16

Automatic Control of the Software Development Process with Regard to Risk Analysis

Marian Jureczko
Wrocław University of Technology
Poland

1. Introduction

In this paper, we present a method for analysing risk in software development. Before describing the details, let us define the basic terms that are important for the further discussion. Among them, the most import is risk. In the context of software engineering it represents any potential situation or event that can negatively affect software project in the future and may arise from some present action. Software project risks usually affect scheduling, cost and product quality. Further details are given in Section 2.2. The risk will be discussed in the context of project requirements and releases. The requirements (or requirement model) represent requests from the customer that should be transformed into features and implemented in the software product. The releases refer to a phase in the software development life cycle. It is a term used to denote that the software product is ready for or has been delivered to the customer. This work takes into consideration agile development (i.e. processes with short iterations and many releases) therefore, it should be emphasized that some of the releases may be internal. These releases are not scheduled to be delivered to the customer; nevertheless they are 'potentially shippable'.

Considerable research has been done on controlling the software development process. The mainstream regards process improvement approaches (e.g. CMMI SEI (2002)) where time consuming activities are performed in order to monitor and control the work progress. In effect the maturity is assessed. Furthermore, mature processes indicate lower project failure risk.

The agile software development shifts the focus toward product Beck & Andres (2005). Moreover, it is recommended to ensure high level of test coverage. The tests are automated Jureczko & Młynarski (2010), and therefore it is possible to execute them frequently in a continuous integration system. Despite of the fact that the agile approach is more automated, risk assessment may be challenging. There are few risk related metrics right out of the box. Fortunately, due to the high level of automation considerable amount of data regarding quality and risk might be collected without laborious activities Stamelos & Sfetsos (2007). It is possible to build simple metrics tools (e.g. BugInfo Jureczko (Retrived on 01/11/2011a), Ckjm Jureczko & Spinellis (Retrived on 06/05/2011)) as well as a complex solution for identifying 'architectural smells' (Sotograph Hello2morrow Inc. (Retrived on 01/11/2011) into the continuous integration system. Unfortunately, it is not evident which metrics should

be collected and what analysis should be performed to accomplish the risk related goals. This chapter discusses which data should be collected and how it should be done in order to obtain accurate risk estimations with smallest effort. It will be considered how to mitigate the effort by using automation. We first suggest a risk analysis method that reflects the common trends in this area (however the focus is slightly moved toward agile software development) and then we decompose the method to well–known in software engineering problems. The potential solutions to the aforementioned problems are reviewed with regard to automation in order to identify improvement possibilities. In result, a direction for further research is defined. Moreover, the main value of this chapter is the review of the status and recent development of risk related issues with regard to automation.

The remainder of the chapter is organized as follows: in the next section an approach to risk analysis is suggested and decomposed into three problems, i.e. effort estimation, defect prediction and maintainability measurement. The next three sections survey the three aforementioned issues. The sixth section discusses the automation possibilities. Finally, the seventh section presents concluding remarks and future research direction.

2. Problem formulation

There are tools that support the risk analysis. Unfortunately, such tools usually do not offer automation, but require expert assistance, e.g. RiskGuide Górski & Miler (Retrived on 01/11/2011) or are proprietary black box approaches, e.g. SoftwareCockpit Capgemini CSD Research Inc. (Retrived on 01/11/2011). The users do not know what exactly is going on inside and the software projects may differ from each other Jureczko & Madeyski (2010); Zimmermann et al. (2009). In order to make good risk related decisions, the inference mechanism should be transparent and understandable.

2.1 Requirement model

The requirement model that incorporates both functional and non-functional features might lay foundations for a method of risk assessment. Each and every requirement is characterized by its specification and so-called time to delivery. Since most of the real world projects undergo not only new feature development, but also maintenance, defect repairs are considered requisite and hence also belong to the requirement model.

In case of any functional requirement, it may or may not be met in the final product, there are only two satisfaction levels: satisfied or not satisfied. In the case of non–functional requirements the satisfaction level might be specified on the ordinal or interval scale depending upon the requirement kind.

Failing to deliver some requirement in due time usually causes severe financial losses. When a functional requirement is missing, a penalty is a function of delivery time, which in turn is a constant defined in the customer software supplier agreement. On the other hand, when a non–functional requirement is not satisfied, a penalty is a function of both: delivery time and satisfaction level. These values should be defined in the very same agreement or evaluated using empirical data from similar historical projects.

Among the non–functional requirements two are especially interesting: quality and maintainability. Both are very difficult to measure Heitlager et al. (2007); Kan (1994) and both may cause severe loss in future. Let us consider a release of a software system with low level of quality and maintainability. There are many defects since the quality is low. The development of the next release will generate extra cost because the defects should be removed. Furthermore, the removal process might be very time consuming since the maintainability is low. There is also strong possibility that developing new features will be extremely expensive due to maintainability problems. Therefore, quality as well as maintainability should be considered during risk analysis.

2.2 The risk

The risk $R(t)$ specified over a set of possible threats (t represents a threat) is defined as follows:

$$R(t) = P(t) * S(t) \tag{1}$$

where $P(t)$ stands for a probability that t actually happens and $S(t)$ represents the quantitative ramifications expressed most often in financial terms. The following threat sets a good example: a product complying with some requirement is not delivered in due time. When the requirement is delivered with delay and the ramifications depends on the value of the delay, the Equation 1 should be drafted into the following form:

$$R(t) = \int_0^\infty P(t,\delta) * S(t,\delta) d\delta \tag{2}$$

where δ denotes the value of delay.

When risk is estimated for a threat on the grounds of delivery time distribution, the mean loss value becomes the actual value of risk. Then, summing up risk values for all requirements belonging to the requirement model of a particular release, the mean cumulative risk for the release r is obtained:

$$RR(r) = \sum_{t \in T_r} R(t) \tag{3}$$

where T_r is the set of threats connected with release r. If all the releases are considered, the mean threat cost is obtained for the whole project p:

$$RP(p) = \sum_{r \in p} RR(r) \tag{4}$$

The risk can be used to evaluate release plan as well as the whole project. When the $RP(p)$ calculated for the whole project surpasses the potential benefits, there is no point in launching the project. Furthermore, different release plans may be considered, and the $RR(r)$ value may be used to choose the best one.

One may argue that summing risks that were estimated for single threats ($R(t)$) in order to evaluate the risk regarding the whole project can result in cumulating estimation errors. Therefore, the provided estimates of $RP(p)$ can be significantly off. Such scenario is likely when the requirements come late in the software development life cycle and are not known

up front. Nevertheless, the suggested approach also has advantages. As a matter of fact, considering each threat separately can be beneficial for agile methods where plans are made for short periods of time, e.g. the Planning Game in XP Beck & Andres (2005). The agile methods usually employ high level of automation Beck & Andres (2005), Stamelos & Sfetsos (2007), and hence create a good environment for automated risk analysis. Furthermore, the mean cumulative risk for the release $RR(r)$ is designed to be the main area of application of the suggested approach and the release is usually the main concern of planning in agile methods Beck & Andres (2005), Schwaber & Beedle (2001).

The suggested risk analysis method partly corresponds with the ISO/IEC standard for risk management ISO (2004). The standard is focused on the whole software life cycle whereas our method can cover the whole life cycle but is focused on shorter terms, i.e. releases. Similar, but slightly more complex method was suggested by Penny Penny (2002), who emphasized the relevance of resources quality and environmental conditions (specifically interruptions and distractions in work). We do not question the value of those parameters. Nevertheless, they may cause difficulties in collecting the empirical data that is necessary to estimate the threat occurrence probabilities. Therefore, data regarding those parameters is not required in our risk analysis method, but when available, definitely recommended.

2.3 Threat occurrence probability

According to Equation 1 the risk consist of two elements : probability of occurrence of a threat $P(t)$ and expected loss $S(t)$. Let us assume that the threat t_i regards failing to deliver the requirement req_i in the due time. Therefore, in order to calculate $P(t_i)$ the distribution of expected effort, which is necessary to fulfill requirement req_i, must be evaluated. The methods of effort estimation are discussed in Section 3.

