A Lightweight Tool Support for Integrated Software Measurement

Bernhard Daubner[1], *Andreas Henrich*[2], *Bernhard Westfechtel*[1]

[1]Bayreuth University, Chair of Software Engineering
bernhard.daubner@uni-bayreuth.de
bernhard.westfechtel@uni-bayreuth.de

[2]Bamberg University, Chair of Media Informatics
andreas.henrich@wiai.uni-bamberg.de

Abstract:

This article shows a lightweight approach to implement a tool supported software measurement process. The basic idea here is to concentrate on five relevant software measures and tie them on the elements of an implicitly given skeletal structure of the project. We show several ways to provide such a skeletal structure for various types of software projects.

Based on this skeletal structure the software measures to collect can be defined in advance and independently of a concrete project. At project runtime the entities to measure are automatically identified by means of the elements of the skeletal structure. The computation of the software measures can then be automated using Open Source tools.

Keywords

Measurement techniques, measurement tools, automatic software measurement, Open Source tools

Zusammenfassung:

In diesem Beitrag wird ein leichtgewichtiger Ansatz zur Implementierung eines werkzeuggestützten Verfahrens zur Softwaremessung dargestellt. Die Grundidee ist dabei, sich auf fünf wichtige Softwaremaße zu beschränken und diese an den Elementen einer implizit gegebenen Grundstruktur des Projekts zu verankern. Wir zeigen verschiedene Möglichkeiten auf, um eine derartige Grundstruktur für unterschiedliche Projektarten anzugeben.

Unter Verwendung dieser Grundstruktur können die zu erfassenden Softwaremaße im voraus und unabhängig von einem konkreten Projekt festgelegt werden. Zur Laufzeit des Projekts werden dann über die Elemente der Grundstruktur die zu messenden Entitäten automatisch erkannt. Die Berechnung der Softwaremaße erfolgt dann automatisiert mittels Open-Source-Werkzeuge.

Schlüsselbegriffe

Messverfahren, Messwerkzeuge, automatisierte Softwaremessung, Open-Source-Werkzeuge

1 Introduction

It is widely accepted that software measurement has to be comprehensively integrated into the software development process [3] and in doing so the gained experiences have to be stored for later reuse [2]. In order to get consistent data quality the measurement process has to be automated as far as applicable [1]. This leads to a demand for integrated measurement tool support within the software engineering environments. For these tools we use the term *Software Project Control Center* (SPCC) [19].

SPCCs shall provide project managers with characteristics about ongoing IT projects. Such characteristics might comprise e.g. test coverage, completion rate, code quality or effort spent. It is however crucial for the implementation of a SPCC that these characteristics can be measured automatically. In order to actually gain decision relevant information, the software measures have to be collected in such a manner that measurement results are comparable between several projects and meaningful in this respect.

Published approaches for such SPCCs either are restricted to certain proprietary software development environments ([23], [17], [8], [14]) or are only available as a conceptual model ([3], [13], [19]).

Since software measurement should be done in a top-down or goal-oriented manner ([13], [14]), for each measurement project it has to be worked out what to measure. Then the measurement framework must be tailored respectively. This makes the establishment of a measurement process a complex and error-prone task. On the other hand it is common consensus that the software measures *size*, *time*, *effort*, *quality* and *productivity* nearly always have to be surveyed [22] and are therefore also called the *Five Core Metrics* [21].

But also these relatively "simple" measures have to be collected together with their context in order to be meaningful. Therefore we want to present a lightweight approach to anchor these measures on the components of an inherently existent *skeletal structure* of the projects or on the elements of an *implicitly given project plan*. We collect these measures using the open source project management tool *Maven* and since we rely on this common inherent structure of the projects also the selection of software measures to collect can be defined in advance and independently of a concrete project. This reduces the complexity of the measurement process and allows to introduce software measurement also within development teams that otherwise would abstain. Additionally our approach for a SPCC makes it easy to reuse already defined measures also for other projects.

2 Related Work

As stated in the previous section the SPCCs known to the authors depend on proprietary software development environments. The *Amadeus* system [23] for example is integrated in a software environment called *Arcadia*. The approaches described in [17] or [8] use workflow engines, that tell the users what tasks they are expected to do and collect data that are associated with the realization of these tasks. And the extension to the *MetriFlame* system described in [14] has to be expensively customized for every project.

