Integrating Manual and Automatic Risk Assessment for Risk-Based Testing

Michael Felderer[1], Christian Haisjackl[1], Ruth Breu[1], Johannes Motz[2]

[1] Institute of Computer Science, University of Innsbruck, Austria
{michael.felderer,christian.haisjackl,ruth.breu}@uibk.ac.at
[2] Kapsch CarrierCom AG, Vienna, Austria
johannes.motz@kapsch.net

Abstract. In this paper we define a model-based risk assessment procedure that integrates automatic risk assessment by static analysis, semi-automatic risk assessment and guided manual risk assessment. In this process probability and impact criteria are determined by metrics which are combined to estimate the risk of specific system development artifacts. The risk values are propagated to the assigned test cases providing a prioritization of test cases. This supports to optimize the allocation of limited testing time and budget in a risk-based testing methodology. Therefore, we embed our risk assessment process into a generic risk-based testing methodology. The calculation of probability and impact metrics is based on system and requirements artifacts which are formalized as model elements. Additional time metrics consider the temporal development of the system under test and take for instance the bug and version history of the system into account. The risk assessment procedure integrates several stakeholders and is explained by a running example.

1 Introduction

In many application domains, testing has to be done under severe pressure due to limited resources with the consequence that only a subset of all relevant test cases can be executed. In this context, risk-based testing approaches are more and more applied to prioritize test cases based on risks. A risk is the chance of injury, damage or loss and typically determined by the probability of its occurrence and its impact [1]. A core activity in every risk-based testing process is the risk assessment because it determines the significance of the risk values assigned to tests and therefore the quality of the overall risk-based testing process.

In current practice, mainly human experts conduct a risk assessment. Therefore the process of risk assessment is expensive, time consuming, and contains non-determinism regarding subjective human decisions. This makes the re-assessment of risks during the testing process a challenging task. To avoid risk assessment to become a critical cost driver in software development projects, we propose an approach to automate parts of the risk assessment. The basis for this partial automation is the fragmentation of its probability and impact into several criteria. Each criterion in the resulting set of probability and impact criteria has an attached metrics which is computed automatically, semi-automatically or manually.

To aggregate all probability and impact criteria to one risk value and to assign this value to tests we attach the risk to so called features. Features specify capabilities of a component and link tests, requirements, and system components together. Risk coefficients are computed for features and propagated to the assigned tests. The probability and impact criteria underlying the risk coefficient are determined based on the model elements assigned to the feature.

Differing from other approaches we additionally consider separate time criteria such as the version history or the test history depending on the temporal development of the project. Time criteria can be computed automatically and are considered as ratios that reduce the probability.

The risk assessment should start early in a software project to provide decision support. Due to changes of the requirements or the design, risks have to be reassessed regularly. Additionally not all artifacts are available in all phases of a project and therefore it may not be possible to automatically compute specific metrics at a specific point in time. Therefore the evaluation type and the evaluation procedure of specific metrics may change during the life cycle of a project. For example a metrics for complexity may be estimated manually at the beginning of a project and automatically in later project phases.

In our approach we focus on product risks where the primary effect of a potential problem is on the quality of the product itself. Since testing is about finding problems in products, risk-based testing is about product risks [2]. Thus, we do not consider project risks related to management and control of the test project like the capabilities of the staff or strict deadlines. The focus on product risks is additionally motivated by the fact that product risks directly influence the test plan and that product risks are typically more objective because they do not consider the capabilities of the project staff. If desired by the software project management, project risks can be considered in our approach as additional criteria [3].

In this paper we contribute to the integration of risk assessment into risk-based testing in several ways. First we introduce a generic risk-based testing process that highlights risk assessment as a core activity. We then define a static risk assessment model that considers a probability, impact, and time factor which have not been considered separately in other approaches. Moreover, each factor is determined by several criteria. Finally, we define a risk assessment procedure based on the risk assessment model and give an example. In the risk assessment procedure each factor is calculated by an aggregation function based on several criteria. Each criterion is determined by a metrics that is evaluated manually, semi-automatically or automatically and that can change over time.

This paper is structured as follows. In Section 2, we present related work, and in Section 3 we define a generic risk-based testing process. In Section 4, we then define the underlying risk assessment model, and in Section 5 we discuss our risk assessment procedure. Finally, in Section 6, we draw conclusions.

2 Related Work

The importance of risk management for software engineering [4] has been addressed in several risk management approaches tailored to software engineering processes. Already the spiral model of Boehm [5] explicitly includes risk management within software development. For instance, the Riskit method [6] provides a sound theoretical foundation of risk management with a focus on the qualitative understanding of risks before their possible quantification. Furthermore, the Riskit method provides a defined process for conducting risk management which is supported by various techniques and guidelines. Ropponen [3] highlights the importance of mature project management for risk management.

Karolak [7] proposes a risk management process for software engineering that contains the activities risk identification, risk strategy, risk assessment, risk mitigation, and risk prediction.

Risk-based testing is a test-based approach to risk management. Amland [8] proposes a risk-based testing approach that is based on Karolak's risk management process comprising the following steps and the corresponding risk management activities: planning (risk identification and risk strategy), identification of risk indicators (part of risk assessment), identification of the cost of a failure (part of risk assessment), identification of critical elements (part of risk assessment), test execution (risk mitigation), and estimation of completion (risk reporting and risk prediction).

Bach [9] presents a pragmatic approach to risk-based testing grounded on a heuristic software risk analysis. Bach distinguishes inside-out risk analysis starting with details about a situation and identifying associated risk, and outside-in risk analysis starting with a set of potential risks and matching them to the details of the situation.

Taxonomy–based risk identification, e.g., as proposed by the SEI [10], supports the risk analysis.

Risk-based testing techniques have rarely been applied on the model level. RiteDAP [11] is a model-based approach to risk-based system testing that uses annotated UML activity diagrams for test case generation and prioritization. This approach does not consider how the risk values are determined and only considers the risk-based generation of test cases. But risk-based testing techniques have so far not been integrated into a model-driven system and test evolution process, and have not been applied for model-based regression testing.

Stallbaum and Metzger [12] introduce a model-driven risk-based testing approach similar to our methodology. But the approach of Stallbaum and Metzger does not consider the system model and time criteria.