There is also a second factor that is relevant to the probability of occurrence. It is the availability of resources for the planned release, which means that the effort estimated for the requirements and the available manpower indicates the probability of failing to deliver a requirement in due time. Furthermore, not every requirement could be implemented with equal effort by any participant of the developers team Helming et al. (2009). Therefore, the effort estimations should be combined with a workflow model. The workflow model is indicated by the employed software development methodology, and hence should be available in every non chaotic process. Nevertheless, what derives from the methodology definition can be on a very high level of abstraction or flexibility, especially in the case of agile methods Schwaber & Beedle (2001), and detailing the release plan is not a trivial task. According to Bagnall et al. Bagnall et al. (2001) the problem is NP hard even without any dependencies between requirements. Fortunately, there are methods for semi-automated planning of release Helming et al. (2009).

Estimation of probability of occurrence of a threat $P(t)$ must take into consideration two factors: effort estimation for different requirements on the one hand and the allocation of resources on the other. Resources allocation can be done semi-automatically Helming et al. (2009) or may come from a manager decision. In both cases the resources allocation will be available and can be used for further analysis. However, in some cases (i.e. in the agile methods) it may be done on a very low level of granularity. Effort estimation is more

challenging. The difficulties that may arise during automating those estimations are discussed in the next section.

2.4 Loss function

As it has been stated in Section 2.1 the value of loss function might be explicitly defined in the customer software supplier agreement. Unfortunately, such simple cases do not cover all possibilities. There are quality related aspects that might not be mentioned in the agreement, but still may cause loss. According to the ISO/IEC 9126 ISO (2001) there are six characteristics to consider: functionality, reliability, usability, efficiency, maintainability, and portability. Some of these characteristics may be covered by the customer software supplier agreement and when are not delivered on the contracted level there is a penalty. However, there are two aspects that should be explicitly considered, i.e. software defects and maintainability (the defects might regard not satisfying one of the five other ISO quality characteristics).

It is a typical case that software contains defects Kan (1994). Therefore, the customers, in order to protect their interests, put defect fixing policies in the agreement. As a result, the defects create loss. First of all they must be repaired, and sometimes there is also a financial penalty.

There are many different grounds for defects. Nevertheless, there is a general rule regarding the relation between cost, time and quality called The Iron Triangle Atkinson (1999), Oisen (1971), see Fig. 1. According to The Iron Triangle cost, time and quality depend on each other. In other words, implementation of a set of requirements generates costs and requires time in order to achieve the desired quality. When one of those two factors is not well supported, the quality goes down, e.g. skipping tests of a component may result in a greater number of defects in this component. However, it should be taken into consideration that in the case of skipping tests we have savings in current release. The issues regarding lower quality and greater number of defects will arise in the next release. Therefore, it is important to estimate the risk regarding lower quality since it sometimes could be reasonable to make savings in current release and pay the bill in the next one. Risk related to low quality regards the loss that comes from defect fixing activities in the next release. Hence, the number of defects as well as the effort that is necessary to fix them should be predicted and used to construct the loss function. The defect prediction is discussed in Section 4.

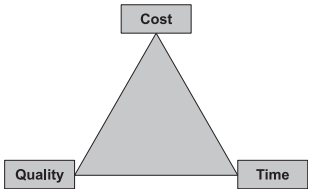

Fig. 1. The Iron Triangle

Low level of maintainability, which reflects problems with analysing, changing and testing the software increases the efforts of developing new features. Since the efforts are greater, the threat occurrence probability for forthcoming requirements grows as well. If the current release is not the last, the low maintainability related loss should be estimated and considered. Methods of estimating maintainability are discussed in Section 5.

2.5 Automation possibilities

The goal of this chapter is to present how risk can be estimated in an automated way. Risk has been decomposed into five factors: effort estimation, release planning, missing functionality penalty, defect prediction, and maintainability measuring (Fig. 2. The release planning is a manager's decision. Nevertheless, the decision may be supported by a tool. Specifically, the suggested risk evaluation approach can be used to assess a release plan. The penalty is usually defined in the customer software supplier agreement and is a function of both: delivery time and satisfaction level of the requirement implementation.

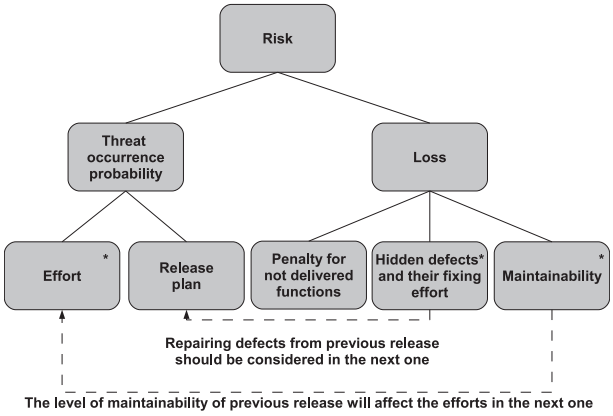

Fig. 2. Problem decomposition; elements denoted with '*' are detailed in forthcoming sections.

The three other aspects (effort, hidden defects and maintainability) are evaluated using empirical data, therefore there is room for automation. When there is no estimation method available, historical data must be collected and used to train a prediction model. If there is no historical data, cross-project prediction methods should be considered. All the aforementioned activities may be automated to some extent. We can employ tools to collect data, we can use tools to make the estimations and finally to generate the risk related reports.

Unfortunately, not all aspects of discussed above risk analysis were investigated closely enough to give conclusive remarks and not all of them are supported by tools. The following sections will discuss what has been already done and where there are gaps that must be filled by practitioners who would like to automate the risk analysis.

3. Effort estimation

Considerable research has been done on effort estimation. It regards different techniques and at least some of them may be used in risk analysis. As it was discussed in Section 2, threats like 'a product complying with some requirement is not delivered in due time' should be considered. Therefore, efforts for each requirement must be estimated to assess the risk.

In order to choose the most appropriate effort estimation approach let us review the available ones. Boehm et al. Boehm et al. (2000) summarized the leading techniques (the content of this

section is mostly based on their work). Later, the estimation methods were also reviewed by Jorgensen and Shepperd Jorgensen & Shepperd (2007).

3.1 Model-based techniques

Most of the models have functional form. Metrics which describe features of the object of estimation or features of the organization, which is going to develop the object, are used as model input. After some calculations the model produces value of the expected effort. Unfortunately, many models are proprietary and hence cannot be analysed in terms of model structure. On the other hand, the proprietary solutions are usually ready to use tools with high level of automation.

3.1.1 Constructive Cost Model (COCOMO 81 and COCOMO II)

The COCOMO model was firstly suggested in Boehm (1981). It is an algorithmic software cost estimation model that uses regression formula. The intuition behind COCOMO model is that the development grows exponentially as the developed system grows in size:

$$Effort_{PM} = a * (KLOC^b) * (\prod_j EM_j) \tag{5}$$

Where, $Effort_{PM}$ is the whole effort in person months, $KLOC$ is thousands of lines of source code, EM is an effort multiplier generated via regression on historical data, a and b denote software type coefficients.

The COCOMO 81 model was based on a study of 63 projects that were created using the waterfall software development model, which reflected the practices of the 80'. Nevertheless, software development techniques changed dramatically, namely highly iterative processes were employed. Therefore, applying the COCOMO 81 model becomes problematic. The solution to the problem was to redesign the model to accommodate different approaches to software development.

The revised cost estimation model was called COCOMO II and was initially described by Boehm et al. Boehm et al. (1995). The model is focused on estimating the effort that regards the whole system. Moreover, it is based on empirical data; it is widely used and hence well validated. Web based tools that support the COCOMO II model are available on the model's web page Center for Systems and Software Engineering (Retrived on 01/11/2011). There are commercial implementations: COSTAR Softstar Systems (Retrived on 01/11/2011) and Cost Xpert Cost Xpert AG (Retrived on 01/11/2011) as well, so the automation is relatively easy. There are also derived models, e.g. COSYSMO Valerdi et al. (2003) or AFCAA REVIC Air Force Cost Analysis Agency (Retrived on 01/11/2011).

3.1.2 Putnam's Software Life-cycle Model (SLIM)

The Putnam's Software Life-cycle Model Putnam (1978) is based on analysis of software development lifecycle in terms of the well–known Rayleigh distribution of resources (project personnel) usage versus time. Quantitative Software Management offers a tools suite based

on metrics collected from over 10 000 completed software projects that employ the Putnam's SLIM model Quantitative Software Management Inc. (Retrived on 01/11/2011).

The tools assure high level of automation. SLIM was originally designed to make estimations for the whole project. However, the project can be interpreted as a set of requirements, what partly corresponds with the described in Section 2 risk analysis.

