Measuring, monitoring and controlling software maintenance efforts*

Markus Zanker
University Klagenfurt, Austria
markus.zanker@ifit.uni-klu.ac.at

Sergiu Gordea
University Klagenfurt, Austria
sergiu.gordea@ifit.uni-klu.ac.at

Abstract

Launching a new software product on the market and maximizing the profit obtained from it is already an art considering the high expectations and conflicting requirements of the software market. Product managers have to take tough decisions regarding the product life cycle like choosing the right moment for a new release, modernization or system replacement. Under these conditions they have to continually monitor the time evolution of efforts and costs required for implementing and maintaining the software product. Software tools that are able to assess these activities constitute a great value for those who need to take product management decisions. In this paper we present an approach for measuring and monitoring maintenance efforts by focusing on measurements of coding related activities. We employ data mining techniques for classifying coding related effort and present a preliminary evaluation of using this approach.

1 Introduction

Software Engineering Measurement (SEM) is a key practice in high maturity organizations. Together with Process/Project Management and Organizational Management it is one of the three Software Engineering Management subcomponents as stated in the Software Engineering Body of Knowledge [18]. The analytic paradigm of process measurement is relying on "quantitative evidence to determine where improvements are needed and whether an improvement initiative has been successful"[14]. Even if empirical evidence proves the importance and effectiveness of process measurement, small and medium sized organizations still have problems in implementing these models and in taking benefits from them. Several empirical research studies show that the success of measurement programs is typically low (about 20% [6]) and some of the highest improvement opportunities lie in activity planning and estimation as well as in schedule and process monitoring [2].

Talking about the legacy crisis", i.e. the high costs and efforts invested in software maintenance, Seacord et. al. present the evolution in time of information systems (see Figure 1). After making the first release of a system, the system enters into a maintenance phase with fewer functional improvements. Modernization or system replacement are periods of time when more new functionality is implemented into the system. A very similar life cycle is observed for systems implemented using iterative development. Many of the Open Source projects - which are already very popular for making scientific studies and evaluations - developed under the Apache Software Foundation [1] umbrella confirm this supposition. Choosing the right moment in time for releasing a new version of a system or replacing is a hard decision. A balance must be kept between different product related and economical requirements such as product reliability, efforts & costs, risks, budget or time to market [1]. Given this situation, monitoring the evolution of a product over time and the trends of its maintenance efforts are critical aspects for product management.

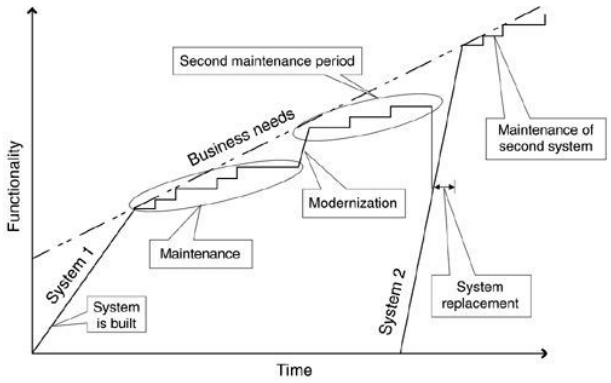

Abbildung 1. Information Systems life cycle [15]

In this paper we present an approach for identifying

*This work is carried out with financial support from the EU, the Austrian Federal Government and the State of Carinthia in the Interreg IIIA project Software Cluster South Tyrol - Carinthia

[1] http://www.apache.org

maintenance activities based on fine grained time information about the development effort and on software metrics documenting the evolution of the software product.

2 Background

According to Seacord et al. [15] we can differentiate reasons for software change and maintenance efforts into four categories:

- **corrective**: repairing system defects
- **perfective**(enhancements): adding user requirements, enhancing performance or usability
- **preventive**: improving future maintainability and reliability of the system
- **adaptive**: adapting the system to work in different environments (e.g. working on different platforms, databases or integrating new technologies)

