A Software Quality Model for RPG

Gergely Ladányi, Zoltán Tóth, Rudolf Ferenc
University of Szeged
Department of Software Engineering
Dugonics tér 13. H-6720 Szeged, Hungary
{lgergely,zizo,ferenc}@inf.u-szeged.hu

Tibor Keresztesi
R&R Software
H-1038 Budapest,
Ráby Mátyás utca 7, Hungary
tibor_keresztesi@rrsoftware.hu

***Abstract*—The IBM i mainframe was designed to manage business applications for which the reliability and quality is a matter of national security. The RPG programming language is the most frequently used one on this platform. The maintainability of the source code has big influence on the development costs, probably this is the reason why it is one of the most attractive, observed and evaluated quality characteristic of all. For improving or at least preserving the maintainability level of software it is necessary to evaluate it regularly. In this study we present a quality model based on the ISO/IEC 25010 international standard for evaluating the maintainability of software systems written in RPG. As an evaluation step of the quality model we show a case study in which we explain how we integrated the quality model as a continuous quality monitoring tool into the business processes of a mid-size software company which has more than twenty years of experience in developing RPG applications.**

Keywords*—*Software maintainability, ISO/IEC 25010, RPG quality model, IBM i mainframe, case study

I. Introduction

There are many legacy systems written for mainframe computers. These systems include critical elements such as bulk data processing, statistics, and transaction handling. In 1988, IBM introduced the very robust AS/400 platform, which became very popular to the end of the century (it was later renamed to IBM i). It has its own programming environment called ILE (Integrated Language Environment), which allows using programs written in ILE compatible languages, like the RPG programming language (Reporting Program Generator).

Business applications developed for the IBM i platform usually use the RPG high-level programming language. In spite of the fact that the language was introduced in 1959, RPG is still widely used nowadays, and it is continuously evolving. It stands close to database systems and it is usually used for processing large amount of data.

Our main contribution in this work is the introduction of a continuous quality management approach specialized to RPG. According to our best knowledge, no or only very little effort was invested into researching state-of-the-art quality assurance techniques for RPG. To explore RPG source code and detect components which carry high risks, we used the SourceMeter for RPG static code analyzer tool. The analyzer provides three types of measurements, which are the following:

- *Software Metrics:* A software metric in this context is a measure of some property of the source code [1]. There is a growing need in software development to define quantitative and objective measurements, so software metrics are calculated for RPG language elements (namely for subroutines, procedures, programs, and for the whole system).
- *Rule violations:* Rule violations [2] can reveal code segments which are prone to errors. These can be coding problems which are introduced e.g. accidentally or because of low-skill programmers, and code smells that can be symptoms in the source code that possibly mean a deeper design problem in the code and can cause incorrect operation in a later stage. Code smells are not equal to bugs but the presence of them is increasing the risk of bugs or failures in the future. Rules are defined for RPG to indicate the existence of possible code smells. Furthermore, coding rules are excellent means for defining company coding standards.
- *Code duplications:* Duplicated code segments come usually from the very productive, but at the same time dangerous source code copy-paste habit. Code duplications [3] could be handled as a code smell; however, they cover an important separable part of quality assurance.

We used these low-level measurements as an input to provide a diagnosis about an RPG system. This diagnosis, which characterizes the quality of a whole RPG system, is provided by the ISO/IEC 25010 [4] standard based new quality model introduced in this paper. Building and testing a quality model like this needs specialists in this specific domain. So, the company called R&R Software[1] (mid-sized company with more than 100 software developers) was involved to help the calibration and evaluation of the quality model with their deep knowledge in developing software systems written in RPG. They clarified the importance of each quality aspect, and also provided a collection of industrial programs written in RPG. As a validation of our approach, we show in a case study how we managed to integrate our method into their development processes, and how they used it for refactoring purposes.

The paper is organized as follows. In the next section we summarize some related studies. Next, after a brief overview of the RPG language, we describe our approach in detail in Section III. Afterwards, in Section IV we present a case study, which shows how our approach works in a real life situation. In Section V, we collect some limitations and threats to validity. Finally, in Section VI we close the paper with conclusions.

[1] http://www.rrsoftware.hu/

II. Related Work

In this section we provide an overview about the most important studies about software quality measurement and its relationship with RPG static code analysis.

A. RPG Analysis

In the last decades several studies have been published dealing with software metrics. As a principle, Chidamber and Kemerer provided a number of object oriented metric definitions [5]. Despite RPG is not an object oriented programming language, this study is essential for further investigations in the area of software metrics. Chidamber and Kemerer also developed an automated data collection tool to collect software metrics. After collecting metrics, they suggested ways in which managers may use these metrics for process improvement.

Many research effort was put into finding effective migration strategies for RPG legacy systems. Canfora et al. presented an incremental migration strategy [6] to transform RPG legacy programs into an object oriented environment. The migration strategy consists of six sequential phases, trying to identify objects in a persistent data storing environment. Applying migration process is mainly used for transforming RPG II and RPG III programs into RPG IV.

Software metrics were usually implemented for the most widely used programming languages like C, C++ [7], Java [8] and C# [9]. Only a few studies are dealing with the quality assurance of the RPG language.

In 1982 Hartman [10] published a study which was dealing with identifying software errors in RPG II and RPG III systems using Halstead [11] and McCabe [12] complexity metrics. The study was evaluated on commercially available RPG modules written within IBM. Using measured metrics, modules were assigned to low, medium, or high metric value ranges. Concentrating upon modules that fall into medium and high metric value ranges increased the effectiveness of finding the most defects in the whole system. McCabe's complexity metric seemed slightly better at identifying modules with errors than Halstead's complexity metric. Another early paper handles the error rate of a software as a measure of code quality. Naib developed a methodology for the prediction of the error rate and hence code quality prior to an application's release [13]. Naib considered the same metrics (mentioned as internal factors) as Hartman, namely Halstead's and McCabe complexity and also lines of code (LOC). Contrary to them, we did not use software metrics to identify errors in the system, but to provide a flexible approach which is able to provide various quality indices.

