Evaluating Software Development Methodologies Based on their Practices and Promises.

Parastoo Mohagheghi
Norwegian Labour and Welfare Services
67 PUBLICATIONS **1,540** CITATIONS

Evaluating Software Development Methodologies Based on their Practices and Promises

Parastoo MOHAGHEGHI
SINTEF, P.O. Box 124-Blidern, N-0314 Oslo, Norway
Parastoo.mohagheghi@sintef.no

Abstract. Software companies must often make decisions about applying new software development methodologies, technologies or tools. Various evaluation methods have been proposed to support this decision making; from those that focus on values (especially monetary values) to the more exploratory ones, and also various types of empirical studies. One common challenge of any evaluation is to choose evaluation criteria. While there have been a growing number of published empirical studies evaluating different methodologies, few of them include rationale for selecting their evaluation criteria or metrics. Therefore they also have problems with explaining their results. This paper proposes an approach for identifying relevant evaluation criteria that is based on the concepts of (core) practices and promises of a methodology. A practice of a methodology is a new concept or technique or an improvement to established ones that is an essential part of the methodology and differentiates it from other methodologies. A promise is the expected positive impact of a practice. Evaluation criteria or metrics are selected in order to evaluate the promises of practices. The approach facilitates identifying relevant criteria for evaluation and describing the results, and thus improves the validity of empirical studies. It will also help developing a common research agenda when evaluating new methodologies and answering questions such as whether a methodology helps improving a quality attribute and how, what the differences are between two methodologies, and which studies are relevant when collecting evidence about a methodology. The proposed approach is applied on software reuse and model-driven engineering as examples based on the results of two literature surveys performed in these areas.

Keywords. evaluation, empirical study, metrics, software reuse, model-driven engineering, GQM.

Introduction

Organizations invest considerable amount of money in adopting new software development methodologies, technologies and tools[1], and are of course interested in evaluating resulting performance, their impact on their practices and organizations, and the positive value created by new investments. Also when involved in R&D projects, they face the challenge of evaluating new methodologies. While cost, time and quality

[1] We use the term "methodology" in the remainder of the paper to cover software development methodologies, technologies and tools.

are often mentioned as the main factors driving software development [40], it is not always straightforward to measure the impact of a methodology on these factors or other external attributes such as quality-in-use or Return-on-Investment (ROI). In addition, the above factors might not always be relevant, since the impact of a methodology may be on other factors that eventually create business revenue for an organization, such as increased flexibility to develop new products or better communication with customers.

MODELPLEX (MODelling solution for comPLEX software systems)[2] is an EU IST research project that started in 2006 with the goal of developing a coherent infrastructure for the application of Model-Driven Engineering (MDE) to the development and management of complex software systems within a variety of industrial domains. Research in MODELPLEX is user driven, which means that the developed methodologies should be evaluated in the context of the four industrial partners. When planning empirical evaluation of MDE, we faced the challenge of identifying relevant evaluation criteria. A survey of earlier empirical studies and experience reports on MDE showed that only a few metrics have been used to evaluate MDE, however, the choice of these metrics is often poorly justified and they do not cover the promises of MDE.

In this paper we discuss a systematic approach for identifying evaluation criteria when facing new methodologies or when trying to analyze results of earlier studies. The approach results in measuring the correct things and describing the results in a better way. Without a systematic approach, we may end with selecting evaluation criteria based on convenience, earlier experience or those often used by others without answering the questions of relevancy and validity. Our approach is based on the concepts of *practices,* which identify the main innovations in a methodology, and *promises* as the expected benefits. While we can measure every methodology from multiple aspects and some of the aspects such as ease of use and compatibility with other methods are relevant anyway, each methodology has also some unique practices that differ it from other methodologies and should be subject of evaluation.

The remainder of the paper is organized as follows. Section 1 is problem statement with discussion of related work and examples on evaluating methodologies. Section 2 presents our approach for identifying evaluation criteria, and Sections 3 and 4 apply it on software reuse and MDE. Section 5 includes discussion of the approach and its relations to other approaches. The paper is concluded in Section 6.

1. How to Evaluate Methodologies?

Evaluating alternative solutions for computer systems, software design or software development has been subject of research for a while and various quantitative or qualitative techniques have emerged. The purpose of this section is not to provide a survey of evaluation methods, but to highlight a common problem in any evaluation process; selecting the relevant evaluation criteria.

[2] https://www.modelplex.org/

1.1. Cost-Benefit Analysis and Empirical Research

In an article published in 1981, Keen identifies three basic techniques used to evaluate proposals for computer systems in most organizations [24]:

- Cost-benefit analysis and related ROI approaches- these techniques view the decision as a capital investment;
- Scoring evaluation- this technique views the decision in terms of weighted scores to some characteristics and is useful when comparing several alternatives;
- Feasibility study- this approach focuses on defining specifications of a complete system and identifying costs and benefits.

According to Keen, all the above approaches require fairly precise estimates of costs and benefits and often do not handle the qualitative issues central to innovation. The author therefore proposes the *Value Analysis Method* which is exploratory and iterative. In short, the method suggests to establish an operational list of benefits and a cost threshold, build version *0* and asses the prototype, take a decision whether to proceed or not; and iterate. The method’s iterative approach and prototyping allow us to observe the outputs and adjust the initial list of costs and benefits. However, the focus is mainly monetary values.

In a paper by Farbay and Finkelstein, they have divided evaluation methods applied on software design in two major groups [17]:

- Quantitative and comparative methods such as cost-benefit analysis, ROI and information economics.
- Qualitative and exploratory methods such as Value Analysis and Multi-Objective Multi-Criteria methods. MOMC is an iterative approach to establish preferences and utilities but from multiple points of view.

The paper also includes taxonomy of evaluation approaches as developed by House [20] and updated by the authors. While the major audiences for the quantitative methods are management or governmental bodies, other audiences such as users or practitioners may prefer other approaches such as experimentation or case studies. The taxonomy includes common questions to answer for each approach, which may be interpreted as evaluation criteria. One key point when studying the taxonomy is that different stakeholders ask different questions in evaluation and thus the viewpoint is important in any evaluation. Examples are:

- From the management view: What ROI is expected? Have the expected effects been achieved?
- For decision making operational research: Will the system be effective in use?

Value-based software engineering is another approach with focus on identifying values of methodologies. In a book by its promoters, the authors emphasize that value considerations should be integrated into software engineering, where value covers more than “the monetary value”; for example stakeholders’ degree of preference of something over alternatives (utility function) or negotiation during requirements are also values to consider [7]. The book includes several examples on how to put value on software engineering assets such as requirements and testing approaches.

While the above methods are proposed to be applied in software engineering research as well, there are some concerns:

- The focus of cost-benefit analysis or ROI is on monetary values. Normally, not all effects are tangible or may be expressed in terms of money.
- Cost is often not the primary concern of early adopters of innovations.
- It is difficult to argue for the monetary values of a methodology that is not mature enough or have not been used in real life projects of proper scale and for some duration.
- Even when taking a broad approach to evaluation that includes other values than the monetary ones, we have to answer the question of what the relevant values are. The challenge is even larger when evaluating new methodologies with unclear impacts. For example, does a methodology improve productivity, software design or ease of work for developers?