The authors of [13] present guidelines for setting up measurement tool support for software development processes and describe also dependencies between goals, activities and products within the measurement process. Hereby the framework allows companies to use their existing tools and processes. Within the presented case studies however much of the data collection was done manually. In [19] a reference model for concepts and definitions around SPCCs is given. Thereby the authors describe very thoroughly the logical architecture of an SPCC. A tool support for the proposed architecture however is not yet available.

3 A Lightweight SPCC

So it is well known how a SPCC should be implemented, whereby most approaches use the *TAME* software development model [3] as basis to construct a SPCC ([17], [13], [14], [19]).

Also our approach roughly fits into the concepts given in [13] and [19]. However in concentrating on the *five core metrics* we can shorten the measurement planning process. We do not perform a complete *Goal Question Metric* analysis to determine the relevant software measures. We claim that collecting these five metrics provides a value per se (see Section 3.2).

In contrast to the above mentioned approaches, which partially dictate the use of certain process models or tools, we want to present a rather lightweight approach to implement a SPCC on basis of Open Source tools.

The basic idea of our approach is to tie the software measures on the elements of the project plan as suggested in [18] in order to be able to conduct software measurement automatically. For that reason the entities that have to be measured (deliverables, issue reports, working hours, etc.) have to be labeled suitably. As we have shown in [9] and [10] it is possible to tag check-ins within the configuration management system, issues within the bug tracking system and time records within the time recording system with the identification code of the corresponding work package of the work breakdown structure.

More generally we anchor the software measures on the elements of an inherently existent *skeletal structure* of the projects or on the elements of an *implicitly given project plan* that the software projects under examination are based on. Since we collect the software measures within the context of this consistent skeletal structure we reach an implicit result packaging as demanded in [3].

3.1 Structuring the Software Development Process

This skeletal structure can be based on a standard work breakdown structure, a process model or an implicitly given generic project plan depending on the type of the process model that will be used for the project. Government software development projects for instance might be based on a very strict process model that dictates to create a work breakdown structure. Agile software development methodologies however, do not use a fixed work breakdown structure. But we show that for every kind of project model it is possible to find a fixed skeletal structure the project is based on. The elements of this skeletal structure are then labeled with an identification code that we need, in order to perform the measurement.

3.1.1 The Work Breakdown Structure

According to [20] the *Work Breakdown Structure* (WBS) is a deliverable-oriented hierarchical decomposition of the work to be executed in the course of the project to reach the project objectives. The WBS subdivides the project work in a hierarchical manner into smaller, more manageable pieces of work that are refered to as tasks, subtasks and work packages. This leads to a tree structure which contains all activities that are within the scope of the project.

Generally each component in the WBS is assigned a unique identifier. These identifiers are often known as *code of accounts* or simply *WBS code*. As the term *code of accounts* suggests, we can use these identification codes to assign software measures to the corresponding portion of the project. As most projects within a given organization will have the same or similar project live cycles [20], it is very common to base the actual WBS on a standard WBS template. By means of the WBS codes of this standard WBS we can assure that given software measures can always be applied to the same corresponding pieces of work. Typically a decade code scheme is used to label the elements of a WBS (see Figure 1). Hereby the position of every element within the WBS is uniquely determined by its identifier.

Especially when a strongly hierarchically structured process model like e.g. the German *V-Modell-XT* is used, it is comparatively easy to derive a standard WBS from the process model.