Kan et al. [14] were dealing with software quality management in AS/400 environment. They identified the key elements of the quality management method such as customer satisfaction, product quality, continuous process improvement and people. Based on empirical data the progress in several quality parameters of the AS/400 software system were examined. They presented a quality action road map that describes the various quality actions that were deployed.

Bakker and Hirdes [15] described some project experiences using software product metrics. They analyzed more than 10 million lines of code in COBOL, PL/I, Pascal, C, C++, and RPG (about 1.8 million lines of code). The goals of the projects written in RPG were maintenance and risk analysis. They found that the problems in legacy systems have their cause in the design, not in the structure of the code. Furthermore, re-designing and re-structuring existing systems are less costly and safer solutions to these problems than re-building.

Sometimes, RPG legacy systems can be hard to maintain, improve and expand, since there is a general lack of understanding of the systems. Suntiparakoo and Limpiyakorn [16] presented a method of flowchart knowledge extraction from RPG legacy code. The extracted flowchart can serve as a quality assurance item that facilitate the understanding of RPG legacy code during a software maintenance process.

B. Quality Model

One can see that the set of previous studies on RPG code maintenance and quality assurance is very limited, thus a well-defined quality assurance method is desired to handle maintainability effort questions.

Once we have quality indicators, like software metrics which are capable to characterize the software quality from various points of view, it is necessary to standardize this information somehow [17]. This is the reason why the ISO/IEC 25010 [4], and its ancestor, the ISO/IEC 9126 [18] international standards have been created. The research community reacted quickly to the appearance of these standards, and several papers have been published in connection to them. Jung and Kim [19] validated the structure of the ISO/IEC 9126 standard based on the answers of a widespread survey. They focused on the connections between the subcharacteristics and the characteristics. They grouped subcharacteristics together based on the high correlation in their values according to the evaluators. The authors found that most of the evolved groups referred to a characteristic defined by the standard.

Since the standards do not provide details about how these characteristics can be determined, numerous adaptations and various solutions have been developed for calculating and assessing the values of the defined properties in the standard [20], [21], [22].

Zou and Kontogiannis introduced a requirement driven system transformation approach [23], [24]. They migrated legacy procedural systems to modern object oriented platforms while achieving specific quality requirements for the target system using Markov model approach and the Viterbi algorithm. They used soft-goal dependency graphs to describe their maintainability and performance quality model for estimating the quality of the target system. Contrary to them our aim was not to improve the result of a migration process, but to provide state of the art quality assurance techniques for legacy RPG systems. Although the examination of quality dependencies between the original and the migrated version of RPG legacy systems is a part of our future work.

Others use benchmarking and quality models to provide higher level information about a software system. Benchmarking in software engineering is proved to be an effective technique to determine how good a metric value is. Alves et al. [25] presented a method to determine the threshold values more precisely based on a benchmark repository [26]

holding the analysis results of other systems. The model has a calibration phase [21] for tuning the threshold values of the quality model in such a way that for each lowest level quality attribute they get a desired symmetrical distribution. They used the $\langle 5, 30, 30, 30, 5 \rangle$ percentage-wise distributions over 5 levels of quality. To convert the software metrics into quality indices we also used a benchmark with a large amount of evaluations, but we applied it in a different way. During the calibration, instead of calculating threshold values we approximate a normal distribution function called benchmark characteristic, which is used to determine the goodness of the software.

III. Approach

In this section we present an approach that is an extension of earlier research achievements concerning software quality models [20] and its integration with a continuous quality monitoring tool called QualityGate[2] [27].

First we provide some general information about the structure of the RPG language. Next, we give an overview about the used software metrics. Afterwards, we briefly introduce our probabilistic software quality model, called ColumbusQM [20] and its implementation to RPG.

A. The RPG Language

In this section we focus on the essential components that we used for defining the quality model for the RPG programming language.

Over the years, the data-intensive RPG language has evolved in many aspects. RPG III (released in 1978) already provided subroutines, modern structured constructs such as DO, DO-WHILE, IF. A great break-through was performed in 1994, when the first version of RPG IV was released. With the release of RPG IV, developers obtained new features like free-form blocks or definition specifications, and the set of available operations was extended as well. RPG supports different groups of data types, namely character data types (Character, Indicator, Graphic, UCS-2), numeric data types (Binary Format, Float Format, Integer Format, Packed-Decimal Format, Unsigned Format, Zoned-Decimal Format), date, time and time stamp data types, object data type (user can define Java objects), and one can define pointers to variables, procedures and programs. In RPG IV there are two types of functions. The first one is called subroutine which takes no parameter and has a limited visibility and usability. Procedures are more flexible constructs since they can have parameters and they can be called from other programs.

Figure 1 lists a sample RPG IV code fragment. The code snippet presents some features and capabilities of the RPG language. From the beginning, RPG uses the column-dependent source format. In this way programming requires the code elements to be placed in particular columns. For instance, the character in the 6th column means the specification type. In Figure 1 there are Control (H), File (F), and Definition (D) specifications. Calculation specifications (column-dependent) can be added to perform an appropriate sequence of calculations. Since RPG IV, it is possible to use the free-form blocks for column-independent calculations. In this sample, payment is calculated by a procedure named *CalcPay* (prototype declaration can be found in the sample, but the definition is not presented here). In this paper, our point is not to present all RPG language capabilities, but just to show the reader how it looks like. For details, see the ILE RPG Programmer's Guide on IBM's website.[3]

```
....+....1....+....2....+....3....+....4....+....5....+...
     H DATEDIT(*DMY/)
     FEMPLOYEE  IF   E           K DISK
     D Pay             S              8P 2
      * Prototype Definition for subprocedure CalcPay
     D CalcPay         PR             8P 2
     D   Rate                         5P 2 VALUE
     D   Hours                       10U 0 VALUE
     D   Bonus                        5P 2 VALUE
      /free
         chain trn_number emp_rec;
         if %found(emp_rec);
            pay = CalcPay (emp_rate: trn_hours: trn_bonus);
         endif;
      /end-free
```

Figure 1. Sample RPG IV code

B. SourceMeter for RPG

A core item in source code quality assurance is a tool set that provides useful information about a system from different points of view. In our study the static code analysis was done by a tool called *SourceMeter for RPG*[4]. Since our focus in this paper is not on static analysis, we will present the capabilities of the tool only briefly.

