The MeTHOOD Approach: Measures, Transformation Rules, and Heuristics for Object-Oriented Design

Thomas Grotehen, Klaus R. Dittrich
University of Zurich
Computer Science Department
CH-8057 Zurich, Switzerland
Email: {grotehen, dittrich}@ifi.unizh.ch

Abstract

The MeTHOOD[1] approach makes design knowledge tangible that will help to improve conceptual object-oriented schemas. Although some of this important knowledge is available in current literature, it is nearly unusable for designers. Therefore, we propose a catalogue of measurements, heuristics, and transformation rules, and introduce an effective integration schema for these elements. This integration schema is the base for a design optimisation process allowing designers to improve conceptual schemata systematically. Besides the integration schema and the process, this paper describes a design support system called MEx[2]. It consists of a design knowledge base (containing definitions of measures, heuristics, and transformation rules) that works on a specification database containing conceptual design schemas.

1 Introduction

Designing a "good" object-oriented conceptual schema is difficult. Although there is an agreement that a methodology should include concrete advice for good design [1, 2], current approaches [3-10] and corresponding tools lack of support for this task. Most of them define an object model (sometimes called object modelling language), graphical notations corresponding to the object model, and a high-level design process. The object model describes what a correct conceptual design schema is, but correctness does not imply "goodness". The high-level design process describes what tasks may be executed when, influencing at best process quality. This means that an important ingredient is missing in object-oriented design methodologies. As a result, many organisations execute design reviews by expert designers to improve the design of a large system and avoid design flaws [11]. However, these reviews are expensive, expert designers are hard to find, and failure to review may result in expensive redesigns.

[1] **Me**asures, **T**ransformation rules, and **H**euristics for **O**bject-**O**riented **D**esign
[2] **M**e**T**HOOD **Ex**pert

The objective of the MeTHOOD approach therefore is to enable a design process allowing designers to review and improve a conceptual schema continuously. It is not necessary to reinvent a complete design methodology (including object model, high level process and notation) to meet this objective successfully. MeTHOOD provides an *extension* to existing methodologies, which is based on a catalogue of "design knowledge" combining and formalising existing and new *design heuristics*, *transformation rules* and *measures,* and adds them to available design methodologies in an effective manner. This catalogue is the main focus of our work. Due to lack of space it is not possible to present the catalogue in detail in this paper. Therefore, we focus here on the conceptual framework of this catalogue consisting of measurements, design heuristics and transformation rules and their integration.

A *design heuristic* [12], e.g. "Keep related data and behaviour in one place.", is a guideline or rule of thumb representing design experience and best practice. It is an advice for making design decisions. A design heuristic is used to describe a family of potential design flaws (e.g. locations in a schema where "related data and behaviour are separated") and help designers to detect them. Explicit descriptions of heuristics are needed, however in most current literature they are implicitly described and hardly recognisable as a useful heuristics ([12] is the only attempt to make them more explicit). Therefore, one of the most important objectives of MeTHOOD is to provide important heuristics explicitly. In fact, our catalogue includes 21 explicit heuristics so far. These heuristics give different kinds of decision support for the correct use of single modelling concepts (e.g. selection of a complex attribute or a new class to model an address), and for the detection of potential design flaws.

If a potential design flaw is detected, often a *transformation rule* can be used to create a proposal for an alternative design. A transformation rule is an advice on how to eliminate occurrences of a design flaw. For example, if a heuristic detects a location in the design where "related data and behaviour are separated", a corresponding transformation rule suggests how to bring this data and behaviour together. Often, there are different ways to eliminate an occurrence of a detected design flaw. Therefore, every design heuristic can have different transformation rules associated with it.

Design heuristics and transformation rules allow to base design decisions on a clear rationale. However, there are two problems with this approach:

- Different heuristics can conflict with each other: the first heuristic may require to "Keep all attributes of a class in its private part." whereas another one demands "Values which are displayed in a user interface should be represented as public attributes of a class."
- A heuristic can suggest different transformation rules to resolve a single design flaw.