3.1.3 SPR KnowledgePLAN

SPR KnowledgePLAN (following in the footsteps of previous SPR products: SPQR/20 and Checkpoint) is a knowledge-based software estimation solution Software Productivity Research (Retrived on 01/11/2009). It is based on a proprietary database of 14 531 projects (version 4.3). The estimation is calculated according to detailed project description.

The tool predicts effort at four levels of granularity: project, phase, activity and task, therefore it is suitable for risk analysis. Furthermore, since there is good support in tools the automation is relatively simple.

3.1.4 Functionality–based estimations

3.1.4.1 Function Points

The function points were suggested by Albrecht Albrecht (1979) and are a standard unit of measure for representing the functional size of a software application. The functional requirements are categorized into five types: outputs, inputs, inquiries, internal files, and external interfaces. Subsequently, the requirements are assessed for complexity and a number of function points are assigned. Since 1984 this method has been promoted by The International Function Point Users Group (IFPUG). This group produced a set of standards, known as the ISO/IEC 14143 series. Function Point Analysis was also developed by NESMA Netherlands Software Metrics Association (Retrived on 01/11/2011). It resulted in method called NESMA Functional Size Measurement that is compliant with ISO/IEC 24570. Moreover, there are good automation possibilities since the Function Point Analysis is supported by a tool called Metric Studio TSA Quality (Retrived on 01/11/2011).

3.1.4.2 MKII FPA

Mk II Function Point Analysis Symons (1991) is a software sizing method that belongs to the function point group of techniques. The most important feature of this method is the simple measurement model. There are only three components to consider: inputs (processes that enter data into software), outputs (processes that take data from software), and entity references (storage, retrieval and deletion of date from permanent storage).

3.1.4.3 Weighted Micro Function Points (WMFP)

The Weighted Micro Function Points (WMFP) is a software size estimating method developed by Logical Solutions Logical Solutions (Retrived on 01/11/2011). The method measures several different software metrics obtained from the source code that represents effort and are translated into time: flow complexity, object vocabulary, object conjuration, arithmetic intricacy, data transfer, code structure, inline data, and comments.

3.1.4.4 Common Software Measurement International Consortium (COSMIC)

The COSMIC project was launched in 1998, it is a voluntary, world-wide grouping of software metrics experts Common Software Measurement International Consortium (Retrived on 01/11/2009). The focus is on developing a method of measuring functional size of software. The method is entirely open, and all the documentation is available for free of charge download. Unfortunately, the consortium provides only the method description and hence automation may be time consuming. However, there are third party vendors that support the method[1].

3.1.4.5 Object Points

Object Points is an object oriented software size estimating method developed by Banker et al. Banker et al. (1991). The method is based on the following object types (type definitions after Banker et al. Banker et al. (1991)): rule set (a collection of instructions and routines written in a high level language of a CASE tool), 3GL module (a precompiled procedure, originally written using third-generation language), screen definition (logical representation of an on-screen image), and user report. The object points are obtained by summing the instances of objects of the aforementioned types and weighting each type by the development effort associated with it.

3.1.4.6 Use Case Points (UCP)

UCP is a measure of software size that takes into consideration use cases. It is calculated by counting the number of actors and transactions that were included in the flow of use case scenarios. Transaction is an event that occurs between the actor and the system and can be made entirely or not at all.

There are tools that support the UCP, e.g. Kusumoto et al. (2004). Nevertheless, identifying transactions in the use cases is not a trivial task. Natural language processing methods are usually employed Ochodek & Nawrocki (2007) and there is a considerable mistake possibility.

3.1.5 PRICE-S

The PRICE-S is a parametric cost model that was originally developed for internal use in NASA software projects. It is implemented in the TruePlanning PRICE Systems (Retrived on 01/11/2011), which is a tool that encapsulates the experience with 3 212 projects from over 30 years.

According to the authors, risk refers to the fact that because a cost estimate is a forecast, there is always a possibility of the actual cost being different from the estimate (such situation very well corresponds with our goal). Furthermore, using TruePlanning tool allows risk analysis that quantifies probability.

The tool makes estimations on work package level hence it is appropriate for the described in Section 2 risk analysis. Unfortunately, according to Boehm et al. (2000) the model equations were not released in public domain, still, it is a well–designed commercial solution and therefore the automation is supported.

[1] http://www.cosmicon.com/vendorsV3.asp

3.1.6 SEER for Software (SEER-SEM)

SEER-SEM is an algorithmic, parametric project management system that supports estimating, planning and tracking effort and resource consumption for software development and maintenance Galorath Inc. (Retrived on 01/11/2011). The estimations are calculated using effective size (S_e). It is a generic metric that covers development and integration of new, reused, and commercial off-the-shelf software modules. Several sizing metrics can be translated into S_e, this includes lines of code (LOC), function points, IFPUG function based sizing method, and use cases. Although the tool is focused on estimation for the whole project, there are great automation possibilities due to the support for integration with MicroSoft and IBM products.

3.1.7 PROxy-Based Estimating (PROBE)

Proxy based estimating uses previous developments in similar application domains. The method is a part of the Personal Software Process Humphrey (1995). A proxy is a unit of software size that can be identified early in a project (e.g. screen, file, object, logical entity, function point). Proxies can be translated into lines of source code according to historical sizes of similar proxies in earlier developed projects.

The concept of proxies corresponds well with the risk analysis. Nevertheless, according to author's knowledge there is no tool that supports the method. Therefore, the automation may be challenging.

3.2 Expert–based techniques

The expert–based techniques capture the knowledge of domain experts and use it to obtain reliable estimations. Those techniques are especially useful when there is not enough empirical data to construct a model. Jorgensen and Shepperd Jorgensen & Shepperd (2007) pointed out two important facts regarding those methods of estimation. Firstly, they concluded that the formal estimation techniques have no documented accuracy higher than the expert–based approach. Secondly, the expert–based technique is the most commonly used method in the industry. Unfortunately, using expert knowledge is very inconvenient with regard to automation.

Two expert–based techniques have been developed. These are the Work Breakdown Structure and the Delphi method.

3.2.1 Work Breakdown Structure (WBS)

The WBS is a method of organizing project into a hierarchical structure of components or activities (as a matter of fact, two structures can be created; one for the software product itself, and one for the activities that must be performed in order to build the software) in a way that helps describe the total work scope of the analysed project. The hierarchical structure is developed by starting with the end objective and dividing it into gradually smaller ones. A tree, which contains all actions that must be executed in order to achieve the mentioned objective is obtained as the result. The estimation is done by summing the efforts or resources assigned to single elements.

3.2.2 Delphi

The Delphi method is a structured, interactive estimation method that is based on experts judgements. A group of experts is guided to a consensus of opinion on an issue. Participants (experts) make assessments (estimations) regarding the issue without consulting each other. The assessments are collected and presented to each of the experts individually. Subsequently, the participants make second assessment using the knowledge regarding estimations provided by other experts. The second round should result in a narrowing of the range of assessments and indication of a reasonable middle value regarding the analysed issue.

The Delphi technique was developed at The Rand Corporation in the late 1940s as a way of making predictions regarding future events Boehm et al. (2000); currently a similar method, i.e. the planning poker is very popular.

3.3 Lerning-oriented techniques

Learning-oriented techniques include a wide range of different methods. There are traditional, manual approaches like case studies as well as highly automated techniques, which build models that learn from historical data, e.g. neural networks, classification trees. Unfortunately, even the most highly automated approaches require some manual work since the historical data must be collected and the estimation models must be integrated with the software development environment. The learning-oriented techniques are commonly employed in other approaches.

3.3.1 Case studies

Case studies represent an inductive process, whereby estimators and planners try to learn useful general lessons and estimation heuristics by extrapolation from specific examples Boehm et al. (2000). Case studies take into consideration the environmental conditions and constraints, the decisions that influenced software development process, and the final result. The goal is focused on identifying relations between cause and effect that can be applied in other projects by analogy. Case studies that result in a well–understood cause effect relations constitute good foundation for designing automated solutions that can be applied to similar project. Nevertheless, in order to define the boundaries of similarity further analysis is usually required.

According to Jorgensen and Shepperd Jorgensen & Shepperd (2007) there are few case studies focused on the effort estimation. In consequence, there is a possibility that we still can learn a lot from well described real–life cases.

