Quantifying the Quality of Object-Oriented Design: the Factor-Strategy Model

Radu Marinescu Daniel Raţiu
LOOSE Research Group
"Politehnica" University of Timişoara
{radum, ratiud}@cs.utt.ro

Abstract

*The quality of a design has a decisive impact on the quality of a software product; but due to the diversity and complexity of design properties (*e.g.*, coupling, encapsulation), their assessment and correlation with external quality attributes (*e.g.*, maintenance, portability) is hard. In contrast to traditional quality models that express the "goodness" of design in terms of a set of metrics, the novel* Factor-Strategy *model proposed by this paper, relates explicitly the quality of a design to its conformance with a set of essential principles, rules and heuristics. This model is based on a novel mechanism, called* detection strategy*, that raises the abstraction level in dealing with metrics, by allowing to formulate good-design rules and heuristics in a quantifiable manner, and to detect automatically deviations from these rules. This quality model provides a twofold advantage: (i) an easier construction and understanding of the model as quality is put in connection with design principles rather than "raw numbers"; and (ii) a direct identification of the real causes of quality flaws. We have validated the approach through a comparative analysis involving two versions of a industrial software system.*

Keywords: *Quality Model, Object-Oriented Metrics, Quality Factors, Design Principles*

1. Introduction

In the last decades the demand for quality in software products has been increasingly emphasized. But, as De Marco pointed out the only way we can control quality is by measuring it, by making quality quantifiable [6]. But in spite of the wide consensus on the previous statements about the need for quality in software, the issues related to the quantification of quality are still far from being solved [1].

Object-oriented design claims to support essential software quality goals like maintainability and reusability [21, 4] by mechanisms like encapsulation of data, inheritance and dynamic binding. But nowadays, the software industry is confronted with a large number of legacy software systems that lack all of the aforementioned qualities: they are instead inflexible and hard to reuse [19, 10]. The reason is that object-oriented programming is a basic technology, that supports the aforementioned quality goals but just knowing the syntax elements or concepts of the object-oriented technology is not sufficient to produce good software [23]. A good object-oriented design needs design rules and practices that must be known and used. Their violation will eventually have a strong impact on the higher-level quality attributes. But as mentioned earlier, quality must be expressed in a quantified manner.

The migration to object-oriented programming has raised the demand for finding adequate measures that would quantify the mechanisms of this paradigm as encapsulation and polymorphism. This has led to the definition of a large number of object-oriented metrics [15, 12]. However, there are a number of problems related to the interpretation of metrics, that have an important negative impact in the context of quality assessment:

- There is a large gap between design principles and design metrics. Thus, there is still an important gap between what we measure and what is important in design.
- There is a lack of relevant feedback link in quality models. We apply metrics we identify suspects, but the metric by itself does not provide enough information for a transforming the code so that it would improve quality. For example, what about a method with more than 1200 LOC? Should it be split, or could a part of it be factored out in a base-class, or should the method (or a part of it) be moved to another class? If the metric is considered in isolation it is hard to say. Thus, the developer is provided only with the problem and he or

Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)
1095-1350/04 $20.00 © 2004 **IEEE**

she must still empirically find the real cause and eventually look for a way to improve the design.

In this paper we propose a novel approach to the issue of quantifying the impact of object-oriented design on the high-level quality factors of software, like maintainability or portability. In order to bridge the gap between qualitative and quantitative statements relate to object-oriented design we propose a quality model that has two major characteristics: (i) an easy and intuitive construction; (ii) a direct link at the design level to the cause(s) and location(s) of quality problems.

We are aware of the fact that high-level quality factors are influenced also by other criteria (*e.g.*, technologies involved, selection of algorithms or database schemas) than the design structure. Because of that, we focus our attention consciously on those aspects of quality that are heavily impacted by design problems, especially on maintainability. We would also like to emphasize that the approach is language-independent, while the toolkit proposed for automation currently supports the JAVA and C++ languages.

The paper is structured as follows: in the next section we discuss the major limitations of the traditional Factor-Criteria-Metrics quality models for the assessment of object-oriented design quality and relate our work to other current approaches. We then introduce *detection strategies* [18], a mechanism that supports the formulation of good-design rules and heuristics in a quantifiable manner. Based on it, we define in Section 4 the new *Factor-Strategy* model together with its quantification mechanism and a stepwise construction methodology. Next, we present our tool support and a validation of the model on two case studies. The paper is concluded with a discussion on the obtained results and an outlook on future work on this topic.

2. Related Work

In this section we discuss the decompositional approach to quality models, known as the *Factor-Criteria-Metric*, and point out the limitations in applicability of this approach together with a very recent contribution that addresses these limitations.

2.1. Factor-Criteria-Metric Models

The most well-known quality models are based on the decompositional approach used by both McCall [20] and Boehm [3], commonly known as the *Factor-Criteria-Metric* quality model (FCM).

