
- All the communication between the colleagues has to pass through the mediator, which makes it easier to understand, maintain and extend the system.
- Encapsulating the control of the communication in one object makes it possible to focus on how objects interact apart from their individual behavior. This can help clarify how the objects interact in a system.

The main drawback of using this pattern is that the mediator object can become very complex and hard to maintain since all of the interactions among the colleagues are encapsulated in the mediator.

2. The Evaluation Process

After redevelopment, we suspected that the new system was an improvement, but how do we evaluate the software architecture? To address this issue, we designed a quantitative comparative case-study based on the concepts of late SAE and the

GQM approach. A generalized description of the process we followed for the study is composed of the following steps:

- Select a perspective for evaluation.
- Define/select metrics and establish guidelines to be used in the evaluation.
- Collect metrics.
- Evaluate/compare the architectures.

Figure 1 illustrates a more detailed view of the measurement/evaluation process used in this study. In order to evaluate and compare the architecture of two versions of a system, we reverse engineered the source code from both versions to obtain the actual architectural designs. We compared the structural differences between the actual designs of the two versions with respect to our selected perspective, maintainability. Then, we compared the actual design of the new version of the system with the design specified in the planned design and our design goals for the system. In this part of the evaluation, we looked for discrepancies between the two designs to find problems with the actual implementation. Then, with each discrepancy, we identified design goal violations.

The next section describes our selected perspective of evaluation and the definition of the metrics. Section 4 provides the results of the evaluation of the architecture of the two versions of our system.

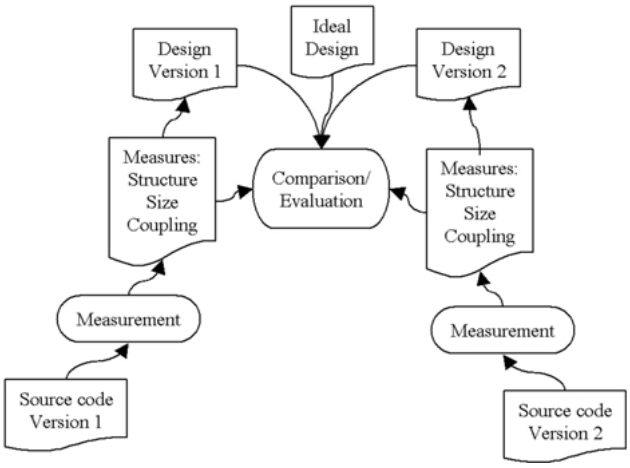

Figure 1. The process for measurement and evaluation in the study.

3. The Study

The study objects are two versions of the EMS, the original version (EMS1) and the version incorporating the new requirements (EMS2). EMS is a client/server system with most of the new functionality and redesigning efforts residing in the client. The general structure of the EMS is shown in Figure 2.

EMS1 is a research prototype developed by two people. There are about 10 K lines of non-commented Java source code (NLOC). Documentation of EMS1 is limited to the comments in the source code and a workshop paper (Seaman et al., 1999) describing the functionality of the system. There was no design document. At the time of the development of EMS1, no data regarding the development of the system was collected. EMS2 represents a significant change from EMS1. EMS2 contains over 15 K lines of non-commented Java source code developed by four local developers plus a team of three developers in Brazil. Twenty-two new requirements were added. There are requirements and design documents for the new system as well as implementation documentation for each component of the system. The characteristics of both versions are summarized in Tables 1 and 2.

Our goal for the study formulated according to the GQM template was to:

Analyze the two versions of the system and the planned design for the purpose of evaluation (by comparing) with respect to maintainability from the point of view of software development in the context of EMS at FC-MD.

3.1. Selecting a Perspective for Evaluation: Maintainability

A system can be evaluated with different goals in mind and from many different perspectives. An evaluation can be based on whether the system implements the specified functional requirements or, more suitable for an architectural evaluation, whether it fulfills the non-functional requirements, i.e., the system qualities or quality attributes. The main goal of the restructuring of EMS was to increase the maintainability, i.e., make the system easier to change. We expected the future changes to the system to include the addition and modification of features. We also

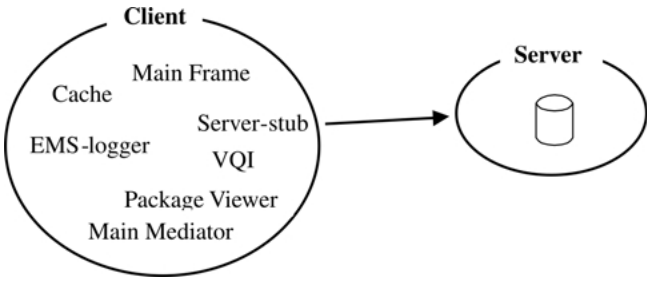

Figure 2. Structure of EMS2.

Table 1. Characteristics of both versions.

Version	Total size (NLOC)	Total developers
EMS1	9,831	2
EMS2	15,382	7

Table 2. Size information (NLOC) for both systems.

	Size			
Version	Client	Server	Common	Total
EMS1	8,405	1,175	251	9,831
EMS2	12,957	1,827	598	15,382

expected future development to be done by different and possibly distributed sets of programmers. The goal was to make it easier to add and modify functionality to the system in the future and to support distributed development. By looking at the maintenance process, one understands better what properties make a system easier to maintain. The process of maintenance can be broken down into the following sub-processes:

1. Understanding the change request.

2. Understanding the system and its structure.

3. Localizing where to change the system in order to implement the change request (primary changes, Lindvall, 1997).

4. Implementing the primary changes.

5. Determining the ripple effects (secondary changes resulting from primary changes, Lindvall, 1997).

6. Implementing the secondary changes

7. Testing that the system fulfills all previous as well as new requirements

Due to the fact that the maintenance process is iterative, the stated order above indicates steps to be taken rather than the order they always occur.