3.3.2 Regression based techniques

Regression based techniques are the most popular method in the effort estimation studies Jorgensen & Shepperd (2007). Moreover, they were used in many of the model based approaches, e.g. COCOMO, SLIM.

The regression based methods refer to the statistical approach of general linear regression with the least square technique. Historical data is employed to train the model and then the model

is used for making estimations for new projects. The regression based methods are well suited for automation since they are supported by almost all statistic tools (including spreadsheets). Unfortunately, the data which is necessary to train the model is not always available and its collection may cause automation related issues.

3.3.3 Neural networks

There are several artificial intelligence approaches to effort estimation and the neural networks are the most popular among them. Nevertheless, according to Jorgensen and Shepperd Jorgensen & Shepperd (2007) less than 5% of research papers oriented to effort estimation considered the neural networks. There is no no good reason for such low popularity since the neural networks have proven to have adequate expression power. Idri et al. Idri et al. (2002) used backpropagation threelayer perceptron on the COCOMO 81 dataset and according to the authors, the obtained accuracy was acceptable. Furthermore, the authors were able to interpret the obtained network by extracting if-then fuzzy rules from it, hence it alleviates to some extent the main neural networks drawback, namely low readability.

Other example of neural networks usefulness was provided by Park and Baek Park & Baek (2008), who investigated 148 software projects completed between 1999 and 2003 by one of the Korean IT service vendors. The authors compared accuracy of the neural networks model with human experts' judgments and two classical regression models. The neural network outperformed the other approaches by producing results with MRE (magnitude of relative error) equal to 59.4 when expert judgements and the regression models produced results with MRE equal to 76.6, 150.4 and 417.3 respectively.

3.3.4 Analogy

The analogy–based estimation methods are based on project's characteristic in terms of features, e.g. size of the requirements, size and experience of the team, software development method. The features are collected from historical projects. In order to forecast effort for a new project the most similar historical projects must be found. Similarity can be defined as Euclidean distance in n-dimensional space where n is the number of project features. The dimensions are standardized since all of them should have the same weight. The most similar projects, i.e. the nearest neighbours, are used as the basis for effort prediction for the new project. According to Jorgensen and Shepperd Jorgensen & Shepperd (2007) the popularity of research on analogy–based estimation models is increasing.

Good prediction abilities of the analogy–based methods were shown by Shepperd and Schofield in a study conducted on 9 datasets, a total of 275 software projects Shepperd & Schofield (1997). The analogy (case based reasoning) was compared upon stepwise regression analysis and produced superior predictive performance for all datasets when measured in MMRE (mean magnitude of relative error) and for 7 datasets when measured in Pred(25) (percentage of predictions that are within 25% of the actual value).

Gupta et al. Gupta et al. (2011) surveyed the recent analogy based techniques for effort estimation and identified following approaches (some of them overlap estimation methods mentioned in other subsections):

- Machine learning based approaches (Multi Layer Perceptrons, Radial Basis Functions, Support Vector Regression, Decision Trees)
- Fuzzy logic based approaches (Fuzzy analogy, Generalized Fuzzy Number Software Estimation Model)
- Grey Theory based approaches (Grey Relational Analysis based software project effort (GRACE), Improved GRA, GLASE, Weighted GRA, GRACE$^+$)
- Hybrid And Other Approaches (Genetic Algorithms with GRA, Fuzzy Grey Relational Analysis, AQUA (case-based reasoning & collaborative filtering))

Gupta et al. also investigated the efficiency of the above listed methods. Moreover, the authors collected and presented information regarding the prediction accuracy of those methods. The best result was obtained in the case of Decision Trees. Nevertheless, it is not conclusive since the methods were evaluated on different data sets.

3.3.5 Analysis effort method

The analysis effort method is suited to producing the initial estimates for a whole project. The method is based on doing some preliminary detailed design and measuring the time consumed to accomplish these tasks. Then, the results are used to estimate effort for the rest of the analysis, design, coding and testing activities. For the purpose of this method, a simplified, waterfall based software development lifecycle is assumed. This method uses three key factors that apply to the estimated project in terms of its size, familiarity and complexity. Unfortunately, the Analysis Effort Method requires some manual work in the initial phase, and therefore impedes the automation. A detailed description of the Analysis Effort Method was given by Cadle and Yeates Cadle & Yeates (2007).

3.4 Dynamics–based techniques

The dynamics based techniques take into consideration the dynamics of effort over the duration of software development process. The environment conditions and constraints (e.g. available resources, deadlines) may change over time. Some of the changes are scheduled, some not, but all of them may have significant influence on productivity and, in consequence, the likelihood of project success.

3.4.1 System dynamics approach

System dynamics is a continuous simulation modeling methodology whereby model results and behavior are displayed as graphs of information that change over time. Models are represented as networks modified with positive and negative feedback loops. Elements within the models are expressed as dynamically changing levels or accumulations (the nodes), rates or flows between the levels (the lines connecting the nodes), and information relative to the system that changes over time and dynamically affects the flow rates between the levels (the feedback loops). Description after Boehm et al. Boehm et al. (2000).

There are examples of application of the system dynamics approach e.g. Madachy (1996), Ruiz et al. (2001). Detailed description of the concept was provided by Abdel-Hamid and Madnick Abdel-Hamid & Madnick (1991).

3.5 Composite techniques

Composite techniques employ two or more different effort estimation approaches (including those mentioned above) to produce the most appropriate model.

3.5.1 Bayesian approach

Bayesian approach is a method of inductive reasoning that minimizes the posterior expected value of a loss function (e.g. estimation error). Bayesian Belief Nets are an approach that incorporates the case effect reasoning. The cause–based reasoning is strongly recommended by Fenton and Neil Fenton & Neil (2000). The authors gave a very interesting example of providing wrong conclusion when ignoring the cause–effect relations. The example regards analysing car accidents. The relation between month of year and the number of car accidents shows that the safest to drive months are January and February. The conclusion, given the data involved, is sensible, but intuitively not acceptable. During January and February the weather is bad and causes treacherous road conditions and fewer people drive their cars and when they do they tend to drive more slowly. According to the authors, we experience similar misunderstanding in the effort estimation since most models use equation with following form: $effort = f(size)$. The equation ignores the fact that the size of solution cannot cause effort to be expanded.

Fenton and Neil employed Bayesian Belief Nets to represent casual relationships between variables investigated in the model Fenton & Neil (1999a). In consequence, they obtained model that has the following advantages:

- explicit modeling of ignorance and uncertainty in estimations,
- combination of diverse type of information,
- increasing of visibility and auditability in the decision process,
- readability,
- support for predicting with missing data,
- support for 'what–if' analysis and predicting effects of process change,
- rigorous, mathematical semantic.

3.5.2 COSEEKMO

COSEEKMO is an effort-modeling workbench suggested by Menzies et al. Menzies et al. (2006) that applies a set of heuristic rules to compare results from alternative models: COCOMO, Local Calibration Boehm (1981), Least Squares Regression and Quinlan's M5P algorithm Quinlan (1992). The model was trained and evaluated using data from 63 projects originally used to create the COCOMO 81 model, 161 projects originally used to create the COCOMO II model and 93 projects from six NASA centers.

The model is focused on estimating the effort that regards the whole application. According to the authors the COSEEKMO analysis is fully automated; all model procedures were published as programs written in pseudo–code or sets of heuristics rules.

3.5.3 ProjectCodeMeter

The ProjectCodeMeter Logical Solutions (Retrived on 01/11/2011) is a commercial tool that implements four different cost models: Weighted Micro Function Points (WMFP), COCOMO 81, COCOMO II, and Revic 9.2. When consider which models are supported, it is not surprising that the tool is focused on making estimations for the whole project. There is a great automation possibility since the tool integrates with many IDEs, i.e. MicroSoft Visual Studio, CodeBlock, Eclipse, Aptana Studio, Oracle JDeveloper, and JBuilder.

3.5.4 Construx Estimate

Estimate Construx (Retrived on 01/11/2011) predicts effort, budget, and schedule for software projects based on size estimates. It is calibrated with industry data. However, it is possible (and recommended) to calibrated Estimate with own data. Estimate is based on Monte Carlo simulation and two mentioned above estimation models: Putnam's SLIM and COCOMO. It is focused on estimations for the whole project.

3.5.5 CaliberRM ESTIMATE Professional

CaliberRM ESTIMATE Professional Borland Software Corporation (Retrived on 01/11/2011) uses two models in order to make initial estimate of total project size, effort, project schedule (duration) as well as staff size and cost: COCOMO II and Putnam's SLIM. Later, a Monte Carlo Simulation may be used to generate a range of estimates with different probabilities later the most appropriate one can be selected and executed.