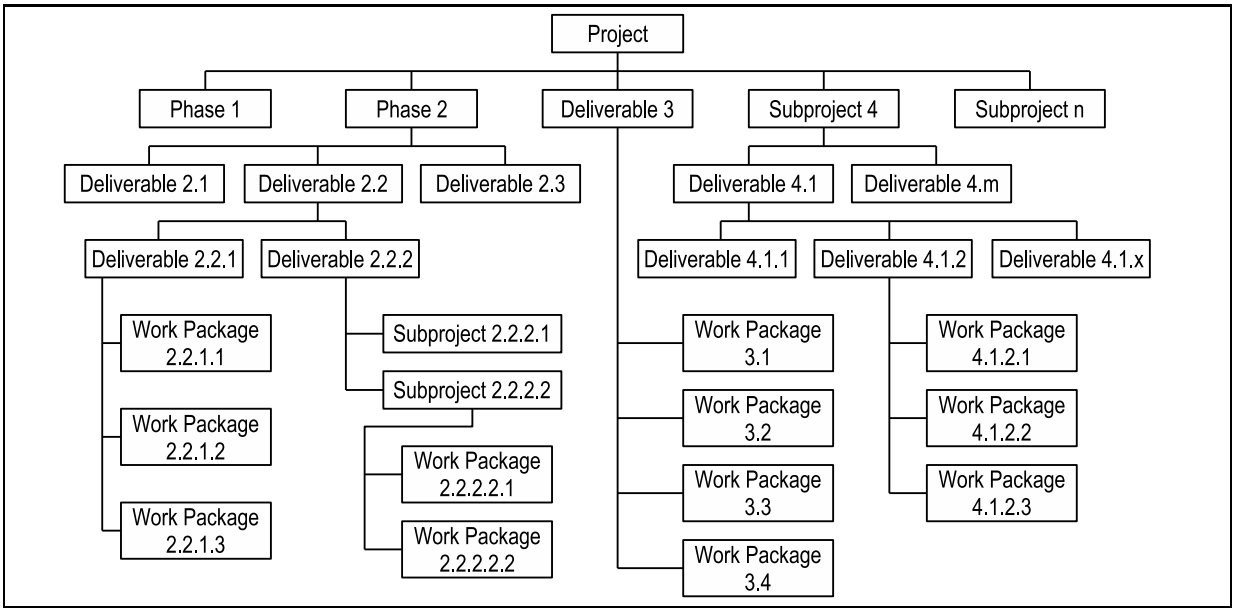

Figure 1: WBS with several decomposition strategies according to [20]

3.1.2 Structure of the Process Model

Especially when planning an iterative project the WBS either is created and revised also iteratively or does not even exist in that form. According to the *PMBOK Guide* [20] the management of the WBS is part of the knowledge area *Project Scope Management* and the maintenance of the WBS is done within every project phase. Therefore the WBS subtrees of deliverables that are dealt with in future phases are relatively unelaborate.

Hence the *Rational Unified Process* (RUP) does not know a WBS in the above sence ([16], [15]). Kroll and Kruchten point out that projects using a deliverable oriented WBS would be organized around a product breakdown. This would lead to the product structure being elaborated to early, when little is known about the product [15].

Therefore planning a RUP project is focused on a process breakdown and the primary planning documents are the *project plan* and the *iteration plans*. The project plan is a coarse-grained plan and describes mainly the milestones of the project and thus the project phases and their objectives. An iteration plan is fine-grained and there exist one per every iteration. It lists the activities and tasks that have to be performed during the particular iteration.

In order to apply our measurement approach on iterative process models like the RUP you can either use again a standard WBS or use the structure of the process model itself as skeletal structure to anchor the software measures.

As it is not uncommon to use the phases of the project life cycle as the first level of decomposition within the WBS [20], it is possible to create a standard