Studies published in the eighties show that the corrective category takes about 20% of the total maintenance effort while adaptive efforts represent about 25%. The largest share, i.e. 50% is devoted to perfective actions. Preventive actions, typically take no more then 5% of the total maintenance efforts [15]. More recent studies confirm similar distributions [19] [11]. Over time information systems increase their size, because rarely effort for removing unused or duplicated code is funded. The average Fortune 100 company, for example, maintains 35 million lines of code and adds 10 percent each year only in enhancements, updates, and other maintenance activities. As a result, the amount of code maintained by these companies doubles in size every seven years [7]. Studies made by Fjelstad and Hamlen in '83 show that the maintainers spent 50% of their time for understanding the source code they maintain [3]. While maintenance effort increases proportionally with the size of software systems, code reduction (by eliminating unused code or by removing duplicates from the source code) provides a way of reducing maintenance costs.
As information systems grow in size and functionality, more effort is required to be invested in maintenance activities, and less resources remain to be allocated for new development. The legacy crisis denotes a situation where due to a continuous increase of maintenance efforts no more resources are available for developing new systems, or new functionality into an existing system. This crisis is typically addressed by developing evolvable systems (which requires additional time and resources to keep systems maintainable), by software re-engineering and modernization as well as by employing advanced programming languages and tools, commercial components, or migrating to web and internet applications. However, none of these alternatives alone is sufficient to prevent or solve the crisis, but only a combination of these techniques might help to keep software maintenance manageable[15]. Iterative software development suffers from a comparable situation when the effort required to maintain old versions of a system reduces the resources initially allocated for developing a newer version.
Looking at maintenance efforts from a marketing perspective, perfective and adaptive maintenance add new functionality into a system or adapt it for working in new environments, therefore they increase the value of a software product. Knowing the amount of effort invested into perfective activities and relating it to total maintenance effort is therefore extremely important for management.
Statistical process control (SPC) is the use of statistical techniques and tools to measure an ongoing process for change or stability. The main purpose of SPC is to prevent defects by monitoring the process while parts or a service is being employed, instead of detecting defects after the process is finished"[17]. Employing SPC when monitoring maintenance activities helps managers to answer questions like: *Should we modernize or replace the system?* or *Do we have to schedule source code refactoring activities?* Building control charts that monitor evolution in time of maintenance efforts (especially corrective and perfective) will identify the cost trends and help managers to make better decisions. For example, when corrective maintenance constantly increases until the point is reached where more effort goes into corrective actions than in perfective ones, refactoring and modernization measures need to be taken. Similarly, these kinds of decisions need to be made when the relative increase in corrective maintenance is too high, or productivity of perfective activities decreases under a prespecified control limit (see Figure 2).

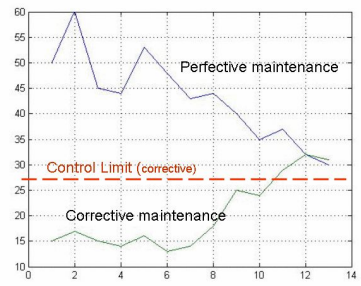

Abbildung 2. Perfective and Corrective maintenance control chart

In the rest of the paper we present our approach for classifying development effort into coding related activities, that can be used for both measuring maintenance efforts (see Table 1) and for monitoring software development process. Manually collected time information is expensive

and the data is often erroneous, therefore we employ the PROM tool [16] for automated collection of the effort spent for implementing new source code or maintaining existing one. Sillitti et al. [16] collect time efforts as well as software metrics at the method level. We analyze the evolution of the software product over time and classify the associated development effort into one of the following maintenance activities:

- implementation of new functionality
- implementation of test cases
- source code improvement (refactoring)
- correction and maintenance of source code
- source code browsing

Each type of activity modifies the source code in a specific way. Implementation of new functionality increases the size and complexity of a software, while the implementation of test cases is reflected by an increased number of test classes and test methods. Corrective maintenance operates on existing code and modifies its complexity without heavily modifying its structure (compare Tables 7,8). Refactoring aims at eliminating duplicates in the source code and enhances reusability of the software, thus it improves class cohesion. The structure of the source code is simplified. We use software metrics to identify these changes in the structure and size of software (see Figure 3) like: lines of code (LOC), the number of methods (NOM), the number of classes (NOC), response for a class (RFC), Fan-in (FI), Fan-out (FO), number of children (NC) and depth of inheritance tree (DIT) [13].