Using the tool, we can calculate different metrics for source code elements located in RPG source files, namely for Subroutines, Procedures, Programs and for the whole System. Metrics are organized into four groups to indicate the role and characteristics of them. These groups are listed in the following (some group members are also presented in parentheses):

- Size (LOC, LLOC)
- Documentation (CLOC, CD)
- Coupling (NII, NOI)
- Complexity (NLE, McCC)

The quality model which will be presented in Section III-C, uses these source code metrics as low-level quality indicators. The description of the used software metrics and quality characteristics can be found in Table I.

SourceMeter also collects information about rule violations found in RPG source files. Rule violations do not result in immediate errors, but they indicate the locations in source code where possibly deeper problems exist. SourceMeter contains a wide range of rules defined for the RPG language. Rules are not categorized like metrics, but classified into three groups according to their criticality. These three groups are the followings (with a given sample):

- Warning P1 (Forbidden operation used)
- Warning P2 (Too Deep Copy/Include nesting)
- Warning P3 (Debug operations should be avoided)

[2]https://www.quality-gate.com/

[3]https://publib.boulder.ibm.com

[4]https://www.sourcemeter.com/

Table I. DESCRIPTION OF THE NODES IN THE RPG QUALITY MODEL.

Sensor nodes	
CC	Clone coverage. The percentage of copied and pasted source code parts, computed for the subroutine, procedure, program, and the system itself.
CLOC	Comment Lines of Code. Number of comment and documentation code lines of the subroutine/procedure/program. Subroutines' CLOC are not added to the containing procedure's CLOC. Similarly, subroutines' and procedures' CLOC are not added to the containing program's CLOC.
CD	Comment Density. The ratio of comment lines compared to the sum of its comment and logical lines of code (CLOC and LLOC). Calculated for subroutines, procedures, and programs.
LLOC	Logical Lines of Code. Number of code lines of the subroutine/procedure/program, excluding empty and comment lines. In case of a procedure, the contained subroutines' LLOC is not counted. Similarly, in case of a program, the contained subroutines' and procedures' LLOC is not counted.
McCC	McCabe's Cyclomatic Complexity [28] of the subroutine/procedure. The number of decisions within the specified subroutine/procedure plus 1, where each if, else-if, for, do, do-while, do-until, when, on-error counts once. Subroutines' McCC are not added to the containing procedure's McCC.
NLE	Nesting Level Else-If. Complexity of the subroutine/procedure/program expressed as the depth of the maximum embeddedness of its conditional and iteration block scopes, where in the if-else-if construct only the first if instruction is considered. The following RPG language items are taken into consideration: if, for, do, do-while, do-until, select, and monitor. The else-if, else, when, other, and on-error operations do not increase the value of the metric; however, they can contain elements defined above, which increase NLE. Subroutines' NLE are not added to the containing procedure's NLE. Similarly, subroutines' and procedures' NLE are not added to the containing program's NLE.
NII	Number of Incoming Invocations. Number of other subroutines which directly call the subroutine.
NOI	Number of Outgoing Invocations. Number of other subroutines which are directly called by the subroutine.
Warning P1	The number of critical rule violations in the subroutine/procedure/program. They can be potential root causes of system faults.
Warning P2	The number of major rule violations in the subroutine/procedure/program. Serious coding issues which makes the code hard to understand, and can decrease efficiency.
Warning P3	The number of minor rule violations in the subroutine/procedure/program. These are only minor coding style issues, makes the source code harder to comprehend.
Aggregated nodes	
Changeability	The capability of the software product to enable a specified modification to be implemented, where implementation includes coding, designing and documenting changes.
Complexity	Represents the overall complexity of the source code. It is represented by the McCabe's Cyclomatic Complexity and the Nested level of the subroutines.
Comprehensibility	Expresses how easy it is to understand the source code. It involves the complexity, documentation and size of the source code.
Documentation	Expresses how well the source code is documented. It is represented by the density and the amount comment lines of code in a subroutine.
Fault proneness	Represents the possibility of having a faulty code segment. Represented by the number of minor, major and critical rule violations.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
Aggregated nodes defined by the ISO/IEC 25010 standard	
Maintainability	Degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.
Analyzability	Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
Modifiability	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
Testability	Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.
Reusability	The degree to which an asset can be used in more than one system, or in building other assets.

These groups are based on priorities. The Warning P1 group represents critical, Warning P2 major, and Warning P3 means minor rule violations. One can see that using a forbidden operation (defined by the actual company standards) is more undesired than a debug message. For every code element (subroutine, procedure, program, system), summarized values are calculated to indicate the amount of rule violations located in that appropriate code segment for each rule. The quality model uses the number of rule violations according to their priorities.