Empirical software engineering involves the scientific use of quantitative and qualitative data to understand and improve the software product, software development process and software management [6]. Empirical research is research based on observation. The researcher often starts with an idea of relations or a hypothesis, collects some data and interprets them to verify the relations or to decide whether to reject the hypothesis or not. The research may even be exploratory without clear definition of relations in the beginning. The viewpoint of evaluation is often those of practitioners. Several books and papers are published during the recent years on types of empirical studies and how to perform them (e.g., [45] and [25]) and how to collect evidence from such studies based on the systematic collection of published results and analyzing them [15]. Our focus here is on the selection of evaluation criteria in empirical studies. These criteria may sometimes be evaluated quantitatively by defining proper metrics and sometimes qualitatively for example by expert judgments. We discuss some proposed methods in the next section together with their advantages and short-comings.

1.2. Selecting Evaluation Criteria beyond Cost and ROI

Any existing or new methodology may be evaluated for some general characteristics. For example, the Technology Acceptance Model (TAM); originally developed by Davis [12] and later extended by others; explains users' intention to use a new system through two beliefs, *perceived usefulness* and *perceived ease of use*. There are also several additions to TAM and those that combine TAM with other models, such as the model used by Dybå et al. [13] and shown in **Figure 1** that includes three additional factors added from other models. The evaluation can be done quantitatively by using questionnaires or qualitatively by using interviews [32]. There are also other models for evaluation as discussed in [14], but few discuss how theory informs their choice of metrics. Besides, characteristics such as usefulness are vaguely defined and a precise definition for the methodology and the context of use is left to the evaluator.

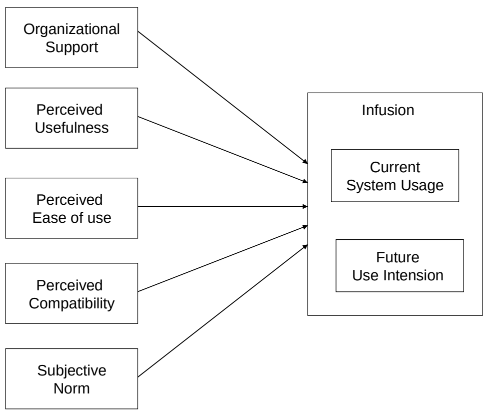


Figure 1. The conceptual model based on TAM [13]. The factors are defined as: Organizational Support is the degree a method is supported by an organization. Perceived Usefulness is the degree to which a person believes that using a particular method will enhance his/her job performance. Perceived Ease of use refers to the degree to which a person believes that using a particular method or tool would be free of effort. Perceived Compatibility is the degree to which a method is perceived as being consistent with existing values, principles, practices and the past experience of potential adopters. And subjective Norm is the degree to which developers think that others who are important to them think they should use a method.

To answer the question of selecting proper criteria (metrics in this case) for evaluation, Basili and his colleagues have proposed the Goal-Question-Metric (GQM) approach [5]. An example is shown in **Figure 2**.

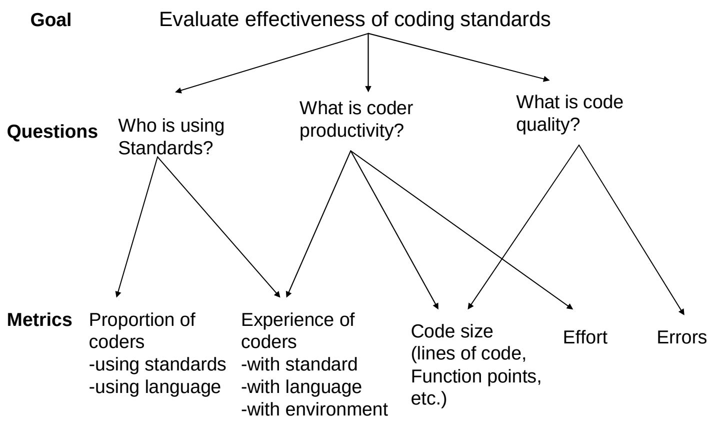

Goal
Evaluate effectiveness of coding standards
Questions
Who is using Standards?
What is coder productivity?
What is code quality?
Metrics
Proportion of coders -using standards -using language
Experience of coders -with standard -with language -with environment
Code size (lines of code, Function points, etc.)
Effort
Errors

Figure 2. Driving metrics from goals and questions. Example is from [18]

GQM starts by expressing the overall goals of the measurement. Then questions are generated whose answers must be known to determine if the goals are met. Finally, each question is analyzed in terms of what measurements are needed to answer the question. GQM is a top-down approach that is useful when we have clear goals for the evaluation; for example when evaluating the impact of a software process improvement activity. However, GQM is difficult to apply when the impacts of a methodology are not clear and an exploratory approach is needed to identify the impacts.

There are also approaches to software process improvement such as the Six Sigma approach that include some evaluation criteria [37]. Metrics are selected based on the customer opinion and are divided into financial (cost and savings), customer (e.g., customer satisfaction, on-time delivery and product quality), internal (e.g., defect, inspection time), and employee learning perspectives (e.g., quality of training)

An alternative approach to GQM and process improvement models that is widely applied in empirical studies is to start with a list of metrics and select those that seem relevant for evaluating a methodology. For example the book of Fenton and Pfleeger classifies the entities to be measured in *processes*, *products* and *resources*, and attributes of measurement in *internal* and *external*, and contains examples of metrics in each group [18]. Existing quality models such as the series of ISO-9126 standards [21] (recently updated in the SQuaRE series of standards [9]) also include examples of metrics. In the ISO standards, attributes are divided into *internal* and *external quality*, and *quality in use*. An example of such approach is [4] where a hierarchical model for the assessment of high-level design quality attributes in object-oriented designs is described. The authors start with the ISO-9126 attributes, drop some and add others, and identify object-oriented design properties to achieve the quality attributes and relevant metrics to measure them. The main disadvantage of this approach is that the selection of quality attributes and metrics is subjective and may be biased towards selecting those that are easier to measure, are most widely used or the evaluator has most experience with. These standards have also been subject of critics for lack of rationale behind classification, and for being ambiguous and incomplete (see [1]). A ranking of top three popular metrics per phase by a group of experts is presented in [27] and shown in **Table 1**. While we may rely on expert opinion in many cases, the rationale for selecting metrics is often not well-documented.

Table 1. Top three popular measures per phase, from [27]

Requirements	**Design**	**Implementation**	**Testing**
Fault density	Design defect density	Code defect density	Failure rate
Requirements specification change requests	Cyclomatic complexity	Design defect density	Code defect density
Error distribution	Fault density	Cyclomatic complexity	Coverage factor

In our opinion, each of the above approaches has some short-comings:

- The top-down approaches such as GQM or Six-Sigma are applicable for evaluating the impact of software improvement activities when the improvement goals have been defined prior to applying the new methodology (for example a new inspection technique). However, they do not answer the question of selecting evaluation criteria when facing new methodologies.
- When selecting general models such as TAM, one should answer the question of how to define usefulness of a methodology.