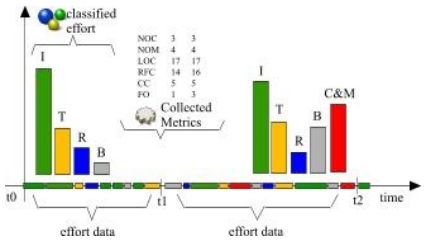

Abbildung 3. Development effort classification

As can be seen from Table 1, maintenance effort can be directly measured by summing up the effort spent in coding related activities with the exception of adaptive maintenance efforts that probably includes source code replacements, exchanges of previously used libraries, or maybe just new system add-ons. Adaptive maintenance is rather complex. Therefore during a fixed period of time typically all resources are focused on the accomplishment of such a specific task. Which means that the adaptive effort can be measured by simply summing all efforts from this period of time.

Tabelle 1. Maintenance category - Development effort

Maintenance	Development activity
corrective	correction & maintenance of source code
perfective	implementation of new functionality
preventive	implementation of test cases, refactoring
adaptive	replacing modules, libraries, technology

3 Analysis

The effects of a specific coding activity are observed through changes of software metrics. Each activity modifies these measures in a particular way. For example, the implementation of new functionality is reflected in software metrics through increased values of NOC, NOM, LOC, RFC, and FO metrics. The implementation of test cases is determined from the class inheritance tree (e.g. using *instance of TestCase* operation) or even from parsing of method names if naming conventions are obeyed. While these examples are quite straightforward, a source code improvement activity is more challenging. It typically changes the structure of the software without modifying its functionality. This activity usually preserves the size of the source code or even reduces it by eliminating duplicated code. The algorithms used to detect refactorings are based on the *origin analysis* (OA) concept [5]. The models proposed by Kontogiannis [10] and Germain [4] aim at identifying structural changes in the source code based on software metrics like: RFC, FI, FO, DIT, NC and NOC. In the following we show our approach for inducing activity patterns starting from concrete code examples.

3.1 Coding related activities

Implementation of new functionality (I). The following code sample gives an example how software metrics evolve when new functionality is implemented. Table 2 represents the values of metrics before and after completion of the maintenance activity. One can observe an increase of evaluations for all metrics that measure the size of the source code.

Example 1. Initial version of class Searcher

```
public class Searcher {
    private static final String FILENAME= "filename";

    public static List search(File indexDir,
        String queryString)throws IOException, ParseException{
```

Tabelle 2. Metrics. Implementation of new functionality

entity	metric	before	after
	NOC	0	1
Searcher	NOM	0	1
Searcher	LOC	0	12
Searcher	RFC	0	11
Searcher	CC	0	2

Tabelle 3. Effort. Implementation of new functionality

entity	time
Searcher.java/Searcher::search	835
Searcher.java/Searcher	136
Searcher.java	10

```
        Directory fsDir = FSDirectory
                .getDirectory (indexDir, false);
        IndexSearcher indexSearcher = new IndexSearcher(fsDir);

        Analyzer analyzer = new StandardAnalyzer();
        Query query = QueryParser.parse(queryString, CONTENT,
                analyzer);

        Hits hits = indexSearcher.search(query);

        List results = new ArrayList(hits.length());
        for (int i = 0; i < hits.length(); i++) {
            Document doc = hits.doc(i);
            results.add(doc.get(FILENAME));
        }
        return results;
    }
}
```

Implementation of test cases (T). For implementing unit tests the JUNIT[2] or CACTUS[3] libraries are quite popular among java developers. Both extend the TestCase class of JUNIT (CACTUS is built on the top of JUNIT). Typically, test classes obey naming conventions like: *all test classes have a test prefix (or suffix) as part of their names*. Knowing these conventions it is even easier to identify test classes. Like implementing new functionality, the values of all metrics are increased for the development of test cases.

Example 2. Test case code fragment

```
import junit.framework.TestCase;

public class TestIndex extends TestCase{
    public void testIndex(){
        ...
    }
}
```

Source code improvement (R). Refactoring is a technique intensively used in agile development. Refactorings are changes of the internal structure of source code that enhances its readability and understandability without changing its observable behavior [5]. Refactorings may occur at different levels of the project structure, i.e. at the method, class, package, or even architecture level. Refactoring activities effect important changes of the structure of the source code without a significant increase in size and complexity (see Table 5). For instance, the *search()* method from the Searcher class was substituted by three other methods namely doSearch, buildQuery and getIndexSearcher in order to increase class modularity and reusability. In the present case, the size of the source code slightly increased while the value of the RFC metric increased by three due to the introduction of calls to the newly created methods (compare Table 5). Code size could even have decreased if some of the methods would have eliminated code duplicates from other methods. The structure of the code was modified by the creation of new methods but the complexity of the *search()* method was reduced, which is reflected by the value of the FO metric.