4. Defect prediction

The described in Section 2 risk analysis considers defect repairs as a vital part of the risk assessment process. Delivering low quality software will result in a number of hidden defects, which is why the development of the next release will generate extra cost (these defects must be removed). In consequence, it is important to estimate the number of defects and the effort that is needed to fix them. Considerable research has been conducted on defect prediction. The current state of the art has been recently reported in systematic reviews Catal (2011); Kitchenham (2010).

4.1 Predicting the number of defects

Traditionally, the defect prediction studies were focused on predicting the number or density of defects Basili & Perricone (1984); Gaffney (1984); Lipow (1982). However, these approaches were extensively criticized Fenton & Neil (2000; 1999b). It was concluded that the models offer no coherent explanation of how defect introduction and detection variables affect defect count and hence there is no cause–effect relation for defect introduction. Furthermore, the employed variables are correlated with gross number of defects, but are poor in predicting. In consequence, these approaches are not adequate for risk analysis.

Recent defect prediction studies are focused on supporting resource management decisions in the testing process. The studies investigate which parts of a software system are especially defect prone and hence should be tested with special care and which could be defect free and

hence may be not tested. The most popular approaches are focused on organizing modules in an order according to the expected number of defects Jureczko & Spinellis (2010); Weyuker et al. (2008) or on making boolean decision regarding whether a selected module is defect free or not Menzies et al. (2007); Zimmermann & Nagappan (2009). Unfortunately, such approaches are not effective in estimating the overall number of defects. Probably the most appropriate approach to defect prediction with regard to risk analysis is Bayesian belief nets Fenton & Neil (1999a),Fenton et al. (2007). This approach can be employed to predict the number of defects Fenton et al. (2008) and can provide the cause–effect relations. The Bayesian belief nets approach might be a source of automation related issues. Unfortunately, designing and configuring such a net without expert assistance is challenging. However, when the net is defined, it is readable and understandable Fenton & Neil (2000) and hence no further expert assistance in necessary.

There is also an alternative approach that is much easier to automate, but unfortunately does not support the cause–effect analysis, namely, reliability growth models. The reliability growth models are statistical interpolation of defect detection in time by mathematical functions, usually Weibull distribution. The functions are used to predict the number of defects in the source code. Nevertheless, this approach has also drawbacks. The reliability growth models require detailed data regarding defects occurrence and are designed for the waterfall software development process. It is not clear whether they are suitable for highly iterative processes. Detailed description of this approach was presented by Kan Kan (1994).

4.2 Predicting the defect fixing effort

The number of residual defects does not satisfy the requirements of risk analysis, when the fixing effort remains unknown. There are two types of studies focused on fixing time. First type estimates the time from bug report till bug fix Giger et al. (2010); Gokhale & Mullen (2010); Kamei et al. (2010). However, this type of estimation is not helpful in risk analysis. Fortunately, the second type of studies is more appropriate since it is focused on the fixing effort.

Weiss et al. Weiss et al. (2007) conducted an empirical study regarding predicting the defect-fixing effort in the JBoss project. The authors used text similarity techniques to execute analogy (nearest neighbor approach) based prediction. All types of issues were considered (defects, feature requests, and tasks). However, the best results were obtained in the case of defects; the average error was below 4 hours. Nevertheless, the study has low external validity and according to the authors should be considered as a proof of concept. Further research in similar direction was conducted by Hassouna and Tahvildari Hassouna & Tahvildari (2010). The authors conducted an empirical study regarding predicting the effort of defect–fixing in JBoss (the same data set as in Weiss et al. (2007)) and Codehaus projects. The suggested approach (Composite Effort Prediction Framework) enhanced the technique suggested by Weiss et al. (2007) by data enrichment and employing majority voting, adaptive threshold, and binary clustering in the similarity scoring process. From the automation point of view, this study provides us with an important finding. The authors developed PHP scripts to crawl and collect issue information from a project tracking system called JIRA. Hence, the authors showed a method of gathering empirical data with regard to defect fixing effort.

Zeng and Rine Zeng & Rine (2004) used dissimilarity matrix and self–organizing neural networks for software defects clustering and effort prediction and obtained good accuracy when applied to similar projects (average MRE from 7% to 23%). However, only 106 samples corresponding to 15 different software defect fix efforts were used to train the model, which significantly affects the external validity.

Datasets from NASA were also used by Song et al. Song et al. (2006). The study investigated 200 software projects that were developed over more than 15 years and hence the external validity was significantly better. The authors used association rule mining based method to predict defect correction effort. This approach was compared to C4.5, Naive Bayes and Program Assessment Rating Tool (PART) and showed the greatest prediction power. The accuracy was above 90%.

The studies regarding predicting defect fixing effort present various approaches. Nevertheless, the external validity is limited and they do not conclude which approaches is optimal or how it can be automated.

4.3 Cross–project prediction

In order to employ a defect prediction model, it must be trained. When analysing the first release of a project, or when no historical data is collected, the model must be trained using data from other projects. Unfortunately, the cross-project defect prediction does not give good results. Some studies have investigated this issue recently He et al. (2011); Jureczko & Madeyski (2010); Zimmermann et al. (2009). However, all of them reported low success rate.

5. Measuring maintainability

A software system that delivers all functionalities correctly may still cause problems in future. Specifically, the effort connected with implementing new features may differ depending on the character of already written source code. There are phenomena that affect the easiness of modifying and improving an application, e.g. the code smells (symptom in the source code that possibly indicates a deeper problem). Pietrzak and Walter Pietrzak & Walter (2006) investigated the automatic methods of detecting code smells. The work was later extended in Martenka & Walter (2010) by introducing a hierarchical model for software design quality. In consequence, the model employs the interpretation of software metrics as well as historical data, results of dynamic behavior, and abstract syntax trees to provide traceable information regarding detecting anomalies, especially code smells. Thanks to the traceability feature it is possible to precisely locate the anomaly and eliminate it. The model is focused on evaluating high–level quality factors that can be a selection of quality characteristic.

Unfortunately, code smells and readability do not cover all aspects of software that are related to the easiness of modifying, improving and other maintenance activities. These aspects are covered by a term called 'software maintainability', which is defined in the IEEE Std 610.12-1990 as: the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment IEEE (1990).

Considerable research has been conducted on measuring the maintainability; therefore there is guidance regarding which approach can be used in the mentioned in Section 2 risk analysis with regard to the automation.

5.1 ISO Quality model

The ISO/IEC 9126 Quality Model ISO (2001) identifies several quality characteristics, among them maintainability. The model also provides us with sub-characteristics. In the case of maintainability they are: analyzability (the easiness of diagnosing faults), changeability (the easiness of making modifications), stability (the easiness of keeping software in consistent state during modification), testability (the easiness of testing the system), maintainability–compliance (the compliance with standards and conventions regarding maintainability). Such multidimensional characteristic of maintainability seems to be reasonable. Software that is easy to analyse, change, test, and is stable and complies with standards and conventions should be easy to maintain.

The ISO Quality Model recommends sets of internal and external metrics designed for measuring the aforementioned sub–characteristics. For instance, for the changeability there is the 'change implementation elapse time' suggested as external metric and the 'change impact' (it is based on the number of modifications and the number of problems caused by these modifications) as internal metric. Nevertheless, the ISO Quality Model was criticized by Heitlager et al. Heitlager et al. (2007). The authors noted that the internal as well as the external metrics are based on the observations of the interaction between the product and its environment, maintainers, testers and administrators; or on comparison of the product with its specification (unfortunately, the specification is not always complete and correct). Heitlager et al. Heitlager et al. (2007) argued that the metrics should be focused on direct observation of the software product.

5.2 Maintainability Index (MI)

The Maintainability Index was proposed in order to assess the software system maintainability according to the state of its source code Coleman et al. (1994). It is a composite number obtained from an equation that employs a number of different software metrics:

$$MI = 171 - 5.2 * ln(\overline{Vol} - 0.23 * \overline{V(g')} - 16.2 * ln(\overline{LOC} + 50 * sin(\sqrt{2.46 * perCM} \quad (6)$$

Where $\overline{Vol}$, $\overline{V(g')}$, and $\overline{LOC}$ are the average values of Halstead Volume, McCabe's Cyclomatic Complexity, and Lines Of Code respectively, and perCM is the percentage of commented lines in module.