The basis for risk-based testing on the model level is model-driven risk analysis. The CORAS method [13] addresses this task and defines a metamodel for risk assessment. But CORAS is a defensive approach restricted to assets that already exist. Additionally, risk-based testing based on CORAS has not been conducted so far. There are several other model-driven risk analysis approaches like fault trees [14], attack trees [15], misuse cases [16] or Tropos Goal-Risk modeling [17] that are useful for risk-based testing.

McCall [18] distinguishes factors which describe the external view of the software, as viewed by the users, criteria which describe the internal view of the software as seen by the developers and metrics which provide a scale and a method for measurement.

Many researchers have addressed the problem of risk assessment using guide lines, checklists, heuristics, taxonomies, and risk criteria [19]. All these risk assessment approaches strongly rely on experts that perform the assessment. Most approaches that enable automation in risk assessment employ metrics based on code artifacts.

There are several approaches, e.g., [20,21] that employ change metrics like the change rate, code metrics like the number of code lines and complexity metrics like the McCabe complexity [22] to predict the failure rate. Experiments indicate that the combination of code and design metrics has a higher performance for predicting the error-proneness of modules than just code or design metrics [23]. These approaches are not directly related to risk-based testing as our approach but they form the basis for determining the probability of risks.

For instance, Nagappan et al. [20] investigate the correlation between classical function, class, and module metrics and failures for selected projects. The authors state that for each project, a set of metrics that correlates with the failure rate exists but that there is no single set of metrics that fits for all projects. This result is very useful for the assessment of the probability of a risk in our approach as it argues for a mix of manual and automatic risk determination.

Illes-Seifert and Paech [21] explore the relationship of a file's history and its fault-proneness, and show that a software's history is a good indicator for its quality. There is not one indicator for quality but the number of changes, the number of distinct authors, as well as the the file's age are good indicators for a file's defect count in all projects. Illes-Seifert and Paech do not consider risk assessment. But the results are a good basis for the determination of time criteria in our approach.

Additionally, also vulnerability databases, e.g., the national vulnerability database [24] or the open source vulnerability database [25] are powerful sources for risk estimation [26].

Similar to our approach the Common Vulnerability Scoring Standard [27] uses a variety of metrics to calculate a risk coefficient. The metrics are organized in three different groups: one basic and two optional. Although the approach focuses on security risks, its flexibility can also be applied to our approach for integrating optional metrics.

3 Generic Risk-based Testing Process

Risk-based Testing (RBT) is a type of software testing that considers risks assigned to test items for designing, evaluating, and analyzing tests [8, 9]. A risk is the chance of injury, damage or loss. A risk is therefore something that might happen and has a potential loss or impact associated with that event.

Typically, in risk-based testing the standard risk model is based on the two factors *probability* (P), determining the likelihood that a fault assigned to a risk occurs, and *impact* (consequence or cost) (I), determining the cost of the assigned fault if it occurs in production.

The probability typically considers technical criteria, e.g., complexity, and the impact considers business criteria such as monetary loss or reputation. These criteria can be determined by automatically, semi-automatically or manually computed metrics. Our risk computation model follows the approach of McCall [18] who suggests using a divide-and-conquer strategy involving a hierarchy of three types of characteristics: factors, criteria, and metrics. On a high level, the factors describe the external view of the software, as viewed by the users. Then, the criteria depict the internal view of the software, as seen by the developer. And the metrics are defined to provide a scale and method for measurement.

Mathematically, the risk (coefficient) R of an artifact a can be expressed by the probability P and the impact I as follows:

$$R(a) = P(a) \circ I(a)$$

The depicted operation $\circ$ is typically a multiplication of numbers or the cross product of two numbers or letters, but can principally be an arbitrary mathematical operation.

Risk-based testing is integrated into a testing process as shown in Fig. 1. Such a process integrates elements of risk management and test management. In most cases, risk-based testing is limited to risk prioritization and its full potential to improve software quality by mitigating risk and optimizing test resource allocation is not exploited. To improve this situation, we define a comprehensive risk-based testing methodology that is aligned with established risk-based testing processes such as Amland [8] or the ISTQB [28]. Our generic risk-based testing process therefore contains activities for the identification of risks, the test planning, the risk assessment, the test design and the evaluation.

In Fig. 1 the activities are depicted as ellipses, and the artifacts as input and output of the activities are depicted as rectangles. This paper focuses on the core activity of Risk Assessment which is therefore highlighted in Fig. 1.

Our generic risk-based testing process depicted in Fig. 1 consists of the phases *Risk Identification*, *Test Planning*, *Risk Assessment*, *Test Design*, and *Static and Dynamic Evaluation* explained in the subsequent paragraphs.