The third component of SourceMeter is responsible for detecting the software code clones, also known as code duplications. A code clone is a source code fragment that occurs more than once in a given software system. Clones are usually undesired, because of their possible side-effects. One harmful effect occurs when a code part with bugs in it is copied. Furthermore, duplicated code instances are hard to actualize and maintain. SourceMeter can detect different types of duplicated code parts in the source code, and also organize the found code parts into clone groups (clone class - CCL). The

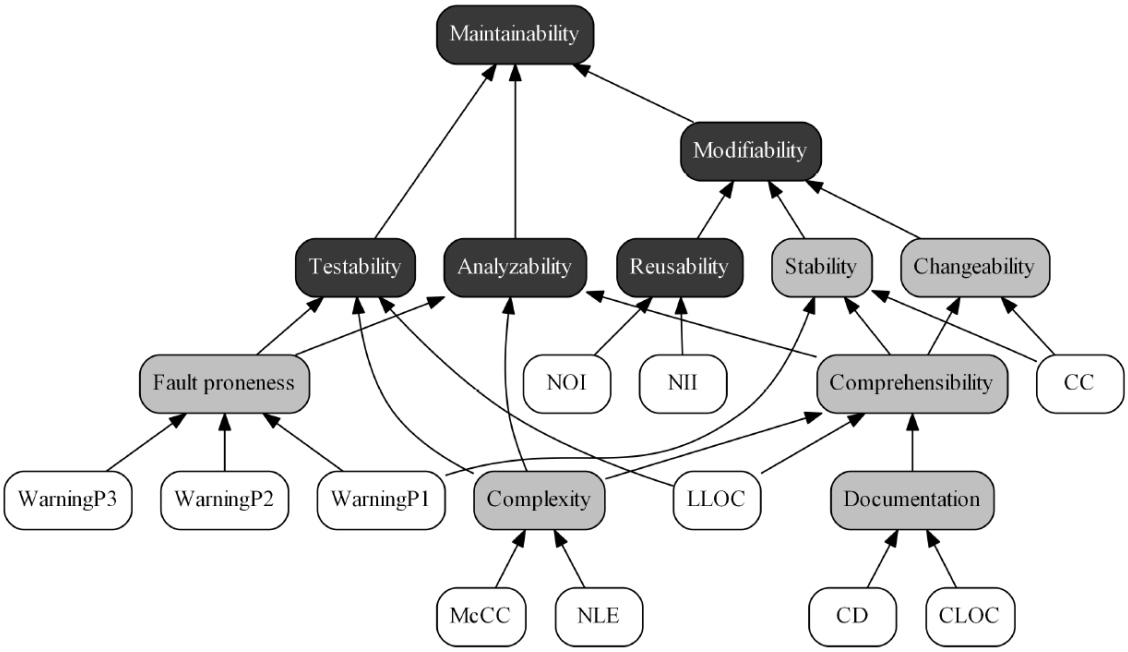

Figure 2. RPG quality model ADG

amount of cloned code is a very important metric to take into consideration if we want to estimate the effort of maintaining a system. SourceMeter for RPG provides different clone metrics for the analyzed software system. Some of them are listed below:

- Clone Class (CCL) – number of clone classes.
- Clone Instance (CI) – number of clone instances.
- Clone Coverage (CC) – percentage of copied and pasted source code parts, computed for the subroutine, procedure, program, and the system itself.
- Clone Age (CA) – number of previously analyzed revisions in which the clone was presented.
- Clone Complexity (CCO) – McCC complexity for clone instance. For clone classes CCO is the sum of CCO of each CI in the clone class.

C. Qualification Method

In this section we provide a brief overview about the probabilistic software quality model called ColumbusQM [20] and show how we implemented the general approach for the RPG language.

The ISO/IEC 25010 international standard defines the product quality characteristics which are widely accepted both by industrial experts and academic researchers. These characteristics are: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. In this study we focused on maintainability because of its obvious and direct connection with the costs of altering the behavior of the software. The standard defines the subcharacteristics of maintainability as well, but it does not provide further details how we could calculate these characteristics and subcharacteristics.

The basic idea behind the ColumbusQM is splitting the complex problem of calculating a high-level quality characteristic into less complex sub-problems. In the quality model the relations between the lower level metrics which can be readily obtained from the source code and the higher level quality characteristics can be described with an acyclic directed graph, called the *attribute dependency graph (ADG)*. The developed ADG for RPG language is shown in Figure 2.

The black nodes in the model are defined by the ISO/IEC 25010 international standard, the white nodes are the source code metrics calculated by the SourceMeter for RPG tool, and finally, the gray nodes are the inner nodes defined by us to help revealing the dependencies in the model. We call the source code metric nodes as *Sensor nodes* and the other nodes as *Aggregated nodes*. Table I contains the descriptions of all nodes.

Although the basic structure of the ADG is based on an earlier Java quality model [20], the differences between the languages caused modifications. The most important difference is that we could not apply object oriented metrics in the RPG quality model because it is a procedural programming language.

An essential part of the approach besides the quality model is the benchmark, which contains several RPG systems. By comparing the system with the applications from the benchmark, the approach converts the low-level source code metrics into quality indices. Formally, each source code metric can be regarded as a random variable that can take real values with particular probability values. For two different software systems, let $h_1(t)$ and $h_2(t)$ be the probability density functions corresponding to the same metric. Now, the

goodness value of one system with respect to the other, is defined as

$$\mathcal{D}(h_1, h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\,\omega(t)\,dt,$$

where $\omega(t)$ is the weight function that determines the notion of goodness. Figure 3 helps us understand the meaning of the formula: it computes the signed area between the two functions weighted by the function $\omega(t)$.

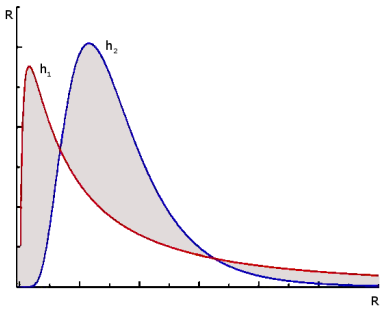

Figure 3. Comparison of probability density functions

For the edges of the ADG, a survey was prepared. In this survey the developers were asked to assign weights to the edges, based on how they felt about the importance of the dependency.