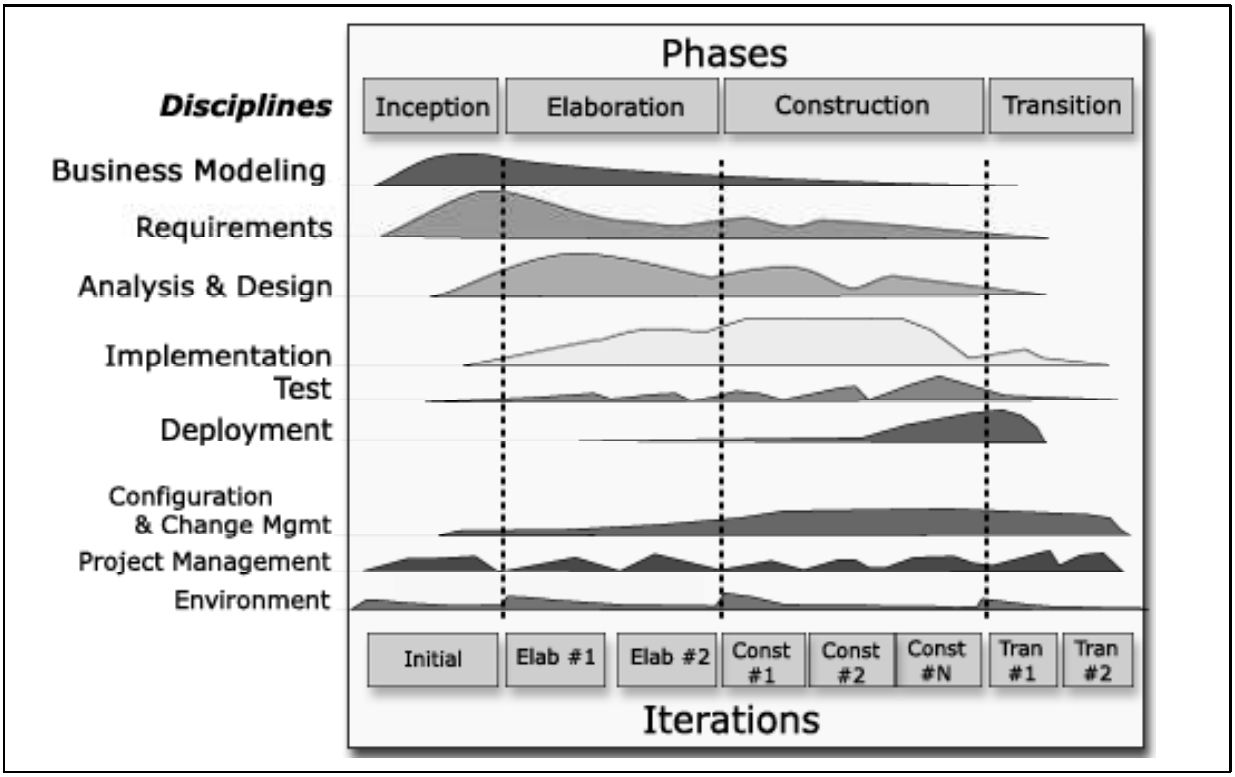

Figure 2: Structure of the *Rational Unified Process* [16]

WBS with the project phases at the first level of decomposition and generic deliverable names (e.g. *User-Interface*, *Database*, *Documentation*) at the second level. This however is only possible, if all projects have the same application area (e.g. Internet Portal Software).

But it seems to be more efficient to use the internal structure of the RUP to anchor software measures. Every activity can be associated with a phase and a discipline it is performed within (Figure 2). Thus information about the activity (time spent, generated code size) can be tagged with a 2-tupel indicating phase and discipline. Since the milestones at the end of each phase are key business-level decision points for the project, where major decisions must be made about the continuation of the project and its scope or schedule [15], it is useful to be able to compute software measures in terms of phases. Also the measurement of activities with respect to the disciplines provides useful information because it is for instance possible to recognise requirement creeps.

What is more, the measurement can even be conducted more fine granular if not the disciplines (also called *core workflows*) but the *workflow details* are used to tag the information about the performed activities. For example the discipline *Requirements* contains amongst others the workflow details *Analyze the Problem* and *Manage Scope of the System*. Therewith a 2-tupel indicating phase and workflow detail can be used as meta information for activities and artifacts pro-

	Analysis	Modeling	Implementation	Test	Deployment	Config. Mgmt	Project Mgmt.
Application Logic	$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$	$(5,1)$	$(6,*)$	$(7,*)$
Presentation	$(1,2)$	$(2,2)$	$(3,2)$	$(4,2)$	$(5,2)$	$(6,*)$	$(7,*)$
Data Management	$(1,3)$	$(2,3)$	$(3,3)$	$(4,3)$	$(5,3)$	$(6,*)$	$(7,*)$
Transaction Mgmt.	$(1,4)$	$(2,4)$	$(3,4)$	$(4,4)$	$(5,4)$	$(6,*)$	$(7,*)$
Logging	$(1,5)$	$(2,5)$	$(3,5)$	$(4,5)$	$(5,5)$	$(6,*)$	$(7,*)$

Table 1: Identification codes of a generic project structure

viding more fine granular measurement options. Here it depends on the project and the project manager, whether the thereby generated higher effort for data collection really provides additional meaningful information.

3.1.3 Generic Project Plan

Finally within the area of *Agile Software Development* it is very uncommon to establish a WBS and also no description of predefined workflows is available ([4], [7]). The agile development methodologies rather are described in terms of *techniques* and *development principles* [5]. Therefore we want to suggest to use an implicitly given generic project plan defined by the activities of the development process and the functions of the created products. The first dimension of our implicit project plan is given by the activities *Modeling*, *Implementation*, *Test*, *Deployment*, *Configuration Management*, *Project Management* and *Environment Maintenance* that normally have to be accomplished on every project. In addition to that the created software units can be categorized according to their function. Here one can think of *Application Logic*, *Presentation*, *Data Management*, *Transaction Management* and *Logging*.

Thus our implicitly given project plan consists of work packages that are defined by the activity to be done and (concentrating on constructive activities) the functions of the thereby created products. Also here the work packages can be tagged with 2-tupels as shown in Table 1.

The advantage of this numbering scheme for the process structure is that it can be applied on every project irrespective of the process model the project is based on. This generic project plan can also be used the compare projects based on totally different process methodologies.