Many studies indicate that there is only a vague connection between the requirements of a system and the structure of the system (Soloway, 1987). This vague connection makes it hard to conduct steps one to three above because they are dependent on each other. It is not possible to fully understand a change request without understanding the system, its structure, and where in the system to make the change. In the same way, it is hard to determine ripple effects without

implementing the primary changes, and the implementation of the secondary changes might cause new ripple effects necessitating another set of secondary changes and so on.

One way of closing the gap between the requirements and the architecture and increase maintainability is to use a component-based approach. Each component implements a specific and related set of requirements making the connection clearer. To exemplify: In the EMS-project the architecture was previously library-based, meaning that classes that provided the same kind of functionality were located together in the same module. For example, all classes related to the user-interface could be found in one branch of the directory structure in the file system. A component, in our terms, implements a well-specified user-oriented functionality. This means that classes that implement a certain feature are located together (in our case, a package) and that it is easier to trace from a requirement to its implementation.

Ripple effects cause problems related to coupling between the pieces that make up the software system, e.g., modules or components (Haney, 1972; Yau and Collofello, 1980). Both Collofello and Haney show that tighter coupling increases the risk of ripple effects. Thus, reduced coupling between components reduces the risk of ripple effects, change propagation and the generation of secondary changes as a consequence of primary changes. The conclusion is that reduced coupling results in systems that are easier to maintain.

As described earlier, maintainability can be viewed in many ways and depends on many different factors. Code that is written according to a set of well-defined design rules and guidelines is easier to understand, and therefore easier to maintain. Low complexity, low coupling and high cohesion in theory indicate a higher level of maintainability. A system that is easier to understand, change, and test is easier to maintain. A system with well-structured, clearly defined, subsystems that is well documented and uses clear and logical names is easier to maintain. A smaller system with equivalent functionality is generally easier to maintain than a bigger system. A system that offers fewer features is easier to maintain.

While all of these characteristics indicate maintainability, some of them cannot be assessed until the system has been operational and undergone changes. Since the new version of EMS has just been implemented, for this study, we focus on those aspects of maintainability that do not require an operational profile. In particular, we examine the structure of both systems. Within this context, a maintainable system is one that has well-structured and clearly defined components. Coupling between those components should be low. With EMS, we wanted to reduce the amount of inter-module coupling without increasing internal coupling, or intra-module coupling.

To summarize, we addressed the problem of the vague connection between functional requirements and the structure of the system by forming components. We used the mediator pattern as the basis of our architectural design to reduce the coupling among the system's components and to make it clearer where a new component should be attached to the system in the future.

3.2. Defining and Selecting Metrics and Establishing Guidelines

The study is based on the notion of static couplings between modules and classes detected in the source code. It should be noted that we use the words object and class interchangeably. We use the word module to denote intentional clusters of classes. The word module also refers to such clusters of classes in EMS1, while the word component refers to clusters of classes in EMS2.

Initially, we had intended to use coupling between objects (CBO) as defined in Chidamber and Kemerer (1994) and implemented in several measurement tools. Using a standard implementation would automate the measurement process and significantly reduce our effort. The problem we had with using CBO is that the common implementation of the metric treats all objects the same and does not distinguish between objects with different characteristics, such as the module where they are defined. CBO would, for example, include the coupling among the objects inside of components. We wanted to exclude this intra-module coupling for part of our study, as we were interested in examining inter-module coupling separately from intra-module coupling. To study inter-module and intra-module coupling separately, we defined a set of new metrics based on the ideas of CBO.

3.2.1. Inter-module Coupling Metrics

To capture inter-module coupling at the module level, we define a metric called coupling-between-modules (CBM). The CBM metric is the number of non-directional, distinct, inter-module references. CBM is a coarse measurement that describes the coupling between modules. With CBM, we ignore the direction of the arrows because we want to capture the number of modules that are involved in the coupling. This metric captures inter-module coupling and ignores intra-module coupling. Importing a class from a module, inheriting from a class in a module, and declaring or instantiating an object from a module are included as forms of coupling in the metric. We define CBM(m) as the number of modules coupled with the module m.

In order to capture the degree of coupling between modules at a finer level, we define the metric coupling-between-module-classes, CBMC(m). The CBMC metric is the number of non-directional, distinct, inter-module, class-to-class references for a module m. This measure captures the number of classes involved in the module coupling.

To illustrate these metrics, consider the example of an architectural design depicted in Figure 3. In this example, an arrow indicates a directional relationship between two classes (e.g. Class A in Module X creates an instance of Class D defined in Module Y. This coupling leads to the arrow from Class A in Module X to Class D in Module Y).

In measuring inter-module coupling, the intra-module couplings such as the arrow from Class A to Class B in Module X in Figure 3 are ignored. Additionally, only

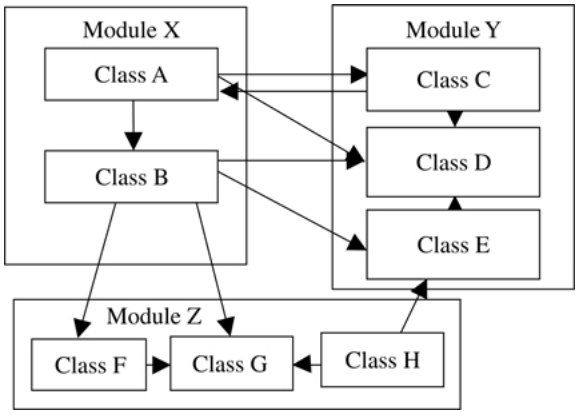

Figure 3. An example of architectural design. Arrows indicate coupling between the classes.

distinct couplings between the modules are considered. Figure 4 illustrates the distinct inter-module connections of the architectural design shown in Figure 3.