Tabelle 4. Effort. Implementation of test cases

entity	time
TestIndex.java/TestIndex::testIndex	701
TestIndex.java/TestIndex	24

Example 3. Class Searcher after refactoring

```
public class Searcher {
    private static String CONTENT = "content";

    public static List search(File indexDir, String
        queryString)throws IOException, ParseException{

        IndexSearcher indexSearcher = getIndexSearcher(
                        indexDir);
        Query query = buildQuery(queryString);
        return doSearch(indexSearcher, query);
    }

    static List doSearch(IndexSearcher indexSearcher,
        Query query)throws IOException {

        Hits hits = indexSearcher.search(query);
        List results = new ArrayList(hits.length());

        for (int i = 0; i < hits.length(); i++) {
            Document doc = hits.doc(i);
            results.add(doc.get(Indexer.FILENAME));
        }

        return results;
        }

    static Query buildQuery(String queryString)
        throws ParseException {

        Analyzer analyzer = new StandardAnalyzer();
        Query query = QueryParser.parse(queryString,
                        CONTENT, analyzer);
        return query;
    }

    static IndexSearcher getIndexSearcher(File indexDir)
        throws IOException {

        Directory fsDir = FSDirectory.getDirectory(
                            indexDir, false);
        IndexSearcher indexSearcher = new IndexSearcher(
                            fsDir);
        return indexSearcher;
    }
}
```

Correction and maintenance of source code (C&M). Each project iteration ends up with testing activities that

[2] see http://www.junit.org

[3] see http://jakarta.apache.org/cactus/

Tabelle 5. Metrics. Source Code Improvement

entity	metric	before	after
	NOC	3	3
Searcher	NOM	1	4
Searcher	LOC	12	17
Searcher	RFC	11	14
Searcher	CC	2	5
search	FO	11	3

Tabelle 6. Effort. Source Code Improvement

entity	time
Searcher.java/Searcher::search	350
Searcher.java/Searcher	136
Searcher.java	36

NOTE: $RFC_after = RFC_before + (NOM_after - NOM_before)$. No effort was registered as being spent for writing the 3 new generated methods.

aim at discovering defects in the system. The removal of these defects is accomplished within a correction and maintenance phase. The modification of the complexity of an existing artifact (source file) without strongly affecting its structure is typical for this category of activity. The following changes in the buildQuery() method include two corrections: the Searcher should use the same analyzer as the Indexer and the search must be over all index fields (changed lines are marked with *).

Example 4. Maintained method buildQuery

```
static Query buildQuery(String queryString)
throws ParseException{

    Analyzer analyzer = Indexer.getAnalyzer();*
    String[] fieldNames = Indexer.getFieldNames();*

    Query query = MultiFieldQueryParser
        .parse(queryString, fieldNames, analyzer);*

    return query;
}
```

Tabelle 7. Metrics. Correction and maintenance of the source code

entity	metric	before	after
	NOC	3	3
Searcher	NOM	4	4
Searcher	LOC	17	17
Searcher	RFC	14	16
Searcher	CC	5	5
buildQuery	FO	3	3

Tabelle 8. Effort. Correction and maintenance of the source code

entity	time
Searcher.java/Searcher:: buildQuery	59
Searcher.java	12

NOTE: The LOC metric remains constant because one line of code was added and one was removed (the CONTENT attribute)

Source code browsing (B). When writing code, software developers search for methods they need to reuse when implementing new functionality. Furthermore, they review source code in order to check for potential bugs and misbehavior. They need to understand what related methods are doing when modifying an existing code fragment. Each type of maintenance activity includes therefore additional effort for accessing source code entities without changing them.