Consequently, a multi-dimensional random variable $\vec{Y}_v = \left(Y_v^1, Y_v^2, \ldots, Y_v^n\right)$ will correspond to each higher level node v. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the higher level node. When dealing with probabilities, one needs to take every possible combination of goodness values and weights, and also the probabilities of their outcome into account. We define the aggregated goodness function for the node v in the following way:

$$g_v(t) = \int_{\substack{t=\vec{q}\vec{r} \\ \vec{q}=(q_1,\ldots,q_n)\in\Delta^{n-1} \\ \vec{r}=(r_1,\ldots,r_n)\in C^n}} \vec{f}_{\vec{Y}_v}(\vec{q})\,g_1(r_1)\ldots g_n(r_n)\,d\vec{r}\,d\vec{q}, \qquad (1)$$

where $\vec{f}_{\vec{Y}_v}(\vec{q})$ is the probability density function of $\vec{Y}_v$, $g_1, g_2, \ldots g_n$ are the goodness functions corresponding to the incoming nodes, Δ^{n-1} is the $(n-1)$-standard simplex in $\Re^n$ and C^n is the standard unit n-cube in $\Re^n$.

Although the formula may look frightening at first glance, it is just a generalization of how aggregation is performed in the classic approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the aggregate node. When dealing with probabilities, one needs to take every possible combination of goodness values and weights, and also the probabilities of their outcome into account. Detailed overview of the ColumbusQM can be found in our earlier paper [20].

D. Integration with QualityGate

Since our intention was to provide valuable information for the managers and developers on a daily basis it was necessary to integrate our approach into a continuous quality management tool. The SourceMeter for RPG tool and the ColumbusQM approach was integrated into a tool called QualityGate [27]. As a comprehensive software quality management platform, QualityGate is capable of calculating quality values using the ColumbusQM from source code using a wide range of software quality metrics provided by the SourceMeter for RPG tool. It is empowered by a built-in quality model conforming to the ISO/IEC 25010 standard and has a benchmark containing analysis data from a large number of RPG systems. This makes it possible to calculate objective quality attributes and estimate upcoming development costs [29].

IV. Case Study

In this section we present our empirical results about how we managed to integrate the introduced quality model for RPG approach and the QualityGate tool suite into the life of a mid-sized software company.

We organized the case study into four consecutive phases. In the first, *initial phase* we calibrated the SourceMeter for RPG tool and created a benchmark and a quality model for RPG. In the next, *integration phase* we prepared the IBM i mainframe to regularly generate spool files and upload it into a subversion repository. In the *refactoring phase* a software module was improved based on the guidelines of the approach. Finally, the last part of the case study is the *discussion phase*, where we discussed our experiences about the approach.

A. Initial Phase

In the initial phase we personalized our approach to the R&R Software company. First we went through every software metric, rule violation and clone properties and calibrated them in cooperation with RPG experts.

We created a survey to set the priorities of each rule violation. Developers classified each rule violation as minor, major or critical. We also implemented some new rule violations which seemed useful based on their opinions. We also asked them to set the limits of specific rule violations, for example what is the Nesting Level (NLE) value which is too much for a subroutine and we need to mark it as a rule violation.

To build a benchmark, we used large, real-life RPG modules provided by the company. The basic statistics of them can be found in Table II. As we can see there are applications from 18K to 129K total logical lines of code (TLLOC) and with thousands of subroutines. Since we wanted to compare the given application to the other application of the company in this study we used these four systems as benchmark.

Table II. Basic statistics of the benchmark systems.

System	TLLOC	Program num.	Subroutine num.
AB	128,146	264	5,355
IV	18,378	62	850
LG	65,655	163	2,336
PR	129,484	288	5,944

Finally, we created and weighted a quality model in cooperation with them. This quality model can be seen in Figure 2 which has been already introduced in Section III-C.

B. Integration Phase

In the second phase we integrated our approach into their IBM i programming environment. The flowchart of the whole approach is shown in Figure 4.

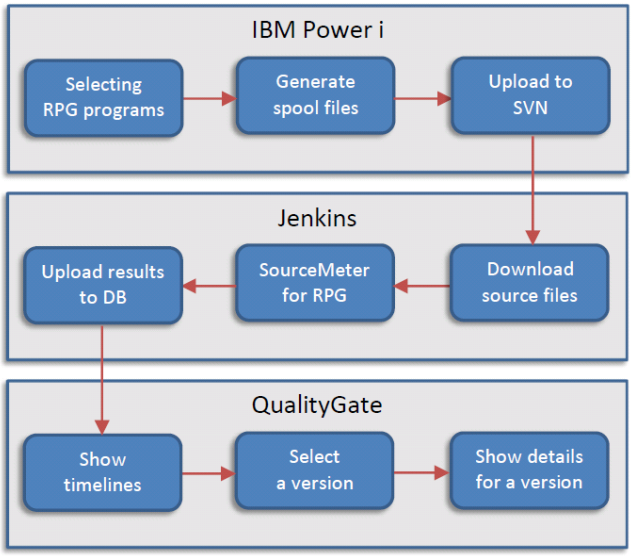

Figure 4. Process chain of the approach.

First of all we had to program the IBM i mainframe to generate the spool files from the chosen system on a daily basis. Spool files are the results of the RPG compilation and besides the source files they contain several useful information about the compilation. SourceMeter for RPG supports both spool files and raw source files, but since spool files contain more information it is recommend to use them. After the spool file generation is finished, the IBM i starts a script which uploads the spool files to an SVN version control system.

We used an extendable open source continuous integration server called Jenkins[5] to analyze every version in the subversion repository. Jenkins makes it possible to extend the build system with various plugins. We implemented a Jenkins plugin, which provides a user interface where the user can set the specific properties of the evaluation, like the used quality model, subversion repository, or frequency of the analysis. By using the plugin, when a new version is committed to the subversion repository, Jenkins automatically downloads the source files and executes the SourceMeter for RPG tool and right after it uploads the results to the database of QualityGate.

Finally, QualityGate shows the quality of the evaluated versions on different timeline and spider charts. On the timelines the developers can choose a specific version and find the root cause why the given quality characteristic has been changed.

