Driving Component-Based Software Development through Quality Modelling

Colin Atkinson[1], Christian Bunse[2], and Jürgen Wüst[2]

[1] University of Mannheim, D-68131, Mannheim, Germany
colin.atkinson@ieee.org
[2] Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6, Kaiserslautern, D-67661 Germany
christian.bunse@iese.fhg.de

Abstract. With the advent of the OMG's new Model Driven Architecture (MDA), and the growing uptake of the UML, the concept of model-driven development is receiving increasing attention. Many software organizations have identified the MDA as being of strategic importance to their businesses and many UML-tool vendors now market their tools as supporting model-driven development. However, most UML tools today support only a very limited concept of model driven development—the idea of first creating platform independent models and then mapping them into executable code. In contrast, true model-driven development implies that the development flow of a project is in some way "driven" (i.e. guided) by models. Quality attributes of models (e.g., measures derived from structural attributes) could be used in this regard, but although many different types of measures have been proposed (e.g. coupling, complexity, cohesion) they are not widely used in practice. This chapter discusses the issues involved in supporting this more general view of model driven development. It first presents some strategies for deriving useful quality-related information from UML models and then illustrates how this information can be use to optimize project effort and develop high-quality components. We pay special attention to how quality modelling based on structural properties can be integrated into the OMG's Model Driven Architecture (MDA) initiative.

1 Introduction

Software organizations are attracted to Component-Based Development (CBD) [16] because it offers the prospect of significant improvements in productivity. By assembling new applications from pre-fabricated and pre-validated components, rather than building them from scratch using traditional developing techniques, organizations can achieve significant savings in cost and time. At present, however, industrial component-technologies such as COM+/.NET, EJB/J2EE and CORBA only support components in the final implementation and deployment stages of development, leaving analysis and design to be organized in traditional, non-component-oriented ways. This not only reduces the potential impact

A. Cechich et al. (Eds.): Component-Based Software Quality, LNCS 2693, pp. 207–224, 2003.

of component-based development, since the design phase of development is precisely where the most critical decisions about software architecture and reuse are made, but also forces notions and metrics for component quality to be highly implementation- and deployment oriented. A more abstract representation of components is desirable because it would allow CBD issues to be considered earlier in the development process and thus to cover a wider part of the software lifecycle.

The software industry has in fact been moving towards more abstract representations of software artifacts for some years, although not specifically in connection with component-based development. Fuelled by the popularity of the Unified Modelling Language (UML) [15], the Object Management Group has recently adopted "Model Driven Architecture" (MDA) [8] as its unifying vision for software engineering. Like component-based development, the aim of MDA is to promote reuse. By separating the description of key application abstractions and logic from the details of specific implementation platforms these key artifacts can be made more stable over time and thus can deliver a greater return on investment. MDA therefore emphasizes the distinction between Platform Independent Models (PIMs), which capture key application abstractions and logic in a platform independent way, and Platform Specific Models (PSMs) which represent the concrete mapping of these artifacts to specific implementation technologies.

Although CBD and MDA offer alternative strategies for reuse, they are in fact complementary and can be used to reinforce each other. MDA addresses the previously mentioned shortcoming of industrial component technologies, namely their focus on the implementation/deployment phases of development, by providing a framework for a model-based (i.e. UML based) representation of components. Several component-oriented development methods, such as Catalysis [9], KobrA [1] and UML Components [6], now emphasize the modelling of logical components in terms of UML diagrams in the early (analysis and design) phases of development. CBD, in turn, addresses a major weakness of the MDA approach — namely its lack of prescriptive support for the modelling process. Although the theoretical advantages of model-driven development are generally accepted, and many companies claim to be using the UML in their software engineering activities, the state-of-the practice is still very primitive. For most companies and tool-vendors model-driven development simply means creating platform independent UML models and then mapping them (as automatically as possible) into platform specific code. While this separation of concerns is valuable, however, it falls far short of the full vision of model-driven development. In particular, there is little attempt to use the information in models developed early in the lifecycle to actually "drive" the development of later models or artifacts. "Model-based development" would thus be a far more accurate term for the current state of the practice rather than "model-driven development".

Adopting a component-based strategy for organizing platform independent models can help address this problem by facilitating the measurement and use of concrete metrics to help move the development process forward. This chapter elaborates on this opportunity by showing how metrics derived from component-

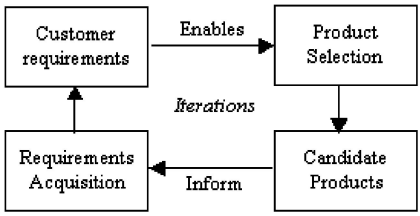

Fig. 1. Component Modelling

oriented models created early in a development project can be used to "drive" activities performed later in the project. The activities that we consider are those associated with quality assurance, since these play a particularly important role in the success of a project. In Section 2 we explain our model of components and composition, and how they can be captured using the UML, while in Section 3 we outline the motivation for quality assurance in software development projects and explain the fundamental role that a quality model can play. Then, in Section 4 we describe in detail how structural properties such as coupling, complexity, and size can be measured for UML artifacts, (i.e., how these quality measures are defined), and give an example based on the component-description models defined by the KobrA method [1]. Section 5 uses the concepts defined in section 4 to show some examples of how information garnered from UML models can be used for decision making in later parts of a project. Section 6 describes some future trends in component-based development. Finally, Section 7 concludes.