3.2 Classification Methodology

Based on their methodology data mining algorithms for effort classification can be grouped into three categories: Math algorithms (based on linear regression, neural networks, statistical analysis, etc.), algorithms based on distance solutions (nearest neighborhood) and algorithms based on logic solutions (decision trees, decision rules) [20]. Considering the specifics of coding related activities, we deemed a logical representation for activity patterns to be appropriate. Indeed, properties like increased complexity, simplified structure, high development effort etc., are very close to Boolean variables and comparison operators. Logic solutions have the advantage of good explanatory capabilities. Small decision trees are very intuitive for human users, they are easy to be implement and quite fast compared with other non linear algorithms. Decision trees and decision rules are related logical representations, a decision tree can be easily transformed into a set of decisions rules by representing each path of the tree as a decision rule.

Activity patterns induction. In the previous section we described the characteristics of each coding related activity and the way they affect the structure and the size of source code. In order to identify test case implementation and refactoring effort we need additional information such as class supertype or origin analysis of refactored software code. Table 9 gives a categorization of development effort with respect to a set of variables including source code structure (STR), size (SIZE), effort (EFF), test class (TC) and origin analysis (OA). Value 0 in the structure column signifies that the structure of the source code is simplified or remains the same, while 1 stands for the opposite. The structural changes of the source code are identified on the basis of NOC,

Tabelle 9. Effort Classification. Table of truth

STR	SIZE	EFF	TC	cat.
0	0	1	0	B
0	0	0	1	T
0	1	0	0	C&M
0	1	0	1	T
0	1	1	0	C&M
0	1	1	1	T
1	0	0	0	R
1	0	0	1	T
1	0	1, OA=0	0	C&M
1	0	1, OA=0	1	T
1	0	1, OA=1	0	R
1	0	1, OA=1	1	T
1	1	1	0	I
1	1	1	1	T
0	1	0	0	T
0	1	0	1	T
0	1	0	0	T
0	1	0	1	T

I= implementation of new functionality, T= implementation of test cases, R= refactoring, B= Browsing, C\&M= correction \& maintenance

NOM and DIT metrics. Comparable, the size of source code increases (value 1) when at least one of the LOC, RFC, FO, or CC metrics is increased (usually all of these variables will change together). OA variable represents the result of the origin analysis test, which verifies if newly created methods implement new code or if they are extracted by refactoring. The TC variable indicates if the edited artifacts were test cases (i.e. they extend junit.TestCase).

The classification rules are learned using top down induction algorithms available in data mining tools. In this particular case we used Matlab statistical toolbox for learning the rules presented in Table 10.

Tabelle 10. Activity patterns rules

$T = TC$
$I = \neg TC \bigwedge STR \bigwedge SIZE$
$B = \neg TC \bigwedge \neg STR \bigwedge \neg SIZE$
$R = (\neg TC \bigwedge STR \bigwedge \neg SIZE \bigwedge \neg EFF)$
$M = (\neg TC \bigwedge \neg STR \bigwedge SIZE) \bigvee (\neg TC \bigwedge STR \bigwedge \neg SIZE \bigwedge EFF \bigwedge \neg OA)$

4 Evaluation

In order to verify the effectiveness of the presented approach we instrumented two empirical evaluations. One using a student project and another collecting data from a commercial development.

Tabelle 11. Effort classification accuracy

category	P(event)	R(event)	P(effort)	R(effort)
I	0.997	1	0.99	1
C&M	0.87	0.75	0.79	0.89
T	1	1	1	1
B	0.69	0.83	0.79	0.64

Evaluation experiment 1 In our first experiment, students were developing a desktop java application on top of the Eclipse Rich Client Platform [4]. The team was composed of 3 developers working part time on the given project. The collected effort was exported into a spreadsheet at the end of each day and the students were asked to classify each entry into one of the activity categories. We collected product metrics each night, and performed an automated classification of the effort data. We conducted the experiment over a period of 3 calendar weeks, during which the students worked effectively for 9 days.

The classification accuracy was evaluated with the help of *precision* and *recall* metrics. Given a development activity, *precision* measures the ratio of efforts which are identically classified by developers and by the classification model as belonging to the same category X, and the total amount of efforts classified by the classification model as category X. Comparable, the *recall* metric computes how much effort is identically classified into category X from the total effort assigned by developers to category X. Table 11 presents the precision and recall measures for each activity type, first by employing the annotations for each event, and second by using the development effort itself.