To calculate the CBM values, the direction of the arrows is ignored. In our example, CBM(X), CBM(Y) and CBM(Z) all have a value of 2.

When calculating the CBMC metric, the number of inter-module class references is considered. For each inter-module arrow depicted in Figure 4, the number of class references is determined. Figure 5 shows the inter-module class references for each arc. For example, the classes in Module X reference classes C, D and E in Module Y. Hence, the arrow from Module X to Module Y in Figure 5 is labeled 3. To calculate CBMC, the directions of the arrows are ignored. In our example, CBMC(X) = 6, CBMC(Y) = 5 and CBMC(Z) = 3.

We could define coupling metrics at finer levels of detail, e.g., by counting each reference from one module to another, but for our purposes (architectural evaluation), the extra level of detail was deemed unnecessary.

Recognizing different kinds of modules. In our study, it was important to recognize that different kinds of modules need to be treated differently. By definition, library-based modules are collections of classes intended to be used by many other modules.

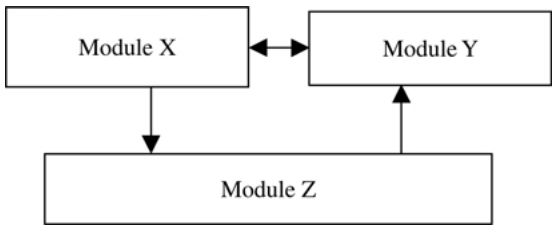

Figure 4. Inter-module couplings of example of architectural design.

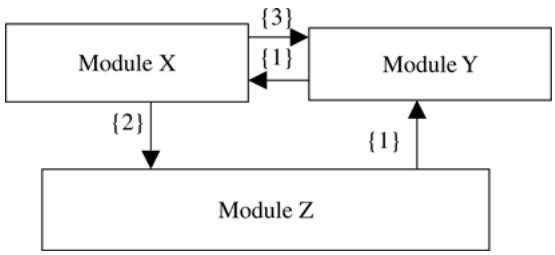

Figure 5. Inter-module class references for the example of architectural design.

Therefore, we would expect other modules to use many classes within a library-based module. To separate this effect, we define $CBMC_{all}$ and $CBMC_{nolib}$ where $CBMC_{all}$ represents class coupling between all modules while $CBMC_{nolib}$ shows class coupling between modules excluding classes from library modules. Since a library is a group of related functions that will be used possibly by many components, we would expect a library-oriented module to be highly coupled with the other modules in the system. That is, we would expect the other modules in the system to use classes and call methods from a library module. However, a coupling from a library module to another non-library module of the system would be an example of an undesirable coupling. Since a component-based module (or non-library module) is a group of related classes, we would expect the module have a low number of couplings outside of the component.

In both EMS1 and EMS2, the Common module is the only library-based module. We handle the couplings to and from the Common module separately since library-based modules are inherently different from component-based modules.

In summary:

- Coupling to the library-based modules is acceptable.

- Coupling from library-based modules to non-library based modules is undesirable.

A summary of the metrics representing coupling between modules (CBM) and coupling between module classes (CBMC) is provided in Table 3.

To conduct deeper analyses of findings, we not only measured the coupling, but also kept lists where the entries represent the couplings and names of classes and modules are preserved. For example, to capture the relationship between class B in module X and class E in module Y shown in Figure 3, the information in Table 4 is stored for later analysis.

The coupling information is stored in database tables for easy search and retrieval. Once all of the coupling information has been stored in the tables, the actual architecture of the system can be visualized in diagrams similar to those shown in Figure 4 and 5.

Table 3. Summary of metrics.

Guideline	Metric	Explanation
Coupling between modules should be low.	CBM(m)	The count of the number of modules coupled to or from m.
	CBM_{hist}	A histogram of the CBM number for each module.
Coupling between module classes should be low.	$CBMC_{all}$(m)	The count of the number of classes coupled to or from m.
	$CBMC_{nonlib}$(m)	Same as $CBMC_{all}$, but class couplings in library modules are ignored.
	$CBMC_{hist}$	A histogram of the class couplings ($CBMC_{nolib}$) for each module.

Table 4. Relationship between classes and modules.

From module	From class	To module	To class
X	B	Y	E

3.2.2. Intra-module Coupling Metrics

In addition to the metrics defined for inter-module coupling, a set of intra-module coupling metrics was defined. When modifying the structure of the system, we did not want to reduce inter-module coupling at the expense of significantly increasing intra-module coupling. To capture intra-module coupling, we defined a metric called coupling inside a module, CIM. To calculate CIM(m) for a module m, a coupling metric for each class within the module must be calculated. CIM(m) is the average of the coupling metric for the classes within the module m. The coupling metric we use for each class is calculated in an analogous way to the CBM for modules. This analogous metric is called coupling between classes, CBC. CBC(c) is the coupling between classes for Class C.

To illustrate how the intra-module coupling metrics are calculated, consider the example module shown in Figure 6. Again, an arrow between classes represents a coupling between the two classes. First, we calculate the coupling between classes metric for each class within the module; $CBC(A) = 2$, $CBC(B) = 1$ and $CBC(C) = 1$. To obtain the intra-coupling metric for the module, we take the average of the CBC values for the classes inside of module X. The $CIM(X) = 1.33$.

We used the intra-coupling measures in our comparison of the two versions of the system. In our efforts to redesign the system, our emphasis was on inter-module coupling. However, we did not want to reduce inter-module coupling by merely ''pushing'' the coupling into the modules. We thought that the old version of the system would have more unrelated (and therefore uncoupled) classes within the same module. While we expected that the intra-module coupling might increase with the new version, we did not expect it to increase significantly. We expected the new

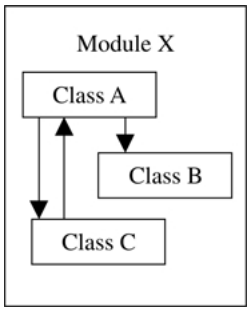

Figure 6. An example module used to illustrate intra-module coupling.

version of the system to have nearly the same amount of intra-module coupling as the old version. To evaluate the intra-module coupling, we used box plot diagrams as described in the next section.