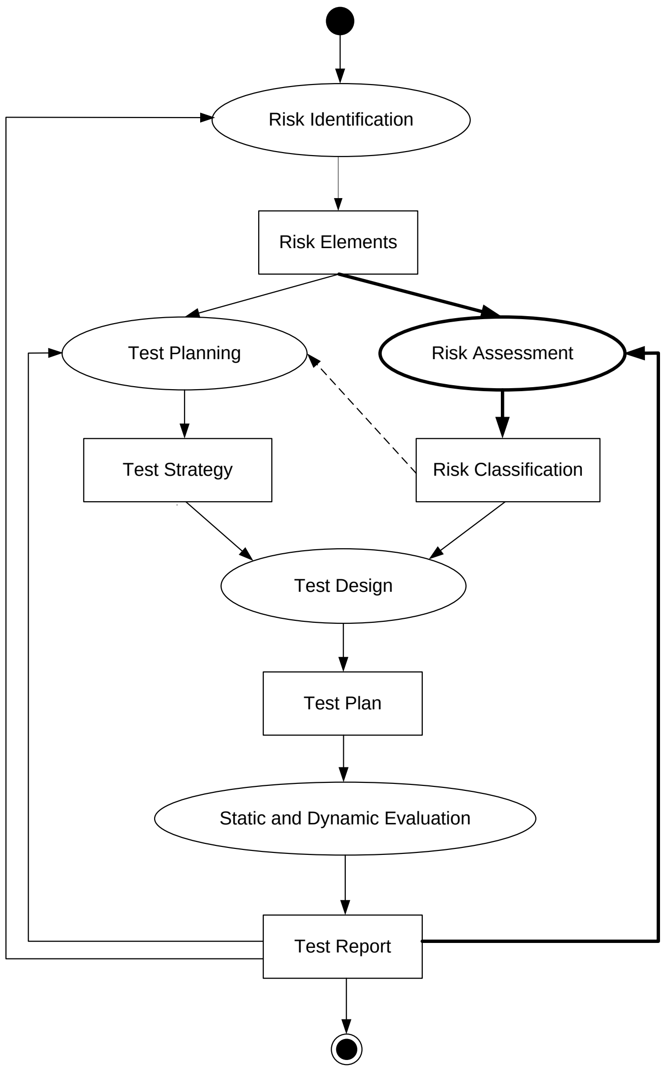

Fig. 1. Generic Risk-based Testing Process

In the **Risk Identification** phase risk items are identified and stored in a list of risk items. Risk items are the elements under test for which the risk is calculated. Therefore risk items need to be concrete such that a risk calculation and an assignment of tests are possible. Risk items can be development artifacts, e.g., requirements or components but also different types of risks such as product, project, strategic or external risks. According to practical experiences [29], the maximum number of risk items should be limited.

In the **Test Planning** phase the test strategy is defined. The test strategy contains test prioritization criteria, test methods, exit criteria and the test effort under consideration of risk aspects. Although our approach is independent of a concrete test strategy, we assume that a given test strategy contains a basic risk classification scheme as depicted in Fig. 4. Such a classification scheme demonstrates how the calculated risk values can be further employed in our risk-based testing methodology.

In the **Risk Assessment** phase the risk coefficient is calculated for each risk item based on probability and impact factors. The probability P and the impact I are typically evaluated by several weighted criteria and can be computed as follows for a specific artifact a:

$$P(a) = \frac{\sum_{j=0}^{m} p_j \cdot w_j}{\sum_{j=0}^{m} w_j} \quad , \quad I(a) = \frac{\sum_{j=0}^{n} i_j \cdot w_j}{\sum_{j=0}^{n} w_j}$$

, where p_j are values for probability criteria, i_j are values for impact criteria, and w_j are weight values for the criteria. The range of p, i, and w are typically natural or real numbers in a specific range, e.g., for p and i we suggest natural numbers between 0 and 9 (so the probability can be interpreted as percentage; we skipped the value 10 (suggesting 100%) because we assume that a component does not fail for sure), and for w we suggest real numbers between 0 and 1 (so the weight can be naturally interpreted as scaling factor).

The values of the metrics are computed automatically, semi-automatically or manually based on an evaluation procedure, and the weights are set manually. The responsibility for setting non–automatically derived values respective checking the automatically derived values should be distributed among different stakeholders, e.g., project manager, software developer, software architect, test manager, and customer.

The impact I is computed by analogy to the probability and typically evaluated manually by customers or product managers. Without loss of generality, we use the same range for impact values as for probability values. In our adapted risk assessment approach we additionally consider time criteria which take metrics into account that change over time because of the products life cycle.

A time factor is a ratio that reduces the probability value which is reflected in our adapted formula for determining the risk coefficient R of an artifact a:

$$R(a) = (P(a) \cdot T(a)) \circ I(a)$$

, where $T(a)$ is the time factor that is multiplied with the probability $P(a)$ and has a range between 0 and 1. The resulting product is then combined with the impact $I(a)$.

The time factor is computed by the mean of the values of k time criteria by the following formula:

$$T(a) = \frac{\sum_{j=0}^{k} t_j}{k}$$

, where t_j are values for time criteria. We assume that over time, stable parts of the software have a lower probability to fail, so we scale the probability in this way.

Based on the impact and probability factor, the classification of risk items into risk levels is done. A typical way to do so is the definition of a risk matrix as shown in Fig. 4. The risk classification is considered in the test planning phase. An example for the risk assessment procedure and the risk classification is presented in Section 5.

In the **Test Design** phase a concrete test plan is defined based on the test strategy and the risk classification. The test plan contains a list of test cases that has been designed under consideration of risk aspects defined in the test strategy, the risk classification and the given resources.

In the **Static and Dynamic Evaluation** phase the test plan is executed manually and/or automatically. Based on the execution result a test report is generated. The evaluation phase may contain dynamic evaluation, i.e., test execution and static evaluation, e.g., inspections of documents that are assigned to risk items that have been classified as risky.

The phases of risk identification, risk assessment and evaluation are together called the risk analysis phase. As shown in Fig. 1, the process of risk-based testing is iterative to consider changes of the risk items and previous evaluation results in the risk analysis.

If the RBT process defined above is integrated with an existing test and software development process, then the benefits of RBT may be as follows:

- Reduced resource consumption (i.e., more efficient testing) by saving time and money
- Mitigation of risks
- Improved quality by spending more time on critical functions and identifying them as early as possible
- Early identification of test issues, which can be started when the requirements specification is completed
- Support for the optimization of the software testing process by moving the test phase from the latter part of the project to the start and allows the concept of life cycle testing to be implemented
- Identifying critical areas of the application in the requirement stage. This assists in the test process but also in the development process because it supports the optimization of resources, e.g., time, budget, or person days.

4 Risk Assessment Model

In this section we present the risk assessment model which is the basis for the risk assessment procedure defined in the next section. The risk assessment model is shown in Fig. 2.