FCM models are usually constructed in a tree-like fashion, where the upper branches hold important high-level *quality factors* related to software products, such as reliability and maintainability, which we would like to quantify. Each quality factor is composed of lower-level *criteria*, such as structuredness and conciseness. These criteria are easier to understand and measure than the factors themselves, thus actual metrics are proposed for them. The tree describes the relationships between factors and criteria, so we can measure the factors in terms of the dependent criteria measures (*e.g.*, the "structurednes" criteria can be associated with a measure of class cohesion, one measuring the complexity of methods, and a third one measuring the coupling to other classes). This notion of divide-and-conquer has been implemented as a standard approach to measuring software quality [22].

2.2. Limitations of FCM Quality Models

Although this approach is cited throughout the whole software engineering literature and is implemented in several commercial CASE tools it has two main drawbacks that limit its usability. These limitations are discussed through the rest of this section.

Obscure mapping of quality criteria onto metrics. When analyzing different FCM models the first question that pops-up is: how are the quality criteria mapped to metrics? The answer to this question is essential because it affects the usability and reliability of the whole model. In the FCM approach this explicit mapping between quality criteria on one hand and rules and principles of design and coding on the other hand implicitly (and obscurely) contained in the mapping between the quality criteria and the quality metrics. Thus, the answer to the previous question is: quality criteria are mapped into metrics based on a set of rules and practices of good-design. But this mapping is "hidden" behind the arrows that link the quality criteria to the metrics, making it in most of the cases impossible to trace back. This observation reveals a first important drawback of the FCM quality models: if the model is a fixed one, it will be *hard to understand*, because we can only guess what are the rules and principles that dictated the mapping. In case of a "user-defined" model, the model is *hard to define* because when we mentally model quality we reason in terms of *explicit* design rules and heuristics, keeping the quality criteria *implicitly* contained in the rules.

Poor capacity to map quality problems to causes. The interpretation of a quality model, must be done in terms of: *diagnosis i.e.*, what are the design problems that affect the quality of my software?, *location i.e.*, where are the problems located in the code? and *treatment i.e.*, what should be changed at the design level, to improve the quality?

When analyzing a software system using a FCM model,

we get the quality status for the different factors that we are interested in (*e.g.*, the maintainability is quite poor, while portability stays at a fair level). We are also able to identify a set of design fragments that are supposed to be responsible for a poor status of a certain quality factor. Thus, FCM solves both the diagnosis and the location issues. But as soon as we arrive at the question concerning the treatment, we reached the limits of the FCM model, because the model doesn't help us find the *real causes* of the quality flaws detected by it. The cause of this is the fact that abnormal metric values – even if the metrics are provided with a proper interpretation model – are just *symptoms* of a design or implementation *disease* and not the disease itself. A treatment can only be imagined when knowing the *disease* not only a set of *symptoms*.

2.3. Improved Approaches

2.3.1. The QMOOD Quality Model Bansiya and Davis [1] correctly emphasize the incapacity of traditional approaches to be proper quantification means for the quality of an object-oriented design. Consequently, based on the framework for quality models defined in [7, 8], they propose QMOOD, a hierarchical quality model dedicated to the assessment of object-oriented design quality. A very good contribution of this paper is the identification of the specific design properties for the object-oriented paradigm (*e.g.*, polymorphism, data abstraction, hierarchies). The paper also introduces a very interesting set of new object-oriented metrics. But the major drawback is that QMOOD still relates design properties directly with metrics, in the same manner criteria are mapped to metrics in the FCM model. Thus, again design characteristics are related explicitly to "raw numbers" (*i.e.*, metrics), while the rules and principles of good-design that determined the mapping remain implicit.

2.3.2. A Hybrid Approach to Quality Models. In [24] the authors criticize the FCM models and propose an improvement to such predictive quality models. The authors emphasize that the use of precise threshold values and their interpretation in the absence of formal models, as well as the crudeness of the derived rules which can only serve to build naïve models are the two diseases of current approaches for building predictive models. They propose a novel approach by building fuzzy decision processes that combine both software metrics and heuristic knowledge [9] from the field. The authors claim that this hybrid approach would improve efficiency of quality prediction and provide a more comprehensive explanation of the relationship that exists between the observed data and the predicted software quality characteristics. A critical view on the paper reveals that while the fuzzification of threshold values seams applicable and well founded, the second part containing decomposition of heuristic knowledge is far from being traceable. In addition to that, no case studies are provided in the paper so that the practicability of the approach is not yet proved. In conclusion, we believe that this approach will become useful in the future especially on the side of a proper parametrization of the interpretation models of the metrics. Yet it does not offer a comprehensive approach for an improved quality model.

3. Detection of Design Flaws

The main issue in working with metrics is how should we deal with measurement results? How can all those numbers help us to improve the quality of the software? As mentioned in the introduction, many times a metric alone cannot help very much in answering this question and therefore metrics must be used together in order to make them efficient. For this purpose we introduced *detection strategies*[17, 18] as a generic mechanism for analyzing a source code model using metrics. Its main goal is to raise the abstraction level in dealing with metrics, by allowing to formulate good-design rules and heuristics in a quantifiable manner, and to detect directly deviations from these rules.

3.1. Structure of a Detection Strategy

The use of metrics in the detection strategies is based on mechanisms of **filtering** and **composition**.