3.3. Evaluating and Comparing the Architectures

We used the measurement/evaluation process shown in Figure 1 using the two versions of EMS. We established evaluation criteria for comparing the two versions of the system and established design goals based on the new design.

3.3.1. Comparing the Actual Designs of the Two Versions

In order to evaluate the two versions of the system, evaluation criteria were established before we examined the actual data. Using our model of maintainability, we created measurable evaluation criteria for the actual designs extracted from the two versions of EMS. Block diagrams of the actual designs of EMS1 and EMS2 are shown in Figures 7 and 8, respectively. These diagrams show the coupling among the actual modules of the system. In the design of EMS1, all modules with the exceptions of the Server and Common modules represent modules of the client (see Figure 2). The Server module is the only module of the server. The Common module is part of (used by) both the client and the server.

EMS2 was designed with the mediator design pattern as its base. The Main Mediator module is responsible for directing all communications on the client side. The Server Stub module is designed to be the only component communicating with the server side of the system. The Common module is a library-based module that holds classes forming the data structures.

For the evaluation of the two actual designs, we are interested in inter and intra-module coupling. In the context of EMS, we would like to see a design that has few couplings between the component-based modules of the system while not increasing intra-module coupling significantly.

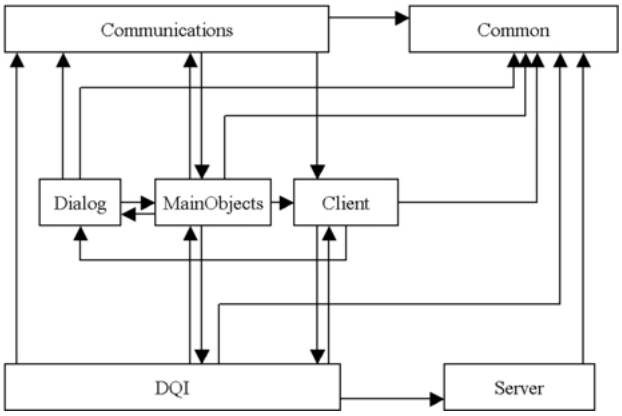

Figure 7. Actual design of EMS1.

With these criteria and measurements in place, we compared the actual design of EMS2 against both the planned design for EMS2 and the actual design of EMS1. When comparing EMS1 and EMS2, we evaluated the module and class coupling measures. We expected the modules in EMS1 to have high CBM and CBMC numbers. In EMS2, we expected the couplings to decrease significantly. We expected the number of modules with high CBM and CBMC to be lower in EMS2. For intra-module coupling, we expected the CIM numbers for EMS2 to be slightly higher, but not significantly higher than the CIM numbers for EMS1. We summarize our expectations in Table 5.

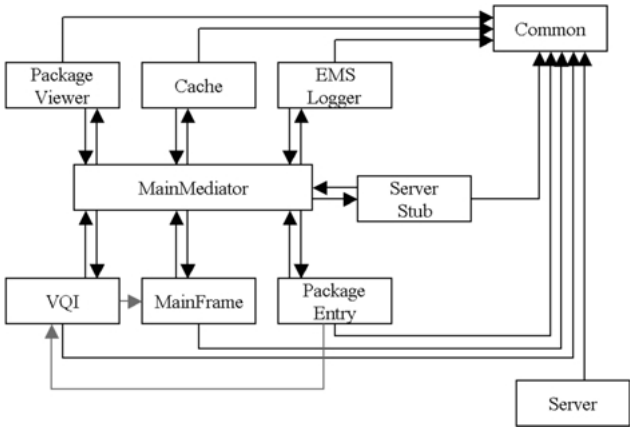

Figure 8. Actual design of EMS2.

Table 5. Summary of expectations for evaluation of the two actual designs.

Expectation	Explanation
CBM_{hist}(EMS2) better than CBM_{hist}(EMS1)	In EMS1, most modules will be highly coupled. In EMS2, most modules will have low CBM, less than or equal to two. The only exceptions will be the Mediator and Common modules.
$CBMC_{hist}$(EMS2) better than $CBMC_{hist}$(EMS1)	In EMS1, most modules will have high CBMC. In EMS2, most modules will have low CBMC, less than or equal to 10. The only exceptions will be the Mediator and Common modules.
CIM(EMS2) is nearly the same as CIM(EMS1)	In EMS1, modules will contain unrelated classes leading to lower CIM numbers. In EMS2, modules will contain related classes leading to higher, but not significantly higher CIM numbers.

3.3.2. Comparing the Actual Design of EMS2 to the Planned Design

The comparison of the actual design of EMS2 against the planned design for EMS2 is based on using the detailed information we kept. When comparing EMS2 against the planned design, we used the metrics as indicators of potential differences and used the detailed coupling information to search for violations of our design goals and for identifying discrepancies between the two designs. Since we were interested mainly in inter-module coupling for this study, our design goals are related to inter-module coupling issues.

The first step was to define violation indicators that can be identified based on the differences in the directed graphs of both designs. There are two general violation indicators to consider:

- [VI1] A reference in the planned design that does not exist in the actual design.
- [VI2] A reference in the actual design that does not appear in the planned design.

For each of these cases, we examined the code and documentation to determine why the discrepancy occurred.

The actual design of EMS2 is shown in Figure 8. The planned design for EMS2 is illustrated in Figure 9.

The design of EMS2 is based on the mediator design pattern. Hence, we have certain design goals that are expected to be preserved in our implementation of the pattern:

- [DG1] The ServerStub of the Client should contain all of the references to the Server (since the ServerStub component is the means of communication with the server).