How do we decide which heuristic and transformation rule fits best in a given context? MeTHOOD provides *measures* [13-15] to give support for these kinds of decisions. A measure is a mapping between an entity and a number and to characterise a feature [16] of the entity (e.g. the number of its dependencies to other entities). Designers use measures to compare the effects of different heuristics and transformation rules on the conceptual schema. The measured values are used to recognise improvements resulting from changes of a conceptual schema. They are continuously monitored during the complete design process and provide designers with instant feedback on design decisions.

The remainder of this paper is organised as follows: Section 2 describes heuristics, transformation rules and measures in more detail. Section 3 illustrates the core of MeTHOOD: the MeTHOOD integration schema and the MeTHOOD process. Section 4 contains a practical example where MeTHOOD is applied to a concrete design problem. Section 5 presents the architecture of MEx, a design support system which is currently under development, Section 6 gives an overview of related work in this area, and Section 7 concludes with a summary.

2 Definitions and basic concepts

In this section we describe the main concepts which are necessary to understand the MeTHOOD approach. These are *design fragments* (parts of a conceptual design schema), *features* of design fragments and *measures* for these features, *heuristics* allowing evaluation of design fragments, and *transformation rules* describing possible improvements.

2.1 Conceptual design schema and design fragment

A conceptual design schema is a high level, implementation environment independent description of a software system. It consists of a set of classes and their relationships. Other terms with a similar meaning include e.g., *a logical design*, *an architecture*, *an object model*, and *a conceptual model*.

A design fragment is a dynamically selected subset of a conceptual design schema. It thus consists of a subset of classes with their structural (e.g. inheritance and associations) and dynamic (e.g. collaborators and messages) relationships. However, it does not include the relationships between classes in the subset to classes outside. The empty design fragment $\in$ is a special case. It contains no classes at all.

2.2 Measures and features of a design fragment

Our definitions of measures are based on [14]. A measure in MeTHOOD is an empirical objective assignment of a number to a design fragment. It is used to characterise a specific *feature* of the fragment. A feature of a design fragment is a property influencing its quality, e.g. its change resistance. Combinations of measured values for these features support designers evaluating different design alternatives for a design fragment.

However, measures do not support the detection of structural misfit in a schema, e.g. a redundant relationship, which is often the reason for a "bad" measure. MeTHOOD supports designers detecting this kind of misfit by means of *design heuristics*.

2.3 Design heuristics and transformation rules

A *design heuristic* is a rule of thumb. It is an advice for designers on how to use design techniques to improve the quality of a design fragment. Sources of heuristics include the literature on object-oriented design, guideline documents of companies, and personal experience. A heuristic is not a law. We see a set of design heuristics as a "paint" which marks potential design flaws violating the heuristic.

There are heuristics where it is not objectively decidable whether any design fragment violates them, sometimes it is not even possible to decide if the heuristic can be applied (the situation described in the heuristic occurs in the fragment). Consider for example the heuristic "Distribute the system intelligence as uniformly as possible among your top level classes." [17]. There are other heuristics where violation can be checked by the designer or by an algorithm. Therefore, we distinguish the following cases:

- A *checkable heuristic* can be checked objectively in a given design fragment. This means that for every given design fragment, there is an algorithm which computes whether the fragment contains conflicts to the heuristic or not. More formally, a checkable design heuristic h is a mapping $h: F \rightarrow \{true, false\}$ where F is the set of all possible design fragments. For a design fragment f, $h(f)$ is *true* if h is not applicable to f or h is fulfilled. $h(f)$ is *false* if h is applicable to f and h is not fulfilled. In other words, if $h(f)$ is *true*, f needs no changes, from the point of view of the heuristic. If $h(f)$ is *false*, f contains a potential design flaw. In this case we say the heuristic "fires". A firing heuristic marks potential design flaws in a design fragment. Examples for checkable heuristics are "Do not change the state of an object without going through its public interface." or "A class in a containment hierarchy should

only depend on its child classes." Checkable heuristics are always achievable, that means a non-trivial design fragment without conflicts to the heuristic exists.