C. Refactoring Phase

As a result of the previous phases, the continuous quality monitoring tool was ready for use. R&R selected a part of the LG module for refactoring, which they wanted to improve. The basic statistics before and after the refactoring of this

[5]http://jenkins-ci.org/

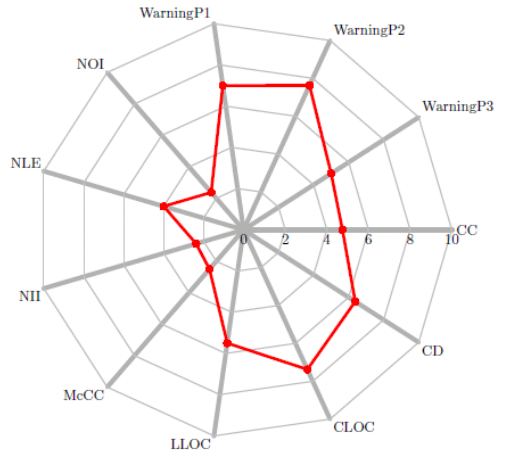

Figure 5. Low-level quality results.

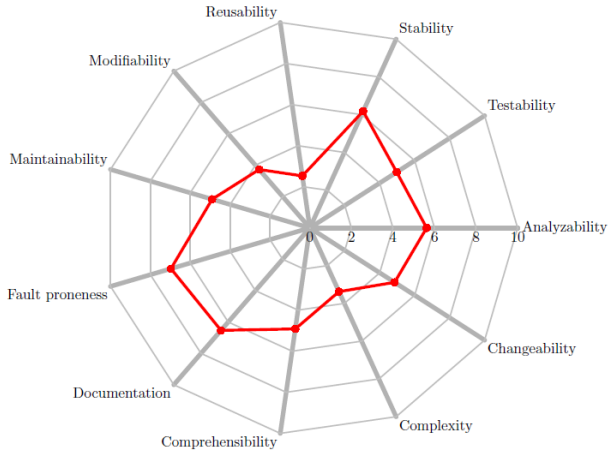

Figure 6. High-level quality results.

submodule can be seen in Table III. As it was hoped and expected, during the refactoring the overall maintainability of the module increased. The number of critical rule violations halved and only about one third of the major rule violations remained in the code. The clone coverage of the module did not change, since no effort was put into reducing clones. Also, the developers did not reduce the number of minor rule violations.

Table III. BASIC STATISTICS OF THE LG SUBMODULE BEFORE AND AFTER THE REFACTORING.

Property	Before refactoring	After refactoring
TLLOC	5,266	5,319
Program num.	9	9
Subroutine num.	226	233
Maintainability	4.87	5.41
WarningP1 num.	109	53
WarningP2 num.	238	80
WarningP3 num.	262	262
Clone Coverage	0.17	0.17

The initial low and high-level quality results of the submodule can be seen in Figure 5 and Figure 6. The maintainability of this system before the refactoring was 4.87 on the scale of [0,10] (larger is better). Since the average is 5, this means the initial maintainability of the selected LG submodule is slightly under the average. Based on Figure 5, we can assume that this

is mostly because of the NII, NOI, McCC and NLE metrics, because their goodness value is the lowest in the model. According to the RPG quality model, the metrics influence the goodness of the Complexity and Reusability aggregated nodes, e.g. Complexity is calculated from McCC and NLE, and it also has a very low 3.38 goodness value.

The business process of a refactoring task is quite simple with QualityGate. The lead developer needs to find a coding issue based on the guidelines of the approach. This issue can be for example a badly maintainable subroutine, a rule violation or a code duplication. The next step is to assign the issue as a ticket to a suitable developer with some comment. Finally, the developer has to fix the specified issue. At the time the developer committed the fix for the issue the new version will be uploaded to QualityGate. If the issue was a rule violation or a code duplication, the validation of the fix is done automatically, since it disappeared from the new version.

The progress of the maintainability quality characteristic of the system during the whole refactoring process can be seen in Figure 7. As it was expected, the maintainability of the submodule increased with almost every new version. After a week of refactoring the quality of the submodule went above 5. It means it became more maintainable than the average according to the benchmark. As we can see in Figure 7, at this time the color of the timeline changed from red to green, meaning the quality of the system reached a specific threshold value. This previously set threshold value determines whether the system is in NO-GO or GO state. In this way managers can easily set the target quality of a specific application.

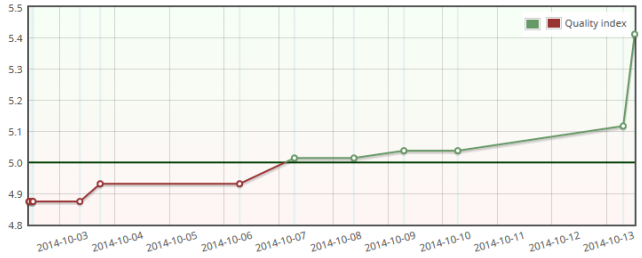

Figure 7. Maintainability quality timeline.

In Figure 8 we can see that in the first week of the refactoring no effort was put into improving the McCC complexity of the module. Although at the end of the refactoring process it increased by 0.5, it remained still under 3. The complexity of the system is still critical, but during the refactoring its goodness increased from 3.38 to 3.69.

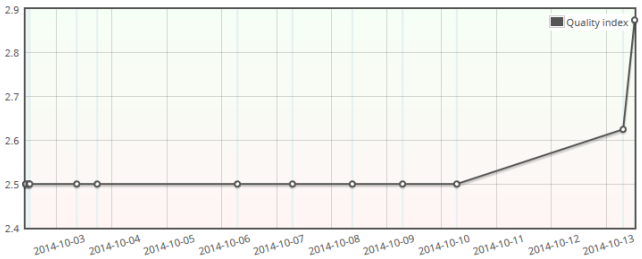

Figure 8. McCC quality timeline.