The MI metric is easy to use, easy to calculate and hence to automate. Nevertheless, it is not suitable to the risk analysis since it does not support cause–effect relations. When analysing risk, it is not enough to say that there is a maintainability problem. An action that can mitigate the problem should always be considered. Unfortunately, in the case of the MI we can only try to decrease the size related characteristics (Vol, V(g'), and LOC) or increase the percentage of comments, what not necessarily will improve the maintainability. The MI is correlated with the maintainability, but there are no proofs for cause-effect relation and hence we cannot expect that changing the MI will change the maintainability.

5.3 Practical model for measuring maintainability

Heitlager et al. Heitlager et al. (2007) not only criticized other approaches, but also suggested a new one, called Practical model for measuring maintainability. The authors argued that the model was successfully used in their industrial practice and to some extend is compatible with the widely accepted ISO Quality Model ISO (2001). The model consists of five characteristics (source code properties that can be mapped onto the ISO 9126 maintainability sub–characteristics: Volume, Complexity per unit, Duplication, Unit size, and Unit testing. The authors defined metrics for each of the aforementioned properties that are measured with an ordinal scale: ++, +, o, –, – –; where ++ denotes very good and – – very bad maintainability.

5.3.1 Volume

Volume represents the total size of a software system. The main metric of volume is Lines Of Code. However, the authors noted that this metric is good only within a single programming language since different set of functionalities can be covered by the same number of lines of code written. Therefore, the Programming Languages Table Jones (Retrived on 01/11/2011) should be considered in order to express the productivity of programming languages. Heitlager et al. gave an ready to use example for three programming languages (Tab. 1).

rank	Java	Cobol	PL/SQL
++	0–66	0–131	0–46
+	66–246	131–491	46–173
o	246–665	491–1,310	173–461
–	665–1,310	1,310–2,621	461–992
– –	>1,310	>2,621	>992

Table 1. Values of the volume metric in correspondence with source code size measured in KLOC according to Heitlager et al. (2007).

5.3.2 Complexity per unit

Complex source code is difficult to understand and hence to maintain. The complexity per unit (method) can be measured using the McCabe's Cyclomatic Complexity (CC). Heitlager et al. pointed out that summing or calculating the average value of CC does not reflect the maintainability well, since the maintenance problems are usually caused by few units, which has outlying values of the complexity metric. To mitigate the problem, the authors suggested to assign risks to every unit according to the value of CC: $CC \in [1,10] \rightarrow$ *low risk*; $CC \in [11,20] \rightarrow$ *moderate risk*; $CC \in [21,50] \rightarrow$ *high risk*; $CC \in [51,\infty] \rightarrow$ *very high risk*. Using the risk evaluations, the authors determine the complexity rate according to Tab. 2.

5.3.3 Duplication

Heitlager et al. suggested calculating duplication as the percentage of all code that occurs more than once in equal code blocks of at least 6 lines. Such measure is easy to automate since there are tools that support identifying code duplication (e.g. PMD). Details regarding mapping the value of duplication onto maintainability are presented in Tab. 3.

rank	Moderate risk	High risk	Very high risk
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
–	50%	15%	5%
– –	–	–	–

Table 2. Values of complexity per unit according to the percentage of units that do not exceed the specified upper limit for percentage of classes with given risk according to Heitlager et al. (2007).

rank	Duplication	rank	Unit test coverage
++	0–3%	++	95–100%
+	3–5%	+	80–95%
o	5–10%	o	60–80%
–	10–20%	–	20–60%
– –	20–100%	– –	0–20%

Table 3. Rating scheme for duplication and test code coverage according to Heitlager et al. (2007).

5.3.4 Unit size

The size per unit should be measured using LOC in a similar way to cyclomatic complexity per unit. Unfortunately, the authors do not provide us with the threshold values that are necessary to map the measured values to risk categories and rank scores.

5.3.5 Unit testing

Heitlager et al. suggested that the code coverage should be used as the main measure of unit tests. Details are presented in Tab. 3.

Measuring test code coverage is easy to automate due to good tool support (e.g Clover[2], Emma[3]. Unfortunately, this measure does not reflect the quality of tests. It is quite easy to obtain high code coverage with extremely poor unit tests, e.g. by writing tests that invoke methods, but do not check their behavior. Therefore, the authors recommend supplementing the code coverage with counting the number of assert statements. Other measure of unit tests quality is becoming very popular nowadays, namely mutation score, see Madeyski & Radyk (2010) for details.

5.4 Other approaches

The presented above approaches do not cover all maintainability measuring techniques that are used or investigated. The researchers also study the usefulness of regression based models, Bayesian Networks, fuzzy logic, artificial intelligence methods and software reliability models. A systematic review of software maintainability prediction and metrics was conducted by Riaz et al. Riaz et al. (2009).

[2] http://www.atlassian.com/software/clover/
[3] http://emma.sourceforge.net/

6. Automation possibilities and limitations

Three areas of risk analysis were discussed: effort estimation, defect prediction and maintainability measurement. The most common techniques and tools were described. Unfortunately, the automation of these analyses is not as trivial as installing off–the–shelf products. There are still gaps that should be filled. This section points out what is missing in this area and what is already available.

6.1 Effort estimation

The research conducted on effort estimation as well as the set of available tools is very impressive. Unfortunately, the focus is oriented towards estimating the whole project what is not suitable for the defined in Section 2 risk analysis. There is support for estimating effort for single requirement (e.g. SLIM, SPR KnowledgePLAN, PRICE-S). However, the methods are not optimised for such task and not as well validated as in the case of larger scope estimation.

For the risk analysis it is crucial to incorporate the uncertainty parameters of the prediction. Let's assume that there are 5 requirements and each of them is estimated to 5 days. Together it makes 25 days. And on the other hand, there are 30 man–days to implement these requirements. If the standard deviation of the estimations is equal to 5 minutes, the release will presumably be ready on time. However, when the standard deviation is equal to 5 days, there will be great risk of delay. Fortunately, some of the effort estimation techniques incorporate uncertainty, e.g. the COCOMO II method was enhanced with the Bayesian approach Chulani et al. (1999); Putnam considered mapping the input uncertainty in the context of his model Putnam et al. (1999).

6.2 Defect prediction

Defect prediction is the most problematic part in risk analysis. There are methods for predicting a number of defects and some pioneer works regarding estimating the effort of the removal process. However, there is no technique of combining these two methods. Furthermore, there are issues related to model training. Cross–project defect prediction has low success rate and intra–project prediction always requires extra amount of work for collecting data that is necessary to train the model.

Full automation of defect prediction requires further research. Currently, there are methods for evaluating the software quality using defect related measures, e.g. the Bayesian belief nets approach or the reliability growth models (both described in Section 4). However, it is not clear how such measures are correlated with the defect fixing effort and hence further empirical investigation regarding incorporating these measures into risk analysis is necessary.

6.3 Measuring maintainability

There are a number of maintainability measuring methods. We recommend the Practical model for measuring maintainability suggested by Heitlager et al. Heitlager et al. (2007). This approach not only measures the maintainability, but also shows what should be done in order to improve it. The authors put the model in use in a number of projects and obtained satisfactory results. Nevertheless, no formal validation was conducted and hence it is hard

to make judgments about the external validity. There is also a possibility that automation related issues will arise, since the authors omitted detailed description of some of the model dimensions. It is also not clear how to map the model output to project risk. Therefore, further research regarding incorporating the model into the risk analysis is necessary.

6.4 Data sources

Further research requires empirical data collected from real software projects. The collection process requires tools that are well adjusted to the software development environment. Such tools can be also beneficial during the risk analysis, since usually they can be used to monitor the projects and hence to assess its current state. A number of tools were listed in the Section 3. We are also working on our own solutions (Ckjm Jureczko & Spinellis (Retrived on 06/05/2011), and BugInfo Jureczko (Retrived on 01/11/2011a)). It is possible to collect data from open–source projects with such tools. Limited number of projects track the effort related data, but fortunately, there are exceptions e.g. JBoss and some of the Codehaus projects Codehaus (Retrived on 01/11/2011). There are also data repositories that can be easily employed in research. That includes public domain datasets: Promise Data Repository Boetticher et al. (Retrived on 01/11/2011), Helix Vasa et al. (Retrived on 01/11/2011), and our own Metrics Repository Jureczko (Retrived on 01/11/2011b); as well as commercial: ISBSG ISBSG (Retrived on 01/11/2011), NASA's Metrics Data Program NASA (Retrived on 01/05/2006), SLIM, SPR KnowledgePLAN, or PRICE-S.