- Selecting most popular metrics or those included in quality models and standards is applicable if we have a theory that relates the subject of evaluation to the selected metrics. Otherwise the selection may be biased or poorly justified.

The examples in the next section show the importance of a systematic approach for selecting evaluation criteria or metrics. Otherwise we end up with metrics that are not relevant, results that cannot be described based on the methodology or data that do not say anything of importance about the methodology.

1.3. Some Examples on Selection of Metrics

We illustrate the problem of selecting metrics in empirical studies with a few examples.

1.3.1. Evaluating MDE

We recently performed a survey to collect industry experiences with MDE and the results are published in [34]. While the motivations behind using MDE are numerous, such as improving productivity and quality, maintenance concerns, formalism and more standardization, and improving communication in development teams and with external stakeholders, the reported quantitative evidence focuses mainly on three aspects:

- Effort and productivity: Effort is time it takes to perform a task, for example creating or changing a model; in person-hours or person-days. Effort is typically used to estimate productivity as output divided by effort.
- Software quality: One report identified that fewer inspections were required to ensure the quality of the developed code and inspection rates are higher. Simulations were also found to be about 30% more effective in catching defects than code inspections.
- Automation degree: this metric focuses on the artefacts that are generated from models in the number or size, relative to all the artifacts.

Productivity and software quality improvement are often given as motivations behind using any new methodology, although software quality means different things in different contexts. Results regarding productivity increase with MDE are contradictory and the rationale behind expecting productivity improvement is not clear either. Any innovation can lead to productivity improvement if it reduces the effort for performing a task but the improvement is significant only if the task is regarded as being time-consuming or of significant size. The studies on MDE are often performed on isolated small tasks and evaluating productivity improvements for such tasks cannot provide reliable results on the impact of MDE on real-life projects. There have been few quantitative results regarding software quality which does not allow drawing any conclusions.

Automation degree is on the other hand a metric specific to MDE and discussions on automation also include when it is achievable and to what degree. Other motivations behind using MDE such as improving communication are not evaluated at all. We clearly see that those studies that have focused on MDE-specific techniques and promises came with interesting observations, while several studies included data that are difficult to explain or are not relevant. For example, automated testing does not depend on having a MDE approach and its benefits cannot be credited to MDE if test cases are not generated from other artifacts.

1.3.2. Evaluating Software Reuse

The lack of clear evaluation criteria for MDE may be related to its being a new methodology. However, another literature survey performed earlier by us on software reuse shows that even in a field that has roots back to 1960s, there is still too much confusion when it comes to evaluation criteria. A total of 12 industrial papers analyzed in [33] included 22 different dependent metrics, with very few examples of a common definition. This diversity makes comparison of results and extracting some generalizable conclusions very difficult. There were only few studies who had focused on the main promises of software reuse which are increased apparent productivity and lower defect density over time (because of the accumulated defect fixes). In most cases, the selection of metrics seems to be based on convenience and the available date, and it covered even examples such as when errors are introduced (requirements, coding etc.) and input-output parameters per Lines of Code (LOC). Although some of these factors may be used to describe other results, they do not seem to be relevant for answering whether reuse is beneficial or when it is beneficial, and there is often no theory behind selecting these metrics.

1.3.3. Evaluating Test-Driven Development

A third example on the importance of selecting right metrics for evaluating methodologies is given in a recent article in the IEEE Software magazine by Janzen and Saiedian [27]. In this article, the authors discuss some misconceptions regarding Test-Driven Development (TDD). One of them is that TDD equals automated testing and therefore positive experiences with automated testing are used as support for TDD (the same as with MDE as discussed in Section 1.3.1).

The authors also refer to earlier empirical studies that have evaluated the impact of TDD on defects and increased productivity and report contradictory results. While TDD may impact productivity and defects, the authors emphasize that TDD should not be evaluated primarily by these external attributes. The focus of TDD is improving design by writing unit test first and constructing the units in short, rapid iterations that iterate over unit testing, coding and refactoring. The iteration allows the design to emerge and evolve. Therefore TDD should be evaluated by improvements in the quality of design and in metrics such as complexity and cohesion.

1.3.4. Evaluating Pair Programming

We conclude this section by an example on evaluating pair-programming. The eXtreme Programming website defines pair programming as [16]:

> All code to be included in a production release is created by two people working together at a single computer. Pair programming increases software quality without impacting time to deliver. It is counter intuitive, but 2 people working at a single computer will add as much functionality as two working separately except that it will be much higher in quality. With increased quality comes big savings later in the project.

Arisholm et al. have identified earlier studies on the effect and cost of pair programming and also performed a large quasi-experiment with professionals [2]. The dependent variables are selected to be *duration*, *effort*, and *correctness*, which are clearly related to the promises of pair programming. They have also added the moderating variables *system complexity* and *programmer expertise*, both of which have an impact on the perceived complexity of programming tasks. The results of this experiment do not support the hypotheses that pair programming in general reduces the time required to solve the tasks correctly or increases the proportion of correct solutions. On the other hand, there is a significant 84 percent increase in effort to perform the tasks correctly. The observed benefits of pair programming in terms of correctness on the complex system apply mainly to juniors, whereas the reductions in duration to perform the tasks correctly on the simple system apply mainly to intermediates and seniors.

This is an example of empirical studies with focus on evaluating the promises of a methodology which has also has answered when it is best to do pair-programming.

1.4. Conclusions on Earlier Work

We discussed the approaches proposed and the challenges related to the selection of evaluation criteria. While cost-benefit approaches are useful for taking decisions on IT investments from the managerial viewpoint, they do not answer the questions of interest to practitioners and are therefore not widely applied in software engineering research. Other methods such as feasibility analysis may be applied but still the criteria for evaluation should be defined. The GQM approach is widely used in selecting metrics but it is a top-down approach based on the goals of software improvement and is less useful in an exploratory evaluation. Finally, using widely-used metrics or those defined by the standards needs justification that shows the relation between metrics and the practices of a methodology.

The examples discussed in this section highlight the need for a systematic approach for selecting evaluation criteria. The approach should provide rationale behind selecting such criteria and facilitate validation of promises. It should also facilitate explaining the results and learning from experiences. We present our approach based on a bottom-up analysis of practices of a methodology in the next section.

2. Evaluating Methodologies Based on Practices and Promises

In the previous section we discussed examples of good and poor selection of metrics. While correct selection simplifies describing the results and collecting evidence from several studies, poor selection makes comparing results and collecting evidence a real challenge. Here we present an approach for selecting evaluation criteria that is based on the concepts of a methodology’s practices and promises.

A *Practice* of a software development methodology (or technology or tool) is a new concept or technique or an improvement to established ones that is an essential part of the methodology and differentiates it from other methodologies. We may also call it a *core practice*. Practices will impact product or project characteristics and benefits will eventually result in cost savings or increased revenue of an organization, which are the motivations behind taking any new methodology in use. Some examples of practices are:

- Short iterations and emphasizing customer involvement in software development are two practices of the agile development approach.
- For TDD, we can mention the practice of writing test cases first.
- For pair programming, it is 2 people working at a single computer.