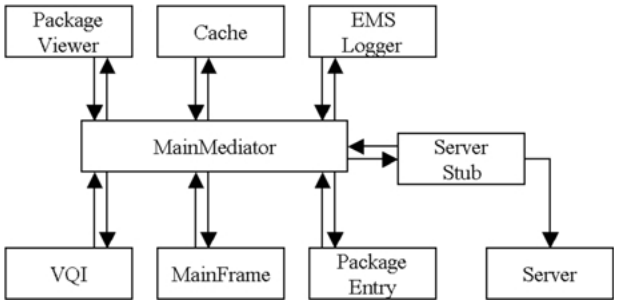

Figure 9. The planned design for EMS2.

- [DG2] The Server should contain no references to the Client (since the server does not initiate communication with the clients).
- [DG3] Ideally, any non-mediator component should only reference the Common and the Mediator components. Since the ServerStub is the link to the Server, it will be the only exception to this rule. It will communicate with the Mediator and the Server.
- [DG4] The Mediator should be coupled with all the components in the client.
- [DG5] No Client component, with the exception of the Mediator, should contact the ServerStub directly.
- [DG6] The Mediator should be coupled with exactly one class per component and vice versa.

We expect that the number of discrepancies between the actual and planned designs will be small, since the developers of EMS2 were working from a document describing the planned design. The results from the analysis are discussed in the next section.

4. Results

In this section, we present and discuss the results of the two evaluations: the comparison between the actual designs of EMS1 and EMS2; and the comparison of the actual design of EMS2 against the planned design for EMS2.

4.1. Comparison Between the Two Actual Designs

The coupling measures for the modules of EMS1 and EMS2 are given in Tables 6 and 7, respectively. The violations to general rules are highlighted in bold and the specific violations to EMS2 are highlighted in italic. For the CBMC of both versions,

Table 6. Coupling measures for EMS1.

Module	CBM	CIM	$CBMC_{all}$	$CBMC_{nolib}$
Client	**5**	1	10	9
Common	**6**	1	31	N/A
Communications	**5**	1.7	30	**22**
Dialog	4	1.3	18	**16**
DQI	**5**	4.5	39	**26**
Main Objects	**5**	1.6	41	**39**
Server	2	2	7	2

we give the class coupling measures with and without the Common module. Since the Common module is a library-based module, it is expected that other modules will access many classes in the Common module resulting in high but desirable $CBMC_{all}$ numbers for those non-library modules that use the Common module. Therefore we do not use the metric $CBMC_{all}$ in our comparison. The CBM numbers presented include module couplings to the Common module. If we exclude the couplings to the Common module, the CBM(m) values of each module would be decreased by one. In our analysis, however we use the CBM(m) numbers that include the module couplings to the Common module.

We now compare expectations to results.

Expectation: $CBM_{hist}(EMS2)$ better than $CBM_{hist}(EMS1)$. We expected the number of components with high coupling between modules to be much lower in EMS2 than in EMS1. The histogram of the CBM for EMS1 and EMS2 is included in Figure 10. Figure 11 shows the structure of couplings in EMS2. Looking at the histogram we can see that EMS1 has more classes with higher CBM than EMS2. However, EMS2 contains two modules that have CBM of 8. Those two modules are the Common and Main Mediator modules. We expected those modules to have high coupling. It is clear that we have reduced the number of highly coupled modules with EMS2, even though the system grew considerably and the number of modules increased from seven to 10. To further visualize the coupling, we use the box plot

Table 7. Coupling measures for EMS2.

Module	CBM	CIM	$CBMC_{all}$	$CBMC_{nolib}$
Cache	2	0	9	3
Common	**8**	2	85	
EMS Logger	2	1	4	3
Main Frame	*3*	3.9	14	7
Main Mediator	**8**	1	37	**25**
Package Entry	*3*	1	13	6
Package Viewer	2	4.3	27	4
Server	1	2.25	12	0
Server Stub	2	0	4	3
VQI	*4*	2.9	23	7

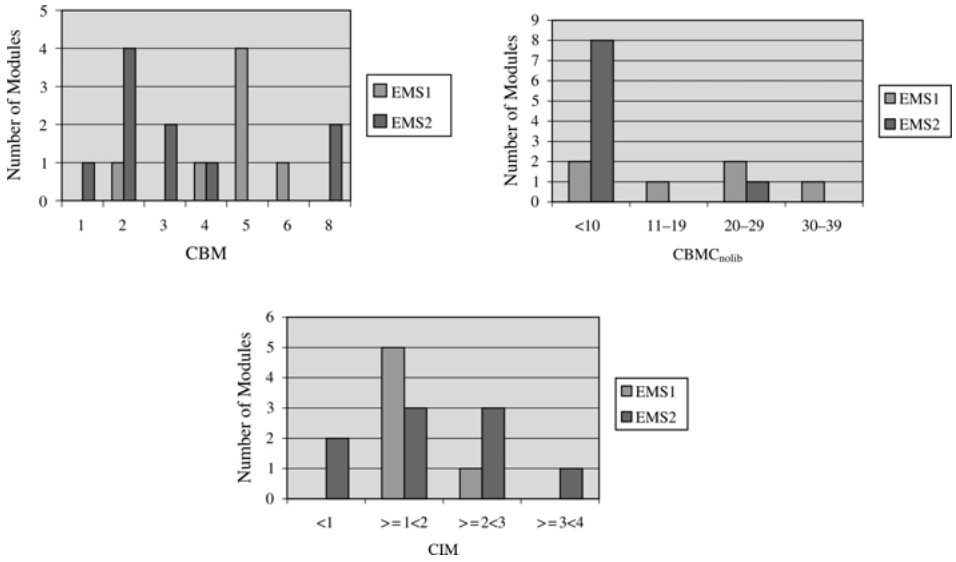

Figure 10. Histograms of CBM, CBMC$_{nolib}$ and CIM.

technique; see Figures 12 and 13. The median CBM value is 5 for EMS1 and 2.5 for EMS2. The box plot of CBM values for the planned design for EMS2 shows an expected median of 2 (see Figure 14) which is considerably lower than EMS1 and slightly lower than EMS2. This difference is not statistically significant, however, because of the outliers represented by the Main Mediator and the Common modules.