7. Summary and future works

Method of software project risk analysis was suggested and decomposed into well known in software engineering problems: effort estimation, defect prediction and maintainability measurement. Each of these problems was investigated by describing the most common approaches, discussing the automation possibilities and recommending suitable techniques.

The potential problems regarding automation were identified. Many of them can be addressed using the available tools (see Section 3). Unfortunately, some of them are not covered and require further research or new software solutions.

Validation in the scope of single requirement is necessary in the case of effort estimation. The tools are adjusted to the scope of the whole project and hence the quality of estimations for single requirement should be proven. Such validation requires great amount of empirical data. Potential data sources were given in Section 6.4. We are developing a tool (extension of our earlier solutions: BugInfo and Ckjm; working name: Quality Spy) that will be able to collect the data and once we have finished we are going to conduct an empirical study regarding the validation.

The discussion regarding defect prediction was not conclusive due to a number of major issues related to incorporating the prediction methods into the risk analysis. It is not clear how to combine the predicted number of defects with the expected fixing effort. Furthermore, the studies regarding prediction of fixing effort have low external validity and the cross-project defect prediction has low success rate. We are going to conduct further studies regarding estimating effort of defect removal and formalizing the method of using results of defect prediction in the risk analysis by providing a mapping to the loss function.

A recommendation regarding measuring maintainability was made (the model suggested in Heitlager et al. (2007)). Unfortunately, we have no empirical data to support it and therefore, we are going to implement the model in a tool and validate it against open–source projects. We are expecting that the validation will help to solve the second maintainability related issue, namely, mapping the Heitlager's model output to values of the loss function.

8. Acknowledgement

Thanks to Z. Huzar, M. Kowalski, L. Madeyski and J. Magott for their great contribution to the design of the risk analysis method presented in this paper.

9. References

Abdel-Hamid, T. & Madnick, S. E. (1991). *Software project dynamics: an integrated approach*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Air Force Cost Analysis Agency (Retrived on 01/11/2011). AFCAA REVIC. URL: *https://sites.google.com/site/revic92*

Albrecht, A. J. (1979). Measuring Application Development Productivity, *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, Monterey, California, USA, pp. 83–92.

Atkinson, R. (1999). Project management: cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria, *International Journal of Project Management* 17(6): 337 – 342.

Bagnall, A. J., Rayward-Smith, V. J. & Whittley, I. M. (2001). The next release problem, *Information and Software Technology* 43(14): 883 – 890.

Banker, R. D., Kauffman, R. J. & Kumar, R. (1991). An empirical test of object-based output measurement metrics in a computer aided software engineering (case) environment, *J. Manage. Inf. Syst.* 8: 127–150.

Basili, V. R. & Perricone, B. T. (1984). Software errors and complexity: an empirical investigation0, *Commun. ACM* 27: 42–52.

Beck, K. & Andres, C. (2005). *Extreme Programming Explained: Embrace Change; 2nd ed.*, Addison-Wesley, Boston, MA.

Boehm, B., Abts, C. & Chulani, S. (2000). Software development cost estimation approaches – a survey, *Ann. Softw. Eng.* 10: 177–205.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R. & Selby, R. (1995). Cost models for future software life cycle processes: Cocomo 2.0, *Annals of Software Engineering*, pp. 57–94.

Boehm, B. W. (1981). *Software Engineering Economics*, Prentice Hall.

Boetticher, G., Menzies, T. & Ostrand, T. (Retrived on 01/11/2011). Promise repository of empirical software engineering data. URL: *http://promisedata.org*

Borland Software Corporation (Retrived on 01/11/2011). CaliberRM ESTIMATE Professional. URL: *http://www.ktgcorp.com/borland/products/caliber/index.html#estimate_pro*

Cadle, J. & Yeates, D. (2007). *Project Management for Information Systems*, 5th edn, Prentice Hall Press, Upper Saddle River, NJ, USA.

Capgemini CSD Research Inc. (Retrived on 01/11/2011). Software Cockpit.
URL: *http://www.de.capgemini.com/capgemini/forschung/research/software/*

Catal, C. (2011). Review: Software fault prediction: A literature review and current trends, *Expert Syst. Appl.* 38: 4626–4636.

Center for Systems and Software Engineering (Retrived on 01/11/2011). COCOMO II.
URL: *http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html*

Chulani, S., Boehm, B. & Steece, B. (1999). Bayesian analysis of empirical software engineering cost models, *Software Engineering, IEEE Transactions on* 25(4): 573 –583.

Codehaus (Retrived on 01/11/2011). Codehaus.
URL: *http://jira.codehaus.org*

Coleman, D., Ash, D., Lowther, B. & Oman, P. (1994). Using metrics to evaluate software system maintainability, *Computer* 27: 44–49.

Common Software Measurement International Consortium (Retrived on 01/11/2009). Cosmic.
URL: *http://www.cosmicon.com*

Construx (Retrived on 01/11/2011). Estimate.
URL: *http://www.construx.com*

Cost Xpert AG (Retrived on 01/11/2011). Cost xpert.
URL: *http://www.costxpert.eu/en/research/COCOMOII/cocomo_comsoftware.htm*

Fenton, N. E. & Neil, M. (1999a). Software metrics and risk, *Proceedings of the 2nd European Software Measurement Conference*, FESMA '99, pp. 39–55.

Fenton, N. E. & Neil, M. (2000). Software metrics: roadmap, *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, ACM, New York, NY, USA, pp. 357–370.

Fenton, N. & Neil, M. (1999b). A critique of software defect prediction models, *Software Engineering, IEEE Transactions on* 25(5): 675 –689.

Fenton, N., Neil, M., Marsh, W., Hearty, P., Radlinski, L. & Krause, P. (2007). Project data incorporating qualitative factors for improved software defect prediction, *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, IEEE Computer Society, Washington, DC, USA.

Fenton, N., Neil, M., Marsh, W., Hearty, P., Radliński, L. & Krause, P. (2008). On the effectiveness of early life cycle defect prediction with bayesian nets, *Emp. Softw. Engg.* 13: 499–537.

Gaffney, J. E. (1984). Estimating the number of faults in code, *Software Engineering, IEEE Transactions on* SE-10(4): 459 –464.

Galorath Inc. (Retrived on 01/11/2011). SEER for Software.
URL: *http://www.galorath.com/*

Giger, E., Pinzger, M. & Gall, H. (2010). Predicting the fix time of bugs, *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, ACM, New York, NY, USA, pp. 52–56.

Gokhale, S. S. & Mullen, R. E. (2010). A multiplicative model of software defect repair times, *Empirical Softw. Engg.* 15: 296–319.

Górski, J. & Miler, J. (Retrived on 01/11/2011). RiskGuide.
URL: *http://iag.pg.gda.pl/RiskGuide/*

Gupta, S., Sikka, G. & Verma, H. (2011). Recent methods for software effort estimation by analogy, *SIGSOFT Softw. Eng. Notes* 36: 1–5.

Hassouna, A. & Tahvildari, L. (2010). An effort prediction framework for software defect correction, *Information and Software Technology* 52(2): 197 – 209.

He, Z., Shu, F., Yang, Y., Li, M. & Wang, Q. (2011). An investigation on the feasibility of cross-project defect prediction, *Automated Software Engineering* pp. 1–33.

Heitlager, I., Kuipers, T. & Visser, J. (2007). A practical model for measuring maintainability, *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, IEEE Computer Society, Washington, DC, USA, pp. 30–39.

Hello2morrow Inc. (Retrived on 01/11/2011). Sotograph.
URL: *http://www.hello2morrow.com/products/sotograph*

Helming, J., Koegel, M. & Hodaie, Z. (2009). Towards automation of iteration planning, *Proc. of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, ACM, New York, NY, USA, pp. 965–972.

Humphrey, W. S. (1995). *A Discipline for Software Engineering*, 1st edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Idri, A., Khoshgoftaar, T. & Abran, A. (2002). Can neural networks be easily interpreted in software cost estimation?, *Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on*, Vol. 2, pp. 1162 –1167.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, *Technical report*, IEEE.