2 UML-Based Component Modelling

The techniques explained in this chapter will work with any approach for the UML-based modelling of components, but in this chapter we use the KobrA method [1] as the underlying foundation. Within the KobrA method, all behavior-rich abstractions, including entire systems, are viewed as components and are modelled in the same basic way. Moreover, the assembling of components is regarded as creating a larger component at the next level of granularity. Thus, a complete application architecture is viewed as a recursive structure in which larger components are realized in terms of smaller components, which in turn are composed of even smaller components, and so on.

The general set of models used to describe a single component is illustrated in Fig. 1. This shows that the models are organized into two groups, those making up the *specification* of the component that describes its externally visible properties, and those making up the *realization* that describe how the component is constructed from other objects and components (i.e. its design). The specification consists of three models which present different, but interrelated, views of the components properties: the structural model describes the data types that the component manipulates and the external component types with which it interacts, the functional model describes the semantic properties of the component's operations in terms of pre and post conditions, and the behavior model

presents the abstract states and transitions exhibited by the component. The realization also consist of three models which present different, but integrated views of how the component is realized: the structural model, which elaborates on the specification structural models to describe the architecture of the component, the interaction model which shows how each of the component's operations is implemented in terms of interactions with other components, and the activity model, which shows the algorithms used to implement the operations. In all cases, strict rules ensure that the various models are internally and externally consistent with one another.

The concept of composition appears in the KobrA method in two basic ways, both derived from the semantics of the UML. At run-time, component instances (created from the abstract component types modelled in the way described above) interact with one another through the traditional client-server model. Client-server (or in KobrA, clientship) relationships are generally loosely coupled, but it is also possible for clientship relationships to imply some form of whole-part relationship. Aggregation is a loose form of the whole-part relationship, in which there is a general notation of the server being a part of the client, but there is no concept of binding or related life-times. The stricter form of whole-part relationship is composition, which as well as indicating that the server is a part of the client also captures the idea that the server is private to the client, and its lifetime is tied to that of the client.

As well as this form of composition, at development time there is also the concept of structural whole-part relationships in which the definition of one component forms a part of the definition of another. In the UML this is captured though the containment (or ownership) relationship between packages. One package contains another package if all the elements within one package are also contained in the other. Packages play the role of name-spaces in the UML, so the nesting of packages serves as the basis for the hierarchical naming of model elements.

Both UML notations of composition are adopted in the KobrA method. Each logical component has an associated UML package, which contains the various models (and model elements) that document the properties of the component (Fig. 1). The model elements in the specification of the component are viewed as public members of the package, while those in the realization are viewed as private. By nesting component packages inside one another, the overall structure of a system can be represented as a tree-based hierarchy of components, each modelled in the way illustrated in Fig. 1. The desired client-server structure of the run-time instance of the components (including any run-time composition or aggregation relationships) is documented explicitly within the components' structural models.

3 Quality Assurance

Quality assurance (QA) encompasses the activities involved in maximizing the quality of the software artifacts developed in a project based on the available resources. In a quantitative sense, therefore, the goal of quality assurance is to

ensure that certain desired quality attributes or properties are attained or exceeded. The ISO9126 standard for software product quality [12] defines a set of six quality attributes: functionality, reliability, usability, efficiency, maintainability, and portability, each broken down into several sub-characteristics. These are intended to cover all quality needs of all stakeholders in a software product. For each quality attribute, specific QA techniques exist, e.g., inspections and testing to ensure functionality and reliability, or change scenario analysis to assess the maintainability of software design. The QA approach in this chapter is the measurement of structural design properties, such as coupling or complexity, based on a UML-oriented representation of components. UML design modelling is a key technology in the MDA, and UML design models naturally lend themselves to design measurement.

In the context of the ISO9126 product quality framework, structural properties are internal quality attributes. Internal attributes can be measured in terms of the product itself. All necessary information to quantify these internal attributes is available from the representation (requirements, design document, source-code, etc.) of the product. Therefore, these attributes are measurable during and immediately after the creation process. Internal quality attributes describe no externally visible quality of a product, and thus have no inherent meaning in themselves. In contrast, the six quality attributes ("-ilities") mentioned above are *external attributes*. Such quality attributes have to be measured with respect to the relationship between the product and its environment; measuring these attributes *directly* requires additional information. For example, maintainability can only be measured directly when the product actually undergoes maintenance, in terms of time spent. Reliability can be measured in terms of mean-time-to-failure (MTTF) of the operational product. External attributes can only be measured directly some time after the product is created (i.e., post-release, when the detection and removal of quality problems is time and cost intensive).