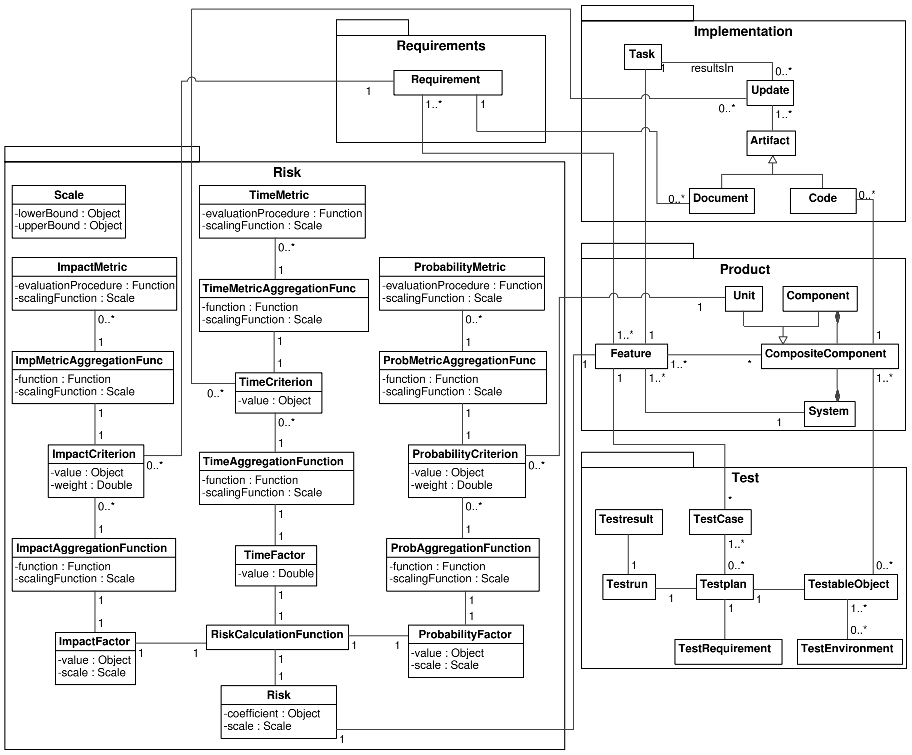

Fig. 2. Model Elements for Risk Assessment

The model contains the package *Risk* which includes the core risk assessment elements and the packages *Requirements*, *Product*, *Implementation*, and *Test* which include requirements, system architecture, implementation, plus test artifacts relevant for risk-based testing. In the following subsections we explain each package in detail. The modeling of the package *Risk* is more granular than the others because it contains the core artifacts of the risk assessment procedure.

4.1 Requirements

The package *Requirements* contains the requirements each defining a certain functional or non-functional property of the system. A *Requirement* is implemented by a set of *Feature* elements and is related to a set of *Document* elements.

4.2 Product

The package *Product* contains the core elements of the system. A *Feature* describes one specific capability of a system or component. It is the central concept to plan and control the implementation. *Feature*s are the core model elements which link all packages together. A *Feature* validates *Requirement*s and is verified by *TestCase*s. Therefore *Feature*s are the natural elements for the assignment of *Risk*s. Each *Feature* is implemented by several *Task*s and assigned to a *System* and several *CompositeComponent*s.

A *System* consists of several *CompositeComponent*s which are either *Component*s or *Unit*s. A *CompositeComponent* is an installable artifact that provides the functionality of several *Feature*s and is defined in a hierarchy of *Component*s and *Unit*s which are the leaves of the hierarchy. Because all modern software products are structured in a hierarchical way, the proposed structure can be applied for most software products. For instance, the components of a software architecture in the modeling language UML or the package structure in the programming language Java are hierarchical and form a hierarchy of *Component*s and *Unit*s.

4.3 Implementation

The package *Implementation* contains the concrete implementation artifacts *Code* and *Document*. *Code* represents implemented or generated source code, and *Document* represents a specification or documentation file, e.g., text files with user requirements or UML diagrams. If an *Artifact* is changed, an *Update* is created containing the new version. A *Task* groups several *Update*s and addresses the implementation or adaptation of a *Feature*.

4.4 Test

Several *TestCase*s form a *Testplan* which has assigned *TestRequirement* elements defining constraints on the test plan such as coverage criteria. The execution of a *Testplan* is a *Testrun* which has several *Testresult* elements assigned. A *Testplan* is executed on a *TestableObject* which is embedded in a *TestEnvironment* and related to a *CompositeComponent*.

4.5 Risk

The *Risk* is computed as defined in Section 3 based on the *ImpactFactor*, the *TimeFactor* and the *ProbabilityFactor* which are combined by a *RiskCalculationFunction*. The *Risk*, *ProbabilityFactor* and *ImpactFactor* have a value and a scale. The *ImpactFactor* is computed by an *ImpactAggregationFunction* based on *ImpactCriteria*, the *ProbabilityFactor* by a *ProbabilityAggregationFunction* based on *ProbabilityCriteria*, and the *TimeFactor* by a *TimeAggregationFunction* based on *TimeCriteria*. In this paper we apply the function mean as aggregation function.

Each *ProbabilityCriterion*, *TimeCriterion* and *ImpactCriterion* has a metric aggregation function of type *ProbabilityMetricAggregationFunction*, *TimeMetricAggregationFunction* or *ImpactMetricAggregationFunction*. The metric aggregation function contains all information to calculate the values of the criteria. Each aggregation function is based on metrics, i.e., *ImpactMetric*, *TimeMetric*, or *ProbabilityMetric*. Each metric has an evaluation procedure which can be of type manual, semi-automatic, or automatic, and a scaling function to normalize values. To make this paper more readable, we often identify metrics and the computed values.

The *Risk*, *ImpactFactor*, and *ProbabilityFactor* have a value and a scale for that value (for the *Risk* the value is called the coefficient). Each *ImpactCriterion* and *ProbabilityCriterion* has an additional weight value defining a ratio that a specific criterion has for the overall risk computation. The *TimeFactor* has in contrast to the other two factors only a value and no scale, because we assume that the time factor is a rational number between 0 and 1. Furthermore, the *TimeCriteria* do not have an additional weight because they are per definition reduction factors for probability criteria.

For evaluating the risk coefficient efficiently, the different factors are assigned to specific artifacts in our model.

- **Feature:** A *Feature* describes a specific functionality of the whole system. The test cases are attached to features. Therefore, the risk coefficient is assigned to this artifact.
- **Unit:** A *Unit* is the technical implementation of a *Feature*. Failures evolve from *Unit*s and the probability that failures occur influences the value of the probability factor and therefore also the risk coefficient. Thus, the *ProbabilityCriteria* are assigned to the *Unit*s.
- **Requirements:** The *Requirement*s represent the view of the customer on a system. If a *Feature* fails, a *Requirement* cannot be fulfilled and damage (monetary or ideally) is caused. Thus, the *ImpactCriteria* are assigned to the *Requirement*s.
- **Components:** The *Component*s consist of several *Units* and form subsystems, where the same directives can be applied. Thus, we attached the weights to the *Components*, so that the same weights can be applied to the associated Units.