Every methodology will be applied in different ways in different organizations, to different degrees, and often combined with existing practices; referred often as *operationalization*. For example pair programming may be done only on selective modules, or may be done in the same office or remote. However, the inclusion of core practices indicates whether the methodology is applied or not, and a total absence of a core practice shows significant modification of a methodology which will impact the expected outcomes or promises. For example, the absence of customer involvement in an agile project will require other means for communication such as developing documentation which is in contrast with the agile spirit.

A *Promise* is the expected improvement of a product or project attribute that is given as the main motivation of applying a practice. For example:

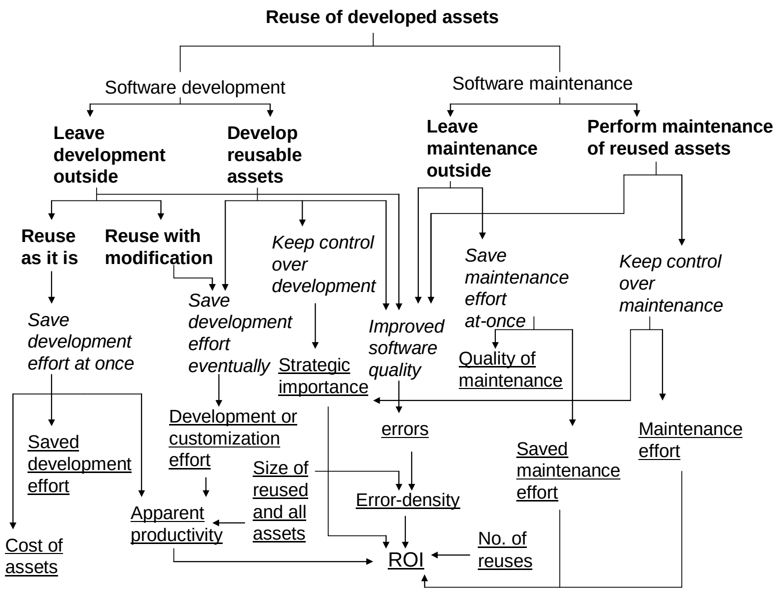
- Short iterations and customer involvement are supposed to reduce the risk of vague and changing requirements.
- TDD promises to improve design quality of software modules.
- Pair-programming promises to improve the quality of coding.

While decision-making on every new methodology includes criteria such as cost and ease of use, practices and promises of a methodology indicate where to expect benefits and how. In the second step, links may be established between the methodology-specific evaluation criteria and external attributes such as cost and quality. Although the idea seems intuitive, the lack of emphasize on (core) practices and promises has led to difficulties in defining the subject of evaluation, identifying relevant metrics and interpreting the results as discussed by the examples in Section 1.3. In the next sections we apply the approach on two areas: software reuse and MDE.

3. A Model for Evaluating Software Reuse

The model in **Figure 3** is based on a literature survey on software reuse [33]. We describe it in the remainder of this section.

Systematic software reuse (or in-short reuse; the term systematic refers to a planned approach to reuse in contrast to ad-hoc) may be done in multiple ways: reusable assets may be developed inside (such as in a product line development) or outside of an organization (so-called Off-The-Shelf products). When it comes to maintenance, the responsibility may be on the developer or the user side. Each practice has some promises and of course disadvantages. The practices (in **bold font**), promises (in *italic font*) and metrics (<u>underlined font</u>) are depicted in **Figure 3**. For example if an organization reuses an asset as it is, the expected benefit is mainly saved development effort. This promise should be evaluated by comparing the estimated development effort with the cost of the reused asset. The improvement in apparent productivity may also be evaluated based on the effort saved and the size of asset. The model also shows how to evaluate ROI based on the defined metrics.



Reuse of developed assets
Software development
Software maintenance
Leave development outside
Develop reusable assets
Leave maintenance outside
Perform maintenance of reused assets
Reuse as it is
Reuse with modification
Keep control over development
Save maintenance effort at-once
Keep control over maintenance
Save development effort at once
Save development effort eventually
Improved software quality
Strategic importance
Quality of maintenance
Development or customization effort
errors
Maintenance effort
Saved development effort
Saved maintenance effort
Size of reused and all assets
Error-density
Apparent productivity
Cost of assets
No. of reuses
ROI

Figure 3. Practices of software reuse (in **bold**), promises (in *italic*) and related evaluation criteria (underlined)

As shown in **Figure 3**, quantitative evaluation should be combined with qualitative evaluation to evaluate some practices. For example, a strategically important reusable part of a system is usually developed and maintained internally, and the evaluation of strategic importance is done by managers or business experts.

4. A Model for Evaluating MDE

Several books and a countless number of papers are written on the OMG's Model-Driven Architecture (MDA), Model-Driven Development (MDD) or Model-Driven Engineering (MDE)[3], and we can not provide an extensive review on what MDE is in this short paper. Besides, this is also a subject of our research. Based on our experience, the survey results described in [34] and other literature that is referred to, the practices of MDE are identified and described in **Figure 4**. We discuss the practices, promises and how to evaluate them with examples of application in the following sections.

[3] In the remainder of the paper we use MDE to refer to a model-driven software development approach, also where MDD is used in the referred literature.

MDE Practice #1. Models everywhere. Modeling is a core practice of MDE and models are considered as primary software artifacts and used directly in the software development chain.

MDE Practice #2. Multiple abstraction levels and separation of concerns. The basic idea of MDE is modeling a system in terms of several models that are at different levels of abstraction or from multiple viewpoints. One known example is the MDA's separation between a CIM (Computational Independent Model), PIM (Platform Independent Model) and PSM (Platform Specific Model). Each model has a separate goal and may be defined in its own language.

MDE Practice #3. Generation of artifacts from models. Transforming models into other models or artifacts such as code by Model-to-Model (M2M) or Model-to-Text (M2T) transformations is the key technique in MDE which allows using models directly in the software production chains as support for the notion of "Models everywhere".

MDE Practice #4. Metamodels and metamodeling. A model always conforms to a unique metamodel. The notion of a metamodel is strongly related to the notion of an ontology used in knowledge representation communities. Metamodeling allows defining relations between models needed to achieve generation and developing domain-specific languages and models.

Figure 4. Practices of Model-Driven Engineering (MDE)

4.1. MDE Practice #1.Models Everywhere

Modeling has mainly been earlier used for facilitating communication, especially with external stakeholders, and for describing high-level design. With MDE, models are primary software artifacts in all or most stages of software development; covering business models, requirements, high-level and low-level design, structure and behavior, and other aspects such as configuration of a system. Staron shows the MDE adoption spectrum in **Figure 5**, which should lead to being model-centric as opposed to code-centric [39].

Applying this practice of MDE will impact the software development process in multiple ways:

- In some organizations, additional phases must be added to the software development process such as adding Business Modeling as done in [36].
- More effort will be spent on modeling and activities related to modeling such as defining languages for modeling and inspecting models for quality [10]. For example, the Enabler company writes that when compared to the traditional development, they noticed that MDE requires a transfer of effort from development to analysis and specification [10].
- Modeling tools must be integrated with other tools such as testing and configuration management.