An artifact's internal attributes are assumed to have a causal impact on its external attributes. For instance, systems with loosely coupled components are expected to be more maintainable than highly coupled components, while components with high structural complexity are more fault-prone than those of low complexity. Making the link from internal attributes to their impact on external attributes is the purpose of a quality model.

Measurement of structural properties is attractive because it is objective and automatable, and thus fast to perform at low cost. Using a quality model, we can identify areas in a design with potential quality problems, and make these the focus of (more expensive) QA activities such as reviews and testing.

3.1 Quality Measures

A large number of measures have been defined in the literature to capture internal quality attributes such as size and coupling for OO systems [2]. Most of these measures are based on plausible assumptions, but, in the light of the discussion of the previous section, the two key questions are to determine whether

- They are actually useful, significant indicators of any relevant, external quality attribute, and how they can be applied in practice
- They lead to cost-effective models in a specific application context.

These questions have not received sufficient attention in the current literature. All too often design measurement is treated not as a means to an end, but as an end in itself. Answering the questions stated above requires a careful analysis of real project data. Some 30 empirical studies have been performed and reported in order to address the above-mentioned questions [2]. The results of these studies can be summarized as follows:

- Most of the studies are concerned with the prediction of class fault-proneness or development effort based on design measurement taken from source code.
- Data sets with fault or effort data at the class level are rare. As a consequence, these data sets tend to be repeatedly used for various studies. Instead, we find a large number of different studies using a small number of data sets. These data sets often stem from small industrial projects or academic environments (student labs).
- Only about half of the studies involve some attempt to build an accurate prediction model for the quality to predicted. The remaining studies only investigate the impact of individual measures on system quality, but not their combined impact.
- From the studies that build quality prediction models, only half of these studies investigate the prediction performance of the model in a relevant application context. Those results, however, look promising. For instance, in the context of fault-proneness prediction, prediction models that pinpoint the location of 80% of post-release faults are possible. Such models can be used to focus other quality assurance activities (inspections, testing).

4 Measurement of Structural Properties

The internal quality attributes of relevance in model-driven development are structural properties of UML artifacts. The specific structural properties of interest are coupling, complexity, and size. These are well understood, and as shown by empirical studies are indicators of various external qualities in OO design [3, 4, 5, 7, 10, 14]. To support the definition of coupling, complexity, and size measures we provide a "plan" for coupling, complexity, and size measurement.

4.1 Size Measurement

Size measures are counts of elements in UML diagrams and are good candidates for developing cost estimation models for designing, implementing, inspecting, and testing modelled entities. Such estimates are used as input for effort planning purposes and the allocation of personnel. The definition of such measures requires the resolution of three questions related to the overall measurement goal.

Table 1. Building Blocks for Size Measures

Entity	**Constituent elements within**
Component	Subcomponents, Packages, Subsystems, Classes, Interfaces
Package	Packages, Subsystems, Classes, Interfaces
Subsystem	Packages, Subsystems, Classes, Interfaces, Operations
Class	Classes, Operations, Attributes, States
Interface	Operations
Object	
Operation	Parameters

Question 1: What entities are to have their size measured?
Question 2: What constituent elements contribute to the size of these entities?
Question 3: How is the size of the entity to be quantified in terms of its constituent elements?

Ad Questions 1 and 2: Table 1 summarizes the possible choices for the first two decisions in the context of component-based development. The left column states the entities for which size is measured, the right column indicates what elements make up each entity and contribute to its size.

Components and classes are likely to be the most useful entities. Investigating size at a lower abstraction level requires an investigation of the external quality properties at that level (e.g., accounting for development effort per operation). However, this may be impractical. When defining size measures, it is not recommended to count entities of different types within one measure, since this would assume that the entities have equal impact on the quality. Therefore, we suggest that entities be measured separately, and that separate measures be used as building blocks for quality models as described in Section four.

Ad Question 3: For the quantification of entity-size there are two options. The size of a "higher-level" entity can be measured in terms of (1) the *number* of a certain type of countable element within it, or (2) the *sum of the size* of certain countable elements. In general, the first option results in a more coarse-grained measurement than the second option. The choice therefore depends on the stability of the number of counted elements at the time when the measurement is performed.

As an example, consider a KobrA [1] class diagram for the realization of a simple banking system of the form illustrated in Fig. 2. To measure the size of the bank component the following steps are performed:

- **Question 1:** entity to measure – the *components*.
  Question 2: constituent elements – component is constituted by a set of *classes*.
  Question 3: count the *number* of constituent elements.

