
Research


Observe-mine-adopt (OMA): an agile way to enhance software maintainability



Jane Huffman Hayes^{*,†}, Naresh Mohamed and Tina Hong Gao


University of Kentucky, 301 Rose Street, Hardyman Building, Lexington, KY 40506-0495, U.S.A.

SUMMARY

 introduce the observe-mine-adopt (OMA) paradigm that assists organizations in making improvements their software development processes without committing to and undertaking large-scale sweeping organizational process improvement. Specifically, the approach has been applied to improve software practices focused on maintainability. This novel approach is based on the theory that software teams naturally make observations about things that do or do not work well. Teams then mine their artifacts and their recollections of events to find the software products, processes, metrics, etc. that led to the observation. In the case of software maintainability, it is then necessary to perform some measurement to ensure that the methods result in improved maintainability. We introduce two maintainability measures, maintainability product and perceived maintainability, to address this need. Other maintainability measures that may be used in the mine step are also examined. Finally, if the mining activities lead to validated discoveries of processes, techniques or practices that improve the software product, they are formalized and adopted by the team. OMA has been studied experimentally using two project studies and a Web-based health care system which is maintained by a large industrial software organization. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: software maintenance; software maintainability; maintainability measurement; process improvement; experimental study; agile methods

1. INTRODUCTION

 observe-mine-adopt (OMA) paradigm introduced in this paper originated from common sense business practices that are also applicable to software development and maintenance. The OMA paradigm is being used to discover ways to build more maintainable software systems. OMA presents a

*Correspondence to: Jane Huffman Hayes, Computer Science, Laboratory for Advanced Networking, University of Kentucky, 301 Rose Street, Hardyman Building, Lexington, KY 40506-0495, U.S.A.

†E-mail: hayes@cs.uky.edu



way to make tactical or 'point' process improvements without committing to large scale, overarching, sweeping process improvement undertakings. However, it should be noted that OMA can also be used as a subset of an overarching process improvement effort.

Observation is a keen human skill, learned at an early age. Long before children understand how to set goals, build plans or pursue objectives, they understand how to observe and learn based on what they see and experience. Many large software organizations have been striving to improve their software processes of late. These efforts have been aimed at improving the quality, reliability and maintainability of the developed software. Organizations are also seeking to establish repeatability of the process to ensure their survival by qualifying them to bid on contracts that require a proven software development maturity. The capability maturity model integrated (CMMI) team [1,2], ISO 9001 [3] and SPICE (now ISO/IEC TR 15504 [4]) have all supported these efforts by providing software process models or guidelines. It appears that headway is being made, but not without a cost. Adopting these models or starting the 'quest for a level 5' (of the staged CMMI) requires a large amount of time, money and effort, and requires an organizational commitment. For smaller companies, agile teams or those companies without the time and resources required to undertake such an expansive task, there must be some other form of improvement. We introduce the OMA paradigm as such an alternative.

1.1. Maintainability as an improvement opportunity

Software is now an integral part of our everyday lives. There are 300 000 lines of code in a cellular telephone, millions of lines of code in an airliner and literally billions of lines of code in the point of sales terminals at our grocery stores, fast food restaurants, department stores, gas stations, as well as in our hospitals, billing departments, etc. During the year 2000 remediation effort, the Gartner Group estimated that globally 180 billion lines of software code would have to be screened [5]. The Internet, with its dependence on software, has also become a part of our daily lives, facilitating communication with each other, shopping, information retrieval and entertainment.

Our increasing dependence on software should be supported by highly reliable, dependable, easily modifiable applications, but this is not the case. The demand for software changes is becoming larger and faster, but we are not able to keep up. We only seem to be able to produce buggy, unreliable software, even if we are producing more of it faster. Web-based software has further exacerbated the problem by decreasing software development cycle times, challenging maintainers to modify running software (operating $24 \times 7 \times 52$) [6], increasing application complexity and performance requirements, etc. Similarly, the increased demand for software and a shortage of adequately trained personnel has further contributed to poor software quality.

Software maintainability is defined as 'the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment' [7, p. 660]. Maintained software refers to software systems or components that are currently in use and are modified (we include perfective, adaptive and corrective maintenance) on a regular basis. Estimates from as far back as 1980 still hold true today; software maintenance typically accounts for at least 50% of the total lifetime cost of a software system [8]. Schach *et al.* found that 53–56% of personnel time is devoted to corrective maintenance [9].

However, all is not lost. We can view maintainability as an opportunity, an opportunity to improve how we build software so that it is more easily maintained. Before the advent of sweeping



software process improvement efforts, managers were making improvements, albeit on a smaller scale. How were they doing this? They did this by observing what worked and what did not and then instituting best practices or processes based on their observations. By formalizing this tried and true management approach, we can improve software maintainability. The OMA approach appears to successfully assist organizations in identifying processes/practices for adoption in order to develop more maintainable software. In the project studies discussed herein, OMA allowed relatively inexperienced students to identify ways to improve the maintainability of their software. It helped a large industrial software developer to place emphasis on the maintainability of software, to identify highly maintainable modules to be emulated, to identify difficult to maintain modules that may be re-designed, as well as to identify best practices for developing easy to maintain software.

1.2. Paper organization

The paper is organized as follows. In Section 2, related work in software maintenance and process improvement is discussed. Section 3 describes the OMA concept. Section 4 discusses the validation of OMA. Finally, Section 5 presents conclusions and directions for future work.

2. BACKGROUND AND RELATED WORK

In this section, related work in the area of software maintainability as well as software process improvement is presented. Section 2.1 discusses software maintenance research. Section 2.2 presents process improvement methods that could be applied to result in more easily maintained software.

2.1. Software maintenance

Many researchers are addressing software maintenance issues including program comprehension, change impact analysis, location and propagation, maintenance processes, maintainability measurement, etc. Program comprehension is the key to allowing modifications to code by people other than the original programmer. Thread analysis [10] has been proposed for understanding object-oriented software, with experimental results showing that such analysis effectively supports program behavior understanding. Also, the proposed testing technique can be exploited to better test an object-oriented system. Dependency analysis, or understanding how components interact with each other to ensure all necessary changes have been made, has also been well researched. For example, Zhifeng and Rajlich present an approach that examines the impact of hidden dependencies on the process of change propagation and an algorithm that warns about the possible presence of hidden dependencies [11].

Many researchers have examined the role that complexity plays in software maintenance. For example, Kafura and Reddy [12] found that there is a strong relationship between software complexity measures and maintenance tasks such as program comprehension, debugging and program modification. They used their measures for effort planning. Kemerer and Slaughter [13] found that frequently repaired modules have high complexity, are large in size and are relatively older. Huffman Hayes and coworkers have examined the role of complexity and in-line documentation in program comprehension [14] and have investigated the use of course projects to validate software maintenance hypotheses [15,16]. Additionally, they hypothesized that it is increased complexity with



each modification that leads to code decay. Along these lines, Banker *et al.* found that complexity does make software harder and more costly to maintain [17], but did not examine the notion of increased complexity with each modification.

A number of metrics have been developed that have been used directly or indirectly to examine maintainability. Chidamber and Kemerer presented a set of metrics for object-oriented software based on Bunge's ontology [18]. These are often referred to as the CK metrics suite. As shown in Table I later in this paper, metrics such as lack of cohesion in methods (LCOM) and response for a class (RFC) were part of this suite. Li and Henry [19] used the CK software metrics, as well as several of their own, to measure maintainability in two independent empirical studies. The MOOD set of metrics was developed by Brito e Abreu and Carapuça [20]. Their metrics include the method inheritance factor (MIF) and the coupling factor (CF).

There is no clear agreement on how to measure maintainability. Welker and Oman suggest measuring it statically by using a maintainability index (MI) [21]. A program's maintainability is calculated using a combination of widely used and commonly-available measures to form the MI. A large value for MI indicates that the program is easy to maintain. The basic MI of a set of programs is a polynomial of the following form (all are based on average-per-code-module measurements):

$$MI = 171 - 5.2 \cdot \ln(\text{ave}V) - 0.23 \cdot \text{ave}V(g') - 16.2 \ln(\text{ave}LOC) - 50 \cdot \sin(\sqrt{2.4(\text{per CM})}) \quad (1)$$

The coefficients are derived from actual usage, with the terms defined as follows: $\text{ave}V$ is the average Halstead volume V [22] per module; $\text{ave}V(g')$ is the average extended cyclomatic complexity per module; $\text{ave}LOC$ is the average count of lines of code (LOC) per module; and, optionally, per CM is the average per cent of lines of comments per module [23,24].