Each *Feature* belongs to one *Component*, implements one or more *Requirement*s and uses one or more *Unit*s. Exactly these relationships are used for calculating the risk coefficient. The aggregated values, e.g., by the aggregation function mean (which we consider as aggregation function in the remainder of this document), of the *ImpactCriteria* from the *Requirement*s and the mean values of the single *ProbabilityCriteria* of the *Unit*s are weighted by the *Component*'s weights. Afterwards, both resulting factors are set into relation (by multiplication, cross product, etc.) for calculating the final *Risk*. The main principle behind our evaluation procedure is the separation of the evaluation of the impact and the probability factor.

The evaluation of the probability factor is based on internal system artifacts, i.e., units in our respect, and the evaluation of the impact is based on external requirements artifacts, i.e., user requirements in our respect. Therefore our risk assessment model can be adapted to other development and testing processes because most practical processes have a separation between internal and external artifacts.

As mentioned above, the *ProbabilityFactor* and *ImpactFactor* are based on several criteria, e.g., code complexity, functional complexity, concurrency, or testability for *ProbabilityCriteria*, and, e.g., availability, usage, importance, image loss, or monetary loss for the *ImpactCriteria*. Each of these criteria has its own weight at the level of *Component*s. If a *Feature* has more *Unit*s or *Requirement*s, the mean value of a specific criterion is calculated afterwards multiplied by the *Component*'s weight. As next step, the mean of these values is generated as final factor for the risk coefficient calculation.

So far we have only considered probability and impact factors but we have not considered their dynamics and the temporal development of *Risk* values. The *Risk* changes only if a *Requirement* is redefined or a *Unit* is adapted. As a *Feature* has a life cycle, the same holds for a *Risk*. But so far we have only considered criteria that are determined statically. To address this problem, we introduced an additional factor, called *TimeFactor*. The criteria of the *TimeFactor* are adapted dynamically and are, e.g., change history, bug reports, test history or software maturity, which change its value during the life cycle of a *Feature*. We have integrated the time factor as a scaling factor between 0 and 1, that adapts the *ProbabilityFactor*. Initially, the *TimeFactor* has the value 1 which means, that the probability is not lowered. If a specific *Feature* turns out to be more stable than other *Feature*s, some *TimeCritera* will be lowered and reduce the probability of that *Feature*.

In the remainder of this document, we often use the singular or plural form of the name of a metamodel element to refer to instances of it.

5 Risk Assessment Procedure

In this section we explain the algorithm for determining the risk coefficient by an abstract example. The example refers to the activity *Risk Assessment* of the generic risk-based testing process shown in Figure 1. The risk elements are *Feature* elements and the resulting risk classification is based on the computed risk coefficients. As the focus of this paper is the risk assessment procedure itself, we do not explicitly consider adequate test reporting techniques.

The example of this paper is based on three *ProbabilityCriteria*, i.e., code metrics factor, functional complexity and testability, two *ImpactCriteria*, i.e., usage and importance, and two *TimeCriteria*, i.e., change history and test history. Without loss of generality, the values of probability and impact criteria have an integer range between 0 and 9. The values of time criteria have a real number between 0 and 1. Additionally, each criterion except the time criteria have a weight. We consider the weight as a real number between 0 to 1. As aggregation function we apply the function mean. All values are rounded to one decimal place. Our model (see Fig. 3) has two components (C1, C2), whereby each component has one feature (C1 is linked to F1, C2 is linked to F2). Furthermore, we have two Units (U1, U2) and three Requirements (R1, R2, R3). F1 uses U1 and U2 and implements R1 and R2. F2 uses U2 and implements R3.

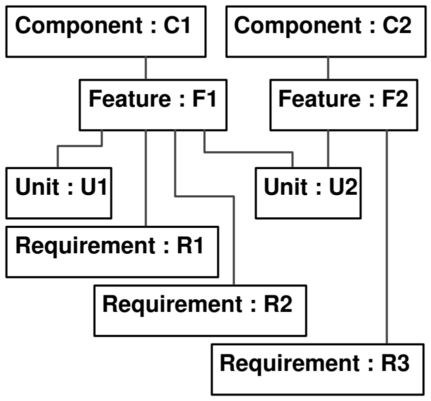

Fig. 3. Example Model for risk assessment

In the risk assessment phase, first the *ProbabilityCriteria* for the Units U1 and U2 are set. We assume, that the project is in the planning phase and therefore the code complexity is estimated manually, e.g., by software developer or architects because there is no source code available for the automatic determination of code complexity. The functional complexity is determined by a function point analysis. The testability is also estimated manually. In our example, we assume that a low code complexity implies a low value for testability. The following table shows the estimated values for the probability criteria of the units U1 and U2 in our example:

Unit	**ProbabilityCriterion**	$\mathbf{p_j}$
U1	Code Complexity	8
U1	Functional Complexity	9
U1	Testability	7
U2	Code Complexity	3
U2	Functional Complexity	2
U2	Testability	2

In the next step, the *ImpactCriteria* of the three requirements R1, R2, and R3 are determined manually, e.g., by the responsible product manager or the customer based on the requirements documents. The following table shows the estimated values for the impact criteria of the requirements R1, R2, and R3 in our example:

Requirement	**ImpactCriterion**	$\mathbf{i_j}$
R1	Importance	9
R1	Usage	9
R2	Importance	4
R2	Usage	3
R3	Importance	8
R3	Usage	9

Additionally, for the determination of the factors the weights attached to the components are needed for the computation of risk values. The weight values are determined manually for the components C1, C2, and C3, e.g., by the project or test manager, and are as follows in our example:

Component	**Criterion**	$\mathbf{w_j}$
C1	Code Complexity	0.4
C1	Functional Complexity	0.7
C1	Testability	0.6
C1	Importance	0.8
C1	Usage	0.5
C2	Code Complexity	0.9
C2	Functional Complexity	0.3
C2	Testability	0.4
C2	Importance	0.7
C2	Usage	0.7

In the last step, the time criteria of the features F1 and F2 have to be set to determine the time factors. Because the example project is in the planning phase, the *TimeCriteria* are set to the initial maximal value of 1, because the change and test history are initially 1 and do not reduce the probability. Thus, the time criteria for the features F1 and F2 are as follows:

Feature	**TimeCriterion**	$\mathbf{t_j}$
F1	Change History	1.0
F1	Test History	1.0
F2	Change History	1.0
F2	Test History	1.0

After the values for criteria and weights have been estimated, the values for the factors are calculated automatically. For feature F1 which is related to the units U1 and U2 (with the corresponding probability factors P1 and P2), and to the requirements R1 and R2 (with the corresponding impact factors I1 and I2), probability, impact and time factors are as follows:

ProbabilityFactor	**Formula**	**P(a)**
P1	(8·0.4+9·0.7+7·0.6)/(0.4+0.7+0.6)	8.1
P2	(3·0.4+2·0.7+2·0.6)/(0.4+0.7+0.6)	2.2

ImpactFactor	**Formula**	**I(a)**
I1	(9·0.8+9·0.5)/(0.8+0.5)	9.0
I2	(4·0.8+3·0.5)/(0.8+0.5)	3.6

TimeFactor	**Formula**	**T(a)**
T1	(1.0+1.0)/2	1.0

By analogy, the probability, impact, and time factors for feature F2 are as follows:

ProbabilityFactor	**Formula**	**P(a)**
P2	(3·0.9+2·0.3+2·0.4)/(0.9+0.3+0.4)	2.6

ImpactFactor	**Formula**	**I(a)**
I3	(8·0.7+9·0.7)/(0.7+0.7)	8.5

TimeFactor	**Formula**	**T(a)**
T2	(1.0+1.0)/2	1.0

Based on the probability, impact, and time factors, the risk coefficients R1 and R2 for the features F1 and F2 can be calculated:

Risk	**Formula**	**R(a)**
R1	((8.1+2.2)/2·1.0)·(9.0+3.6)/2	32.4 (5.2×6.3)
R2	(2.6·1.0)·8.5	22.1 (2.6×8.5)

The risk coefficients for R1 and R2 are shown in Fig. 4. Even though, the *ImpactFactor* of R1 is not as high as the *ImpactFactor* of R2, it is categorized into a higher risk category than R2 because of the high probability that a failure occurs.

We assume that the next risk assessment session takes place after the first prototype has been implemented. Furthermore, we suppose, that the different impact and probability criteria have been estimated precisely, such that their values do not change in the actual assessment session. The result of the first risk assessment session was that the risk value of F1 is roughly 10 units higher than F2's risk. Therefore, in the first evaluation phase F1 has been tested more

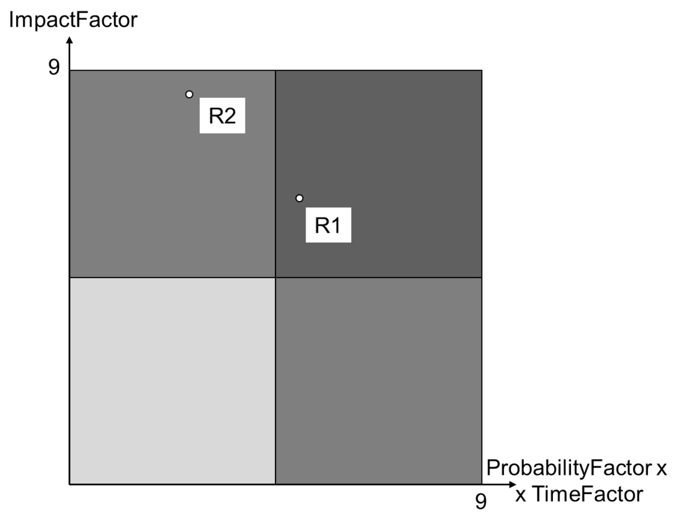

Fig. 4. R1 and R2 after first assessment

deeply than F2. When testing the features after their implementation, we noticed that F2 has more failures than F1. In our example, we assume that the the value of *failed test cases/number of test cases* for F1 is only 40% of F2's value. Additionally, we assume that it is observed that F2 has been changed more frequently than F1. In the example, F1 has only 60% of the number of updates of F2. Thus, the change history and the test history value for F1 are reduced, but are constant for F2:

Feature	**TimeCriterion**	$\mathbf{t_j}$
F1	Change History	0.6
F1	Test History	0.4
F2	Change History	1.0
F2	Test History	1.0

The recalculated time factors T1 and T2 are then as follows:

TimeFactor	**Formula**	**T(a)**
T1	(0.6+0.4)/2	0.5
T2	(1.0+1.0)/2	1.0

As mentioned before, we assume that none of the probability and impact criteria have changed its value, such that the resulting risks coefficients R1 and R2 are as follows:

Risk	**Formula**	**R(a)**
R1	$((8.1+2.2)/2 \cdot 0.5) \cdot (9.0+3.6)/2$	16.2 (2.6×6.3)
R2	$(2.6 \cdot 1.0) \cdot 8.5$	22.1 (2.6×8.5)

The updated risk coefficients and their classification are shown in Fig. 5 which illustrates the effect of the *TimeFactor*, because it can seen, that the *ImpactFactor* of R1 and R2 has not changed but the *ProbabilityFactor* of R1 decreased. Therefore, R1 and R2 are now both in the same risk category and the risk coefficient R1 is lower than the risk coefficient of R2.