Due to its straightforward classification rule, based on only one variable, implementations of test cases were identified with a precision of nearly 100%. The classification accuracy when detecting source code corrections (C&M) was a little less accurate (with a precision of $\sim$87%). There were two reasons that led to wrong classifications. First, the classification of development efforts was only done once in a day for all activities based on metrics collected from the source code repository. This way, the algorithm is not able to identify different activities performed by different users on the same source code fragment during this time interval. The second reason was the fact that not all source code was committed into the repository at the end of each day. The students were inexperienced in using continuous source code integration practices, and were afraid of committing source code that might not compile and break the metric

[4] See http://www.eclipse.org/ercp/

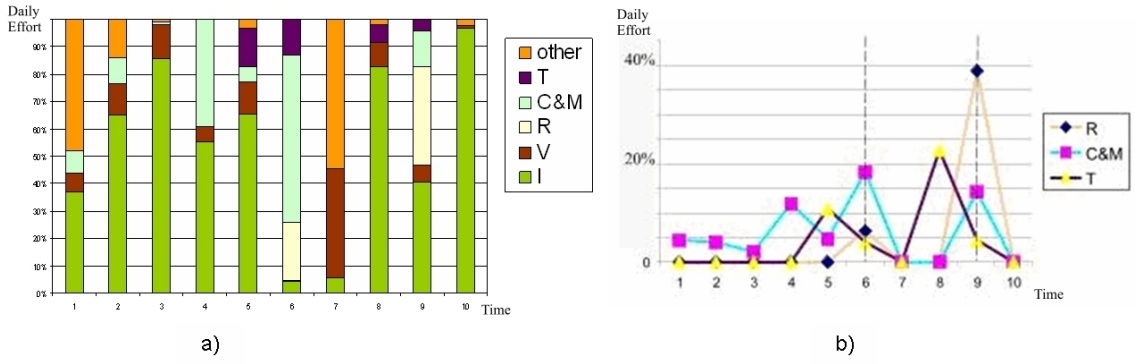

Abbildung 4. Distribution over time period of coding related effort. a) Measuring daily distribution of coding related effort (percentage scale) b) monitoring development cycles

collection [5]. Considering the fact that source code browsing effort is relatively small in comparison with the other effort categories (approximatively 5%), the classification accuracy is considered at least satisfactory. The student were not using refactoring techniques, therefore we were not able to verify the classification accuracy for this category.

Evaluation experiment 2. In our second experiment we monitored the activities of a more experienced team of commercial developers that used agile development techniques in their work. The team used a two weeks iteration cycle, therefore we run an evaluation over a period of 10 working days. The project team develops a web based java application, therefore some effort was spent on editing non java artifacts (e.g. html files, templates, properties files, or css files). These efforts were classified as *other effort*. In Figure 4a) we represent the classified efforts on a percentage scale, representing them as a share of the total daily development time. Contrasting the student experiment, the largest amount of the time was devoted to implement new functionality ($\sim 59\%$). Hence, refactoring and unit testing techniques were more intensively used. The second chart denotes the effort spent in refactoring, correction & maintenance, and testing related activities. This allows monitoring the implementation life cycle: implement test refactoring correct & maintain (in this case). In the presented time interval we captured two implementation cycles of major functionality that ended on the 6th and 9th day with a relatively high amount of effort spent in writing test cases, code refactoring and bug fixing. The chronology of the effort spent in writing test cases reveals the fact that the development team was not using the test first development style, promoted by agile developers, but a test after development style. The test cases were implemented in both cases at the end of the implementation cycle.

[5] Compilable code was a prerequisite for metric collection.

Both experiments collected information only over a relatively short period of time (only a few weeks) and represented therefore only an initial evaluation of our approach. In order to support product life cycle decisions of management, it is required to repeat the second experiment over a longer time period, which is one of our goals for future research.

5 Related work

Maintenance of software can become very costly: There are studies that show that corrective maintenance costs might reach 90% of the total costs spent during the product life time leading into the so called legacy crisis [15]. Therefore growing interest for measuring maintenance costs can be observed and approaches to reduce these costs are in the focus of research. Studies published in the eighties measured the distribution of maintenance efforts over the four categories of maintenance in order to better understand the process before trying to reduce its associated cost [7]. Due to the important changes of software development processes and working environments, more recent studies try to verify if the results of previous measurements are comparable to new situations. Lee and Jefferson measure how maintenance effort is distributed for web application development [11]. Other research aims at improving decision support by monitoring the evolution of software metrics on the complexity and quality of software products over time. Bhawnani et. al. present a model for providing decision support for software release decisions by monitoring the evolution of defect rates [1].