Oman evaluated the MI and found that the above metrics are good and sufficient predictors of maintainability [24]. Ramil [25] suggests using a model using A and B , the model parameters that are to be derived from historical data by, for example, least squares regression [26]. By detecting changes to A and B as a system evolves, one may infer changes in evolvability. Ramil's M3 model is:

$$\Delta\text{Effort}(t) = A \cdot \text{Modules handled}(t) + B \quad (2)$$

where $\Delta\text{Effort}(t)$ is the effort in person-months applied during a one-month interval (from month t to $t + 1$), and $\text{Modules handled}(t)$ is the number of modules which were either added to the system or modified, or both (if both, the module is counted only once) during the interval [25].

Polo *et al.* use the number of modification requests, mean effort in hours per modification request and type of correction to examine maintainability [27]. They found no meaningful influence of size metrics on the number of faults and failures. The OMA approach examines MI as well as other maintainability measures such as hours of effort per change, comment ratio, etc. in the mine step.

The present authors have also developed several maintainability measures [28]. Our observations (observation step of OMA) have shown us that if program A and program B satisfy the same set of requirements but program A is more maintainable (i.e., it is easier to make changes to A than to B), then: (1) it will take less time to make changes to program A; (2) it will take less time to understand how to make the changes to program A; (3) less modules or components or classes will have to be changed or added for program A than for program B; and (4) it will be easier to leave program A in an 'easy to keep maintaining' state than program B.



Based on this, we calculate our dependent variable, the maintainability product (MP), as the product of the effort for the change and the percentage of the program that has been changed:

$$MP = \text{Change scope} \times \text{Effort} \times 100 \quad (3)$$

Maintenance effort measures the personnel resources expended on a maintenance activity. It is generally measured in person-hours. It can also be measured in relative terms. In (3), Effort is evaluated on a scale of 0 to 1. Thus, an activity that has an Effort of 0.9 would require much more effort than an activity that has an Effort of 0.2. For example, if four changes totaling 133 person-hours were made to release 2.7 of an application, each taking respectively 50, 70, 10 and 3 hours, their Effort values would be, respectively, 0.38, 0.52, 0.08 and 0.02 (this is each change's percentage of the total hours). The Change scope measure examines what percentage of the program's components were added or modified to implement a change. So if 10% of the program's components were modified and/or added for the 50 hour change, MP would be $0.38 \times 0.10 \times 100$ for a value of 3.8. There is no unit for the MP measure. MP is measured on a scale of 0 to 100 where 0 is highly maintainable and 100 is highly un-maintainable. Imagine a change that was a 1.0 for effort and required every component to change (100%), its MP of 100 indicates that it is hard to maintain. This is also intuitively obvious. Note that the Effort value can also be assigned on a scale of 0 to 1 (with 0 being no effort and 1 being the most effort) if there is not enough person-hours data for a particular release to calculate the value.

We also measure 'perceived maintainability' (PM) by asking the maintaining software engineer, after all the changes have been made, to assign a value from 1 to 10 to each component modified, where 10 is code that was very easy to change.

2.2. Software process improvement

As mentioned in Section 1, overarching process improvement models such as CMMI for software [1,2], SPICE [4] and ISO 9001 [3] have been adopted by many organizations, with varying results. A telling commentary on the effort, budget, staff and commitment required to undertake such an improvement project is that on average it takes an organization approximately 24 months to move to level 2, 21.5 months to move from level 2 to 3, 33 months to move from level 3 to 4 and 18 months to move from level 4 to 5 in the SEI's staged CMM [29]. Although one could theoretically work on small elements of these models, it has not been our experience that organizations do so. Instead, a serious commitment of time, money and effort is required before the undertaking starts.


In this vein, a paradigm, goal/question/metric (GQM), has been used in many process improvement undertakings [30]. GQM is part of the University of Maryland's tailoring a measurement environment (TAME) project. The paradigm is very simple and elegant: establish high-level process improvement goals; develop a set of questions that, when answered, indicate whether or not the goals have been met; and develop a set of metrics to help answer the questions. Although high-level process improvement goals are first required, we have successfully used this in practice (the lead author successfully used this while working at Science Applications International Corporation (SAIC)).


Process improvement advances aimed specifically at maintenance have also been made. Higo *et al.* have developed a maintenance environment called Gemini that assists maintenance programmers in identifying duplicate sections of code called clones [31]. Basili *et al.* [32] developed a predictive cost model for maintenance releases that are primarily enhancements. Their model is based on the estimated size of a release. The model was developed by examining 25 releases of ten different projects.



We have also noticed maintainability-related process advancements ‘in the small’, some based on experimentation. For example, Siy and Votta [33] came up with a new hypothesis when an earlier experiment on code inspections showed that reorganizing the inspection process by changing the size and number of inspection teams had little effect on defect detection effectiveness. Their observation was that the benefit of code inspections is in maintainability rather than in defect detection [33]. Shirabad *et al.* discuss the mining of source code and software maintenance records, an organization’s repository of past software maintenance experience, to extract relations. These maintenance relevance relations help developers know which files in a legacy system are relevant to each other in the context of program maintenance [34].

3. OMA

The ent of agile development methods such as XP [35], Scrum [36], and Crystal [37] indicate the desire to decrease the amount of process and formality in software development while still developing large quantities of software rapidly to some acceptable quality level. These methods stress individuals and interactions over processes and tools, they de-emphasize documentation and stress coding and they use concepts such as pair programming, fearlessness, communication, etc. OMA was developed to serve organizations using agile methods as well as organizations that do not have the time, money or resources to undertake a major process overhaul, or perhaps that have noticed that some things work well and other things do not and want to make ‘point’ improvements. OMA is being used to identify and adopt ways to build more maintainable software systems.

 OMA can be categorized as an agile method because it is people-oriented instead of process-oriented and it has many of the characteristics of other agile methods: it is adaptive, quick and self-organizing. It possesses all of the advantages of agile methods [35] as follows.

- First, the operations in OMA are clearly defined and close to the natural ways of human thinking. Thus, it is easy to learn and master.
- Second, the overhead of the approach is small. People can perform this by holding a few OMA meetings, which do not need to take a long time. With the OMA concepts in mind, a software development group can frequently intertwine the three steps with their regular meetings without spending much additional effort.
- Third, OMA can help a development team obtain early observations and feedback so that people can quickly adjust their processes to adapt to changes in the environment.
- Fourth, OMA is largely dependent on human experiences. It is actually an empirical paradigm that ‘can expect the unexpected’ [36] like other agile methods. Software development relies on human intelligence and creativity, making it very complex and unpredictable and well suited for empirical methods.

In addition to the above advantages that both OMA and other agile methods possess, OMA is scalable. The duration and labor intensity of each step of OMA can be easily adjusted to satisfy the current needs. Thus, a team with tens of people can easily grasp the method as well as a group with only a few people. In fact, unlike most agile methods that finely attune the software development process, OMA helps to identify and adopt more ‘good’ methods. This includes the adoption of both agile and traditional methods, with an end goal of building more maintainable software systems.

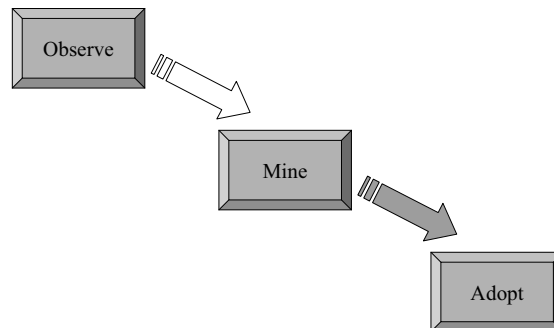


Figure 1. The OMA paradigm consists of three steps.

The rest of Section 3 is organized as follows. Each of the OMA steps will be described in Sections 3.1–3.3. Section 3.4 presents a discussion of OMA as it relates to the GQM paradigm. Specifically, the two approaches are compared and criticism of the approaches is addressed.

3.1. Observe

OMA has three major steps, observe, mine and adopt, as shown in Figure 1. In the qualitative observe step, an observation (or many) is made related to software development, in our case related to software maintainability. For example, a manager may note from a software problem report summary sheet that one particular component is repeatedly changed in response to customer enhancement requests, but is always quickly and easily modified. So the manager identified a maintainable ‘piece’. They could have just as easily identified an un-maintainable piece. There is no temporal limitation to this step. The observation may occur immediately after an activity has been very successful or a product has achieved success. The observation may occur much further in the future, when an engineer is reflecting back on past experience. The only mandate is that the observation be documented (note that this can be as simple as an e-mail to oneself).

For a focus on maintainability, the observe step has three sub-steps.

- First, an organization must agree on what is meant by ‘maintainable’. For example, if ranking the maintainability of components on a scale of low to high (high means highly maintainable, easy to change component), these rankings must first be defined. Highly maintainable might be defined as having a low comment ratio (say 5–10 meaning that there is one comment for every 5 to 10 LOC), low cyclomatic complexity [38] and low coupling (coupling factor (CF) [20]).
- Second, the group must agree on what and when to observe. Perhaps a brainstorming session on ‘what highly maintainable components from release 1.7 pop to mind?’ may be followed by a perusal of a list of frequently executed components in that release.
- Third, the observation must be documented.