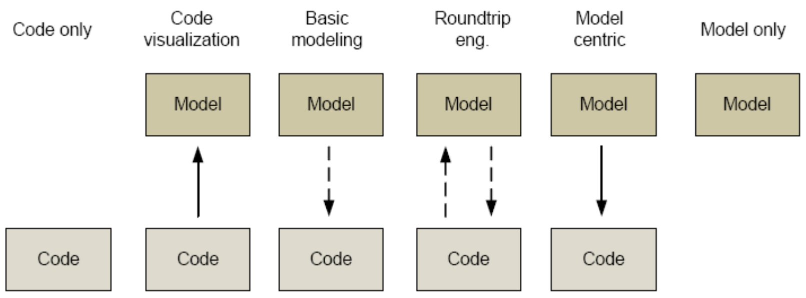


Figure 5. MDE adoption spectrum according to [39]

Some indicators of applying this practice (or operationalization) are:

- To what extent are models used in various phases of software development?
- Are models detailed enough as input for generation?
- How much effort is spent on modeling relative to other tasks such as manual coding?
- To what extent are the various tools for software development integrated?
- Is there extensive legacy code that should be reused? There are cases where organizations have refrained from using MDE due to the difficulties and workarounds required to integrate modeling with legacy systems (see [34]).

The main promises of this practice are described below.

4.1.1. Improved Communication

Modeling (especially graphical modeling) is expected to facilitate communication within development teams and with external stakeholders. This benefit is often given as one of the motivations of applying MDE but it is also a benefit that is hard to quantify. MacDonald et al. write that increasing the abstraction of design allows the design to be communicated to a range of people including customers and other developers [28]. The development of domain-specific languages is also motivated by the fact that using concepts close to those known by domain experts may ease communication between technical experts and domain experts. Improved communication may eventually make maintenance easier. According to [28], a design which is easier to communicate may reduce the “mythical man month”, the time that is normally taken to learn a new system, which will improve the cost effectiveness of maintaining systems. Also [36] names improving communication as one of the motivations behind applying MDE but does not evaluate it in practice.

4.1.2. Improved Software Quality by Using Models for Analysis and Testing

Detailed models provide possibility for analysis and testing on the model level. Model-based simulation and testing is given as one of the benefits of applying MDE [3, 11 and 44]. Motorola data in [44] shows that simulation is about 30% more effective in catching defects than the most rigorous inspections, and that defects are detected earlier in the software development lifecycle. That models are verified through simulation and checked for completeness also improves software quality significantly according to [11] and [44]. In [11], France Telecom writes that being able to validate the specification using simulation, allows them to "to eliminate uncomfortable ergonomics that would be difficult to detect otherwise". And Baker et al. write that due to MDE, fixes can be done on models, and the fix may be tested by running a full regression test suite on the model itself, regenerate the code from scratch, and run the same regression test suite on the generated code [3] .

4.2. MDE Practice #2.Multiple Abstraction Levels and Separation of Concerns in Models

The separation of domain concerns from implementation details should provide better communication, better quality, maintainability and portability of the developed systems across platforms [42]. Abstraction is also the main technique to handle complexity of software development. The practice is then combined with stepwise refinement (MDE Practice #3); either vertically or horizontally. Again the level of implementing this practice should be evaluated; for example by answering whether there is a need for having models from various viewpoints and abstraction levels, and whether there is tool support for that. Separation of concerns is also supported by the aspect-oriented approach where so-called cross-cutting concerns such as security are defined in separate models that are combined with base models. However, MDE adds transformation of models and multiple abstraction levels to the technique.

While there are benefits with separation of concerns, Mellor and Balcer refer to several challenging issues that inevitably arise from the multi-view and multi-notational approach in MDE, such as keeping models consistent with one another [30]. Thus applying this practice raises some challenges that should be handled by tools and new techniques such as traceability. MDE facilitates solving some of these problems by generation as discussed later. The promises of this practice are discussed below.

4.2.1. Improved Communication

Separation of concerns can lead to better communication between stakeholders; for example between business experts and IT experts. An example is given in [28]: the Tau Developer tool has been sufficiently flexible with good detail suppression to allow different views of the models, which can be used to describe the model to different people at different levels of abstraction. Tau Developer allows the nesting of behaviour descriptions one layer behind the diagram. This nesting allows the top layer of the activity diagram to be used to describe the design very abstractly, for example to a customer. However when describing the design in detail to another developer, the same diagram can be used with the behaviour descriptions visible.

4.2.2. Improved Software Quality

Software quality may be improved since developers focus on one aspect of development at a time. For example, [31] writes that: "When using MDA, you model your system using UML—not just by modelling Java classes, but high-level domain entities as well. This procedure forces you to actually *think* about the architecture and object model behind your system, rather than simply diving into coding, which many developers still do". Improved software quality may be evaluated based on feedbacks, surveys or inspecting models and analyzing when defects are injected or removed.

4.2.3. Portability of Solutions

A PIM may be implemented in multiple platforms. The word “platform” in this case applies to multiple aspects such as hardware, programming languages and integration solutions such as CORBA and XML. Increased portability leads to greater flexibility of organizations in case of platform changes. For example, [29] describes a case where the hardware was to be developed in parallel with the software and there would be little time to integrate the two, leading to a significant risk of errors and misunderstandings that would have to be handled by very late changes in the software. Therefore they needed to test the software on a range of platforms, from host PC to real target hardware. In case of changes in the platform, PIMs do not change and therefore these models have longer life time. Portability to multiple platforms is still not achieved in many development environments since most tools are bound to specific platforms.

4.3. MDE Practice #3.Generation of Models and Other Artifacts from Models

Generation of artifacts from models is the key technology to achieve automation and reducing manual work. In [34] we discussed that there are various levels of automation reported in industrial studies. While some papers report generating all or most of the code from the models, others report that only part of the code could be generated. Motorola evaluates the potential of MDE in generation to be between 65 to 96 percent depending on the type of the code (low level code is not captured in the design and is unlikely to be generated), and perceives the status of code generators as satisfactory in producing code with no introduced defects [3 and 44]. Companies should evaluate for their context which artifacts are best suitable to be generated and to what degree.

Generation is done through transformations; either Model-to-Model (M2M) or Model-to-Text (M2T). A transformation takes one or several models as input and produces a model (or models), or text, as output [38]. During transformation, output models are supplied with information not present in the input model. An example of such information is the *platform* concept. Thus generation actually supports separation of concerns and adding details later; not by manual work but by applying transformations. As shown in **Figure 6**, defining transformation depends on the concept of metamodeling identified as the fourth practice of MDE.

Various tools and development environments support transformations. In some cases generation is done by developing Domain Specific Languages (DSLs) or UML profiles and developing own code generators, as done in [8] and [41]. The main promises of the practice are described below.