- An *implied heuristic* is an advice to optimise the value of a measure. Consider for example the heuristic "Minimise the number of classes a class collaborates with." It is implied by the measure "Number of classes a class collaborates with" (small values of this measure are better than large values). Implied heuristics have no "hard" checking algorithms. Sometimes they are not achievable because there is no "best value" for the corresponding measurement (e.g. "Minimise the number of methods in a class.").
- *Uncheckable design heuristics* cannot be checked automatically for a given design fragment. An example of such a rule is "Object-oriented designers should never allow physical design criteria to corrupt their logical designs." These kinds of advice are usually very important. They often subsume and give rationale for "lower level" checkable heuristics. In most cases they are hints towards an ideal design which is not achievable, e.g. a design which is completely open for extension and completely closed for modification (open/closed principle).

Heuristics are used to *detect* misfit in a conceptual schema but they do not help to *correct* it. A *transformation rule* describes a process, which is used to give designers proposals for (restructured) alternative design fragments. These alternative fragments preserve the existing functionality and the semantics of the original fragment and do not contain potential design flaws.

3 Integration of measures, heuristics and transformation rules in MeTHOOD

We have observed that designers seek support to reason more objectively about a design, find design flaws, and decide between various possible design alternatives. Therefore, we have three main requirements to any approach that claims to support the design of a good conceptual schema:

1. It must help designers to evaluate consequences of schema modifications.
2. It must give designers information about possible design flaws.
3. It must provide proposals on how recognised design flaws can be removed.

Current approaches which use measurements and design heuristics could be very effective to provide this kind of support. However, they address the above requirements only partly.

In these approaches, measures are used at certain well-defined points in the software development cycle to find out how certain projects perform. They are used to compare different projects, to predict the development and maintenance costs, to make statements about the quality of the software, for example in terms of defects per module, and to optimise the development process. One of the main disadvantages of current product measures is that they only deliver passive numbers. Using these numbers, the designer knows that something is wrong, and eventually he knows the location. However, there is no systematic approach to making improvements.

The MeTHOOD approach is different because here measures are used to give designers *continuous* feedback about their design decisions and their consequences (requirement 1). Furthermore, measures are combined with heuristics to find locations where the measured values can be improved (requirement 2). This is different to current approaches that provide "stand alone" heuristics. Another difference to these approaches is that we propose to combine heuristics and transformation rules. This allows proposing useful design fragment transformations to

designers (requirement 3). The necessary integration between measurements, heuristics and transformation rules is facilitated using the MeTHOOD integration schema.

3.1 The MeTHOOD integration schema

The MeTHOOD integration schema describes the interesting relationships between measures, heuristics and transformation rules. Consider Figure 1: Heuristics are used to find potential design flaws (1.1) in a design fragment, transformation rules are used to generate alternative design fragments (1.2) and measures assess features of the design fragment (1.3).