3.2. Mine

Once the observation has been made and documented, mining can begin. As in data mining, we view mining as discovering useful knowledge from unstructured artifacts or events [39]. From more traditional definitions, we look at mining as digging out or extracting the ‘rich’ elements while leaving behind the rest [40]. The mine step is comprised of sub-steps: categorize the observation; investigate the observable ‘item’ to learn more about what caused it to occur or what factors contributed to its occurrence; determine if there is a causal relationship (as opposed to happenstance); and possibly validate and/or replicate the finding.

There are some guidelines for the mine step.

- First, the object of the observation should be determined [15,41]. For example, the observation ‘release 1.3 was very easy to modify when we went to release 1.4’ would have *product* as its object. Other possible objects include, but are not limited to, process, model, metric, theory [15,41].
- Second, the object is investigated. This may involve examining a number of criteria such as number of changes made (by severity), effort required to make the changes, comment ratio, currency of documentation, etc., to uncover what made release 1.3 so easy to maintain. Note that the initial observed object may be *product* but *process* issues might surface quickly in the investigation sub-step. This is to be expected. This sub-step is quantitative as will be illustrated in Sections 4.1–4.3.
- Third, the collected data and criteria are mined for causal relationships. An object and a best practice/lesson learned have a causal relationship if we can show that the best practice resulted in the object being observed. We can mine for causal relationships much in the same way that root-cause analysis is performed. This mining process may include a review of historical reports, queries of software problem report databases and trend analysis of trouble reports by category or severity. For example, one might find that a very high comment ratio existed (comments per LOC) and that the comments were not out of date (currency).
- Fourth and optionally, this finding may be validated. That is, the team might try using a high comment ratio with current comments in release 1.4 and see if it is also highly maintainable.

3.3. Adopt

The adopt step involves formalization of the successful process and introduction to and adoption by the organization. This is arguably the most important step of OMA. Without this step, the knowledge that we have gained from the previous two steps will not be applied to the maximum advantage of the organization. In this way, it is similar to the ‘publish’ step in an academic’s career. Excellent research results cannot influence the further advancement of an area if not published for possible use by others.

The successful process or practice needs to be formalized to the extent needed by the organization. For a CMM-level seeking organization, this will involve a written procedure with entry and exit criteria, steps to be performed, and process controls such as management, quality assurance and configuration management. The new process document will itself be the subject of several written processes such as document review, assignment of a configuration control number to the process document, etc. For agile



organizations, on the other hand, this step may simply be an e-mail or addition to the ‘best practices’ or ‘lessons learned’ list.

It should be noted that, in practice, the OMA approach will often resemble a series of successively smaller funnels. That is, many observations may be made and documented. Some observations will not be pursued further. Similarly, after mining, some items will not be deemed repeatable or worthy of adoption. A very informal risk analysis can be used to facilitate this decision. Instead of calculating the product of probability of a risk occurring and cost to the project if that risk does occur, the possible cost should be replaced by possible benefit to the organization.

3.4. OMA and GQM

Part of OMA’s strength lies in its simplicity and practicality. The GQM paradigm [30] of Basili *et al.* also possesses these characteristics. Basili commented ([42] quoted with permission) that:

‘The GQM paradigm represents a practical approach for bounding the measurement problem. It provides an organization with a great deal of flexibility, allowing it to focus its measurement program on its own particular needs and culture. It is based upon two basic assumptions (1) that a measurement program should not be ‘metrics-based’ but ‘goal-based’ and (2) that the definition of goals and measures need to be tailored to the individual organization. However, these assumptions make the process more difficult than just offering people a ‘collection of metrics’ or a standard predefined set of goals and metrics. It requires that the organization make explicit its own goals and processes.’

OMA is similar to GQM in that it requires organizations to be goal-based while it also allows great flexibility. It does not provide a predefined set of goals and metrics. Also, OMA consists of three major steps as does GQM, each with defined activities and guidance. OMA is different from GQM in that it focuses on discovery during the mine process. It also focuses more on the collection and analysis of information than GQM. This is to facilitate development and adoption of important findings.

GQM has been criticized for being ‘common sense’ and ‘something that good engineers do anyhow’. This criticism would be correct were it not the fact that a study of management practices shows that common sense is not always used. The decision processes used frequently are based more on folklore and true belief than on hard facts and actual measurements [43]. The same criticism can be levied at OMA. The same rebuttal can also be used. Gathering hard facts and measurements to identify good engineering practices and then using them can only improve the software process.

3.5. Measuring success

From our experience, understandability and complexity are the two key components of maintainability. This agrees with the factors found in the literature, such as the MI [21] and with what we have found from surveying the industrial partners of our Computer Science Department. Based on this, we have collected the following metrics in the mine step of the OMA process to assist us in looking for causal maintenance relationships (note that some only apply to the applications that were developed using object-oriented techniques/languages).



Product effort (number of person hours to make the changes) by phase—that is: (1) the understanding phase (figuring out what the program does and how); (2) the analysis phase (deciding how to modify the analysis model); (3) design (modifying the design); (4) code; and (5) test. We also collected the following.

1. Structural complexity of each module.
2. Data complexity of each module.
3. System complexity.
4. Module coupling indicator for each module.
5. Weighted methods per class (for each class).
6. Depth of inheritance tree.
7. Coupling between object classes.
8. Defects (found when making the changes) by phase—that is: (1) analysis; (2) design; (3) code; and (4) test.
9. Number of modules added.
10. Number of modules changed.
11. Number of classes added.
12. Number of classes changed.
13. Total effort on adding modules.
14. Total effort on changing modules.

We also calculate MP and PM where possible.

4. VALIDATION

In this section, we discuss the validation of OMA and our methods for measuring maintainability. The validation took two forms: two controlled experiments involving graduate and undergraduate students and their maintained applications; and a study of an industrial partner's long-term project.

4.1. Project experiment I

The first study involved graduate students who took a survey course on Software Engineering in January 2001 at the University of Kentucky. The students undertook a real-world project to develop and then modify a phenylalanine (phe) milligram tracker. The product, developed to run on a personal digital assistant (PDA), allows phenylketonuria (PKU) disease sufferers to monitor their diet as well as assists PKU researchers to collect data. The project was also used as an experimental study, testing numerous software engineering and software maintenance hypotheses [15]. The students, who were concentrating on maintainability, then applied the OMA approach.

Similarly, in CS 499 Senior Design Project, we developed a stereology tool to support the Otolaryngology Department of the U.K. Medical School. The tool allows them to determine the volume of various matter types such as old bone, new bone, blood, etc. based on an image (TIFF) of a cross section slide. This real-world course project also was used as an experimental study in software maintenance. The undergraduates were interested in the maintainability of their programs, so the OMA approach was applied.



Table I. Data gathered during phe tracker and stereology projects.

RFC	Response for class [18]
LCOM	Lack of cohesion in methods [18]
CBO	Coupling between objects [18]
NOO	Number of operations [44]
WMPC	Weighted methods per class [18]
NOC	Number of children [18]
CC	Cyclomatic complexity [38]
NOA	Number of attributes
DIT	Depth of inheritance tree [18]
AC	Attribute complexity [45]
CF	Coupling factor [20]
CR	Comment ratio
Hdiff	Halstead's difficulty [22]
Heff	Halstead's effort [22]
LOC	Lines of code
NOAM	Number of added methods
MIC	Method invocation coupling
TCR	True comment ratio
FO	Fan out
HPLen	Halstead program length [22]
Effort	Effort by activity (requirements, design, etc.)

For both projects, significant data were collected at all stages of the project. Some of the sample data collected is shown in Table I along with applicable abbreviations [15].

In both courses, the students were told that a major change to their projects would be specified late in the semester. This motivated the students to build highly maintainable applications. The students completed the implementation of their applications and then were given a major change specification. As they made the major revision they collected data for our study. In the OMA experiment, we looked at a minor and a major change to the student applications. A minor change is defined as not affecting any of the core requirements for the specification and not requiring prior knowledge of the domain area. A major change does affect the core processing requirements of the system and requires domain knowledge as well as strong knowledge of the system.

4.1.1. Hypotheses

The following four research questions and hypotheses were addressed by this OMA study.

1. Are students good judges of what is or is not highly maintainable (i.e., can they benefit from an approach like OMA)? The null hypothesis is that the MP means are the same for the 'observed to be maintainable' applications and for the applications that were not so observed (at all change levels).



Table II. Course phe tracker projects OMA study subjects.