4.3.1. Saved Effort on Manual work

Automation will results in less manual work which may be evaluated by estimating the saved effort. However, the impact on productivity depends on the saved effort due to automation compared with the effort spent on modeling and related activities such as developing transformations. Saved effort also depends on the percentage of artifacts generated from models. For example, if a significant amount of code or other assets are generated from models, saved effort will be significant.

As an example, we can mention the results of a comparative study performed by the Middleware Company, on behalf of Compuware [31]. Two teams developed the same application, one using MDE and the other using a non-MDE approach. The result was that the MDE team developed their application 35% faster than the other team – needing 330 hours compared to 507,5- and this change is mainly credited to the MDE-based code generation.

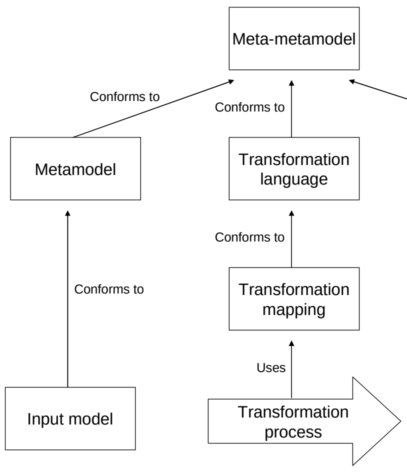


Figure 6. A transformation is described by its transformation mapping. It takes a model as input and produces a different model as output (we can also view code as a model). Each model – the transformation mapping included – conforms to a metamodel.

4.3.2. Consistency and Traceability between Artifacts

In traditional software development, models and code or test cases cannot be kept in synch with one another. With transformations, links between these artifacts can be preserved. These links can be used for analyzing the impact of changes, model debugging or synchronizing artifacts. As discussed in Section 4.2, the multi-model approach introduces consistency and traceability concerns which should be solved by generation and preserving the trace links. Unfortunately, there is still a long way to achieve full traceability, which is also one of the subjects of research in the MODELPLEX project.

Generating other artifacts from models such as documentation has benefits such as keeping documentation up-to-date with the design, which is a major challenge in non-MDE projects [28]. Evaluating the effort saved on maintenance (covering also implementing changes and impact analysis) depends on the case. Also test-cases can be generated from models to support model-based testing.

4.3.3. Improved Software Quality

Generation facilitates improving the quality of models and other artefacts such as their syntactic correctness and completeness. Quality of the generated code can also be improved by integrating design patterns and coding standards in the transformation rules. This is done in [31] which report that: “by using an MDA tool to generate your code with a consistent algorithm, rather than writing it by hand, you give all developers the ability to use the same underlying design patterns, since the code is generated in the same way each time”.

4.4. MDE Practice #4.Metamodeling

The concepts of metadata, OMG's Meta Object Facility[4] (MOF) and the MOF-like Eclipse's metamodel[5] (Ecore) allow definition of new languages or extending the existing ones; for example as DSLs or UML profiles. Developing domain-specific solutions is therefore also a practice of MDE that is based on the concept of metamodeling. While defining new languages or modifying the existing ones needs knowledge of language engineering and appropriate tools (examples are Eclipse GMF[6], MetaEdit+[7], and XMF[8]), the promises have motivated many companies to invest effort in developing their own languages and code generation engines; such as in [35], [41] and [23]. Metamodeling allows defining model transformations as relations between metamodels (see **Figure 6**) which is necessary for generation. The main promises are discussed below.

4.4.1. Improved Communication

Sharing the same language between domain and IT experts and narrowing the gap between them [43], and involving domain experts in all stages of design [8 and 23] are some of the promises of developing DSLs or UML profiles. Domain appropriateness has been defined as one of the quality goals of languages in [26] and may be evaluated qualitatively by experts or by comparing developed with existing domain ontologies. When developing own languages, the semantics may also be defined precisely which is not supported by some General Programming Languages such as UML.

4.4.2. Flexibility or Adapting Development Environments to Needs

The possibility of defining own languages and generation engines allows supporting other programming paradigms than those supported by existing modeling tools (mostly Object-Oriented) [35 and 36] and achieving flexibility which is of outmost importance for some domains.

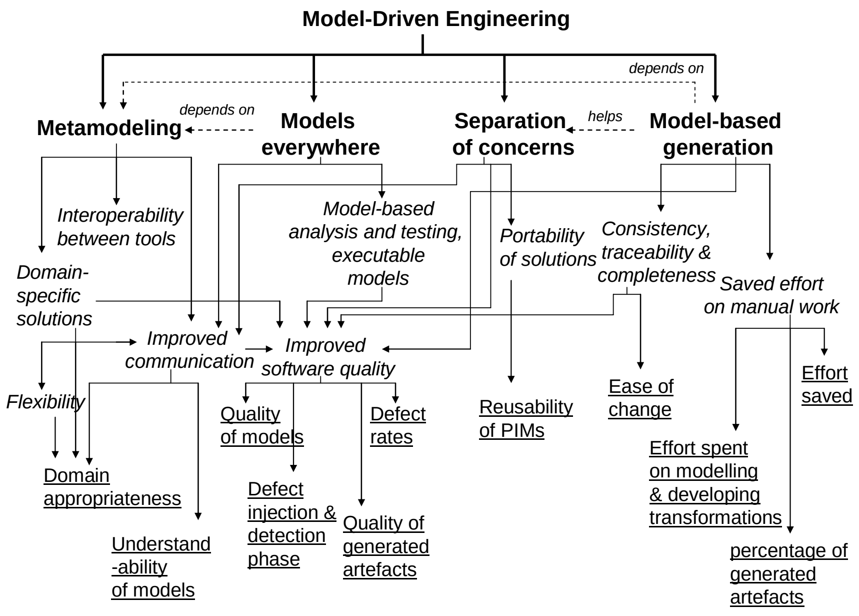
4.4.3. Tool Interoperability

The practice of metamodeling allows defining relations between metamodels or instances of them and exchanging models between tools; thus achieving interoperability between tools. This is an achievement of MDE which has significantly impacted the way software is developed.

4.5. Summary of MDE Practices, Promises and Metrics

We have summarized the practices and promises of MDE, with some evaluation criteria as depicted in **Figure 7**. The model is based on literature on MDE and empirical research analyzed in [34].

[4] http://www.omg.org/mof/
[5] http://www.eclipse.org/modeling/emf/?project=emf
[6] http://www.eclipse.org
[7] http://www.metacase.com/
[8] http://www.xactium.com/



Model-Driven Engineering
depends on
depends on
helps
Metamodeling
Models everywhere
Separation of concerns
Model-based generation
Interoperability between tools
Model-based analysis and testing, executable models
Portability of solutions
Consistency, traceability & completeness
Domain-specific solutions
Saved effort on manual work
Improved communication
Improved software quality
Flexibility
Quality of models
Defect rates
Reusability of PIMs
Ease of change
Effort saved
Domain appropriateness
Defect injection & detection phase
Quality of generated artefacts
Effort spent on modelling & developing transformations
Understand -ability of models
percentage of generated artefacts

Figure 7. MDE practices (in **bold**), promises (in *italic*) and evaluation criteria (<u>underlined</u>)

We do not have space to discuss evaluation criteria in more details and others may be added depending on the context. For example, improved software quality may be evaluated in multiple ways. Besides, in some cases it is easy to perform a quantitative evaluation, while in other cases the evaluation will be qualitative. Identifying appropriate evaluation methods depend on the context, the importance of a criterion for the involved stakeholders and whether it is easy to quantify a criterion or not. In [19], Glinz argues that not every quality requirement can be measured quantitatively; either because there is no metric defined or application of a metric would be too expensive. The author advises taking advantage of other methods such as inspections in these cases.