3.2 Selection of Appropriate Software Measures

As mentioned above the selection of the software measures to collect should be based on a strategy like the *Goal Question Metric Paradigm* [3]. But we claim that the computation of the *Five Core Metrics* ([21], [22]) provides a value per se. Also in [3] it is explained that besides the *goal orientation* software measurement must follow the principles *experience reuse*, *appropriate data collection* and *real-time feedback*. We reach experience reuse by defining in advance that the following five software measures have to be collected for every project. As our approach allows the automatic computation of these measures also the other two above mentioned requirements are fulfilled.

Product Size: To measure the product size we have to identify artifacts that have been created or modified within the scope of project activities and to measure their sizes or changes in size respectively. There are many publications about the appropriate measurement of the size of software. The *Function Point Method* [12] is widely accepted, but the computation of function points cannot be done completely unattendedly. The computation of *Lines of Code* (LOC) is strongly dependent on the programming style. To us the computation of *Non Commenting Source Statements*[1] appears quite promising, since there already exists adequate tool support.

Effort: Development effort normally is given in man-days or man-months. In order to compute the effort spent automatically the project members have to maintain their efforts within a time recording system. For the purpose of correct assignment of the stored time records these have to include not only the length of time and an activity category but also the account code of the work package in the scope of which the effort has been conducted. The additional effort for time recording can be limited by proper tool support.

Productivity: As productivity measure it is quite common to compute the ratio of product size and effort spent [11]. The most important application of this software measure however is not to assess the current development team but to have a key to compare several projects and to have a problem indicator. We think that a comparison of the dimension of the productivity between two projects or between several phases of a single project can provide some clues about shortcomings in the scope of the process performance.

Time: Here we talk about calendar time that is actually consumed by the project. This measure is simple to measure but relatively hard to estimate in

[1] http://www.kclee.de/clemens/java/javancss

advance. For the customer of the software project this measure is very important. The time need of a software project results from the difference between the end and the start of the project. Times during which the project rests are not counted.

Quality: According to the ISO 9126 standard the quality of a product is defined as combination of the attributes functionality, usability, portability, maintainability, efficiency and reliability. According to [21] the first five attributes are the result of an appropriate software design, which can be assessed by corresponding design measures. The reliability of a product is defined as the likelyhood of error free usability over a certain period of time. Thus the defect rate, i.e. the number of defects occuring during a certain period of time, is an appropriate measure in order to assess the reliability and hence the quality of a product. Defects that occur during product development or during product use usually are maintained with a bug tracking system. We suggest to store therein not only information about the type and the impact of defects but also the account code of the project step the defect has been detected and has originated within respectively.

4 Implementation of a Lightweight SPCC

4.1 Architecture

The basis of our lightweight SPCC implementation is the project managmenent tool *Maven*[2]. As described in [9] and [10] we implemented some maven plugins which act as measurement transmitters for software measures. A measurement transmitter is a maven plugin that can be configured in such a manner that it computes a certain software measure only within the context of a certain element of the project structure. For example it can compute the *code size* of an artifact that has been implemented witin the *analysis activities* of the project. This datum would give information about the amount of code that has been produced during the creation of prototypes. Or the measurement transmitter is configured to count only the *effort* for the creation of the *database layer* for the application that has to be implemented within the project.

Therefore, the relevant information has to be provided beforehand by the project members. As mentioned above changes of software artifacts have to be labeled with an account code when they are committed into the configuration management system. Time records for the spent time have to be maintained and must also be tagged with the proper account information.