Filtering Operators. A filtering operator is a statistical means by which a *subset* of the measurement results is extracted based on the particular focus of the measurement, in the context of the detection strategy. For example, if our goal is to detect design problems we use them in order to capture those program elements (*i.e.*, methods, classes, subsystems) that have abnormal values for a given metric. For the moment we considered using the following set of filtering mechanisms:

- **Thresholds** – `HigherThan` ; `LowerThan`. These filtering mechanisms can be parameterized with a numerical value, representing the threshold. They are well known and widely used in connection with metrics and therefore we will not detail them any further.
- **Extremities** – `TopValues` ; `BottomValues`. These filters are useful in contexts where rather than indicating precise thresholds we would like to see on the highest (or lowest) values from a data set. The filters above can be parametrized using *absolute* (*e.g.*, "get the 20 entities with the highest values") or *percentile* (*e.g.*, "get the 10% of all measured entities having the lowest values) parameters.

Composition Operators. In a detection strategy we usually need more than one metric and one filtering mechanism. Thus, the strategy is built as a *composition* of metrics and filtering mechanisms. The operators by which the rule is "articulated"(composed), are called **composition operators**. A detection strategy uses three operators: `and`, `or` and `butnotin`.

3.2. Detection Strategy. An Example

Let's assume that we want to find the poor encapsulated classes *i.e.*, those classes that exhibit their data in the interface. For this we use two metrics: the first one to count the number of public attributes (NOPA) and the other one to count the number of accessor methods (NOAM) [16]. We decide that the classes we want to find are those with the most public data from the project, but that they should not have less than 3 public attributes or 5 accessor-methods. Therefore, the detection strategy can be expressed as follows[1](see Section 5):

```
PoorEncapsulatedClasses :=
   (NOPA, HigherThan(3) and NOPA, TopValues(10%))
or (NOAM, HigherThan(5) and NOAM, TopValues(10%))
```

As you might have already noticed, the most sensitive part in a detection strategy is the selection of threshold values. Although this is not discussed in this paper, we addressed this issue in [18, 17].

3.3. A Suite of Detection Strategies

In order to support the assessment of quality we defined a suite of detection strategies, that either quantify deviations from different good-design rules found in the literature [19, 23] or they identify "bad smells" of design [10]. As a result of this process we successfully quantified around 15 such design problems. Due to the space limitations of this paper we cannot provide a full description of all these detection strategies, but this is available in [17]. In Figure 1 we summarized the detection strategies defined in the context of quality assessment. The flaws are classified in conformity with the design entities that they affect *i.e.*, methods, classes and subsystems. In addition to these we introduced a fourth category: *micro-design flaws*. Design flaws that fall in this category affect not only a class, but a cluster of classes (or subsystems). In this category we included those cases where a particular design pattern [11] should have been applied, but the pattern solution was ignored. Each of the design flaws in Figure 1 is put in relation with the four criteria of good design identified by Coad and Yourdeon in [5]. These are: *low coupling* (COUPL), *high cohesion* (COHES), *manageable complexity* (COMPLX) and *proper data abstraction* (ENCAPS). This mapping will help us in constructing the *Factor-Strategy* quality model (see Section 4.2.2).

Category	Name	Source	Impact on:			
			COUPL	COHES	COMPLX	ENCAPS
Class	GodClass	[10, 23]	X	X	X	
	DataClass	[10, 23]				X
	ShotgunSurgery	[10]	X			
	RefusedBequest	[10]		X	X	
	ISPViolation	[19]	X			
Method	GodMethod	[10]			X	
	FeatureEnvy	[10]	X	X		X
	TemporaryField	[10]			X	
Subsystem	GodPackage	[19]	X			
	MisplacedClass	[19]		X		
Micro-Design (missing patterns)	LackOfBridge	[11]	X		X	
	LackOfStrategy	[11]		X	X	
	LackOfState	[11]			X	
	LackOfSingleton	[11]	X			X
	LackOfFacade	[11]	X			

Figure 1. Overview of design flaws

4. Factor-Strategy Model

Based on the *detection strategy* mechanism we propose a new type of quality model, called **Factor-Strategy**(FS). This approach is intended to improve the FCM paradigm with respect to the two major drawbacks discussed in Section 2.2.

In Figure 2 we illustrate the concept of a Factor-Strategy model. FS models still use a decompositional approach, but after decomposing quality in factors, these factors are not anymore associated directly with a bunch of numbers, which proved to be of a low relevance for an engineer. Instead, quality factors are now expressed and evaluated in terms of detection strategies, which are the quantified expressions[2] of the good-style design rules for the object-oriented paradigm.