5. Discussion

In Section 1, we examined some methods that either include some general evaluation criteria (e.g., TAM) or define an approach for identifying these based on the software process improvement goals (e.g., GQM). The approach proposed in this paper meets the challenge that rises when trying to use either of these methods; i.e., what are the relevant quality goals or usefulness criteria? We propose performing a deep analysis of a methodology and identifying its core practices and promises before selecting metrics. The approach is useful both when evaluating innovations where the impacts are unclear or not verified and when analyzing results of other studies in order to evaluate their relevance.

Our approach can be combined with the above mentioned approaches as depicted in **Figure 8**. The promises of a methodology can be used as input to GQM as goals.

Evaluation criteria may also be used as input to TAM to precisely define usefulness, ease of use or compatibility. While the practices identified for software reuse or MDE may be subject of discussion and modification, the approach's emphasize is on building models of practices and promises in order to evaluate a methodology.

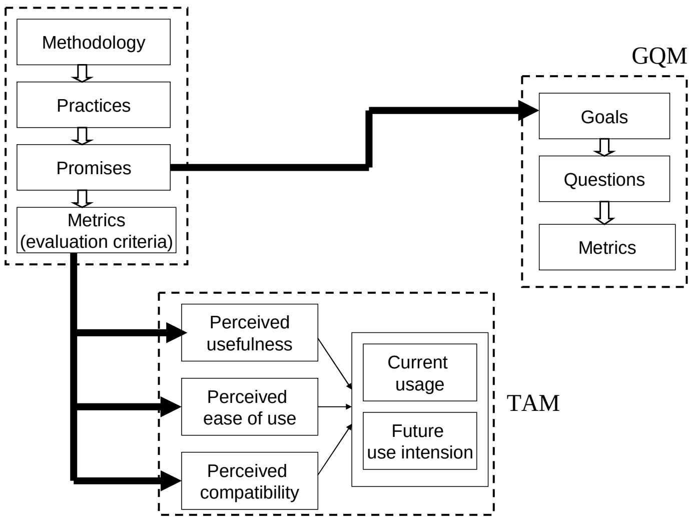


Figure 8. Our approach (Methodology, Practices, Promises & Metrics) can be used as input to other methods of evaluation for identifying relevant evaluation criteria.

Achieving promises of a methodology also depends on the operationalization of the practices, which is implementing them in a context and often combined with other practices. The process of methodology usage and evaluating organizational performance (or promises in our approach) is in fact a complex one as shown in **Figure 9.** It is therefore important to also evaluate methodology usage (whether practices are applied and to what degree); for example by subjective self-reported measures [14], objective computer-recorded measures [14], observation by an external reviewer, or ideally a combination of these approaches. In Section 4 we presented some questions to answer to evaluate operationalization of some practices; for example the degree of automation and the extent of using models in different phases of software development.

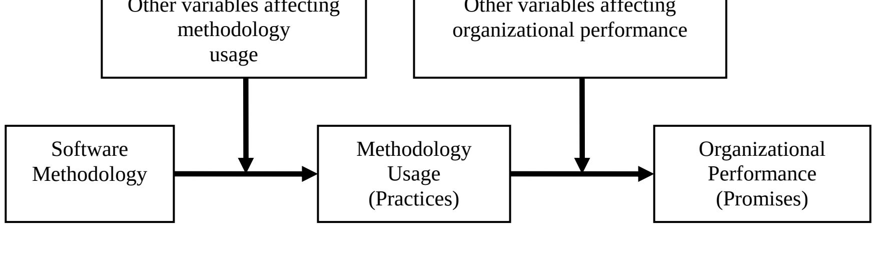


Figure 9. Methodology usage as a mediating variable [14]

Our approach should be operationalized as well for a specific context; which is refinement and precise definition of how practices are implemented (the actual usage), identifying other practices and important organizational factors, defining procedures for measurement and collecting data, and identifying some baseline data to compare the results with.

6. Conclusions and Future Work

The need for a systematic approach for identifying relevant metrics has been emphasized before, and approaches such as Goal-Question-Metric (GQM) are proposed based on software improvement goals. These approaches are useful when a methodology is applied to achieve certain quality goals and the effect is under evaluation. However, they do not guarantee defining relevant goals when evaluating a methodology. Selecting metrics from standards or earlier evaluations does not guarantee relevance either. The approach proposed in this paper – based on practices and promises – fills this gap and leads to a hypothesis-driven measurement approach by linking practices to the expected benefits or promises, and improving internal validity of empirical studies by identifying the right data to collect.

We applied the approach on two software methodologies; i.e. software reuse and Model-Driven Engineering (MDE); and showed that using the approach can help identifying relevant evaluation criteria. The approach is also applicable when comparing software methodologies by identifying relevant criteria for comparison; for example whether generated artefacts in a MDE approach have higher quality than those developed manually because of generation. The approach can be complementary to more general evaluation frameworks such as the Technology Acceptance Model (TAM) to identify methodology-specific metrics.

Like any other methodology, our methodology should be evaluated in practice. We propose applying the proposed approach in evaluating the approach itself. The main promise of our approach is facilitating identification of relevant evaluation criteria based on identifying the practices of a methodology. In the MODELPLEX project, we are concerned with developing MDE methodologies for modelling of complex software systems and evaluating these solutions. Earlier work on evaluating MDE has focused only on a few metrics, without describing the results by MDE practices and their operationalization. The evaluation plan developed in the MODELPLEX project (see the project website) has focused on promises of MDE and contains most of the evaluation criteria for MDE that are discussed in this paper; identified by the project industrial partners depending on their context. We will also take advantage of TAM and combine evaluation from a MDE perspective with TAM criteria such as ease of use. The evaluation of our methodology will be done in the first step by collecting feedback on the evaluation plan and whether the identified metrics are useful for explaining usefulness of MDE and the future use intensions.

7. Acknowledgements

Part of the ideas presented in this paper are based on conclusions obtained in the context of the MODELPLEX project (IST-FP6-2006 Contract No. 34081), co-funded by the European Commission as part of the 6th Framework Programme. (cf.

http://www.modelplex.org) and the “Quality in Model-Driven Engineering” project at SINTEF (cf. http://quality-mde.org/).

References

[1] H. Al-Kilidar, K. Cox, B. Kitchenham, The use and usefulness of the ISO/IEC 9126 quality standard, 2005 *International Symposium on Empirical Software Engineering*, 7 p.

[2] E. Arisholm, H. Gallis, T. Dybå, D.I.K. Sjøberg, Evaluating pair programming with respect to system complexity and programmer expertise, *IEEE Trans. Soft. Eng.* 33(2), 65-86, 2007.