Project	Team	Language	Size (LOC)	Observed?
Phe tracker	Team 5	Java	701	No (control)
Phe tracker	Team 8	C++, C	1015	Yes (experimental)
Phe tracker	Team 3	Java	1523	Yes
Stereology	Team 10	C++	9542	Yes
Stereology	Team 5	Java	1613	No

2. Are some applications easy to change in a minor way? The null hypothesis is that the MP means are the same for all applications at the minor change level.
3. Are some applications easy to change in a major way? The null hypothesis is that the MP means are the same for all applications at the major change level.
4. Are there metrics that measure/correlate with maintainability? The null hypothesis is that the MP means, MI means, number of hours per change means, comment ratio and percentage of components changed do not correlate with the 'PM' value (on a scale of 0 to 10) at all change levels (no change, minor change, major change).

4.1.2. Design

The experiment was designed as follows. For the observe step, the students from both courses were contacted and asked if any projects stood out in their minds as being maintainable. The students were asked three to ten months after their course ended. The majority of the students were still enrolled at the University and did respond. 'Maintainable' was not defined to the students in the request (although it had been defined in the two courses). Note that the students had made status presentations on requirements, architecture, design, etc. to their colleagues throughout the semester. Based on student responses to our request, two of the 11 phe tracker projects were identified as being maintainable while only one of the 11 stereology projects was so identified. We also randomly selected one 'control' project from each course to study. The three identified projects were treated as experimental and were the 'observed' objects. Table II shows some characteristics of the projects used as subjects in the experiment: the project name, team number, programming language used, size in LOC, and whether or not the program was identified as being highly maintainable.

For the mine step, we performed the following. First, we took a 'snapshot' of the project (application). This included retrieving the project's metrics from our repository and generating metrics, using the Together computer-aided software engineering tool [46], for the project. Second, a graduate research assistant (RA) was asked to make a minor change to the projects. The minor change was to make a change to the GUI (for example, in phe tracker, 'protein grams' was changed to 'phenylalanine milligrams'). We then made another 'snapshot' of the projects. Next, the RA was asked to make a major change to the projects. For example, in phe tracker, the application was modified to include calories as a displayed item along with food servings and phenylalanine milligrams. Note that the original application was modified, not the application with the minor change applied. A snapshot was



then made of the modified application. At each step, the RA also gathered much of the data discussed above: how long the changes took, how much time was spent reviewing the design, etc.

Our dependent variables, as mentioned above, are MP and PM. Our independent variables are observation and change level. Formally, we had a factorial design with two treatments: observation, with two levels (observed to be more maintainable or not so observed) and degree of change, with three levels (none, minor and major). We had five subjects, three experimental and two control. Given this design, we generated 30 observations. Each observation included the metrics discussed above.

4.1.3. Threats

There were several threats to the validity of our experiment. Internal validity threats deal with the causal relationship between the independent and dependent variables. We attempted to limit the threats by validating the tools and processes we used for data collection. For example, we used a commercially available CASE tool. Also, our experimental subjects were based on student's observations. Had a different set of programs been identified as highly maintainable, the conclusions may have been different. Also, the student who made the changes to the programs was familiar with the domain area (he had been in the original CS 650 course). It is possible that a programmer unfamiliar with the programs and domain areas may have felt that the minor changes were quite major. We controlled this by making the minor changes specific to the GUI. There may have also been a learning effect as the same student made all the changes.

A major threat to external validity (generalization of results) for our experiment is the representativeness of our subject programs and changes as well as our small sample size. Also, we worked with students and with student applications. However, Høst *et al.* [47] found that students perform the same as professionals on small tasks of judgements. Tichy [48] found that using students is acceptable as long as they are adequately trained and data is used to test initial hypotheses prior to experiments with professionals. To further address these threats to validity and to utilize Tichy's findings, we worked with an industrial organization to perform a larger study addressing a wider range of change types on industrial applications. There is also the threat to construct validity (are the measures appropriate?). There is no commonly accepted dependent variable for measuring maintainability.

4.1.4. Results

The results of these OMA studies are shown in Table III. As can be seen, none of the applications identified as being highly maintainable (projects 3 and 8 for phe tracker and project 10 for stereology), were significantly easier to maintain than the other applications. Note that phe tracker project 3 was actually harder to maintain than any of the control projects (took 180 minutes to make the major change, had a low MI of 15.8 and was perceived as a 3 (where 10 is easiest to maintain)). We examined this application more closely to understand why. First, they failed to design their project according to the original project specifications. Thus much effort was required to understand the actual design. Second, in addition to a large effort for understanding, their design of the GUI required major rework and redesign to allow the addition of one element. From this, we must conclude for H1 that students incorrectly identify hard to maintain projects as being maintainable! There is a strong indication that these two groups of students require additional training in order to accurately observe maintainability.



Table III. Maintainability results for phe tracker and stereology projects.

Language	Java									
	Phe tracker					Stereology				
	Tm 5	Tm 5 Min	Tm 5 Maj	Tm 3	Tm 3 Min	Tm 3 Maj	Tm 5	Tm 5 Min	Tm 5 Maj	
AC	16	16	9.8	11.8	11.8	12	21.8	21.8	22.1	
CR	13.3	13.3	13.5	31.4	31.4	31.2	10.5	10.6	10.6	
LOC	701	701	766	1523	1523	1558	1613	1613	1646	
TCR	17.5	17.5	17.8	62.6	62.6	62	13.3	13.4	13.4	
MI	71.5	71.5	81.2	16.7	16.9	15.8	72.6	72.2	71.8	
Prod Eff - reqts (minutes)	Basecase	15	No Add. Min	Basecase	15	15	Basecase	30	No Add. Min	
Prod Eff - design	—	10	45	—	5	60	—	10	60	
Prod Eff - code	—	10	100	—	5	180	—	10	120	
Prod Eff - test	—	5	15	—	5	30	—	5	10	
Number classes added	—	0	23	—	0	2	—	0	3	
Number classes changed	—	2	2	—	2	5	—	2	2	
Total effort changing modules	—	15	100	—	20	180	—	10	120	
Effort value for change	—	0.2	0.4	—	0.2	0.9	—	0.2	0.4	
(0 to 1)	—	—	—	—	—	—	—	—	—	
Modules change/add (%)	—	2.3	22.3	—	2.4	12.2	—	15.4	31.3	
(0–100)	—	—	—	—	—	—	—	—	—	
MP	—	0.5	8.9	—	0.5	10.9	—	3.1	12.5	
'Perceived maint.' (1 to 10)	—	10	6	—	9	3	—	9	5	

Language	C++									
	Phe tracker					Stereology				
	Tm 8	Tm 8 Min	Tm 8 Maj	Tm 10	Tm 10 Min	Tm 10 Maj	Tm 8	Tm 8 Min	Tm 8 Maj	
AC	9.6	9.6	9.8	50	50	50	NA	NA	NA	
CR	30.1	30.1	29.2	34.2	34.2	34.2	25	30	30	
LOC	1015	1015	1540	9542	9542	9542	135	135	135	
TCR	52.8	52.8	51.6	67.1	67.1	67.1	20 min	20 min	20 min	
MI	NA	NA	NA	NA	NA	NA	3	3	3	
Prod Eff - reqts (minutes)	Basecase	30	30	Basecase	10	25	40	40	40	
Prod Eff - design	—	55	55	—	10	30	0.8	0.8	0.8	
Prod Eff - code	—	30	65	—	10	135	29.4	29.4	29.4	
Prod Eff - test	—	5	10	—	5	5	23.5	23.5	23.5	
Number classes added	—	0	3	—	0	3	1	1	1	
Number classes changed	—	3	7	—	4	4	—	—	—	
Total effort changing modules	—	15	50	—	15	15	—	—	—	
Effort value for change	—	0.3	0.7	—	0.3	0.3	—	—	—	
(0 to 1)	—	—	—	—	—	—	—	—	—	
Modules change/add (%)	—	11.5	34.5	—	28.6	28.6	—	—	—	
(0–100)	—	—	—	—	—	—	—	—	—	
MP	—	3.5	24.1	—	8.5	8.5	—	—	—	
'Perceived maint.' (1 to 10)	—	10	7	—	8	8	—	—	—	

* Failed design (see Table I for definition of terms). NA = not available.



For H2, we see that all of the phe tracker projects were easy to change in a minor way. Although the results are not statistically significant, there is a noticeable difference in the effort required to change the applications in a minor way.

For H3, we see that project 5 for phe tracker was easy to change in a major way (low MP, high MI, high PM, low comment ratio (less LOC per comment)). In observing this application as maintainable, the students noted that it would be highly maintainable because the application was GUI-intensive and Visual C++ had been used. The Visual C++ development environment thus facilitated interface changes. This finding represents a possible ‘best practice’ to be evaluated and possibly adopted in the adopt step of OMA. Finally, team 3’s application (observed as maintainable) was not easy to change in a major way, as discussed above. As with H2, the results are not statistically significant, but there is a noticeable difference in the effort required to change the applications in a major way.