Therefore we may state in more abstract terms that *in a Factor-Strategy model, quality is expressed in terms of principles, rules and guidelines of a programming paradigm.* The set of detection strategies defined in the context of a FS quality model encapsulate therefore the *knowledge-box of good design* for the given paradigm. The larger the knowledge-box, the more accurate the quality assessment is. In our case the detection strategies are defined for the object-oriented paradigm, and thus in the right side of Figure 2 we depicted a sample of a *knowledge-box* of object-oriented design. The knowledge-box, as such, is indispensable for any quality model. Although not visible at first

1 The sequence is written in SOD, a script language defined as part of our toolkit for implementing detection strategies

2 The acronyms that appear in the ovals on the right side of Figure 2 represent metrics, and the arrows show which metric appears in which detection strategy (rectangles). The concrete metrics are not relevant for the understanding of this picture

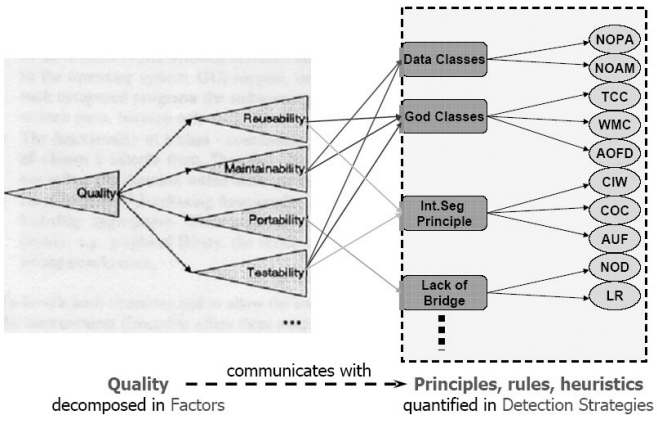

Figure 2. Factor-Strategy model.The concept

sight, it is also present in the FCM models. The knowledge-box is not obvious in the FCM approach because of its *implicit character*, while it becomes *explicit* in the FS model.

4.1. Stepwise Construction Methodology

The main issue in constructing any type of quality model is how to build the *association* between the higher and the lower levels of the model *e.g.*, in building a FCM model we are concerned with associating the quality factors with the proper criterion, or how to choose the metrics for a given criterion. As the Factor-Strategy models are also based on a decompositional approach the association is still the relevant issue. Based on the previous considerations, we identify two distinct aspects on this matter of association: a *semantical* and a *computational* aspect.

- The *semantical aspect* must sustain the validity of choosing a particular decomposition for a higher-level element into lower-level ones. In other words, it must explain the rationale behind the association *i.e.*, **why** *and* **how** *do we choose a particular decomposition for a higher-level element of the model?*
- The *computational aspect* must tackle the issue of quantifying the association *i.e.*, how the quality score for the higher-level element is to be computed from the quality scores of the lower-level elements associated with it.

Obviously, we must first define an association that "stands" semantically, and only then the focus must be set to finding the *association formula* that quantifies it. The association formula must reflect the "participation level" of each lower-level element within the higher-level aspect of quality. In constructing a Factor-Strategy model there are two association that must be done: the decomposition of the quality *goal* in quality *factors* and the association between these factors and detection strategies that can detect design flaws that affect the given quality factor.

4.1.1. Decomposition of the Quality Goal in Factors There are two possible approaches to address the semantical aspect of the association between a quality goal and a set of quality factors: we can either rely on a *predefined decomposition* found in the literature or go for a *user-defined one*. The former option has the advantage of a wider acceptance, while the latter is more flexible and adaptable to the particular investigation needs. In this paper we rely on an existing and widely accepted decomposition *i.e.*, the one found in the ISO9126 standard [22]. For the general case we recommend using a hybrid solution: start from a predefined decomposition found in the literature that comes closest to your ideal model and then slightly customize it until it matches your perspective on quality.

This association is orthogonal to the programming paradigm used for the design and implementation of the system. The decompositions found in literature, in spite of many differences, keep the higher level of quality decomposition [3], abstract enough to make it independent of the development paradigm. As a consequence, in a FS model the decomposition of a quality goal in factors is not different in any aspect to that found in the FCM approach. Therefore, the computational aspect of this association does not raise additional discussions at the conceptual level.

4.1.2. Association of Factors with Detection Strategies Detection strategies used in FS models capture deviations of a design from design rules and guidelines. In [18] we have described in detail the process of transforming informal design rules into detection strategies. The process of identifying the metrics needed to a quantify a given rule and the way the metrics are correlated to capture that precise aspect is done very much like in the "Goal-Question-Metric" approach [2]. The authors of such good-design rules implicitly or explicitly relate them to quality factors [23, 19, 10], or to abstract design principles [21, 13] (*e.g.*, abstraction, modularity, simplicity) that can be easily mapped to quality factors. Therefore, the *semantical aspect* of the association between quality factors and detection strategies becomes self-evident in the FS approach. This is one of the main advantages of the FS models over the FCM approach, where the correspondent association is subject to severe drawbacks, as we have already pointed out in Section 2.2.

There are two actions related to the *computational aspect* of the association between factors and their set of strategies:

3 By intention, we didn't name this level the *factors level* because some authors refer to it as the criteria level. In order to avoid confusion in terminology we referred to it as the "higher-level of quality decomposition".

1. *Compute a quality score for each detection strategy.* We have defined two mechanisms for computing a quality score from the results of a detection strategy: first, a formula to compute the *raw-score* must be established (*i.e.*, based on the number of suspects); second, we need a *matrix of ranks* based on which the computed *raw-score* is transformed into a *quality score*, like a school grade. In other words, using this matrix we place the *raw-score* in the context of quality *i.e.*, the matrix of ranks tells us how good or bad a raw-score is with respect to the quality factor. In Section 4.2.2 (Step 2) we provide a concrete example of such a *ranks matrix*.