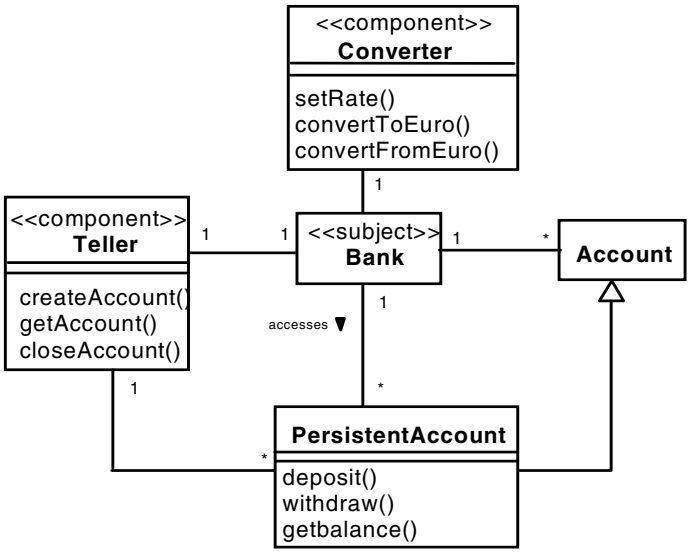

Fig. 2. Class Diagram for Bank

The resulting measure is simply the number of classes in the component. Considering the component Bank depicted in Fig. 2, the measure of this component is 5, since it is composed by 5 classes (Converter, Teller, Bank, Account and PersistentAccount).

– **Question 1:** entity to measure – the *components*
Question 2: constituent elements – *classes* of the component
Question 3: *sum of the size* of the constituent elements.

We need a size measure for classes, which we define in the following.
– **Question 1:** entity to measure – the *classes*
Question 2: constituent elements – the operations they *contain*
Question 3: count the *number* of constituent elements.

The resulting measure is the number of operations in the classes of the component. In Fig. 2, the size of the Bank component is 9 operations (only operations for three classes have been identified at this stage).

Considered in isolation, these numbers give us little, if any, useful information. However, when compared they can help allocate resources. For example, measuring the size of all entities (e.g., components) in the banking system allows us to make statements of the form "Bank" is twice as large as some other system or component. Such information can be used as input for the allocation of personnel. If empirical data is available, size measures can also be used to predict the effort for implementation and testing, thus supporting project planning and management.

4.2 Coupling Measurement

Coupling is the degree to which various entities are connected. These connections cause dependencies between the entities that have an impact on various external system quality attributes. Examples are maintainability (a modification of an entity may require modifications to its connected entities), testability (a fault in one entity may cause a failure in a completely different, connected entity), and fault-proneness (dependencies usually define a client-server relationship in which the client developer needs to know exactly how to use the service). Thus, a common design principle is to minimize coupling. The definition of such measures requires six questions to be answered:

Question 1: What entity is to have its coupling measured?
Question 2: By what mechanism is the entity coupled to other elements?
Question 3: Should multiple coupling connections between two elements (e.g. multiple associations between two classes) be counted separately or not?
Question 4: What is the direction of coupling (import or export coupling)?
Question 5: Should direct and/or indirect connections be counted?
Question 6: Are there any special relationships that should exist between connected entities?

For each question there are a number of options to choose from. These are detailed in the following: The actual selection should be driven by the measurement goal in mind and the design practices at hand.

Ad Question 1: the entities for which we may quantify coupling are the same as for size measures (see column "Entity" in Table 1): components, packages, subsystems, classes, interfaces, objects, or operations.
Ad Question 2: the possible types of links between a client and a server entity[1] (operation, class, component) are:

- Associations, e.g., between two classes. Aggregations and compositions may be subsumed under "associations" or measured separately as unique coupling mechanisms.
- UML Dependencies between any two elements.
- UML Abstractions (e.g., between interfaces and implementing classes).
- Operation invocation (object instance of class C sends message to object instance of class D).
- Parameter type (operation of class C receives/returns parameter of type class D).
- Attribute type (attribute of class C has attribute of type class D).

Ad Question 3: If there can be multiple coupling connections between a pair of elements, we have to decide whether to count these connections individually, or whether to just take into account the fact that there are connections

[1] Most, but not all connectors are directed and thus impose client and server roles on the connected items. For undirected connectors, the connected items are just two equal peers.

between the element pair, regardless of their precise number. The answer depends on the stability of the number of connections. If it is likely to change, a precise count is not needed.

Ad Question 4: For directed connectors, it is possible to count import or export coupling. Import coupling analysis the entities in their role as clients of other attributes, methods, or classes, while export coupling analysis the entities in their role as servers. High import coupling indicates that an entity strongly relies on other design elements to fulfill its job. Thus, import coupling has to be considered together with the following external attributes: understandability, fault-proneness, and maintainability. High export coupling states, that one entity is heavily used by others. Thus, it has to be considered together with the following external attributes: criticality and testability.