One concern we had as we undertook these two OMA studies was whether the changes requested were accurately described as minor and major. Almost every metric we examined for H4 indicated that minor changes were clearly ‘easier’ than major changes. For example, examining the MP of the minor and major changes to each application using analysis of variance, minor changes average 3.2 as compared to major changes at 14.1 with a p-value of 0.01 (we accepted results with a p-value of 0.05 or smaller as being statistically significant). Also, we examined whether a learning effect could be detected. If learning took place (the RA got better at changing the applications as he progressed), we would expect that the last applications modified would be the easiest to change. In order, the changes were made to: team 5 phe tracker, team 3 phe tracker, team 8 phe tracker, team 5 stereology, team 10 stereology. The last application changed (team 10) was the least maintainable. In fact, the RA was not able to successfully make the major change. Similarly, the last phe tracker application modified was harder to change in an easy way (high effort value, high percentage of modules changed/added and high MP) than the two phe tracker applications that were changed before it.

For H4, we again had mixed results. When looking at all changes (minor and major), there was a 0.82 correlation between the percentage of changed/added modules and MP. Very strong correlation was seen between the maintainability measures for the major changes to the applications. For example, the percentage of modules changed/added and MI had a 0.81 correlation, and the percentage of modules changed/added and PM had a 0.82 correlation. More impressively, the comment ratio and MI had a -0.96 correlation (negative correlation) and PM and MI had a 0.98 correlation. Note that none of the metrics correlated with PM for all 10 changes (five minor, five major), so we cannot reject the null hypothesis.

The students who participated in the two courses (a subset of whom assisted us by ‘observing’ maintainable applications) found that collecting various metrics throughout the development and subsequent maintenance efforts was worthwhile. This is also a possible lesson learned to be evaluated and adopted in the third step of OMA.

Although the hypothesis results were not conclusive, several possible best practices were identified. We plan to formalize these and ask the students in future classes to adopt them.

4.2. Project experiment II

The objective of our second validation study was to have graduate student programmers concerned with maintainability use of OMA. This study was used to further test three of the four hypotheses listed in Section 4.1. This time, all the experimenters involved possessed software engineering knowledge and



were more experienced in programming. This study was performed on another group of three projects, all of which were written in Java.

The first project was a test data generation tool based on a probability distribution. The test cases generated by this tool using a spathic approach were shown to be more effective in identifying software faults in some cases than random test cases [49]. The tool was designed and coded by a graduate student in the Software Verification and Validation Laboratory during an independent study. It has since been frequently used in the laboratory as a research application. Personnel changes frequently in the laboratory, so the tool was designed to be maintainable.

The other two projects were course projects assigned as part of the University of Kentucky graduate networking curriculum [50]. One project, named PA0, was aimed at helping students understand length-based and delimiter-based framing protocols. A server implemented by the instructor sent out frames fragmented from a certain message through each protocol alternatively. Students were required to implement a client that could correctly receive the frames and assemble them into a message which was the same as the original piece. In another project, PA2, students implemented a tic-tac-toe protocol. The protocol allowed the game of tic-tac-toe to be played over the Internet. With the exception of the server part of PA0, all the source code for the two projects examined in our experiment were originally written by a graduate RA who also took the networking course.

As in the first study, the students were very interested in the maintainability of these programs and used the OMA paradigm to assess maintainability. We intensively examined the original source code of all three projects as well as the modified versions (after a minor or a major revision). The major and minor revisions were similar in scope and size to those found in Section 4.1.

4.2.1. Hypotheses

In this study, we addressed hypotheses H2, H3 and H4 described in Section 4.1. Moreover, we examined the correlation between additional metrics and maintainability. Since we only had one data point for each project, we did not test H1 in this study.

4.2.2. Design

The design of this experiment was nearly the same as for experiment I (Section 4.1.2). One difference is that in the observe step, the designated maintainers (graduate students) observed all the projects under examination and assigned PM to each. Then the mine step was performed. A minor revision and a major revision were made to each project. For example, the minor revision to PA0 involved a change made to the output. The major revision to the Spathic project enabled the tool to make an informed decision on how many test values to generate.

We took three snapshots of the source code by collecting various data points and metrics using the Together tool [46]. We took snapshots of the original source code, the source code after a minor revision and the source code after a major revision. We asked each maintainer to keep a record of the time they spent on each phase of the development lifecycle, just as in the first experiment. We had the maintainers use a NASA-developed workload tool [51], TLX, to evaluate their effort on each revision. The tool prompts a user to perform comparisons such as ‘was this task more mentally demanding or frustrating?’. Based on these ratings, the tool calculates a value between 0 and 100 where 0 required



no effort and 100 required maximum effort. We also asked that they provide a subjective effort value for the minor and major revision. We then examined the three hypotheses based on this data.

4.2.3. Results

The results of this OMA study are shown in Table IV. The rows of Table IV are identical to Table III with the exception of new rows for number of operands (NOprnd) and number of operators (NOprtr) and the percentage of changes in each.

For H2, we find that all the projects were easy to change in a minor way. The maintainers spent no more than 20 minutes on the minor changes and estimated their effort as no more than 0.1 (on a scale of 0 to 1 with 1 being maximum effort). The TLX value was also significantly lower for each minor revision. Thus, we can reject the null hypothesis.

For H3, we see that it was easy to make a major revision to PA0 (low MP and high PM). We also found that the percentage of operators and operands changed provides useful information on how hard a program is to revise. When the percentage of modules changed does not give us conclusive information (or gives us non-intuitive results), we should examine the information on changed operators and operands. This represents a possible 'best practice' that we may choose to Adopt.

For H4, we tested the correlation between MP and PM (PM). For this experiment, as in the last, we still calculated MP with the percentage of modules changed as the change scope. Thanks to the observations made in evaluating H3, we also calculated MP' and MP'' by using the percentage of operands changed and the percentage of operators changed as the change scope, respectively. The correlation coefficient obtained from the regression analysis is 0.69 between PM and MP, 0.98 between PM and MP' and 0.96 between PM and MP''. The p-value is 0.042 between PM and MP and less than 0.001 between PM and both MP' and MP''. These are very strong results that are consistent with our findings above that MP is more accurate when calculated with the changes in operands and operators. As stated above, this finding from the mine step is a candidate for adoption.

We also analyzed the correlation between PM and MI as well as other metrics given by the Together tool. The results show that PM is not strongly correlated with any of these metrics. For example, the p-value between MI and PM for major revisions is 0.65 and the correlation coefficient is 0.27. Next, we computed the correlation between %MI changed and PM. Here, the correlation coefficient is 0.81 and the p-value is 0.014. Thus, %MI changed can also be used to measure maintainability and we can reject the null hypothesis.

The students identified several best practices as a result of this exercise. The identified practices are listed below.

1. Perform requirement elicitation well.
2. Analyze whether the requirement is feasible.
3. Think about the requirements in the form of reusable modules performing one particular function.
4. After modularizing, get all the interactions of a module and the parameters needed for interaction.
5. Start coding each module one by one keeping its interaction with other modules.
6. Remember to add comments when you are coding to explain what each main statement intends to do.



Table IV. Maintainability results for Spathic, PA0 and PA2 projects.