2. *Compute a quality score for the quality factor.* This score is also computed based on an algorithm that again involves two mechanisms: first, an *association formula* in which the operands are the *quality scores* computed for the strategies; second, a *matrix of ranks* that transforms the raw-score for the quality factor into a quality score.

In the next section (4.2) these computation mechanisms will be illustrated as we build a concrete FS model for the assessment of maintainability.

4.2. A Factor-Strategy Model for Maintainability

We want to open this section with a disclaimer: the quality model that we are going to present below raises no claim of completeness. Moreover, we believe that a complete and universally acceptable quality model is impossible to define at least because of the following reasons: (i) there is no objective argument for adding or removing a component from the model [14]; (ii) the "knowledge-box" used in the model – *i.e.*, the detection strategies defined for the model – is limited and we see no possibility of claiming completeness in this aspect.

In the context of this disclaimer, there is still a twofold motivation for proposing a concrete model in this paper: First, the concrete model illustrates the steps and mechanisms involved in the construction of a FS quality model. Thus, it will illustrate how metrics are encapsulated in detection strategies and how quality factors are associated with these strategies that quantify deviations from good design rules. Second, based on this concrete model we will show how the drawbacks of the FCM approach are eliminated, by evaluating its applicability on a concrete case-study (Section 6)

For describing FS quality models, we defined a simple description language, called QMDL[4]. The language is inspired from the description language used in a commercial quality assessement tool [25] to describe FCM quality. The decision to define QMDL as a variant of a description language used in connection with FCM quality models was deliberate, as we believe that this would simplify the understanding of both the commonalities and the differences between the FCM and the FS approach.

4.2.1. Decomposing Maintainability in Quality Factors In conformity with the ISO-9126 standard [22] maintainability is decomposed in four factors: analysability, changeability, stability and testability. Because we want to weight equally the four factors in the evaluation of maintainability, we will use the average value of the scores computed for each quality factor. The association formula for maintainability is expressed as follows:

```
Maintainability := avg(Changeability, Testability,
                       Analysability, Stability)
```

Obviously, any other mathematical formula might have been used depending on the special emphasis of the quality evaluation. For example, if the emphasis would have been on the *analysability* aspect of maintainability, the previous formula could have been replaced by an weighted average:

```
Maintainability := (Changeability + Testability +
                    3*Analysability + Stability) /6
```

4.2.2. Associating Factors with Detection Strategies We briefly illustrate the process of association between a factor and a set of strategies using the *Stability* factor, which is in this model an aspect of maintainability. Stability is defined in ISO 9126 [22] as the *"attributes of software that bear on the risk of unexpected effect of modifications"*.

In conformity with the definitions of the design flaws that are detectable using the current set of detection strategies, and based on their impact on the desirable design properties (see Figure 1) we have selected the strategies associated with five of these flaws *i.e.*, those that affect directly stability. These are: *GodClasses, ShotgunSurgery, DataClasses, GodMethod* and *LackOfState* [5]. After the semantical association between the *Stability* factor and the strategies, we focus on the computational aspect, using the following sequence of steps:

Step 1: *Choose a formula for computing the* **raw-score** *for each strategy.* In our case we have used for all the strategies the simplest formula, *i.e.*, the raw score is the *number of suspects* detected by that strategy.

Step 2: *Choose an adequate* **matrix of ranks** *for each strategy*. We used three levels of tolerance in ranking the raw-scores for the strategies, and consequently we defined three matrices: a severe one (*SevereScoring*), a permissive one (*TolerantScoring*) and one in between the two (*MediumScoring*). For the design flaws that in our view had the highest impact on stability we applied the *SevereScoring* matrix of ranks.

4 QMDL stays for **Q**uality **M**odels **D**escription **L**anguage

5 The detection strategies for these design flaws were introduced in [17]

For example, we believe the design flaws that affect stability in the highest degree are *ShotgunSurgery* and *GodClasses*. Thus, while for the other design flaws we used the *MediumScoring* matrix, for the aforementioned two flaws we computed their quality score using the *SevereScoring* matrix, which is defined as follows:

```
SevereScoring {
  0   0   10,   /* EXCELLENT  */
  1   1   9,    /* VERY GOOD  */
  2   4   7,    /* GOOD  */
  5   7   5,    /* ACCEPTABLE */
  8   +oo 3     /* POOR */
},
```

Note that any *matrix of ranks* has three colums: the first two columns define the range (upper and lower limits) of the *raw score*, while the last column is the "mark"[6] that corresponds to that range. For example, the third line of the previous matrix is interpreted as follows: we grant a 7 mark for a system that has between 2 and 4 classes affected by the design flaw to which the matrix is attached (in this case *ShotgunSurgery* and *GodClasses*). The number of lines of such a matrix is dependent on the engineer who defines the model, and it could any value higher than 2 (*i.e.*, differentiate only between good and bad). Yet, we believe that in practice the number should not exceed 5.