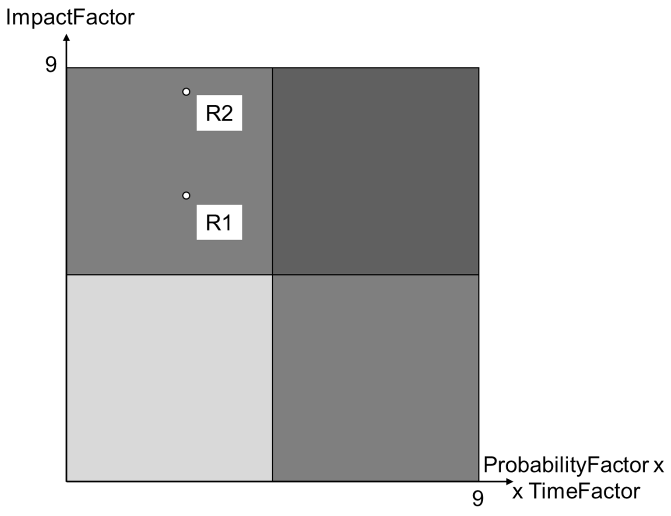

Fig. 5. R1 and R2 after second iteration

With the procedure at hand, resources can be saved by prioritizing the test cases and not implementing and/or executing all of them. Nevertheless, there is an overhead of time and money needed for evaluating the different risk factors. Therefore, if possible most of the criteria should be evaluated automatically. We have categorized criteria according to the degree of automation into *automated evaluation*, *semi-automated evaluation*, and *manual evaluation*. A criterion is evaluated automatically, if all underlying metrics are evaluated automatically, semi-automatically, if all underlying metrics are evaluated automatically or semi-automatically, and manually, if at least one underlying metrics is evaluated manually. Typically code complexity metrics are evaluated automatically, functional complexity metrics are evaluated semi-automatically and customer-related metrics like importance of a product are evaluated manually.

We assume that automatic evaluation is easier to perform and more meaningful for *ProbabilityMetrics* than for *ImpactMetrics* This assumption normally holds because the *ProbabilityMetrics* are derived from formal artifacts like the source code and its architecture for which many analysis tools are available. The *ImpactMetric*s are derived from requirements which are typically only available as informal text and evaluated manually.

The degree of automation of the evaluation can change for specific metrics. For instance, time metrics like the change or test history can only be estimated manually when a project starts but can later be computed automatically based on data from version and test management systems.

Besides the automation, the precision of the assessment of manually and semi-automatically determined metrics can be improved by distribution among the stakeholders. Software developers and software architects may evaluate probability metrics, the project manager sets the component weights, and the impact metrics are evaluated by customers. Time metrics are determined by the test manager before the release and additionally by the customer support after the release of the system.

In the following subsections, we give examples for the automated, semi-automated and manual evaluation of metrics.

5.1 Automated Metrics

As automated metrics we consider e.g., *code complexity metrics* because there exist many code metrics, e.g., the number of code lines or the McCabe complexity, that can be determined automatically. In general, we suggest to project the automatically evaluated values to a scale. The lowest value derived, will be projected on the lowest value of the scale, and the highest value on the highest scale value. The other derived values have to be interpolated on the remaining values of the scale. There are many tools for automatically measuring the code complexity, e.g., CAST [30], Understand [31], Sonar [32] or iPlasma [33].

The results of the automatic evaluation can be quantitative or qualitative. Quantitative results are numeric values (see [34] for an overview of typical metrics) like lines of code (LoC), McCabe complexity [22] or depth of the inheritance tree. Qualitative results are verbal description, e.g., of error-prone statements. In this case, the different warnings and error messages have to be quantified. Both types of metrics are derived by automatic static code analysis [35]. Thus, if the evaluation procedure is based on several code metrics, only an algorithm for automatically determining the value of a criterion has to be implemented.

5.2 Semi-Automated Metrics

Semi-automated metrics are derived automatically but the resulting value is only an approximation, e.g., because relevant information is not taken into account by the derivation algorithm. Therefore, the value has to be corrected manually and the automatically derived value only provides decision support for the user. As an example for semi-automated metrics we consider *functional complexity metrics*. There are several fuzzy metrics for measuring functional complexity, e.g., the length of the textual description of an algorithm, the occurrence or absence of keywords, the complexity of the computation graph of the algorithm, and the computational complexity of the algorithm.

In many cases, none of the mentioned metrics is solely a precise measure for the functional complexity. But they provide valuable input for a subsequent manual estimation of the functional complexity.

5.3 Manual Metrics

As a manually determined metrics, e.g., the predicted *usage*, i.e., the frequency of use and importance to user, can be considered. The usage can only be estimated depending on the expected user number of the *System* and the privileges needed to use the specific *Requirement*. Even though the text might contain hints for a possible usage, the real usage only can be estimated with experience and background knowledge. As support for the human estimators, a scale with textual values can be provided. The following table shows such a scale with textual values. The scale defines five values (seldom, sometimes, average, often, highest) which are projected on the numeric values of the metrics.

Usage Metrics Ordinal Scale Values	
seldom	1
sometimes	3
average	5
often	7
highest	9

Another possible interpretation of the usage criterion is shown in the next table, which represents a cumulative scaling function. The selected entries are summed up to determine the final value of the metric.

Usage Metrics Cumulative Scale Values	
many different users	+3
barrier-free	+1
often used	+3
untrained staff	+2

The different values are defined during the risk identification phase and stakeholders who conduct the risk assessment only have to select the proper values. For the cumulative scale, the sum of all values has to be lower or equal to the maximal scale value (in our case 9). As mentioned before the evaluation type of metrics can change during the application life cycle. For instance, the usage has to be manually estimated before the system is in use, but can be determined automatically in the maintenance phase.

5.4 Metric Estimation Workshop

Many metrics have to be evaluated manually or semi-automatically. For the evaluation process, estimation workshops may be conducted. For such workshops where human experts estimate values, it is very helpful if the scales for manually or semi-automatically evaluated metrics do not only consist of numeric values, but additional provide textual descriptions for each value of an ordinal or cumulative scale (see Section 5.3 for examples).

As mentioned before, time metrics are typically computed in an automatic way as soon as version and test management systems are available and are therefore not separately considered in an estimation workshop. In our process the semi-automatically and automatically evaluated probability and impact values are estimated by two separate teams. The impact metrics are estimated by customer-related stakeholders such as product managers or customers themselves, and the probability metrics by technical stakeholders such as software developers or architects. Each member of the particular estimation team gets an estimation form for each requirement or unit for determining probability or impact values. Afterwards, the estimated values of the single team members are compared and discussed by the whole team, until a consensus is achieved and the final value can noted by the moderator of the estimation workshop. It may be useful to document the activities in the estimation workshop because a complicated decision-finding can be an indicator for a high risk. The separated estimation of probability and impact values guarantees an independent and efficient estimation process.

6 Conclusion

In this paper we have defined a risk assessment model and a risk assessment procedure based on a generic risk-based testing process. The risk-based testing process consists of the phases risk identification, test planning, test design, evaluation, and risk assessment which is the core activity of the process.