Another area of research focuses on applying data mining techniques to software engineering. They usually extract knowledge from source code repositories and issue tracking systems, to measure and predict the quality of soft-

ware products. Software metrics are basically employed to estimate the development and maintenance efforts, as well as to measure different aspects of the quality of software products. [12] uses a similar approach, using metrics based rules for detecting design flaws. Applying *detection strategies* (a concept equivalent to activity patterns) the authors report good results in detecting classes and methods affected by particular design flaws (e.g. God Class). Hassan [8] mines the evolutionary history of software systems in order to guide software practitioners (managers, developers) to develop better software. The authors analyze source code changes introduced by three types of activities: fault repairing, introduction of new features, and maintenance in general.

6 Conclusions

The measurement of maintenance costs became a hot topic since the legacy crisis of software industry. Despite all technological improvements, the maintenance costs remained relatively high for most information systems. Due to strong competition on software markets, the sales prices of standard software products decrease continuously even if their quality and functionality increases. These developments give priority to support management decisions by automated measurement and monitoring tools. In this paper we presented an approach for measuring and monitoring the development and maintenance efforts, and propose the usage of statistical process control for supporting decision making regarding software products release and evolution. When employing tools like PROM [16] or HackyStat [9] for data collection the process can be fully automated. This contrasts other approaches that are based on manually collected information.

The classification rules have been induced by domain knowledge that was transformed into boolean variables. The categorized effort information can be used for monitoring development activities and supports managerial decision making. Our first experimental evaluations gave us promising results, and encourage us to continue research into this direction.

Literatur

[1] P. Bhawnani, B. H. Far, and G. Ruhe. Explorative study to provide decision support for software release decisions. In *21st IEEE International Conference on Software Maintenance Proceedings (ICSM'05)*, 2005.

[2] M. Brown and D. Goldenson. Measurement analysis: What can and does go wrong? In *METRICS'04 Proceedings*, pages 131–138, 2004.

[3] R. Fjeldstad and W. Hamlen. Application program maintenance report to our respondents. *Tutorial on Software Maintenance*, pages 13–27, 1983.

[4] E. Germain and P. N. Robillard. Activity patterns of pair programming. *The Journal of System and Software*, pages 17–27, 2005.

[5] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2), 2005.

[6] W. Goethert and W. Hayes. Experiences in implementing measurement programs. Technical Report 2001-TN-026, Carnegie Mellon University/Software Engineering Institute (SEI), 2001.

[7] K. W. H. A. Mller and S. R. Tilley. Understanding software systems using reverse engineering technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, pages 41–48, 1994.

[8] A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers*. PhD thesis, University of Waterloo, Ontario, 2004.

[9] P. Johnson, H. Kou, J. Agustin, C. Chan, C. M. J. Miglani, S. Zhen, and W. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *ICSE '03 proceedings*, 2003.

[10] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE '97 proceedings*, 1997.

[11] M.-G. Lee and T. L. Jefferson. An empirical study of software maintenance of a web-based java application. In *21st IEEE International Conference on Software Maintenance Proceedings (ICSM'05)*, 2005.

[12] R. Marinescu. Metrics-based rules for detecting design flaws. In *ICSM '04 proceedings*, 2004.

[13] S. L. P. Norman Fenton. *Software Metrics: a rigurous and practical approach (second edition)*. PWS Publishing Company, 1997.

[14] Project Management Institute, editor. *A guide to Project Management Body of Knowledge*. Project Mangement Institute, 2004.

[15] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. P Addison Wesley Professional, 2003.

[16] A. Sillitti, A. Janes, G. Succi, and T. Vernazza. Measures for mobile users: an architecture. *Journal of Systems Architecture*, pages 17–27, 2004.

[17] D. Stamatis. *Six Sigma and Beyond: Statistical Proces Control*. St. Lucie Press, 2003.

[18] The Institute of Electrical and Electronics Engineers, editor. *Software Engineering Body of Knowledge*. IEEE Computer Society Press, 2001.

[19] H. V. Vliet. *Software engineering: principles and practice*. John Wiley, 2000.

[20] S. M. Weiss and N. Indurkhya. *Predictive Data Mining*. Morgan Kaufmann Publishers, 1998.