Step 3: *Define a formula for computing the* **raw score for the factor**. Because we intended to weight equally the five strategies when computing a score for stability, we used the *average* function(*avg*). Throughout the model we applied the same function for computing the raw-scores for quality factors.

Step 4:*Choose the* **matrix of ranks** *for computing the quality score for the factor*. The *raw-score* computed during the previous step must also be placed in a matrix of ranks in order to retrieve a normalized quality score. Having reached the last step, we can "reveal" how *Stability* is quantified (in QMDL):

```
Stability := avg(ShotgunSurgery(SevereScoring),
                 GodClasses(SevereScoring),
                 DataClasses(MediumScoring),
                 GodMethod(MediumScoring),
                 LackOfState(MediumScoring)
    {
          9   10  10,  /* EXCELLENT  */
          7   9   8,   /* GOOD  */
          5   7   6,   /* ACCEPTABLE */
          0   5   4    /* POOR */
    };
```

5. Tool Support

On large-scale software systems any quality model is useless without an adequate toolkit that supports it automatic assessment. Therefore, we developed the PRODETECTION toolkit that support quality-driven code inspections based on detection strategies. PRODETECTION can be used for the analysis of object-oriented systems written in C++ and Java. The inspection process, consists of the following activities:

1. Construction of the System Model. All design information needed in order to compute metrics on a given software system, is stored in a *meta-model*. The meta-model captures information about the design entities (*e.g.*, classes, methods, variables) and about the main existing relations (*e.g.*, inheritance, cross-referencing) among these entities.

2. Executing Detection Strategies. A detection strategy implemented can be automatically run using the PRODEOOS tool on the design model of the system to be analyzed. The result is a set of design entities that are reported at suspect for that particular detection strategy.

3. Evaluation of the Quality Tree. Eventually, based on the results of detection strategies, PRODEOOS computes the FS quality models. As a first result, the engineer first gets a score for the quality of the evaluated attribute. The tool also allows the engineer to browse through the quality tree and inspect all the intermediary scores down to the detection strategies. This way it becomes easy to trace back the causes of a possible low score. So in applying a given FS quality model everything is automatized. Obviously the construction and adaptation of the model (*e.g.*, customizing the matrixes of ranks) is still to be done manually. Yet, as we emphasized over and over the construction of the quality model is facilitated by the intuitiveness of putting in relation quality factors with violations of design rules and heuristics.

6. Evaluation of the Approach

In this section we will describe the approach used for evaluating the *Factor Strategy* quality model. We first describe the two systems involved case-study as their characteristics play an important role in the evaluation approach. Next, the evaluation assumptions and the approach itself are introduced. The last part of the section is dedicated to the presentation and extensive discussion of results.

6.1. Case-Study

Detection strategies and the Factor-Strategy quality model have already been applied successfully in the past on multiple large industrial case-studies, in the size of 700 KLOC up to 2000 KLOC especially on software for telecommunication. In this paper we present the results obtained on two successive versions of a medium size business application related to computer-aided route planning.

6 We chose a score (ranks) scale with the two limits being 1 (worst) and 10 (best) in order to enhance our intuitive understanding of the quality scores, as the in our country we use in schools the same scale for marks. Of course, this "intuition aid" has only a regional applicability; yet, the idea that stays behind the scale selection is reusable.

The size characteristics of the two systems[7] is summarized in Table 1. Concerning the relation between the two

System	KLOC	Packages	Classes	Methods
SV1	93	18	152	1284
SV2	115.6	29	387	3446

Table 1. The case-study in numbers

systems we know that the second one is a re-engineered and enhanced version of the first system. We also know from the designers and developers of the system that the design was substantially improved at the design level in order to increase its *maintainability*. Notice the fact that the number of classes has doubled in SV2. This reveals that intentions of the refactorings operated on SV1 in order to improve its quality: they are justified by the need to "prepare the ground" for a massive functional extension.

6.2. Evaluation Approach

The evaluation approach is based on two assumptions, derived from the aforementioned observations:

Assumption 1 : All major design problems that troubled the developers in the first version, were eliminated during the reengineering process, and consequently will not be found anymore in the second version. We do not assume that all the design problems have been eliminated, but we assume that most relevant problems have been dealt with.

Assumption 2 : The level of maintainability for the reengineered version of the system is higher (better) than the one of the initial version of the system, as the reengineering goal was to increase maintainability.

The evaluation approach for the *Factor-Strategy* approach to quality models is very simple. We take the entire quality model for assessing maintainability – partially described in Section 4.2 and fully presented in [17] – and apply it on the two versions of the system. As the second version was reengineered with the goal of eliminating maintenance difficulties (**Assumption 2**), we want to check if the *Factor-Strategy* quality model properly reflects the improvements. A second evaluation criterion is to see how the model can help us identify the major design flaws that made the difference in maintainability between the two versions.

7 Through the rest of this section, we will designate the two versions of the system as SV1 and SV2, standing for *System Version 1* and *System Version 2*.