[2] http://maven.apache.org

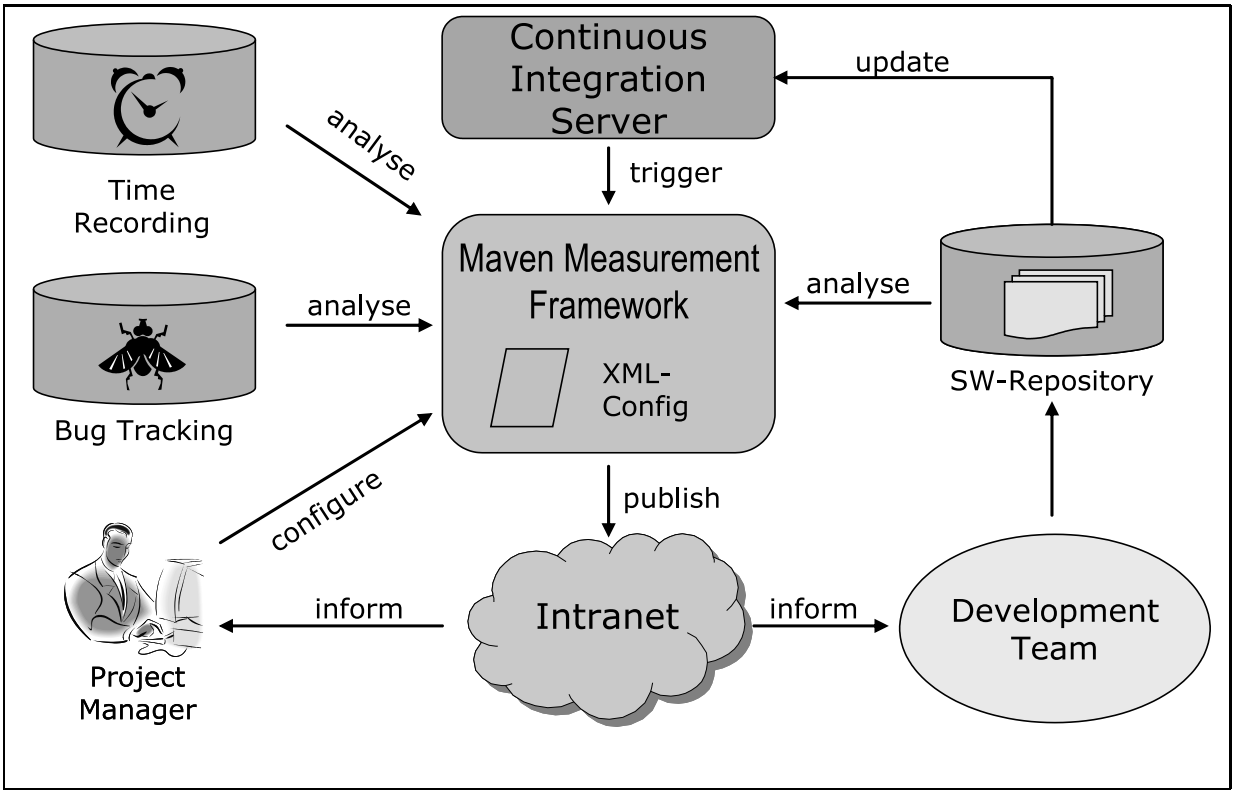

Figure 3: Integrated Software Measurement

Then however it is possible for the measurement transmitters to automatically identify the entities that have to be measured at runtime. Therefore, the Maven plugin can access e.g. the configuration management system, the bug-tracking system or the time recording system (see Figure 3).

The project manager defines by means of an XML config file (see Figure 3) which software measures have to be computed and in what context. The *continuous integration server* [6] not only performs the regularly build and test processes of the software project but triggers also the Maven based measurement framework which computes the defined software measures. The results are presented on an intranet web site as it is common practise for Maven.

4.2 Integrating our Approach into Existent Project Environments

Since especially large organizations within the software engineering field have already set up a project framework which is supported by corresponding tools, it is crucial that a new measurement tool can be integrated into an existing environment. For this integration the use of Maven as basis for our framework brings a large benefit. Within the Maven configuration file also information about the project environment are maintained including the configuration management system, the bug tracking system or the continuous integration system. More-

over, Maven provides methods to transparently access e.g. the software configuration management system (SCM system). For example a user who wants to check-out a certain software revision from the SCM system can simply call the appropriate Maven command and does not have to know what SCM system actually is used. Since our measurement framework is designed to use only the appropriate Maven commands to access these systems our approach can also be used within environments that use commercial software engineering tools. Either the existent systems are already supported by Maven (e.g. many SCM systems) or an adapter for Maven has to be implemented.

4.3 Case Study

We evaluated our approach on a small software development project that was conducted by students within the scope of the study courses in computer science at Bayreuth University in March 2006. The students had to form two teams and to implement each a client for a *Mancala*[3] game. The Mancala client should connect to a corresponding Server which contained the current state of a game and the data model. So the students had to implement the application logic of the client and a graphical user interface (GUI). The implementation of the application logic included also a game tree which is used to compute the next move of the game.

Accordant to an agile programming style the students at first implemented a *Spike* in order to be able to communicate with the mancala server. Then the actual implementation phase followed. After the deployment of the implemented software on two computers the two Mancala clients had been able to play against each other unattendedly.

The development progress of the two teams (see Figure 4) was measured with our Maven based SPCC implementation. Hereby, the program size of the Java programs was counted in *Non commenting Source Statements*[4]. It can be seen that the first team at the beginning mainly addressed the implementation of the GUI and started not until the last four days to implement the application logic. In contrast the second team developed right from the start both components in parallel. This measurement-based analysis could be "confirmed" by the final client applications. It turned out that the GUI of the first team's client was very polished but the client of the second team was technically superior and won every game.