Expectation: CBMC$_{hist}$(EMS2) better than CBMC$_{hist}$(EMS1). The histogram of the CBMC$_{nolib}$ for EMS1 and EMS2 is included in Figure 10. Looking at the histogram we can observe that most modules from EMS2 have CBMC$_{nolib}$ lower

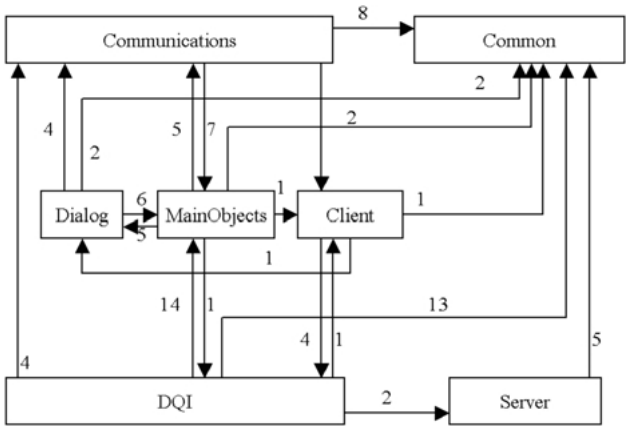

Figure 11. Structure of the couplings in EMS2.

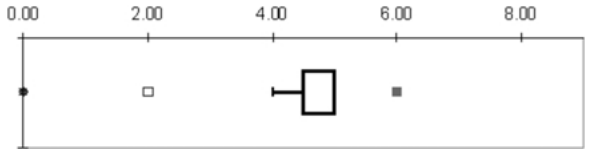

Figure 12. CBM for EMS1. Median = 5.

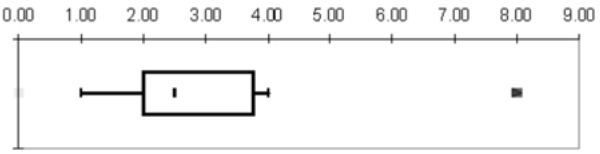

Figure 13. CBM for EMS2 (actual). Median = 2.5.

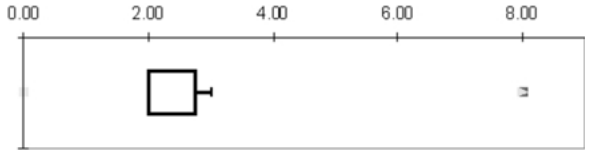

Figure 14. CBM for EMS2 (planned). Median = 2.

than 10. Eight of the ten modules in EMS2 have $CBMC_{nolib}$ less than 10. The other two modules are the Main Mediator and the Common modules. Common is not included in the histogram since we are not considering couplings with the common modules. In EMS1, only two out of the six modules have $CBMC_{nolib}$ less than 10. In EMS1, the communication among the modules is scattered, not localized. The low $CBMC_{nolib}$ numbers with EMS2 demonstrate that we have localized the communications among modules in EMS2.

Expectation: CIM(EMS2) is nearly the same as CIM(EMS1). The histogram of the CIM for EMS1 and EMS2 is included in Figure 10. With the new architecture, since modules are component-based, we expected to have slightly more coupling of classes inside the modules. The classes were grouped in modules based on their functionality. When we grouped the classes in modules, we expected to be moving some of the couplings between the modules to couplings inside the modules. The histogram shows that EMS1 has indeed more modules with fewer class couplings than EMS2, but the median value is actually slightly higher for EMS1 (1.6) than for EMS2 (1.5). Figures 15 and 16 show respectively the box plots with CIM values for EMS1 and EMS2. The box plots show that the difference is not significantly large; i.e., despite the fact that the intra-module coupling decreased for EMS2, it did not decrease significantly. The difference between the medians is not statistically significant. One reason for the decrease could be the addition of new components with only one class.

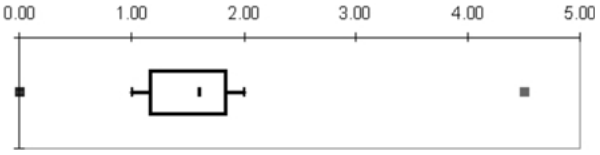

Figure 15. CIM for EMS1. Median = 1.6.

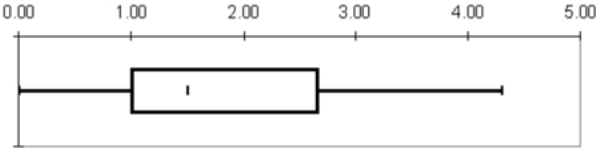

Figure 16. CIM for EMS2 (actual). Median = 1.5.

4.2. Comparison of the Actual Design of EMS2 against the Planned Design for EMS2

Using the evaluation technique, we found that the actual design nearly matched the planned design, but there were a few problems. The evaluation quickly highlighted the problems and allowed us to determine how to fix those problems. The CBM and CBMC measurements give indications of the problems with the actual design. According to our design goals, only two components, the Common and MainMediator components should have a CBM higher than 2. However, there are three components that have a higher CBM than expected. Using the database tables that store the detailed coupling information we were able to find the actual design violations quickly. The design goals DG2, DG4 and DG5 were not violated. We found three violations that are explained in the following paragraphs (see Figure 8). These violations are highlighted in italic in Table 7.