Ad Question 5: A yes/no decision has to be made as to whether the coupling measure should only count direct connections between elements or indirect ones as well. Indirect connections can be relevant when estimating the effort of run-time activities such as testing and debugging or the impact of modifications (ripple effects). Direct connections are sufficient for the analysis of understandability. To understand a component it is sufficient to know the functionality of directly used services. In contrast, knowledge about their implementation is not needed.

Ad Question 6: In some cases, we may only want to count coupling between entities that have a special relationship. For example, only coupling between classes with an inheritance relationship may be counted; or coupling to classes that are variation spots in a framework or product line architecture.

As an example of a coupling measure, we use the above schema to count operation invocations for the classes of a component.

Q1: entity to measure – the classes
Q2: coupling mechanism – messages invoking operations
Q3: direction of coupling – import coupling (outgoing messages)
Q4: multiple connections – multiple invocations of the same operation are counted individually
Q5: direct/indirect connections – count direct connections only
Q6: no special relationships between caller and callee.

Assuming the following collaboration diagrams for the withdraw() and deposit() operations of the Bank component (Fig. 3), the import coupling of class Bank is 7 (2 × getAccount(), 2 × convertToEuro(), 1 × getBalance(), 1 × deposit(), 1 × withdraw()).

If we change the definition of the coupling measure to count export coupling (Question 3), the resulting coupling values are 2 for classes Teller and Converter, 3 for PersistentAccount.

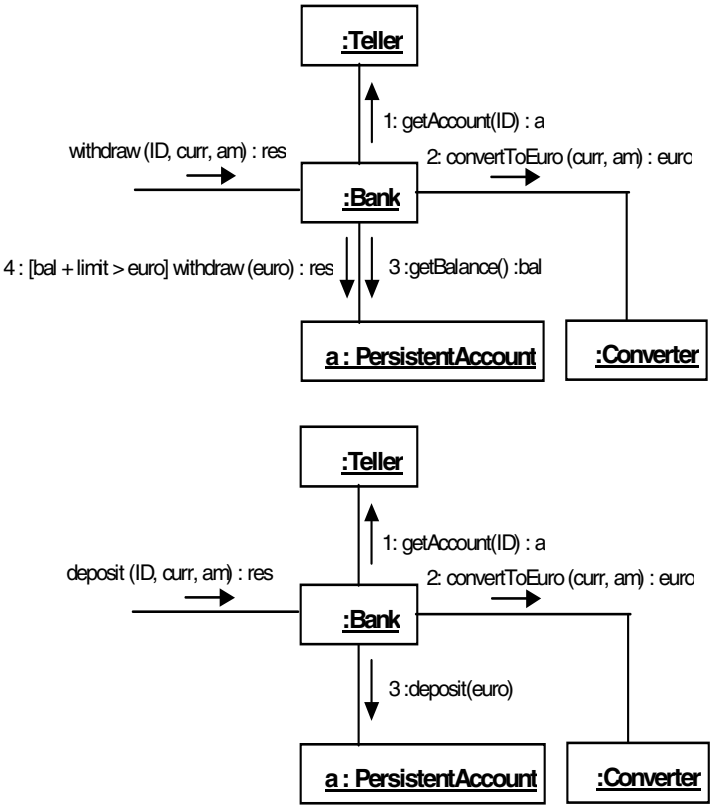

Fig. 3. Collaboration Diagram for the withdraw() and deposit() operations

4.3 Complexity Measurement

Whereas coupling is concerned with how a design entity is connected to other design entities, complexity measures are concerned with how connected the elements within a design entity are (e.g., associations between classes of a component, or invocations between operations of a class). The higher the connectivity of elements within the design entity, the higher its complexity. The questions to be answered when defining complexity measures are therefore similar to those for coupling measurement:

Question 1: What entity is to have its complexity measured?
Question 2: By what mechanism are the elements in the design entity connected with each other?
Question 3: Should multiple connections between two elements be counted separately or not?
Question 4: Should direct and/or indirect connections be counted?
Question 5: Are there any special relationships that should exist between connected entities?

Ad Question 1: the entities for which we may quantify complexity are the same as for size measures (see column "Entity" in Table 1): components, packages, subsystems, classes, interfaces, objects, or operations.
Ad Question 2: The mechanisms by which elements in the design entity are connected are the same as the coupling connectors in listed in Question 2 of Section 4.2. In addition, transitions between states contribute to the complexity of a class.

The options for the remaining questions are the same as for coupling measurement as described in Section 4.2. As an example of a complexity measure for a component, we count the number of associations between classes of the component:

Q1: Entity to measure – the component
Q2: Connection mechanism between design elements – associations between classes of the component
Q3: multiple connections – multiple associations between the same two classes are counted as one, not individually
Q4: direct/indirect connections – count direct connections only
Q5: no special relationships between associated classes.