Relevance of the Approach. The experimental approach that we are going to use is relevant for the evaluation of the approach defined by this paper because of a number of reasons, enumerated in the following:

1. Analyzing a large-scale industrial case-study gives us the opportunity to evaluate the *scalability* of the approach.
2. By analyzing two successive versions of a system which are also part of a "before-and-after reengineering" scenario, allows us to set up an evaluation methodology that assesses the *accuracy* of the detection strategies. Thus, based on **Assumption1** we can automatically identify the *false positive* suspects *i.e.*, entities that were erroneously reported as flaws by a given detection strategy.
3. Taking advantage of the supplementary information that the goal of the original reengineering process was to improve maintainability, we can evaluate the *accuracy* and *relevance* of the information provided by the *Factor-Strategy* quality model for maintainability, defined in Section 4.2.

6.3. Results Summary

The results of applying the FS quality model for maintenance on the two versions of the system are synthesized in Figure 3. The numbers in the two "Score" columns are given

		SV1		SV2	
Quality Goal	Factor	Score	Qualifier	Score	Qualifier
Maintainability		6,95		8,07	
	Changeability	7,58	8	8,33	8
	Analyzability	6,67	6	8	8
	Testability	6,2	6	7,6	8
	Stability	7,33	8	8,33	8

Figure 3. Results for maintainability model.

by the number of suspects reported by each detection strategy, while the "Qualifier" is the corresponding score taken from the scoring table.

The first result that we notice is the difference of 1.12 points of the maintainability score, which indicates that the quality model captured a sensible improvement of the level of maintainability. More than that, for each of the four quality criteria, we can see that the model shows an improvement, varying from 0.75 points (for Changeability) up to 1.4 (for Testability) and 1.33 (for Analysability).

Analyzing in more detail the results for SV1 we notice that the lowest quality "marks" are associated with Analysability and Testability. We took a closer look to the

detection strategies associated with each quality factor in order to see which design flaw had the largest negative impact on the design quality. For the two aforementioned quality factors, the "diseases" that seem to be most widespread in the system are *Refused Bequest* (22 occurrences), *Shotgun Surgery* (15 or 33 occurrences, depending on the used variant) and *God Class* (5 occurrences). If we recall the semantics of these "bad smells" we identify easily the key problems of this system: an improperly designed class hierarchy (*Refused Bequest*) and an improper distribution of the system's complexity, which is reflected by the high dependency on some classes (*Shotgun Surgery*) and an excessive complexity centralization in some classes (*God Class*). Furthermore, a correlated analysis of the results has revealed that 4 of the "god classes" where also affected by *Shotgun Surgery*, which makes the negative impact on the maintenance of the system even worse.

6.4. Lessons Learned from the Case Studies

The first observation concerning the accuracy of the FS model on maintainability is that an increase of quality was "sensed" for all four quality factors. In addition to this, the fact that the highest improvements were registered for *Testability* and *Analysability* is a further positive signal concerning the accuracy of the model, as these two quality attribute play a key role for a system that is going to be extended with new functionality, like the analyzed system is. Thus, these results tend to confirm our first hypothesis *i.e.*, that FS quality models are efficient means for defining associations between external quality goals and the quality of the design structure.

Next, we wanted to identify what caused the difference between the maintainability scores of the two versions? For this, we went through the strategy-level of the quality-model and selected those with the most important score increases. The results are synthesized in Figure 4. These results help us drive immediate conclusions on the major design problems in SV1 that hindered its maintenance.

Strategy	Initial System (SV1)		Re-engineered System (SV2)		Improvement	
	Score	Qualifier	Score	Qualifier	Score	Qualifier
Wide Subsystem Interface	5	5	1	9	4	4
God Class	5	5	2	9	3	4
Refused Bequest	22	3	6	6	16	3
Data Class	3	7	2	9	1	2
Shotgun Surgery	15	3	7	3	8	0

Figure 4. The highest score improvements

The answer to the question above illustrates best the advantages of the Factor-Strategy approach: we do not receive – like in a usual FCM approach – a "bunch of numbers" together with statements like "the results for metric X and Y improved in the second version compared to the first one", which require further effort to be converted into useful information for an engineer. Instead, we can now reason and pick up conclusions *directly* from our quality model!

In this concrete case, when we analyze Figure 4 we can immediately drive the following conclusions. First, we notice that the main problems are related to the class and package design rather than methods implementation. We reached this conclusion, while observing the small number of *God Methods* (4), compared to the five *God Classes*. This contrasts with other systems that we analyzed, *God Classes* were the "cumulative" result of a large number of *God Methods*.

The problems at the package level were strongly related to the interface of the packages *i.e.*, the number of classes from outside the package directly using it (*Wide Subsystem Interface*). At the class level, the problems were basically related to the tendency to centralize the intelligence of the system in a small number of classes (*God Class*) that used several "dumb" data-holders (*Data Class*), an improper class hierarchy (*Refused Bequest*) and an increased coupling level between the classes (*Shotgun Surgery*)

The manual analysis of the SV1 system, together with further discussion with the developer concerning the initial weaknesses of the system fully confirmed the conclusions above.