There is one coupling from the Server Stub to the Server in the planned design that does not exist in the actual design of EMS2 (VI1, DG1 and DG3). Recall that the Server Stub module is the means of communication between the client and the server of EMS. In the actual design, there is no coupling between the Server Stub and the Server. This communication is handled through an RMI interface. Upon examination of the code, we found that the RMI interface was mistakenly placed in the Common module. It should have been placed in the Server module and implemented by the Server Stub module. By placing the RMI interface in the Server module, one of the couplings from the Server to the Common module would be eliminated and the coupling from the Server Stub to the Common module would move to the Server module, as designed.

The actual design of EMS2 contains two couplings that do not exist in the planned design. The first coupling is from the VQI module to the Main Frame module. The second coupling is from the Package Entry module to the VQI. Both of these couplings are considered undesirable (VI2). When we examined the code for these couplings, we found that the coupling between the VQI and Main Frame was not a

true coupling. The VQI module imports the classes from the Main Frame module, but does not use them. This coupling is an artifact from the previous version of EMS and should be removed. The coupling between the VQI and the Package Entry is a true coupling: the classes of the Package Entry use exception definitions defined by the VQI. Thus, it is a violation of our planned design (DG3).

There is an extra coupling from each module to the Mediator module that is not in the planned design (VI2). In the implementation of the mediator pattern in EMS2, each class that communicates with the Mediator module extends a class called MediatorObject that implements the MediatorObjectInterface. These class and interface definitions (MediatorObject and MediatorObjectInterface) are part of the Mediator module, and explain the second coupling from the components to the Mediator. This is a violation of the planned design (DG6). These extra couplings could be eliminated moving the classes MediatorObject and MediatorObjectInterface to the Common module. However, we felt that it was best to leave them in the Mediator module and modify our design goal.

5. Summary and Conclusions

In 1999 we identified maintainability problems with the architecture of one of our main assets: the experience management system (EMS) written in Java. In an effort to increase maintainability and to facilitate distributed development, we based the new architecture on components and the mediator design pattern. During this restructuring we also implemented a set of new requirements. It was obvious from looking at the system that it had changed structure and grew considerably as an effect of the restructuring and the implementation of new requirements. We were however interested in measuring this change in order to evaluate in architectural terms whether we were better off or not. One reason was to be able to show management that the new architecture had improved.

We defined two metrics based on inter-module couplings (CBM(m) and CBMC(m)). The metrics were used in order to measure the level of interrelationship between the modules in the two versions of the system. More fine-granular coupling metrics are available in the literature, but we decided that for architectural evaluation, these two metrics were the correct granular level to use. The metrics can be used to demonstrate to management that an existing system has architectural problems and is in need of restructuring. Once the restructuring is complete, the metrics can be used to verify the design goals and to show that the structure of the architecture has improved. We used the metrics as indicators of potential problems with the revised system. Then, we investigated further using more detailed information.

We also defined a third metric to measure intra-module coupling (CIM(m)). When we modified the structure of the system we did not want to reduce the inter-module couplings at the expense of significantly increasing intra-module coupling. CIM was used to show that there was actually some decrease in the intra-module coupling but the decrease was not significantly higher. We expected the intra-module coupling to

increase slightly since the new system was based on functional components. Thus, we expected classes within the modules of the new system to be more coupled than those of the old system. With the slight decrease, we believe the system is more maintainable since the classes are grouped into modules based on their functionality. When new developers join the team, they can start working on one piece of functionality without having to understand all of the code upfront.

The study was based on two different kinds of comparisons. We compared the architectural design of the new version with the architecture for the old version of the system and with the planned architecture for the new version. The former comparison used mainly the metrics, while the latter was mainly based on evaluation according to our design goals.

The comparison and evaluation show that we now have an architectural design with independent components that are only loosely coupled to each other. The first observation is that while the structure of the old version of the system indicates that almost every module interacts with every other module, the new structure indicates that all components, except for the mediator and the common library, are very loosely coupled (high vs. low CBM). The second observation is that while the structure of the old version indicates that communication between classes in different modules is relatively scattered, the new structure indicates that communication now is localized to a few classes (high vs. low CBMC).

The modules of the new version are components, which easily map to the functional requirements of the system. This mapping makes it easier to find where end-user functionality is implemented in the code. The evaluation of the new architecture as compared to the planned architectural design shows that there are still a few problems. Even though the problems are minor, they clearly violate our design goals. They are potential threats to the maintainability of the system and should be fixed in the next release of the system. The fact that we were able to detect these architectural problems illustrates that our method is good for identifying undesirable interactions between modules in a system.

In order to implement the new architecture, we turned our library-oriented modules into components with specific functions. We created a mediator module to manage and coordinate the interactions of the components. The definition of how the communication among the components and the mediator was to be implemented took some time for developers to understand, and was not easily accepted by the developers at first. But, eventually the mediator pattern was very useful. Due to the fact that the modules were component-oriented and that the architecture was based on the mediator design pattern, geographically distributed development was enabled. The components were assigned to developers that could implement and test them independently, even before the other components were ready. Developers did not need to know about the interface of the other objects or what the other objects were. The integrator implemented the mediator module that provided the communication among the components.

From our evaluation, it appears that the current architectural design is well structured and the components only loosely coupled and therefore should be more maintainable than the previous version. The mediator design pattern helped to

enable distributed development. Future studies of further evolution of this system are needed to determine the maintainability of this design in practice. The process used to evaluate the architectural structure was helpful in highlighting the progress of the system and identifying areas for improvement. This evaluation helped us convince management that the architecture had improved and that we had been able to stop architectural degeneration. The evaluation process is general and can be used in different, object-oriented contexts. We believe that using this and similar approaches gives technical staff the tools needed to convince upper management that stopping architectural degeneration is worthwhile.