Title	Spathic			PA0			PA2		
	Original	Minor	Major	Original	Minor	Major	Original	Minor	Major
AC	48.00	48.00	37.14	8.11	8.11	8.11	14.55	14.55	13.95
CBO	6.00	6.00	5.86	4.96	2.96	4.96	5.91	5.91	5.77
CC	10.60	10.60	10.29	4.33	5.04	5.00	9.77	9.86	9.27
CR	10.40	10.40	14.14	19.11	18.96	18.93	42.73	42.68	43.95
HPVol	1829.00	1829.00	1592.14	475.56	483.89	500.70	967.14	972.18	900.00
LOC	79.00	79.00	76.14	34.37	34.89	35.04	61.59	61.91	58.63
TCR	12.40	12.20	17.42	29.26	29.04	29.00	84.68	84.00	87.95
MI	106.72	106.72	82.20	57.15	57.84	57.84	98.54	98.18	104.96
Other metrics									
NOprnd	707.00	707.00	865.00	1093.00	1115.00	1151.00	1773.00	1780.00	1650.00
NOprtr	700.00	700.00	860.00	1160.00	1181.00	1219.00	1729.00	1735.00	1623.00
Total number of classes	5.00	5.00	7.00	275.00	27.00	275.00	22.00	22.00	21.00
Changes									
Number of classes added	Basecase	0.00	2.00	Basecase	0.00	0.00	Basecase	0.00	0.00
Number of classes changed		2.00	2.00		1.00	2.00		1.00	8.00
% Modules changed/added (0–100)		40.00	40.00		3.70	7.41		4.55	36.36
% Operands changed (0–100)		0.00	22.35		2.01	5.31		0.39	6.94
% Operators changed (0–100)		0.00	22.86		1.81	5.09		0.35	6.13
% MI changed (0–100)		0.00	22.97		1.20	1.20		0.36	6.51
Effort									
Prod Eff—reqts (minutes)	Basecase	3.00	53.00	Basecase	3.00	20.00	Basecase	5.00	5.00
Prod Eff—design		5.00	25.00		3.00	110.00		5.00	15.00
Prod Eff—code		7.00	297.00		6.00	50.00		5.00	110.00
Prod Eff—test		5.00	746.00		0.00	15.00		5.00	20.00
Total effort changing modules		20.00	1121.00		12.00	195.00		20.00	150.00
Effort value for change (0 to 1)		0.10	0.40		0.00	0.70		0.10	0.70
Workload		7.00	66.00		46.00	70.00		20.00	53.00
Conclusive values									
MP	Basecase	4.00	16.00	Basecase	0.00	5.19	Basecase	0.45	25.45
MP (% Oprnd)		0.00	9.94		0.00	3.71		0.04	4.86
MP (% Oprtr)		0.00	9.14		0.00	3.56		0.03	4.29
MP (workload)		0.70	26.40		0.00	49.00		2.00	37.10
MP (% Oprnd, workload)		0.00	1474.96		92.59	371.45		7.90	267.68
MP (% Oprtr, workload)		0.00	1508.57		83.28	356.03		6.94	324.93
*Perceived maint.		10.00	7.00		10.00	9.00		10.00	8.00



7. Prepare a small document on each module and its interaction.
8. Use proper structure while programming (e.g., proper indentation).
9. When the percentage of modules changed is not conclusive (or is not intuitive), examine changed operators and operands.

Some of these practices were already in use by the students, but were reinforced by their experience on this experiment. Some practices were ‘discovered’ during the experiment because they were lacking in the original program. For example, one student noted that practices 3, 6 and 7 would have made his major revision much easier. As a result of the negative experience (modifying a complicated program that was not modular or documented), he ensured that he performed those practices and plans to continue to do so. Another student found that the original author had used practices 6 and 8 and this made his maintenance work easier. As in the first experiment, the students who participated in this experiment found that collecting various metrics throughout the development and subsequent maintenance efforts is worthwhile. This is a possible lesson learned to be evaluated and adopted.

4.3. Health care system at an industrial partner

In our third validation study, we worked with an industrial partner on a large commercial system it has developed to investigate the practicality of OMA. Perot Systems [52] is a software solutions and services provider, with clients and offices worldwide. Their Lexington, Kentucky office has developed a system for a health services provider to serve as a tool for patients to access medical care via the Internet. This is a typical Web application system, developed by a team of five developers, plus a technical lead and a graphic/layout designer. Development was based on the Java 2 Enterprise Edition (J2EE) platform and consisted of Java, JavaServer Pages (JSPs), HTML and XML. Its business logic is mostly implemented in Java. It is composed of 538 classes grouped into 19 modules. The length of development of the first fully functional version lasted nine months and they have been maintaining it for one year. In this particular situation, Perot Systems is interested in small improvement and particularly in improved maintainability. Perot Systems decided to collaborate with us and use OMA to understand more about maintainability and to identify best practices for maintaining software. Specifically, they want to understand maintainability more so that the cost of maintaining software can be more accurately estimated.

4.3.1. Hypotheses

Two research questions and hypotheses were addressed by the OMA study with Perot Systems.

- Are there any metrics that measure/correlate with maintainability? The null hypothesis is that the metrics being measured do not correlate with the PM value. This hypothesis is similar to the one we addressed in the first validation study. However, since we perform this study on a real, large-scale system developed under strict business requirements, the results can assist us to more accurately assess the maintainability of similar applications (H5).
- Are there any best practices at Perot Systems for improving software maintainability when developing similar systems? The null hypothesis is that no such practices exist or that none can be found by using the OMA paradigm (H6).



4.3.2. Design

The study was designed and performed as follows. First, Perot Systems made ‘observations’ about maintainability by indicating whether modules were maintainable on a scale of 1 to 10 (with 10 being easy to maintain and 1 being hard to maintain). Hard to maintain code (1) was characterized by Perot Systems as ‘poorly organized code, much complicated business logic, one character variable names used, no comments’. Highly maintainable code (10) was characterized as ‘code that basically maintains itself, so robust that changes only need to be made to its configuration or initialization’. This ranking represents the observe step.

Second, we helped Perot Systems to mine the observations. We collected 64 metrics related to software maintenance on each module using Together ControlCenter TM 5.5 [46] and also calculated the MI of each module by using Welker’s equation [21]. Then, we checked the strength of each metrics’ correlation with maintainability through linear regression analysis. This helped us assess our first hypothesis (H5). Using all of the metrics that had the strongest correlation with PM as provided by Perot Systems, we diagnosed each module as easy or hard to maintain. Based on this analysis, Perot Systems assessed the activities they had performed in the development of those modules that received a definite diagnosis. If the second hypothesis we proposed is supported over the null hypothesis, much of their work would be categorized into good practices and possibly ‘bad’ practices.

The last step in the design, and OMA, is for Perot Systems to adopt the best practices and examine practices that may lead to reduced maintainability.

4.3.3. Threats

There were several threats to the validity of our study. We attempted to limit the internal validity threat by validating the tools and processes we used for data collection. For example, we used a commercially available CASE tool. A major threat to the external validity (generalization of results) for our experiment is the representativeness of our subject organization and subject program as well as our small sample size. There is also the threat to construct validity (are the measures appropriate?). There is no commonly accepted dependent variable for measuring maintainability.

4.3.4. Results

After receiving an introduction to OMA, Perot Systems felt that thinking back over the development time would help them make observations. They also felt that they could go into much more detail than our students did. Perot’s ratings for PM of all 19 modules are listed in Table V. The first column of the table lists the module names. We replaced the actual module names with an uppercase letter because we have agreed not to disclose the names of the modules. The second column shows the PM assigned by Perot Systems. Two modules, F and I, received ratings of 9. Perot explained that both modules received such high ratings because they required little effort, even to add a new business process, and because they were less complicated and easy to understand. On the other end of the spectrum, R received a rating of 6. This module had very complicated business logic and was very hard to maintain.

The rest of the table shows the values of a subset of the metrics collected in the second step, mine. The third to the ninth columns show those metrics that have the strongest correlation with PM. The third



Table V. Perot systems' maintainability observations and data analysis.

Item (module)	PM (high) [6, 9]	MI (high) [-120.46, 85.94]	CBO (low) [17, 66]	CC (low) [12, 220]	CR (high) [6, 55]	HPVol (low) [1402, 224640]	LOC (low) [238, 18592]	TCR (high) [6, 120]	Diagnosis
A	8	-21.87	46	89	14	75 868	4809	16	Mixed results*
B	8	-71.80	42	94	27	21 331	1679	36	Mixed results
C	8	-33.22	25	47	31	8984	902	44	Easy to maintain
D	8	-18.14	32	42	33	9823	721	50	Easy to maintain
E	8	-9.69	46	66	13	68 356	5572	15	Easy to maintain
F	9	85.94	17	12	55	1402	238	120	Easy to maintain
G	7	-43.19	36	91	6	224 640	18 952	6	Hard to maintain
H	7	-50.34	29	46	28	12 284	1190	39	Mixed results
I	9	11.01	20	31	36	5587	527	56	Easy to maintain
J	8	-120.46	66	220	28	75 181	4263	40	Mixed results
K	8	-53.60	25	70	29	13 230	1211	41	Mixed results
L	7	-32.52	29	37	22	7953	725	29	Mixed results
M	7	-29.52	35	82	17	22 532	2154	20	Hard to maintain
N	8	-62.82	43	83	24	15 336	1316	32	Mixed results
O	7	-54.39	20	32	24	17 360	1550	31	Mixed results
P	8	5.13	22	32	18	4601	491	21	Easy to maintain
Q	7	-70.17	44	111	27	17 114	1280	37	Hard to maintain
R	6	-93.26	64	189	21	45 548	2801	26	Hard to maintain
S	7	-37.73	33	48	20	15 696	1403	25	Hard to maintain

*'Mixed results' indicates that the data analysis results are mixed, with some inconsistency between the PM and the measured metrics.



column shows calculated MI. The MI calculations differed based on using either average values or the representative values of metrics provided by Together 5.5. When performing linear regression analysis, we found that the MI calculated with representative values showed a strong correlation with PM while those calculated with the averages did not. Our first study used the Together values. To be consistent with that earlier study, we also used the representative Together values in this study. The full names of the metrics CBO, CC, CR, LOC and TCR can be found in Table I. HPVol is an acronym for Halstead's program volume. The tenth column of the table shows our diagnosis of each module's maintainability. For each measure, the parenthetical statement indicates the level of the metric that we hypothesize is correlated with high maintainability (e.g., low cyclomatic complexity (CC) correlates with high maintainability). The numbers in the square braces give the minimum and maximum values of the measures of all the modules in the system (e.g., the minimum LOC was 238 and the maximum was 18 592).