ISBSG (Retrived on 01/11/2011). The global and independent source of data and analysis for the it industry.
URL: *http://www.isbsg.org*

ISO (2001). ISO/IEC 9126-1:2001, Software engineering – Product quality – Part 1: Quality model, *Technical report*, International Organization for Standardization.

ISO (2004). ISO/IEC 16085:2001, systems and software engineering – life cycle processes – risk management, *Technical report*, International Organization for Standardization.

Jones, C. (Retrived on 01/11/2011). Programming languages table.
URL: *http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm*

Jorgensen, M. & Shepperd, M. (2007). A systematic review of software development cost estimation studies, *IEEE Trans. Softw. Eng.* 33: 33–53.

Jureczko, M. (Retrived on 01/11/2011a). BugInfo.
URL: *http://kenai.com/projects/buginfo*

Jureczko, M. (Retrived on 01/11/2011b). Metrics repository.
URL: *http://purl.org/MarianJureczko/MetricsRepo*

Jureczko, M. & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction, *PROMISE'2010: Proceedings of the 6th International Conference on Predictor Models in Software Engineering*, ACM.

Jureczko, M. & Młynarski, M. (2010). Automated acceptance testing tools for web applications using test-driven development, *Electrotechnical Review* 86(09): 198–202.

Jureczko, M. & Spinellis, D. (2010). *Using Object-Oriented Design Metrics to Predict Software Defects*, Vol. Models and Methodology of System Dependability of *Monographs of System Dependability*, Oficyna Wyd. Politechniki Wroclawskiej, Wroclaw, Poland, pp. 69–81.

Jureczko, M. & Spinellis, D. (Retrived on 06/05/2011). Ckjm extended.
URL: *http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/*

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B. & Hassan, A. (2010). Revisiting common bug prediction findings using effort-aware models, *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1 –10.

Kan, S. H. (1994). *Metrics and Models in Software Quality Engineering*, 1st edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kitchenham, B. (2010). What's up with software metrics? - a preliminary mapping study, *J. Syst. Softw.* 83: 37–51.

Kusumoto, S., Matukawa, F., Inoue, K., Hanabusa, S. & Maegawa, Y. (2004). Estimating effort by use case points: Method, tool and case study, *Proceedings of the Software Metrics, 10th International Symposium*, IEEE Computer Society, Washington, DC, USA, pp. 292–299.

Lipow, M. (1982). Number of faults per line of code, *IEEE Trans. Softw. Eng.* 8: 437–439.

Logical Solutions (Retrived on 01/11/2011). Project Code Meter.
URL: *http://www.projectcodemeter.com*

Madachy, R. J. (1996). System dynamics modeling of an inspection-based process, *Proceedings of the 18th international conference on Software engineering*, ICSE '96, IEEE Computer Society, Washington, DC, USA, pp. 376–386.

Madeyski, L. & Radyk, N. (2010). Judy – a mutation testing tool for java, *Software, IET* 4(1): 32 – 42.

Martenka, P. & Walter, B. (2010). Hierarchical model for evaluating software design quality, *e-Informatica* 4: 21–30.

Menzies, T., Chen, Z., Hihn, J. & Lum, K. (2006). Selecting best practices for effort estimation, *IEEE Trans. Softw. Eng.* 32: 883–895.

Menzies, T., Greenwald, J. & Frank, A. (2007). Data mining static code attributes to learn defect predictors, *IEEE Transactions on Software Engineering* 33: 2–13.

NASA (Retrived on 01/05/2006). Nasa's metrics data program.
URL: *http://mdp.ivv.nasa.gov/repository.html*

Netherlands Software Metrics Association (Retrived on 01/11/2011). NESMA Function Point Analysis.
URL: *http://www.nesma.nl/section/home/*

Ochodek, M. & Nawrocki, J. (2007). Automatic transactions identification in use cases, *Proceedings of the Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques*, Springer, pp. 33–46.

Oisen, R. P. (1971). Can project management be defined?, *Project Management Quarterly* 2(1): 12 – 14.

Park, H. & Baek, S. (2008). An empirical validation of a neural network model for software effort estimation, *Expert Syst. Appl.* 35: 929–937.

Penny, D. A. (2002). An estimation-based management framework for enhancive maintenance in commercial software products, *Software Maintenance, 2002. Proceedings. International Conference on*, pp. 122 – 130.

Pietrzak, B. & Walter, B. (2006). Leveraging code smell detection with inter-smell relations, Vol. 4044 of *Lecture Notes in Computer Science*, pp. 75–84.

PRICE Systems (Retrived on 01/11/2011). TruePlanning.
URL: *http://www.pricesystems.com/products/true_h_price_h.asp*

Putnam, L. H. (1978). A general empirical solution to the macro software sizing and estimating problem, *IEEE Trans. Softw. Eng.* 4: 345–361.

Putnam, L. H., Mah, M. & Myers, W. (1999). First, get the front end right, *Cutter IT Journal* 12(4): 20–28.

Quantitative Software Management Inc. (Retrived on 01/11/2011). SLIM Suite.
URL: *http://www.qsm.com/tools*

Quinlan, J. R. (1992). Learning With Continuous Classes, *Proceedings of the 5th Australian Joint Conf. Artificial Inteligence*, pp. 343–348.

Riaz, M., Mendes, E. & Tempero, E. (2009). A systematic review of software maintainability prediction and metrics, *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, IEEE Computer Society, Washington, DC, USA, pp. 367–377.

Ruiz, M., Ramos, I. & Toro, M. (2001). A simplified model of software project dynamics, *Journal of Systems and Software* 59(3): 299 – 309.

Schneider, G. & Winters, J. P. (2001). *Applying use cases, Second Edition*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Schwaber, K. & Beedle, M. (2001). *Agile Software Development with Scrum*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

SEI (2002). Capability Maturity Model Integration (CMMI), Version 1.1 - Continuous Representation, *Technical Report CMU/SEI-2002-TR-011*, Software Engineering Institute.

Shepperd, M. & Schofield, C. (1997). Estimating software project effort using analogies, *Software Engineering, IEEE Transactions on* 23(11): 736 –743.

Softstar Systems (Retrived on 01/11/2011). Costar.
URL: *http://www.softstarsystems.com/*

Software Productivity Research (Retrived on 01/11/2009). SPR KnowledgePLAN.
URL: *http://www.spr.com/project-estimation.html*

Song, Q., Shepperd, M., Cartwright, M. & Mair, C. (2006). Software defect association mining and defect correction effort prediction, *IEEE Trans. Softw. Eng.* 32: 69–82.

Stamelos, I. & Sfetsos, P. (2007). *Agile Software Development Quality Assurance*, IGI Publishing, Hershey, PA, USA.

Symons, C. R. (1991). *Software sizing and estimating: Mk II FPA (Function Point Analysis)*, John Wiley & Sons, Inc., New York, NY, USA.

Thaw, T., Aung, M. P., Wah, N. L., Nyein, S. S., Phyo, Z. L. & Htun, K. Z. (2010). Comparison for the accuracy of defect fix effort estimation, *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, Vol. 3, pp. 550 – 554.

TSA Quality (Retrived on 01/11/2011). Metric Studio 2011.
URL: *http://www.tsaquality.com/pages/english/nav/produtos.html*

Valerdi, R., Boehm, B. & Reifer, D. (2003). Cosysmo: A constructive systems engineering cost model coming age, *Proceedings of the 13th Annual International INCOSE Symposium*, pp. 70–82.

Vasa, R., Lumpe, M. & Jones, A. (Retrived on 01/11/2011). Helix - Software Evolution Data Set.
URL: *http://www.ict.swin.edu.au/research/projects/helix*

Weiss, C., Premraj, R., Zimmermann, T. & Zeller, A. (2007). How long will it take to fix this bug?, *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, IEEE Computer Society, Washington, DC, USA, pp. 1–.

Weyuker, E. J., Ostrand, T. J. & Bell, R. M. (2008). Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models, *Empirical Softw. Engg.* 13: 539–559.

Zeng, H. & Rine, D. (2004). Estimation of software defects fix effort using neural networks, *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, Vol. 2, pp. 20 – 21.

Zimmermann, T. & Nagappan, N. (2009). Predicting defects with program dependencies, *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, IEEE Computer Society, Washington, DC, USA, pp. 435–438.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E. & Murphy, B. (2009). Cross-project defect prediction: a large scale experiment on data vs. domain vs. process, *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, ACM, New York, NY, USA, pp. 91–100.