Applying this measure to the class diagram for Bank (Fig. 2) we see that there are 2 associations between classes of the component.

5 Example Quality Models

The previous Section identified several internal quality attributes that can be measured directly from UML diagrams. However, these internal attributes are of little direct value by themselves. In the following the use of different quality models to assess various quality attributes is discussed. We show how such models can be used to identify potential risk areas in the system models and how this information can be used to drive the inspection and testing activities.

5.1 Prediction Models

Prediction models try to estimate the future quality of a system from internal quality attributes. This is achieved by exploring the relationships between internal and external attributes from past systems as well as by applying insights into the system under development. In the following the construction and usage of such a prediction model is described in the context of fault-proneness and structural properties of a class. Fig. 4 depicts the steps involved in building the quality model.

A measurement tool is applied to a set of previously created documents to obtain the relevant structural properties. This results in a set of documents enriched with structural properties. In addition, fault data (e.g., from inspections) has to be selected for these documents. Statistical analysis (e.g. classification

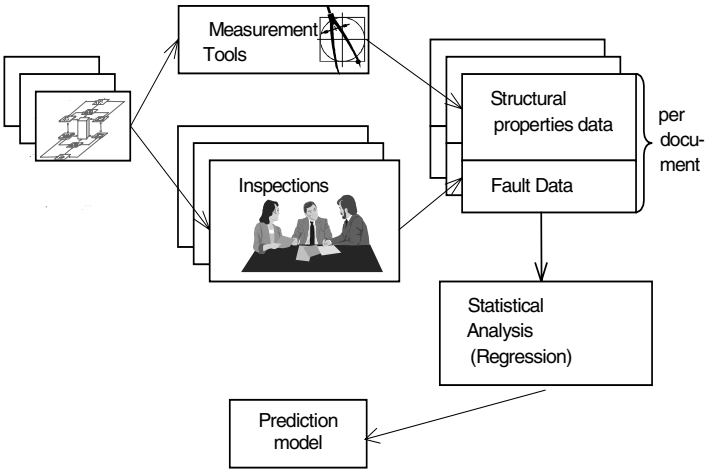

Fig. 4. Building a Prediction Model

or regression analysis) is then used to explore the relationship between fault data and structural properties. The result is a prediction model (i.e., a regression equation) that computes a predicted fault-proneness or predicted number of faults. This model can be used to make predictions for new entities as depicted in Fig. 5.

For a new entity the measurement tool is reapplied to obtain data on structural properties. This is an automatable, cheap, and fast process. The resulting data is fed into the prediction model, which uses the now known relationships between coupling, complexity and faults to calculate, for instance, a predicted fault-proneness for the document — that is, the probability that a fault will be found in the document upon inspection[2].

The resulting output is useful for deciding what activities to perform in the ensuing development process and what these activities should focus on. For instance, if the predicted fault-proneness is below 25%, it is not necessary to inspect the document but to proceed immediately to the next development activity. However, if the predicted fault-proneness is above 25% an inspection can be triggered. Thus, effort can be focused on the artifacts that are more likely to contain faults. Prediction models for other system qualities (e.g., models to predict implementation and test effort from size) can be built in the same way.

The advantage of prediction models is that they provide a mapping from non-interpretable internal quality data to interpretable external qualities. The result is an absolute, quantitative statement about the external quality of a system, which can be understood by developers and expressed in the same units in which the external quality is measured. However, a disadvantage is that the data requirements are high, and that the building and use of such models requires expertise in statistics.

[2] The kind of prediction depends on the used regression analysis, which in turn depends on the type of (fault) data collected.

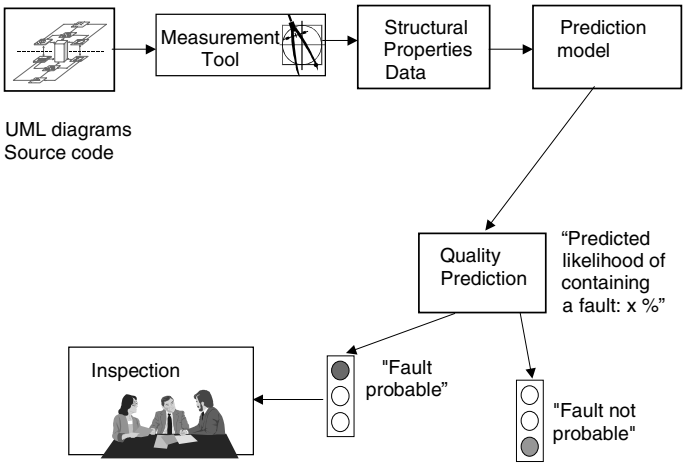

Fig. 5. Using the Prediction Model

Industrial experience with prediction models for fault-proneness/effort prediction in the context of OO system development (not limited to component development) is encouraging [3, 4, 7, 10, 14]. When predicting fault-proneness for classes, the best models have consistently obtained a percentage of correct classifications of about 80% and find more than 80% of faults (these figures assume that classes predicted fault-prone are inspected, and all faults are found during the inspection). Overall, the results suggest that design measurement-based models for fault-proneness predictions of classes may be very effective instruments for quality evaluation and control of components. From the results presented in studies predicting development effort, we may conclude that there is a reasonable chance that useful cost estimation models can be built during the analysis and design of object-oriented systems. System effort predictions with relative errors below 30% (an acceptable level of accuracy for cost estimation models) seem realistic to achieve.