In Table III we saw that the number of critical rule violations (WarningP1) decreased a lot during the refactoring. This improvement can be seen on the quality timeline of the WarningP1 node in Figure 9. If we compare it with the maintainability timeline, we can see that fixing critical rule violations had an important role in the two highest improvement on the maintainability timeline. This improvement was caused mainly by the elimination of rule violations.

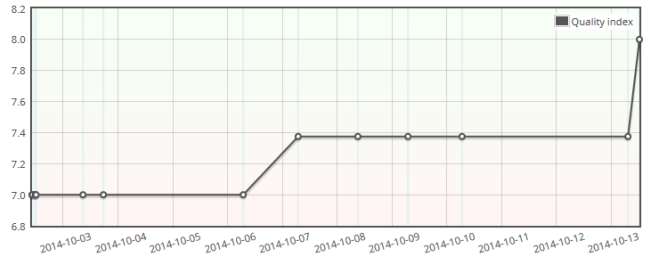

Figure 9. WarningP1 quality timeline.

In the following, two simple cases will be shown to demonstrate the refactoring mechanism applied on the elimination of rule violations. Companies have different standards for the allowed set of operations to be used in the code. Since the given set can differ in companies, the list of operations to avoid in code is customizable. At R&R Software, *ADDDUR* is one of these operations to avoid. Using such an operation is forbidden according to the company's standard.

Look at the code snippet from the LG module, shown in Figure 10. A subroutine definition can be seen named *fk0601*. In the first step a checking subroutine is called, then if no error was found in the date format an assignment is done, and finally another check is performed by calling *ckfm01*. The ADDDUR operation adds the duration specified in the second operand called factor 2 (*"7:*days"* which means that the number of days is increased by seven) to a date or time and places the resulting Date, Time or Timestamp in the result field (*@@date*). Since the ADDDUR operation is on the avoid operation list, a critical rule violation will occur pointing to the relevant code location.

To eliminate ADDDUR from the code, refactoring should be performed on the code. In Figure 11 the same *fk0601* subroutine is shown with some modifications. After checking whether the date is in the desired format, the two *eval* and the *adddur* operations were combined into one single *eval* operation that has the same functionality.

Another critical rule violation is when a programmer does not specify an error handling subroutine on a File Specification

```
....+....1....+....2....+....3....+....4....+....5....+...
     C     fk0601        begsr
     C                   exsr      ckdtfr
      * No error --> Process
     C     *in99         ifeq      *off
     C                   eval      @@date = %date(wsdtfr)
     C                   adddur    7:*days       @@date
     C                   eval      wsdtfr = %dec(@@date)
     C                   exsr      ckfm01
     C                   endif
     C                   endsr
```

Figure 10. Using avoid operation in a subroutine

```
....1....+....2....+....3....+....4....+....5....+...
C     fk0601        begsr
C                   exsr      ckdtfr
 * No error --> Process
C     *in99         ifeq      *off
C                   eval      wsdtfr = %dec(%date(wsdtfr)
                                          +%days(7))
C                   exsr      ckfm01
C                   endif
```

Figure 11. Eliminate avoid operation rule violation

line, thus error(s) occurred during file management will not be handled correctly. In Figure 12 a file named *lro1000c* is specified as an external file and also a fully procedural file is stored on a local disk for only input purposes (in this program). A line continuation is added with a keyword *rename* to reference the file more easily from code. In this case, to handle input errors on file *lro1000c*, we need to add another keyword that is *infsr*. The given parameter of the *infsr* keyword is the name of the subroutine that will handle occurring input errors. The corrected source snippet can be seen in Figure 13. A keyword has been added that marks *srinfs* as the error handling subroutine.

```
....1....+....2....+....3....+....4....+....5....+...
Flro1000c  if   e           k disk
F                                     rename(rorc:ro1000cr)
```

Figure 12. Missing error handling on File Specification

```
....1....+....2....+....3....+....4....+....5....+...
Flro1000c  if   e           k disk
F                                     infsr(srinfs)
F                                     rename(rorc:ro1000cr)
```

Figure 13. Eliminate missing error handling on File Specification

By eliminating these critical rule violations it is very likely that the quality of the system will improve. Nevertheless, as we have seen the quality index is calculated from a wide set of metrics. Consider a case when a programmer is told to eliminate a code clone. So, the developer extracts the cloned code parts into a procedure and calls it from the right places where formerly the clones were located (assume that clone instances were located in two subroutines). In this case, the value of the CC (Clone Coverage) metric will be lower, but the NOI (Number of Outgoing Invocations) values will increase for both subroutines. To sum up, improving one characteristic of the system does not necessarily result in the improvement of the quality index of the system.

D. Discussion Phase

The last phase was the discussion phase where we concluded our experiences about the approach in a workshop. Altogether they found our approach useful and easy to use. We organized their opinions into the following four points.

- According to them one of the most important feature of the approach is that they were able to check the RPG coding conventions inside the company using the precisely calibrated rule violations and metrics. This feature helped them to force their developers to avoid undesirable solutions. This makes the source code more maintainable since when a few months later a different developer would like to maintain the code part it will be much easier.
- The personalization of the approach can be done easily. QualityGate provides user interfaces to create new benchmarks and quality models. In the future they would like to use this feature to validate and upgrade their source code generator with a quality model which is not designed for maintainability, but for their special task.
- The best time to use the method is before testing, since the tool could help preventing unnecessary test cycles by revealing faults. Another good occasion to use the method is when a new project is started or an application needs to be upgraded.
- A long term benefit of using this approach is that developers will learn what are the common patterns solving different problems and what needs to be avoided. In this way not only the company will benefit using the approach but the developers as well.

V. Threats to Validity, Limitations

Altogether the results are promising, but we have to highlight the limitations and the threats to validity of our approach as well.

One major limitation of our approach is generality, since the entire process was designed within a cooperation of one software company. The quality model reflects their and our opinion, the benchmark was created from their software systems and they designed several rule violations. The approach can be used in other companies as well, but it is highly recommended to create a new benchmark and calibrate the weights of the used quality model.