[3] http://en.wikipedia.org/wiki/Mancala

[4] http://www.kclee.de/clemens/java/javancss

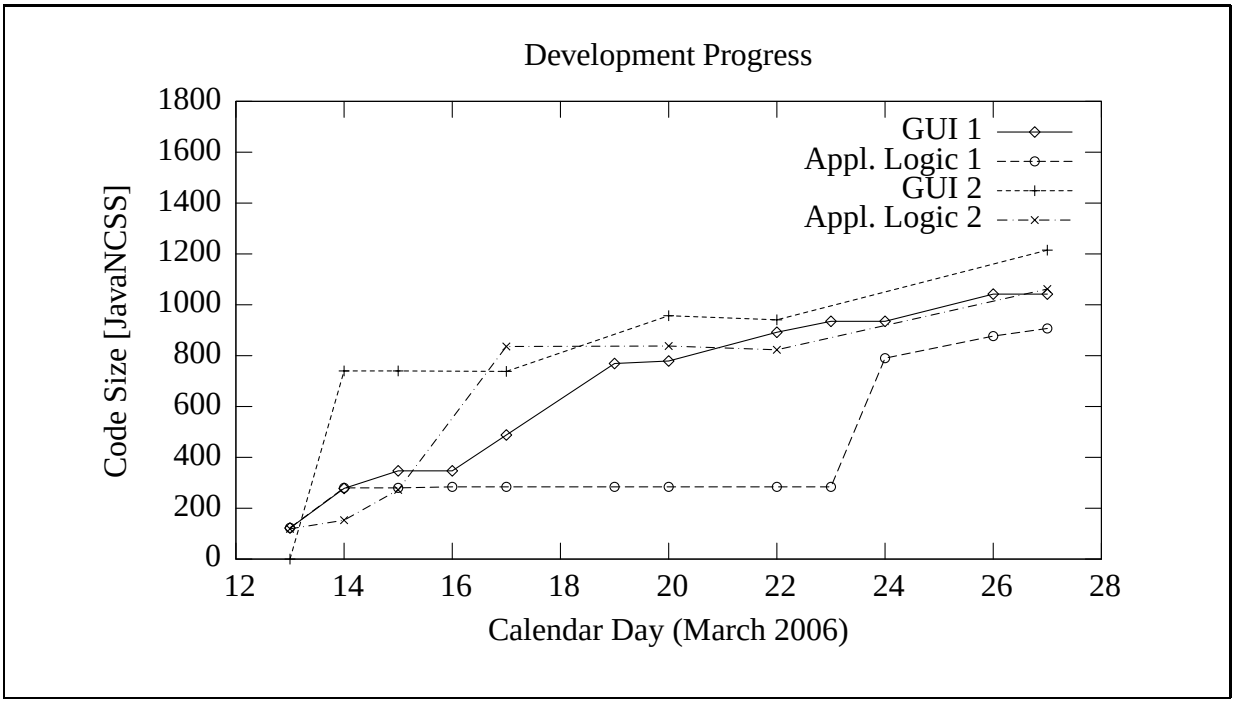

Figure 4: Comparison of the Implementation Progress of two Teams

5 Conclusion and Future Work

The benefit of our approach is that software measurement can relatively easily be added to a software project. Our approach is based on a practicable trade-off between the need for data collection and the additional effort the developers have to perform. The additional effort is caused by the need to store information about the development activites within the SCM system, the time recording system and similar databases. It is not uncommon that these pieces of information are maintained anyway.

Our approach provides the possibility to collect software measures in a standardized way. Thereby it is possible to compare measurement results both between several projects and between individual phases or steps within a single project.

We have restricted our approach to a small subset of the concepts that make up a software project control center according to [13] or [19]. We also define in advance the software measures we are most interested in. However, it is always possible to extend our measurement framework by additional measurement transmitters in order to compute further software measures.

So far we implemented API functions that enable the measurement transmitters to access both the SCM system and a time recording system we built ourselves. We also plan to provide functions to make also bug tracking systems and perhaps requirements management systems accessible.

Currently each Maven plugin within our measurement framework implements one measurement transmitter. If for instance the productivity shall be measured, a designated plugin for this task has to be used, since the architecture of Maven does not allow to easily invoke one plugin from another one. That means in order to perform new or new combinations of software measures also new plugins (written in Java) have to be created.