As shown in Table V, the two modules with the highest MI were F (85.94) and I (11.01). The two modules with the lowest MI were J (−120.46) and R (−93.26). Other modules' MI values fell within this range. It is worth noting that both F and I had the highest PM and R had the lowest PM. This indicated to us that MI might have a strong correlation with PM. Consequently, we performed linear regression analysis on MI and other metrics.

Using ANOVA tables, we extracted the metrics strongly correlating with PM and listed them in Table V. Given alpha as 0.05 (that is, there is a 5% or less chance that the correlation is due to chance), we can conclude that PM has a significant linear relationship with MI, CR and TCR. Although the p-values of HPVol and LOC were much higher than alpha, the p-values of the correlation between PM and natural logarithms of these two metrics were 0.062 and 0.077, respectively, a littler larger than alpha. This indicated that there were also a linear relationship between PM and these two metrics, although it was not as strong as PM's correlation with MI. This result was consistent with Welker's equation, which included the logarithms of HPVol and LOC to calculate MI.

As for CBO and CC, the regression results did not show that they were strongly correlated with PM in a linear fashion. However, we noticed that module J had some conflicting results. It had the lowest MI value while still being perceived as an 8. Perot Systems explained that J was a module critical to the whole system and that they had applied a great amount of additional effort to make this module highly maintainable. This was indicated by a low LOC as well as a high CR and TCR. Thus, we thought the metric values for this module might be the one that contributed the most noise to our analysis. When we reanalyzed without this module, we found that we were correct. The p-values of the correlation between PM and the two metrics, CBO and CC, were 0.032 and 0.012, less than alpha. Based on these results, we believe that CBO and CC can still influence maintainability in a linear fashion. In some cases, we might need to consider some other particular factors such as domain complexity, module structure, design considerations, etc. Our future work will include accurate modeling of these factors in the measurement of maintainability.

These few exceptions of module J and a few other modules aside, the analysis showed that all the listed metrics had strong correlation with PM in the same direction as we hypothesized. MI, CR, and TCR were positively correlated with PM, while CBO, CC, HPVol, and LOC were negatively correlated with PM. We felt that the objective metrics could provide us with a reliable measurement of software maintainability. Based on this analysis, we diagnosed each module. Eleven of the 19 modules were diagnosed as easy or hard maintain, while the remaining eight modules were not diagnosed due to inconclusive results. It was very interesting to note that five of the six modules diagnosed



to be highly maintainable, C, D, F, I and P, were developed by the same person. This developer was also the team's architect. He is a very experienced programmer who strictly adheres to his own coding guidelines. The sixth 'easy-to-maintain' module, E, was also developed by another domain expert. We were unaware of this when we were performing data analysis but we did notice a strong similarity between the metrics for these module groupings. This strongly supports our hypothesis (H5) that the metrics we picked include measures that effectively help evaluate maintainability and that the mine step provides us with important clues for investigating the development and maintenance process.

We also learned from this experience that in addition to the measurement of subjective metrics, knowledge of the domain area and application can also be very important for evaluating maintainability. We then asked Perot Systems to think about the practices they used in developing this system. Some of the good practices they used in implementing easy-to-maintain modules are as follows:

- outlining rules and standards at the very beginning of the development effort in a written format;
- enforcing some good coding styles;
- holding code reviews periodically;
- keeping professional log and configuration files;
- designing a good architecture; and
- placing increased emphasis and attention on critical business logic modules.

Perot Systems also reflected on practices (or lack thereof) that may have lead to hard-to-maintain modules. They decided that there were several contributing factors: they allowed several people to take charge of a single module and they included too many independent functions in a module. The former practice can lead to non-cohesive code as well as code with very high coupling. The latter practice leads to very low cohesion. We were not surprised by any of these conclusions. Our own experience (in industry and academia) has borne these out. OMA provided a quantitative standpoint for evaluating these practices and their contribution to improved software maintainability. After we helped Perot Systems perform the mine step, we discussed these findings with them. They agreed that certain practices will continue to help them write more maintainable code and that other practices should be curtailed (H6). Perot Systems is very willing to adopt the 'best' practices in this and other projects in the future. We plan to work with Perot Systems to evaluate modifications of this and other systems to ensure that the best practices do indeed increase maintainability.

Finally, we asked Perot Systems to help us perform a 'process improvement' review of the OMA paradigm and our validation study. Perot Systems felt that the OMA process was very beneficial as well as efficient. They felt that it helped them gain quantitative insight into their 'best' practices as well as some possible problems in their software development process. The entire validation study only took 8 hours of Perot System's time spread over several weeks (in the spirit of an agile method). The time was distributed as follows:

- 1 hour initial meeting;
- 2 hours first metrics session (failed attempt to install tool and gather data);
- 1 hour 'observe' session;
- 1 hour metrics session;
- 1.5 hour metrics session;



- 1 hour analysis meeting ('mine'); and
- 0.5 hour analysis meeting ('adopt').

This validation effort demonstrates that our approach can be easily grasped and flexibly applied in developing large industrial systems. We plan to revisit Perot Systems to collect quality metrics on the modules they develop and modify after they adopt the practices identified by using OMA. We are very interested in studying whether there are any significant changes in maintainability between now and then, which would further demonstrate the effectiveness of the approach.

5. CONCLUSIONS AND FUTURE WORK

It should be noted that in the OMA study with Perot Systems, the relationship between PM and MI is slightly different from its relationship in project experiment II. This is due to the different ways in which maintainers estimated maintainability and developed PM values. In project experiment II, PM was given by the maintainers who were performing the revisions. In this case, the perceived hardship of making the changes became a factor for evaluating maintainability (as we feel it should). This explains why %MI changed appeared to be more related to PM than actual MI did. However, when being requested to perceive maintainability in the same way, Perot Systems intuitively related PM to the complexity of the business logic of each module. Since all the independent variables used in the MI calculation are indicators of complexity, PM is strongly correlated with MI itself in the Perot Systems study. Our future work will include examining whether certain approaches are more suitable in given circumstances. We will also provide more detailed instructions during the mine step.

Through the project experiments and the industrial study, we have found that small teams, agile organizations and even 'major process reform' undertakers can benefit from OMA for point improvements in software maintainability. The projects examined ranged in size from very small to large Web-based commercial products. The paradigm can also be used to facilitate the feedback loop of any improvement effort (where the organization has collected data and performs trend analysis to determine areas for improvement). Perot Systems has benefitted from OMA. They have identified difficult to maintain modules that may be redesigned as future changes to them are required. They have identified best practices, as well as some practices to be avoided. The two project studies also identified many best practices for building maintainable software.

Work remains to be done though. The OMA paradigm is in its infancy. We need to perform further refinement of OMA, using the OMA approach itself. We plan to formalize the observations that we have made about OMA as we have been using it. We will then mine, investigating ways to improve the approach. For example, we can add more guidance to codify it and make it easy to implement repeatedly. Based on the mining findings, we will adopt improvements to OMA. We will also seek others to validate the benefits of OMA and to assist in its enhancement. In addition, we will continue pursuing our work on maintainability measurement. For example, these results have demonstrated that major changes or enhancements have significant differences from minor changes. These differences make major changes a subject for further study. Also, by working with Perot Systems and others we hope to validate our hypotheses about software maintainability measurement and to develop maintenance estimators.



ACKNOWLEDGEMENTS

Our thanks go to Perot Systems for their participation and support. We thank the students of CS 650 (January 2001), CS 499 (August 2001), Inies Raphael Michael Chemmannoor and Senthil Karthikeyan Sundaram for their contributions. We thank Dr. Ken Calvert for allowing us to use specifications and code from his CS 571 course. We thank Togethersoft for their donation of Together to our research program.