The validation of our method was based barely on the opinions of the developers about the improvement of one submodule. Examining more refactoring processes, collecting exact values and running statistical tests would have provided more precise results, but only a few developers were working on the refactoring project and we could not run such tests because of the lack of data.

A big question about software metrics and rule violations affects the reliability of our approach as well. We know that they are useful, they are proved to be good indicators for some aspects of software quality, but maintainability is still a subjective high-level characteristic, and we cannot be sure that we took every important aspect into consideration.

VI. Conclusion and Future Work

In this paper we introduced a software quality model for the RPG programming language which conforms to the ISO/IEC 25010 international standard, and integrated it into the continuous software quality monitoring tool called QualityGate. We used software metrics, rule violations and code duplications to estimate the maintainability of RPG software systems and to reveal various source code issues.

As a validation of our approach we presented a case study in cooperation with the R&R Software development company. We organized the case study into four phases. In the initial

phase we calibrated the parameters of the tools, and in the second phase we set up the IBM i compiler to generate spool files into a subversion repository on a daily basis. The third phase of the case study was the refactoring of one of their submodules by the company. Finally, as a conclusion, we discussed their experiences in using our approach.

Based on the opinions of the developers, the industrial application of our method was a success. During the refactoring phase, the number of critical and major rule violations halved. Despite the fact that the complexity goodness of the examined system is still under the average, its maintainability value increased and crossed the baseline value, so the state of the project was changed from NO-GO to GO.

In the future we would like to analyze more RPG systems and create domain specific benchmarks. By extending the quality model with new metrics, more aspects could be taken into consideration. We would also like to collect more votes to the edges of the quality model to improve the generality of our method. Based on a widespread survey we would like to investigate the relationship between the exactly calculated quality values and the subjective opinions of RPG experts. Since we are able to measure the quality of the original and the migrated version of a legacy system it would be interesting to study their quality differences and using these experiences to provide better migration algorithms.

We also plan to extend the presented approach to further programming languages and perform similar industrial case studies to the one presented in this paper.

ACKNOWLEDGMENT

This research was supported by the Hungarian national grant GOP-1.1.1-11-2012-0323.

REFERENCES

[1] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design," in *IEEE Transactions on Software Engineering 20,6(1994)*, 1994, pp. 476–493.

[2] C. Boogerd and L. Moonen, "Assessing the Value of Coding Standards: An Empirical Study." in *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*. IEEE, 2008, pp. 277–286.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*. IEEE Computer Society, 1998, pp. 368–377.

[4] ISO/IEC, *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. ISO/IEC, 2005.

[5] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476 –493, jun 1994.

[6] G. Canfora, A. De Lucia, and G. A. Di Lucca, "An incremental object-oriented migration strategy for rpg legacy systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 01, pp. 5–25, 1999.

[7] R. Ferenc, I. Siket, and T. Gyimóthy, "Extracting Facts from Open Source Software," in *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society, Sep. 2004, pp. 60–69.

[8] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, 2003.

[9] P. Hegedűs, "A Probabilistic Quality Model for C# – an Industrial Case Study," *Acta Cybernetica*, vol. 21, no. 1, pp. 135–147, 2013.

[10] S. D. Hartman, "A counting tool for RPG," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 11, no. 3. ACM, 1982, pp. 86–100.

[11] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[12] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, July 1976.

[13] F. A. Naib, "An application of software science to the quantitative measurement of code quality," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 11, no. 3. ACM, 1982, pp. 101–128.

[14] S. H. Kan, S. Dull, D. Amundson, R. J. Lindner, and R. Hedger, "AS/400 software quality management," *IBM Systems Journal*, vol. 33, no. 1, pp. 62–88, 1994.

[15] G. Bakker and F. Hirdes, "Recent industrial experiences with software product metrics," in *Objective Software Quality*. Springer, 1995, pp. 179–191.

[16] K. Suntiparakoo and Y. Limpiyakorn, "Flowchart Knowledge Extraction on RPG Legacy Code," *Advanced Science and Technology Letters (ASEA 2013)*, vol. 29, pp. 258–563, 2013.

[17] T. L. Alves, P. Silva, and M. S. Dias, "Applying ISO/IEC 25010 Standard to prioritize and solve quality issues of automatic ETL processes," *IEEE International Conference on Software Maintenance (ICSM 2014)*, pp. 573–576, 2014.

[18] ISO/IEC, *ISO/IEC 9126. Software Engineering – Product quality*. ISO/IEC, 2001.

[19] H.-W. Jung, S.-G. Kim, and C.-S. Chung, "Measuring Software Product Quality: A Survey of ISO/IEC 9126," *IEEE Software*, pp. 88–92, 2004.

[20] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, "A Probabilistic Software Quality Model," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. Williamsburg, VA, USA: IEEE Computer Society, 2011, pp. 368–377.

[21] R. Baggen, K. Schill, and J. Visser, "Standardized Code Quality Benchmarking for Improving Software Maintainability," in *Proceedings of the Fourth International Workshop on Software Quality and Maintainability (SQM 2010)*, 2010.

[22] J. Bansiya and C. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, 2002.

[23] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos, "Requirements-driven software re-engineering framework," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 71–80.

[24] Y. Zou and K. Kontogiannis, "Migration to object oriented platforms: A state transformation approach," in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 530–539.

[25] T. L. Alves, C. Ypma, and J. Visser, "Deriving Metric Thresholds from Benchmark Data," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, 2010.

[26] J. P. Correia and J. Visser, "Benchmarking Technical Quality of Software Products," in *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 297–300.

[27] T. Bakota, P. Hegedűs, I. Siket, G. Ladányi, and R. Ferenc, "QualityGate SourceAudit: A Tool for Assessing the Technical Quality of Software," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 440–445.

[28] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[29] T. Bakota, P. Hegedűs, G. Ladányi, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, "A Cost Model Based on Software Maintainability," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*. Riva del Garda, Italy: IEEE Computer Society, 2012.