As this is not very convenient we plan for the near future to make it possible for the end users, i.e. the project managers, to define new measurement transmitters using a Java-based scripting language like *BeanShell*[5] or *Groovy*[6]. These scripting languages have a Java-like Syntax, but can be implemented with less syntactical overhead than Java programs. Moreover, they can be evaluated at runtime by Java programs and consequently by Maven plugins. The project manager can then define his software measures by combining simple API calls withing a small script, which then is invoked by a Maven plugin.

References

[1] Martin Auer, Bernhard Graser, and Stefan Biffl. A survey on the fitness of commercial software metric tools for service in heterogeneous environments: Common pitfalls. In *Proceedings of the Ninth International Software Metrics Symposium (METRICS 2003)*. IEEE Computer Society, 2003.

[2] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 511–519. John Wiley & Sons, 1994.

[3] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6):758 – 773, 1988.

[4] Kent Beck. *Extreme Programming explained: Embrace Change*. Addison Wesley, Upper Saddle River, New Jersey, second edition, 2004.

[5] Barry Boehm and Richard Turner. *Balancing Agility and Discipline*. Addison-Wesley, Boston, 2005.

[6] Mike Clark. *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications*. Pragmatic Bookshelf, Raleigh, N.C, 2005.

[7] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, Boston, MA, 2001.

[8] S. Dami, J. Estublier, and M. Amiour. APEL: A graphical yet executable formalism for process modeling. *Automated Software Engineering: An International Journal*, 5(1):61–96, January 1998.

[5] http://www.beanshell.org

[6] http://groovy.codehaus.org

[9] Bernhard Daubner, Andreas Henrich, and Bernhard Westfechtel. Integrierte Softwaremessung durch Verankerung der Softwaremaße an Elementen des Vorgehensmodells. In *Software Engineering 2006 – Fachtagung des GI-Fachbereichs Softwaretechnik*, volume P-79 of *LNI*, pages 157–162. GI, 2006.

[10] Bernhard Daubner, Andreas Henrich, and Bernhard Westfechtel. Towards Anchoring Software Measures on Elements of the Process Modell. In *Proceedings of the First International Conference on Software and Data Technologies (ICSOFT 2006)*, Setubal, Portugal, to appear 2006. INSTICC.

[11] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, Boston, Massachusetts, second edition, 1998.

[12] IFPUG, editor. *Function point counting practices manual, Release 4.1*. International Function Point Users Group, Princeton Junction, New Jersey, 2000.

[13] Ralf Kempkens, Peter Rösch, Louise Scott, and Jörg Zettel. Instrumenting measurement programs with tools. In *PROFES '00: Proceedings of the Second International Conference on Product Focused Software Process Improvement*, pages 353–375, London, 2000. Springer.

[14] Seija Komi-Sirvio, Paivi Parviainen, and Jussi Ronkainen. Measurement automation: Methodological background and practical solutions – a multiple case study. In *Proceedings of the Seventh International Software Metrics Symposium (METRICS 2001)*, page 306 ff. IEEE Computer Society, 2001.

[15] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy*. Addison Wesley, Boston, 2005.

[16] Philippe Kruchten. *The Rational Unified Process - An Introduction*. Addison Wesley, Boston, third edition, 2004.

[17] Christopher M. Lott. *Measurement-based Feedback in a Process-centered Software Engineering Environment*. PhD thesis, University of Maryland, 1996.

[18] Christopher M. Lott and H. Dieter Rombach. Measurement-based guidance of software projects using explicit project plans. *Information and Software Technology*, 35(6/7):407–419, 1993.

[19] Jürgen Münch and Jens Heidrich. Software project control centers: Concepts and approaches. *Journal of Systems and Software*, 70:3–19, 2004. Issues 1–2.

[20] Project Management Institute. *A Guide to the Project Management Body of Knowledge: PMBOK Guide*. Project Management Institute, Newtown Square, Pennsylvania, third edition, 2004.

[21] Lawrence H. Putnam and Ware Myers. *Five Core Metrics: The Intelligence behind successful Software Management*. Dorset House Publishing Co., New York, 2003.

[22] Janet Russac. Cheaper, better, faster: A measurement program that works. In International Function Point Users Group, editor, *IT Measurement: Practical Advice from the Experts*. Addison-Wesley, Boston, Massachusetts, 2002.

[23] Richard W. Selby, Adam A. Porter, Doug C. Schmidt, and Jim Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *Proceedings of the 13th International Converence on Software Engineering (ICSE 1991)*. IEEE Computer Society, 1991.