7. Conclusions. Future Work

In this paper we presented a new approach to quality models, the *Factor-Strategy* models. We have shown that the gap between qualitative and quantitative statements, concerning object-oriented software design can be bridged. While we used *detection strategies* as a higher-level mechanism for measurement interpretation, the FS quality model provides a goal-driven approach for applying the detection strategies for quality assessment.

The *Factor-Strategy* approach has two major improvements over the traditional approaches: First, the *construction* of the quality model is easier because the quality of the design is naturally and explicitly linked to the principles and good-style rules of object-oriented design. Our approach is in contrast with the classical Factor-Criteria-Metric approach, where in spite of the decomposition of external quality factors into measurable criteria, quality is eventually linked to metrics in a way that is less intuitive and natural. Second, the *interpretation* of the strategy-driven quality model occurs at a higher abstraction level *i.e.*, the level of design principles, and therefore it leads to a direct identification of the real causes of quality flaws, as they are reflected in flaws at the design level. As we pointed out earlier, in this new approach quality is expressed and evaluated

in terms of an explicit knowledge-box of object-oriented design.

Besides the improvement of the quality assessment process, the detection strategies used in the context of a Factor-Strategy quality model proved to have a further applicability, at the conceptual level: for the first time a quality factor could be described in a concrete and sharp manner with respect to a given programming paradigm. This is achieved by describing the quality factors in terms of the detection strategies that capture design problems that affect the quality factor, within the given paradigm.

We have also shown based on two versions of an industrial case study that the FS quality model is usable in practice and provides us with information that is not only relevant for quality assessment, but also for the further improvement of the system's design. It is clear that the reduced number of case studies is by no means statistically relevant, but nevertheless it shows that the approach warrants further study and experimentation.

We will focus our future work on three main fronts: First, we intend to widen the evaluation approach in order to have a statistically relevant validation for the accuracy of the FS model for maintainability. We believe it is also necessary to conduct a more elaborate study on the relevance of the refactoring hints provided by the FS quality model. Second, we plan to research how detection strategies could return a more refined evaluation *i.e.*, by returning a number rather than a boolean, allowing us to know not only *if* but also *how much* is a design entity affected by a design problem. Based on this we want to research further quantification schemas within the FS quality model. Last but not least, we are concerned with the "instantiation" of the FS model for other quality goals than maintainability. In this context we intend also to address the issue of a *domain-specific* quality models (*e.g.*, models for accounting applications, frameworks, or highly interactive applications).

8. Acknowledgments

This work is supported by the Austrian Ministry BMBWK under Project No. GZ 45.527/1-VI/B/7a/02. We owe a lot to Tudor Gîrba for all the encouragement, fruitful discussions and for reviewing this paper. We would also like to thank Ciprian Chirilă and Petru Mihancea for helping us with a lot of the implementation effort. Last but not least we would like to thank the LOOSE Research Group (LRG) for being such a great team.

References

[1] J. Bansiya and C. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1), jan 2002.

[2] V. Basili and D. Rombach. The TAME project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Softw. Engineering*, 14(6), jun 1988.

[3] B. Boehm, J. Brown, and J. Kaspar. *Characteristics of Software Quality*. TRW Series of Software Technology, Amsterdam, North Holland, 1978.

[4] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, 2 edition, 1994.

[5] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice Hall, London, 2 edition, 1991.

[6] T. DeMarco. *Controlling Software Projects; Management, Measurement and Estimation*. Yourdan Press, 1982.

[7] G. Dromey. A Model for Software Product Quality. *IEEE Transactions on Software Engineering*, 21(2), febr 1995.

[8] G. Dromey. Cornering the Chimera. *IEEE Software*, 13(1), Jan 1996.

[9] N. Fenton and M. Neil. Software metrics: A Roadmap. In *ICSE - Future of SE Track*, pages 357–370, 2000.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] B. Henderson-Sellers. *Object-Oriented Metrics – Measures of Complexity*. Prentice-Hall,Sydney, 1996.

[13] R. Johnson and B. Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[14] B. Kitchenham and J. Walker. The meaning of quality. In *Conference on Software Engineering*, 1986.

[15] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, Englewood Cliffs, NY, 1994.

[16] R. Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In *Proceedings of TOOLS USA 2001*, pages 103–116. IEEE Computer Society, 2001.

[17] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Universitatea "Politehnica" Timişoara, http://loose.utt.ro/download/papers/thesis.zip, 2002.

[18] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *ICSM 2004 Proceedings, IEEE Computer Press*, 2004.

[19] R. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[20] J. McCall, P. Richards, and G. Walters. *Factors in Software Quality, Volume I*. NTIS Springfield, 1977.

[21] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[22] I. S. Organization. ISO 9126 - Quality Characteristics and Guidelines for Their Use. *Brussels*, 1991.

[23] A. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[24] H. Sahraoui, M. Boukadoum, and H. Lounis. Building Quality Estimation models with Fuzzy Threshold Values. *lObjet*, 17(4), 2001.

[25] Telelogic. *Telelogic Tau Logiscope 5.1. – Audit Basic Concepts*. Telelogic AB, Malmoe, Sweden, 2000.