The package risk of the risk assessment model defines factors, criteria, metrics and functions for the computation of risks. Additionally, the model considers requirements, implementation, plus system elements as basis for the risk assessment, and tests to which the resulting risks are attached. The static view of the risk assessment model is the foundation for the dynamic view of the risk assessment procedure that is presented by an example considering impact, probability and time criteria.

The concrete algorithms for the determination of risks are based on manually, semi-automatically, and automatically evaluated metrics. For each type of metrics evaluation we present examples and determination strategies. The integration of manual, semi-automatically and automatic metrics as proposed in our approach significantly improves risk assessment because it supports more efficient determination of metrics by automation and distribution among stakeholders. Efficient risk assessment is the prerequisite for the successful application of risk-based testing in practice.

As future work we evaluate the effort of the risk assessment process empirically by its application to industrial projects. Based on the results of the risk assessment proposed in this paper, we investigate the other activities of the risk-based testing process. The risk classification influences the definition of test methods and test exit criteria in the test strategy. The prioritization of tests, where areas with high risks have higher priority and are tested earlier, is considered for the test design. In the evaluation phase residual risk of software delivery are estimated. All these tasks are evaluated empirically by the application to industrial projects.

Acknowledgment

This work is sponsored by the project QE LaB - Living Models for Open Systems (FFG 822740).

References

1. Merriam-Webster: Merriam-Webster Online Dictionary (2009) available at `http://www.merriam-webster.com/dictionary/clear` [accessed: July 12, 2011].
2. Bach, J.: Troubleshooting risk-based testing. Software Testing and Quality Engineering **5**(3) (2003) 28–33
3. Ropponen, J., Lyytinen, K.: Components of software development risk: How to address them? a project manager survey. Software Engineering, IEEE Transactions on **26**(2) (2000) 98–112
4. Pfleeger, S.: Risky business: what we have yet to learn about risk management. Journal of Systems and Software **53**(3) (2000) 265–273
5. Boehm, B.: A spiral model of software development and enhancement. Computer **21**(5) (1988) 61–72
6. Kontio, J.: Risk management in software development: a technology overview and the riskit method. In: Proceedings of the 21st international conference on Software engineering, ACM (1999) 679–680
7. Karolak, D., Karolak, N.: Software Engineering Risk Management: A Just-in-Time Approach. IEEE Computer Society Press Los Alamitos, CA, USA (1995)
8. Amland, S.: Risk-based testing: : Risk analysis fundamentals and metrics for software testing including a financial application case study. Journal of Systems and Software **53**(3) (2000) 287–295
9. Bach, J.: Heuristic risk-based testing. Software Testing and Quality Engineering Magazine **11** (1999) 99
10. Carr, M., Konda, S., Monarch, I., Ulrich, F., Walker, C.: Taxonomy-based risk identification. Carnegie-Mellon University of Pittsburgh (1993)
11. Stallbaum, H., Metzger, A., Pohl, K.: An automated technique for risk-based test case generation and prioritization. In: Proceedings of the 3rd international workshop on Automation of software test, ACM (2008)

12. Stallbaum, H., Metzger, A.: Employing Requirements Metrics for Automating Early Risk Assessment. Proc. of MeReP07, Palma de Mallorca, Spain (2007) 1–12
13. Lund, M.S., Solhaug, B., Stolen, K.: Model-driven Risk Analysis. Springer (2011)
14. Lee, W., Grosh, D., Tillman, F.: Fault tree analysis, methods, and applications - a review. IEEE transactions on reliability (1985)
15. Mauw, S., Oostdijk, M.: Foundations of attack trees. Information Security and Cryptology-ICISC 2005 (2006) 186–198
16. Alexander, I.: Misuse cases: Use cases with hostile intent. Software, IEEE **20**(1) (2003) 58–66
17. Asnar, Y., Giorgini, P.: Modelling risk and identifying countermeasure in organizations. Critical Information Infrastructures Security (2006) 55–66
18. McCall, J., P.K., R., G.F., W.: Factors in software quality. . Technical report, NTIS, Vol 1, 2 and 3 (1997)
19. Haimes, Y.Y.: Risk Modeling, Assessment, and Management. Wiley (2004)
20. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th international conference on Software engineering, ACM (2006)
21. Illes-Seifert, T., Paech, B.: Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs. Information and Software Technology **52**(5) (2010)
22. McCabe, T.: A complexity measure. IEEE Transactions on software Engineering (1976) 308–320
23. Jiang, Y., Cuki, B., Menzies, T., Bartlow, N.: Comparing design and code metrics for software quality prediction. In: Proceedings of the 4th international workshop on Predictor models in software engineering, ACM (2008) 11–18
24. NIST: National Vulnerability Database available at `http://nvd.nist.gov/` [accessed: July 12, 2011].
25. : The Open Source Vulnerability Database available at `http://osvdb.org/` [accessed: July 12, 2011].
26. Frei, S., May, M., Fiedler, U., Plattner, B.: Large-scale vulnerability analysis. In: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense, ACM (2006) 131–138
27. Mell, P., Scarfone, K., Romanosky, S.: Common vulnerability scoring system. Security & Privacy, IEEE **4**(6) (2006) 85–89
28. Spillner, A., Linz, T., Rossner, T., Winter, M.: Software Testing Practice: Test Management. dpunkt (2007)
29. Erik van Veenendaal: Practical risk–based testing, product risk management: the prisma method. Technical report, Improve Quality Services BV (2009)
30. : CAST available at `http://www.castsoftware.com/` [accessed: July 12, 2011].
31. : Understand available at `http://www.scitools.com/` [accessed: July 12, 2011].
32. : sonar available at `http://www.sonarsource.org/` [accessed: July 12, 2011].
33. : iPlasma available at `http://loose.upt.ro/iplasma/index.html` [accessed: July 12, 2011].
34. Zhao, M., Ohlsson, N., Wohlin, C., Xie, M.: A comparison between software design and code metrics for the prediction of software fault content. Information and Software Technology **40**(14) (1998) 801–810
35. Nagappan, N., Ball, T.: Static analysis tools as early indicators of pre-release defect density. In: Proceedings of the 27th international conference on Software engineering, ACM (2005) 580–586