The empirical results also indicate that prediction models are highly context-sensitive, affected by factors such as project size, development processes at hand, and so forth. Therefore, prediction models need to be built and validated locally in the development environment in which they are used. Also note that the above results were achieved performing measurement on source code artifacts, not UML design artifacts. A demonstration of these principles using UML design measurement remains future work.

5.2 Quality Benchmarks

The idea of benchmarks is to compare structural properties of an entity with properties of previous systems that are 'known to be good'. To this end, measurement values for selected size, coupling, and complexity measures are stored in a database. If a new or modified component has to be evaluated, the same

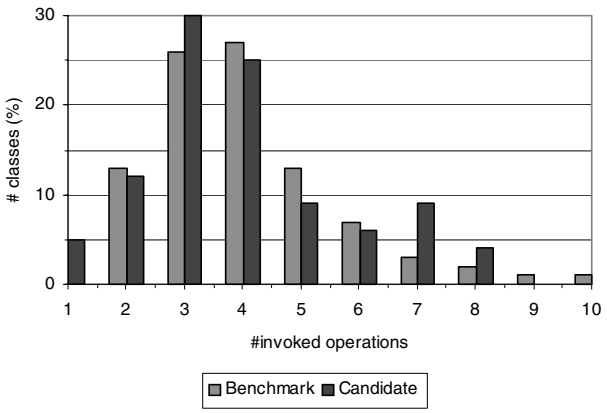

Fig. 6. Benchmark

measurements are applied. Afterwards the distribution of each size, coupling, and complexity measure is compared to the distribution of those stored in the database.

As an example, Fig. 6 shows a (fictitious) distribution of the number of operations invoked by the classes of a component. The vertical axis indicates, for each value on the horizontal axis (number of invoked operations), the percentage of classes in the component that have that particular value. In the example, the distribution of the new component follows closely the distribution of the benchmark, except for values 7 and 8, which occur more frequently. Such deviations pinpoint potential risk areas. The example also illustrates that the examination of distributions provides more information than simply defining thresholds for each measure. Although the number of invoked operations is not exceedingly high (the benchmark suggests 10 as an upper value), there are more than an average number of classes with 7/8 invoked operations. This may point to a problem.

Unlike the prediction model approach, benchmarking does not require external quality data. No absolute statement about the quality of entities can be made. A database of past 'known to be good' entities has to be acquired before the approach can be used operationally. However, a quality benchmark can be used to:

1. provide guidance for design decisions — a simple example, a quality benchmark for the size of class, object, or interaction diagram provides objective criteria as to whether the diagram is too big.
2. define design exit criteria. For instance, deviations from the quality benchmark must either be justified, or the design reworked.
3. identify potential trouble spots in the system that should be the focus of inspection and testing.

An industrial application of quality benchmarking in the context of large-scale software acquisition (COTS, outsourced development) is reported in [13].

Table 2. Example of a Simple Ranking Model

Component	Rank Measure 1	Rank Measure 2	Rank Measure 3	Average Rank
Component2	24	3.5	12	13.17
Component18	9	16	14	13
Component7	17	3.5	7	9.17
...	...			

5.3 Simple Ranking Models

The ranking model approach uses selected structural property measurements to select potentially critical areas in a design. Although it does not require the measurement of external quality data, it is important that the measures are indicators of specific qualities (see Section 5.1). The steps involved are as follows:

1. Apply design measurement to the system
2. Rank entities in decreasing order, independently for each measure, and assign ranks. The entity with the lowest value is assigned rank 1, the entity with the next lowest value is assigned rank 2, etc.
3. For each entity, take the average rank of that entity for all measures.
4. Sort the entities in decreasing order of their average rank.

An example application is shown in Table 2. The first column indicates the entity name, the next three columns the ranks for three selected measures, and the last column the average of these ranks. Rows are sorted in decreasing order. Entities at the top are potentially critical, because they display less desirable structural properties. Thus, such entities should be selected first for inspections or testing until the allocated resources are depleted.

The advantage of this approach is that it is immediately applicable — neither external quality data nor a database of past 'known to be good' systems is required. Unfortunately, there is no absolute statement about the quality of the system. In addition, it is not possible to make statements about the quality of individual entities in isolation since only relative rankings of entire sets of entities are possible. Furthermore, the creation of such a ranking may result in lost information since measures are not usually defined on an interval or even ratio scale but are only used on an ordinal scale. In summary, this approach is therefore less powerful than the others. To use this approach effectively it must be ensured that the used measures are related to the external quality in mind.