REFERENCES

1. CMMI Product Development Team. CMMISM for systems engineering/software engineering/integrated product and process development, Version 1.02. *Continuous Representation (CMMI-SE/SW/PPD, V1.02, Continuous)* CMU/SEI-2000-TR-031, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 2000.
2. CMMI Product Development Team. CMMISM for systems engineering/software engineering/integrated product and process development, Version 1.02. *Staged Representation (CMMI-SE/SW/PPD, V1.02, Staged)*, CMU/SEI-2000-TR-030, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 2000.
3. Technical Committee 176 (TC 176). Quality Management Systems—Requirements, *ISO 9001:2000*, International Standards Organization: Geneva, Switzerland, 2000.
4. Joint Technical Committee 1 (JTC 1). Information Technology—Software Process Assessment, *ISO/IEC TR 15504-1:1998*, International Standards Organization: Geneva, Switzerland, 1998.
5. Time Inc. Industry wakes up to the year 2000 menace. *Fortune* 1998; **137**(8):163–171.
<http://www.fortune.com/fortune/1998/980427/imt.html> [14 July 2003]
6. Offutt J. Quality attributes of web software applications. *IEEE Software* 2002; **19**(2):25–32.
7. IEEE. Authoritative Dictionary of IEEE Standards Terms, *ANSI/IEEE Std. 100*. Institute of Electrical and Electronic Engineers, New York NY, 2000.
8. Lientz BP, Swanson EB. *Software Maintenance Management*. Addison-Wesley: Reading MA, 1980.
9. Schach S, Jin B, Wright D, Heller G, Offutt AJ. Determining the distribution of maintenance categories: survey versus empirical study. *Kluwer's Empirical Software Engineering* 2003; **8**(4) in press.
10. di Lucca GA, Fasolino AR, de Carlini U. An algebraic notation for representing threads in object-oriented software comprehension. *Proceedings 9th International Workshop on Program Comprehension (IWPC 2001)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 176–185.
11. Zhifeng Y, Rajlich V. Hidden dependencies in program comprehension and change propagation. *Proceedings 9th International Workshop on Program Comprehension (IWPC 2001)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 293–299.
12. Kafura D, Reddy R. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering* 1987; **13**(3):335–343.
13. Kemerer CF, Slaughter S. Determinants of software maintenance profiles: An empirical investigation. *Journal of Software Maintenance* 1997; **9**(4):235–251.
14. Huffman Hayes J, Burgess C. Partially Automated In-line Documentation (PAID): Design and implementation of a software maintenance tool. *Proceedings Conference on Software Maintenance—1988 (ICSM 1988)*. IEEE Computer Society Press: Los Alamitos CA, 1988; 60–65.
15. Huffman Hayes J. Energizing software engineering education through real-world projects as experimental studies. *Proceedings IEEE Conference on Software Engineering Education and Training (CSEET 2002)*. IEEE Computer Society Press: Los Alamitos CA, 2002; 192–206.
16. Huffman Hayes J, Offutt AJ. Product and process: Key areas worthy of software maintainability empirical study. *Sixth IEEE Workshop on Empirical Studies of Software Maintenance (WESS 2000)*.
<http://members.aol.com/geshome/wess2000/janeHwess.PDF> [3 July 2003].
17. Banker RD, Datar SM, Kemerer CF, Zweig D. Software complexity and maintenance costs. *Communications of the ACM* 1993; **36**(11):81–94.
18. Chidamber S, Kemerer C. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.
19. Li W, Henry S. Object-oriented metrics that predict maintainability. *Journal of Systems and Software* 1993; **23**(2):111–122.
20. Brito e Abreu F, Carapua R. Object-oriented software engineering: Measuring and controlling the development process. *Proceedings of the Fourth International Conference on Software Quality (ICSQ 1994)*. American Society for Quality: Milwaukee WI, 1994; 1–8.
21. Welker KD, Oman PW. Software maintainability metrics models in practice. *CrossTalk: The Journal of Defense Software Engineering* 1995; **8**(11):19–23.



22. Halstead M. *Elements of Software Science*. Elsevier Science: New York NY, 1977.
23. VanDoren E, Sciences K, Springs C. *Maintainability Index Technique for Measuring Program Maintainability*. Software Engineering Institute, Carnegie Mellon University: Pittsburgh PA, 2002.
<http://www.sei.cmu.edu/activities/str/descriptions/mitmpm-body.html> [3 July 2003].
24. Oman P, Hagemeister J. Construction and validation of polynomials for predicting software maintainability. *Journal of Systems and Software* 1994; **24**(3):251–266.
25. Ramil JF, Lehman M. Metrics of software evolution as effort predictors—a case study. *Proceedings International Conference on Software Maintenance (ICSM 2000)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 163–172.
26. Gujarati DN. *Basic Econometrics* (3rd edn). McGraw-Hill: New York NY, 1995.
27. Polo M, Piattini M, Ruiz F. Using code metrics to predict maintenance of legacy programs: A case study. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 202–208.
28. Huffman Hayes J, Patel S. Examining the relationship between maintainability and reliability. *Technical Report CS901-03*, Department of Computer Science, University of Kentucky, Lexington KY, 2003.
29. Software Engineering Institute. Process maturity profile of the software community 2001 update, SEMA.8.01, slide 24. Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 2001.
30. Basili VR, Rombach HD. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering* 1998; **14**(6):758–773.
31. Higo Y, Ueda Y, Kamiya T, Kusumoto S, Inoue K. On software maintenance process improvement based. *Report*, Graduate School of Information Science and Technology, Osaka University, Toyonaka, Osaka, Japan, 2002.
32. Basili VR, Briand LC, Condon SE, Kim Y, Melo L, Valett JD. Understanding and predicting the process of software maintenance release. *Proceedings International Conference on Software Engineering (ICSE 1996)*. ACM Press: New York NY, 1996; 464–474.
33. Siy H, Votta L. Does the modern code inspection have value?. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 281–291.
34. Shirabad JS, Lethbridge TC, Matwin S. Supporting software maintenance by mining software update records. *Proceedings IEEE International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 22–31.
35. Constantine L. Methodological agility. *Computers in Libraries* 2001; **21**(6):67–69.
<http://www.sdmagazine.com/documents/s=730/sdm0106f/0106f.htm> [3 July 2003]
36. Schwaber K, Beedle M. *Agile Software Development with Scrum*. Prentice-Hall: Upper Saddle River NJ, 2001.
37. Cockburn A, Highsmith J, Johansen K, Jones M. Crystal methodologies. <http://crystalmethodologies.org/> [3 July 2003].
38. McCabe T, Butler C. Design complexity measurement and testing. *Communications of the ACM* 1989; **32**(12):1415–1425.
39. Nahm U, Mooney R. Text mining with information extraction. *Proceedings AAAI 2002 Spring Symposium on Mining Answers from Texts and Knowledge Bases*. American Association for Artificial Intelligence: Stanford CA, 2002; 60–67.
40. Wordsmyth Organization. Wordsmyth: on-line dictionary and thesaurus. <http://www.wordsmyth.net> [3 July 2003].
41. Basili V, Selby R, Hutchens D. Experimentation in software engineering. *IEEE Transactions on Software Engineering* 1986; **12**(4):733–743.
42. Basili V. Foreword to the goal/question/metric method. <http://www.iteva.rug.nl/gqm/forum.html> [3 July 2003].
43. Goldberg A. Measuring for progress. www.cse.ogi.edu/colloquia/event/20.html [3 July 2003].
44. Lorenz M, Kidd J. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall: Upper Saddle River NJ, 1994.
45. Chen JY, Lu JF. A new metric for object-oriented design. *Journal of Information and Software Technology* 1993; **35**(4):232–240.
46. Borland Software Corporation. Borland Together main product page. <http://www.borland.com/together/index/html> [3 July 2003].
47. Host M, Regnell B, Wohlin C. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering* 2000; **5**(3):210–214.
48. Tichy WF. Hints for reviewing empirical work in software engineering. *Empirical Software Engineering* 2000; **5**(4):309–312.
49. Huffman Hayes J, Zhang P. Fault detection effectiveness of spathic test data. *Proceedings IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002)*. IEEE Computer Society Press: Los Alamitos CA, 2002; 183–192.
50. Calvert K. *Curriculum for CS 571, Spring 2002*. Computer Science, University of Kentucky, Lexington KY, 2002.
51. Hart SG, Staveland LE. Development of a multi-dimensional workload rating scale: Results of empirical and theoretical research. *Human Mental Workload*, Hancock PA, Meshkati N (eds.). Elsevier Science: Amsterdam, 1988; 139–183.
52. Perot Systems Corporation. Welcome to Perot systems. <http://www.perotsystems.com/> [3 July 2003].



AUTHORS' BIOGRAPHIES

Jane Huffman Hayes is an Assistant Professor of Software Engineering in the Computer Science Department of the University of Kentucky. Previously, she was a Corporate Vice President and Operation Manager for Science Applications International Corporation. She received a MS degree in Computer Science from the University of Southern Mississippi and a PhD degree in Information Technology from George Mason University. Her research interests include software verification and validation, requirements engineering, software testing and software maintenance.



Tina Hong Gao is a research assistant in the Department of Computer Science at the University of Kentucky. Her research interests include software maintenance, testing, software architecture and Web engineering.



Naresh Mohamed is a tester with SWIFT Corporation in Culpeper, Virginia. He received a MS degree in Computer Science from the University of Kentucky. His research interests include software testing, software reliability and software maintenance.