[3] P. Baker, P.S. Loh, F. Weil, Model-driven engineering in a large industrial context - Motorola case study. *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005), Springer LNCS 3713,* 476-491, 2005.

[4] J. Bansiya, C.G. Davis, A hierarchical model for object-oriented design quality assessment. IEEE Trans. Soft. Eng. 28(1), 4-17, 2002.

[5] V.R. Basili, D. Weiss, A methodology for collecting valid software engineering data, *IEEE Trans. Soft. Eng.* 10(6), 728-738, 1984.

[6] V.R. Basili, *The Role of Empirical Study in Software Engineering*, URL http://www.cs.umd.edu/~basili/presentations/2006/Role%20of%20E%20in%20SE%20Irvine.pdf

[7] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, P. Grünbacher, *Value-Based Software Engineering*, Springer, 2006.

[8] B. Burgstaller, E. Wuchner, L. Fiege, M. Becker, T. Fritz, Using domain driven development for monitoring distributed systems. *1st European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA’05), LNCS 3748*, 19-24, 2005.

[9] J. Bøegh, A new standard for quality requirements, *IEEE Software* 25(2), 57-63, 2008.

[10] *Modelware D5.3-2, Enabler ROI, Assessment, and Feedback*, URL: http://www.modelware-ist.org/, 2006.

[11] *Modelware D5.3-4, France Telecom ROI, Assessment, and Feedback.* Revision 1.1, URL: http://www.modelware-ist.org/, 2006.

[12] F. Davis, Perceived usefulness, perceived ease of use, and user acceptance of information technology, *MIS Quarterly* 13(3), 318-339, 1989.

[13] T. Dybå, N.B. Moe, E.M. Mikkelsen, An empirical investigation on factors affecting software development acceptance and utilization of electronic process guides. *10th International Symposium on Software Metrics (Metrics’04)*, 220-231, 2004.

[14] T. Dybå, N.B. Moe, E. Arisholm, E., Measuring software methodology usage: challenges of conceptualization and operationalization, 2005 *International Symposium on Empirical Software Engineering*, 11 p.

[15] T. Dybå, B. A. Kitchenham, M. Jorgensen, Evidence-based software engineering for practitioners, *IEEE Software* 22(1), 58-65, 2005.

[16] *Extreme Programming, a gentle introduction*. http://www.extremeprogramming.org/

[17] B. Farbay, A. Finkelstein, Evaluation in software engineering: ROI, but more than ROI. *3rd International Workshop on Economics-Driven Software Engineering Research (EDSER-3)* at ICSE 2001, online at http://www.cs.ucl.ac.uk/staff/A.Finkelstein/papers/edser3eval.pdf

[18] N.E. Fenton, S.L. Pfleeger, *Software Metrics. A Rigorous & Practical Approach*. Thomson Computer Press, 2nd edition, 1996.

[19] M. Glinz, A risk-based, value-oriented approach to quality requirements. *IEEE Software* 25(2), 34-41, 2008.

[20] E.R. House. *Evaluating with validity*, Sage Publications, 1980.

[21] *Software engineering -- Product quality -- Part 2: External metrics*, ISO 9126-2, http://www.iso.org/

[22] D.S. Janzen, H. Saiedian, Does test-driven development really improve software design quality? *IEEE Software* 25(2), 77-84, 2008.

[23] H. Jonkers, M. Stroucken, R. Vdovjak, Bootstrapping domain-specific model-driven software development within Philips. *6th OOPSLA Workshop on Domain Specific Modeling (DSM’06)*, 10 p, 2006.

[24] P.G.W. Keen, Value analysis: justifying decision support systems, *MIS Quarterly* 5(1), 1-15 (1981)

[25] B. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, *IEEE Trans. Soft. Eng.* 28(8), 721-734, 2002.

[26] J. Krogstie, Evaluating UML using a generic quality framework", chapter in *UML and the Unified Process*, Idea Group Publishing, 1-22, 2003.

[27] M. Li, C.S. Smidts, A ranking of software engineering measures based on expert opinion. *IEEE Trans. Soft. Eng*. 29(9), 811-824, 2003.

[28] A. MacDonald, D. Russell, B. Atchison, Model-driven development within a legacy system: an industry experience report. *Australian Software Engineering Conference (ASWEC'05)*, 14-22, 2005.

[29] A. Mattsson, B. Lundell, B. Lings, B. Fitzgerald, Experiences from representing software architecture in a large industrial project using model driven development. *2nd Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI'07) at ICSE 2007, IEEE Press*, 6 p., 2007.

[30] S.J. Mellor, M.J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[31] *Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach. Productivity Analysis.* Report by the Middleware Company on behalf of Compuware, URL: http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf, 2003.

[32] N.B. Moe, T. Dybå, The use of an electronic process guide in a medium-sized software development company. *Software Process Improvement and Practice* 11 (2006), 21-34.

[33] P. Mohagheghi, R. Conradi, Quality, productivity and economic benefits of software reuse: a review of industrial studies, *Empirical Software Engineering* 12(5), 471-516 (2007)

[34] P. Mohagheghi, V. Dehlen, Where is the proof? - a review of experiences from applying MDE in industry. *European Conference on Model Driven Architecture Foundations and Applications (ECMDA 2008), Springer LNCS 5095,* 432-443, 2008.

[35] L. Safa, The practice of deploying DSM, report from a Japanese appliance maker trenches. *6th OOPSLA Workshop on Domain Specific Modeling (DSM'06)*, 12 p., 2006.

[36] D. Shirtz, M. Kazakov, Y. Shaham-Gafni, Adopting model driven development in a large financial organization. *3rd European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA'07), Springer LNCS 4530*, 172-183, 2007.

[37] Six Sigma website, last visited in May 2008, http://www.isixsigma.com/library/content/c011008a.asp

[38] I. Solheim, T. Neple, Model quality in the context of model-driven development, *2nd International Workshop on Model-Driven Enterprise Information Systems (MDEIS'06)*, 27-35, 2006.

[39] M. Staron, Adopting model driven software development in industry- a case study at two companies. *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006), Springer LNCS 4199*, 57-72, 2006.

[40] *Cost, quality, time – choose two. Factors that drive product development.* URL: http://www.stdtime.com/whitepapers/cost-quality-time-choose-two.pdf.

[41] B. Trask, D. Paniscotti, A. Roman, V. Bhanot, Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, 846 – 853, 2006.

[42] A. Ulrich, A. Petrenko, Reverse engineering models from traces to validate distributed systems- an industrial case study. *3rd European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA'07), LNCS 4530*, 185-193, 2007.

[43] H. Wegener, Balancing simplicity and expressiveness: designing domain-specific models for the reinsurance industry. *4th OOPSLA Workshop on Domain Specific Modeling (DSM'04)*, 12 p., 2004.

[44] T. Weigert, F. Weil, Practical experiences in using model-driven engineering to develop trustworthy computing systems. *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, 208-217, 2006.

[45] C. Wohlin, P. Runneson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, *Experimentation in Software Engineering. An Introduction*. Kluwer Academic Publishers, 2000.