6 Future Trends

The use of structural measures to help drive the allocation of resources in software development projects will undoubtedly increase in the future. However, further research is needed to improve the usefulness of measures derived from early, analysis and design artifacts of the kind usually associated with platform independent models.

Most of the measurement activity reported to date is actually based on measurement of source-code. Early analysis and design artifacts are, by definition, not complete and only represent early models of the actual system to be developed. The use of predictive models based on early artifacts and their capability to predict the quality of the final system still largely remains to be investigated.

One reason for the lack of empirical work carried out on design artifacts to date is missing tool support to automatically collect design measures. While there are many static analyzers for source-code of all programming languages, tools to collect design measures from UML diagrams are scarce. Some commercial UML modelling tools have measurement capabilities, though often limited to size measures. SDMetrics[3], a flexible tool using the XMI standard to read UML designs and calculate user-specified design measures as outlined in Section 4, was released only recently.

Future research needs to focus on collecting large data sets involving large numbers of systems from the same environment. In order to make the research on quality prediction models of practical relevance, it is crucial that these models be usable from project to project. Their applicability depends on their capability to accurately and precisely predict quality attributes (e.g., class fault-proneness) in new systems, based on the development experience accumulated from past systems. Though some studies report the construction of such models, few report their actual application in realistic settings. Our understanding has now reached a sufficient level to enable such studies to be undertaken.

Cost-benefit analysis related to the use of quality prediction models are scarce. Little is known about how the predictions made by statistical models compare with those made by system experts in terms of correctness and completeness. Furthermore, there might be a way to combine expert opinion with object-oriented design quality models to obtain more reliable statements about system quality. Future empirical studies should also take this into account to improve the case for the use of design-measurement-based quality prediction models.

7 Summary and Conclusions

The model driven architecture (MDA) is a powerful approach for improving long-term productivity by increasing the return on investment derived from software development effort. Although its theoretical advantages are widely accepted, however, the state-of-the practice is still very primitive. For most companies and tool-vendors model-driven development simply means creating platform independent UML models and then mapping them (as automatically as possible) into platform specific code. There is currently very little support for using platform independent model created early in the software lifecycle to "drive" ensuing development activities. Much of the potential benefits of model driven development therefore remain to be realized.

[3] SDMetrics – software design measurement tool for the UML, see `http://www.sdmetrics.com`

This chapter has pointed the way towards true model driven development by briefly illustrating how the measurement of UML structural properties can help drive the quality assurance activities in component-based development. Such measures are objective, can be collected automatically and provide early feedback on the quality of entities (e.g., components). This feedback can help an organization allocate quality assurance effort and thus "drive" part of the overall development process from model-derived data.

References

1. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel. *Component-based Product Line Engineering with UML*. Addision-Wesley, Component Series, 2001.
2. L. Briand, J. Wüst. Empirical Studies of Quality Models in Object-Oriented Systems. *Advances in Computers*, 59:97–166, 2002.
3. L. Briand, J. Wüst, J. Daly, V. Porter. A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems. *Journal of Systems and Software*, 51:245–273, 2000.
4. L. Briand, J. Wüst, H. Lounis, S. Ikonomovski. Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study. In *Proc. of ICSE'99*, pages 345–354, Los Angeles, USA, 1999.
5. L. Briand, C. Bunse, J. Daly. A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs. *IEEE Transactions on Software Engineering*, 27(6), 2001.
6. John Cheesman and John Daniels. *UML Components: A simple process for specifying component-based software*. Addison-Wesley, 2000.
7. S. Chidamber, D. Darcy, C. Kemerer. Managerial use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, 1998.
8. Desmond F. D'Souza. OMG's MDA - An Architecture for Modeling, OMG MDA. Seminar, October 2001. Available at `www.omg.org`
9. Desmond F. D'Souza and Allan Cameron Wills. *Objects, Components, and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, Object-Technology Series, 1999.
10. N. Fenton, S. Pfleeger. *Software Metrics, A Practical and Rigorous Approach*. International Thompson Computer Press, 1996.
11. W. Li, S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23(2):111-122, 1993.
12. ISO/IEC FCD 9126-1.2, "Information Technology – Software Product. Quality– Part 1: Quality Model". 1998.
13. C. Mayrand, F. Coallier. System Acquisition Based on Software Product Assessment. In *Proc. of ICSE'96*, pages 210–219, Berlin, Germany, 1996.
14. P. Nesi, T. Querci. Effort estimation and prediction of object-oriented systems. *Journal of Systems and Software*, 42:89–102, 1998.
15. Object Management Group (OMG). *Unified Modeling Language (UML)*. Version 1.4, 2001. Available at `www.omg.org`
16. Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, 1998.