The study object is a relatively small system and in order to scale up and apply this process and metrics to a larger system, some tailoring would be necessary. Examples are automatic generation of reports and diagrams pointing out differences and problems in the architecture would make the approach applicable to larger systems. Applying this process and metrics to larger systems would be a good idea in order to validate the ideas and results presented in this paper.

Acknowledgments

We would like to thank Dr. Marvin Zelkowitz for reviewing this paper, Dr. Manoel Mendonca for developing the EMS, Students at University of Maryland for enhancing EMS. Dave Cohen for developing the analysis tool and Dr. Victor Basili for supporting this work.

References

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., and Zaremski, A. 1996. Recommended best industrial practice for software architecture evaluation. *Software Engineering Institute*, CMU/SEI-96-TR-025

Antoniol, G., Fiutem, R., and Cristoforetti, L. 1998. Using metrics to identify design patterns in object-oriented software. *Proceedings of the Fifth International Symposium on Software Metrics—METRICS'98*, 23–34.

Avritzer, A., and Weyuker, E. J. 1998. Investigating metrics for architectural assessment. *5th International Symposium on Software Metrics* 4–10.

Basili, V. R., Caldiera, G., and Rombach, D. H. 1994a. The experience factory. *Encyclopedia of Software Engineering*, 2 volume set, pp. 469–476.

Basili, V. R., Caldiera, G., and Rombach, D. H. 1994b. The goal question metric approach. *Encyclopedia of Software Engineering*, 2 volume set, Wiley.

Bengtsson, P., and Bosch, J. 1999. Architecture level prediction of software maintenance. *Proceedings of 3rd EuroMicro Conference on Maintenance and Reengineering*, 139–147.

Chidamber, S. R., and Kemerer, C. F. 1994. Towards a metrics suite for object-oriented design. *Transactions on Software Engineering*, 476–493.

de Bruijn, H., and van Vliet, H. 2001. Scenario-based generation and evaluation of software architectures. *Lecture Notes in Computer Science* 2186: 128–139.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 27: 1–12.

Fiutem, R., and Antoniol, G. 1998. Identifying design-code inconsistencies in object-oriented software: a case study. *Proceedings of International Conference on Software Maintenance*, 94–102.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design patterns elements of reusable object-oriented software*. Reading, MA: Addison Wesley.

Haney, F. M. 1972. Module connection analysis—A tool for scheduling software debugging activities. *AFIPS Joint Computer Conference*, 173–179.

Kazman, R., Abowd, G., Bass, L., and Clements, P. 1996. Scenario-based analysis of software architecture. *IEEE Software*, 47–55.

Lassing, N., Bengtsson, P., van Vliet, H., and Bosch, J. 2002. Experiences with ALMA: architecture-level modifiability architecture analysis. *Journal of Systems and Software* 61: 47–57.

Lehman, M. M., and Belady, L. 1985. Program evolution. *Process of Software Change*. London: Academic Press.

Lindvall, M. 1997. An empirical study of requirements-driven impact analysis in object-oriented systems evolution. PhD thesis No. 480, Linköping Studies in Science and Technology.

Meyer, S., Duby, C. K., and Reiss, S. P. 1993. Constraining the structure and style of object-oriented programs. *Brown University Computer Science Technical Report CS-93–12*. First workshop on principles and practice of constraint programming.

Murphy, G., Notkin, D., and Sullivan, K. 1995. Software reflexion models: bridging the gap between source and high-level models. *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 18–28.

Schwanke, R. W. 1991. An intelligent tool for reengineering software modularity. *Proceedings of the 13th International Conference on Software Engineering*, 83–92.

Seaman, C., de Mendonca Neto, M. G., Basili, V. R., and Kim, Y.-M. 1999. An experience management system for a software consulting organization. *24th NASA SEL Software Engineering Workshop (SEW'24)*.

Sefika, M., Sane, A., and Campbell, R. H. 1996. Monitoring compliance of a software system with its high level design models. In *18th International Conference on Software Engineering (ICSE)* 387–397. Los Alamitos, CA: IEEE Computer Society Press.

Shereshevsky, M., Ammari, H., Gradetsky, N., Mili, A., and Ammar, H. H. 2001. Information theoretic metrics for software architectures. *International Computer Software and Applications Conference (COMPSAC 20001)*, IEEE Computer Society.

Soloway, E. 1987. I can't tell what in the code implements what in the specs. In *The Second International Conference on Human-Computer Interaction*, 317–328.

Yacoub, S. M., and Ammar, H. 2002. A methodology for architectural-level reliability risk analysis. *IEEE Transactions on Software Engineering* 28: 529–547.

Yau, S., and Collofello, J. 1980. Some stability measurements for software maintenance. *IEEE Transactions on Software Engineering* 6.

Mikael Lindvall is a scientist at Fraunhofer Center for Experimental Software Engineering, Maryland. He specializes in work on software architecture and impact analysis as well as experience and knowledge management in software engineering. He is currently working on defining cost-efficient approaches to evaluate software architectures. He received his PhD in computer science from Link—pings University,

Sweden. His PhD work was based on a commercial development project at Ericsson Radio and focused on the evolution of object-oriented systems.

Roseanne Tesoriero Tvedt is an Assistant Professor in the Department of Mathematics and Computer Science at Washington College in Chestertown, Maryland and a Scientist at the Fraunhofer Center for Experimental Software Engineering in College Park, Maryland. She received a PhD in Computer Science from the University of Maryland. Her research interests include software architecture evaluation, agile methods, and computer science education.

Patricia Costa is a scientist at the Fraunhofer Center for Experimental Software Engineering, Maryland. She has a BSc (1996) and a MSc (1999) in Computer Science from the Federal University of Minas Gerais, Brazil and a MSc (2001) in Telecommunications Management from University of Maryland University College. She has experience in software development and in the areas of knowledge management and evaluation of software architectures. She is currently interested in using evaluation of software architectures as a tool to assess/assure quality attributes like security and maintainability of software systems.