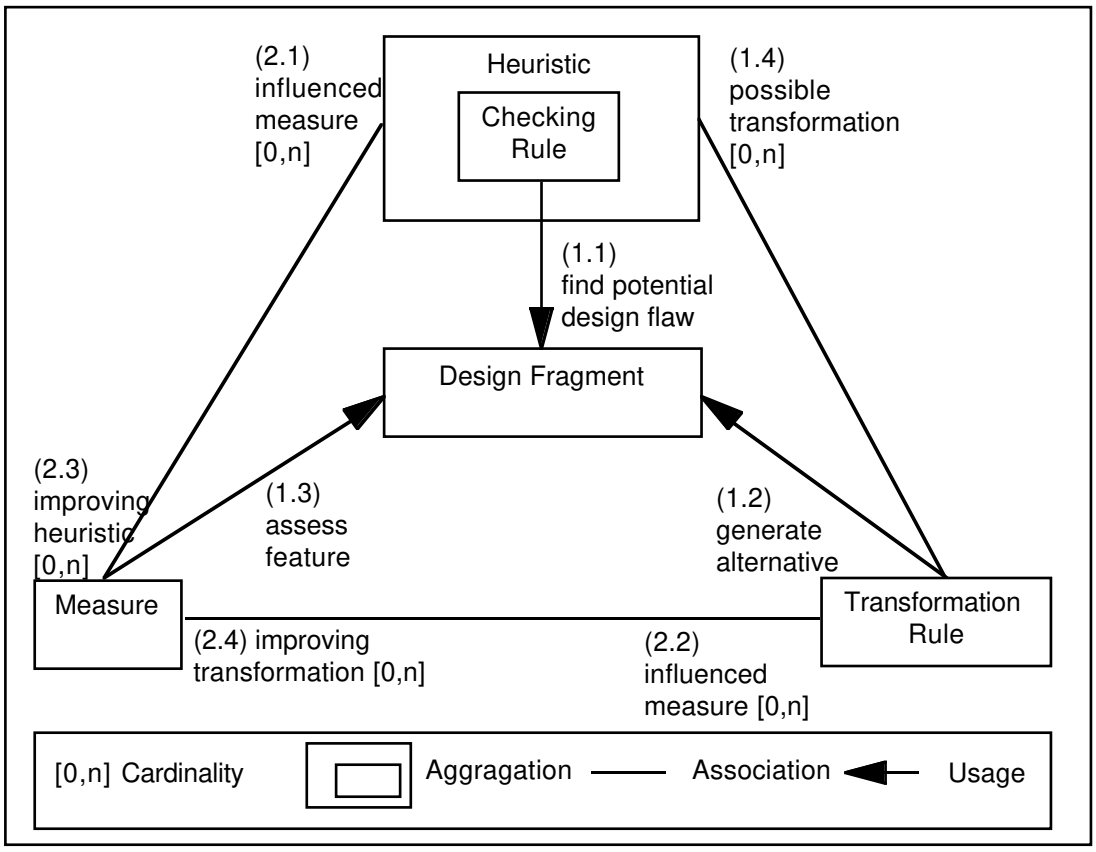

Figure 1: The MeTHOOD integration schema

If a designer has found a potential design flaw (using a firing heuristic *h*), he can decide to remove the violation to *h*. All modifications that remove violations to a certain heuristic *h,* change the structure of all design fragments in similar ways. Consider for example the heuristic "A class in a containment hierarchy should only depend on its child classes." Removing a violation to this heuristic always means to remove a certain kind of relationship between contained classes and classes different to their child classes. The fact that these structure changes are similar allows describing and formalising the change process. Therefore, we propose to associate heuristics with transformation rules (1.4) in the MeTHOOD integration schema. The heuristic describes how to detect a design flaw; the transformation rule describes how to eliminate it. More formally, a transformation rule is a mapping *t: F->F* on the set *F* of all design fragments and a transformation rule belongs to a checkable or an implied heuristic if it fulfils one of the following transformation criteria:

- Checkable heuristic: Let *h* be a checkable heuristic, and *f* in *F* a violating design fragment with *h(f)= false*. *t* is a transformation rule for *h* in *f* if *h(t(f))= true*.
- Implied heuristic: Let *j* be an implied heuristic and *m: F-> R* the measure of *j*. Let *f* be a fragment in *F*. *t* is a transformation rule for *j* in *f* if *m(t(f))* is better than *m(f).*

Furthermore, since transformation rules change design fragments in similar ways, the specific measured values of the resulting design fragment will also change. These value changes are predictable for many heuristics and transformation rules. Making a violating design fragment compliant to, e.g., "A class in a containment hierarchy should only depend on its child classes." always improves the values for isolability measurements of this fragment. Therefore, the MeTHOOD integration schema also includes associations between heuristics and measures they influence (2.1)/(2.3) and between measures and transformation rules (2.2)/(2.4). This allows selecting heuristics systematically according to desired improvements of measured values. Furthermore, it helps designers to resolve conflicts between colliding heuristics, one of the main problems of "pure" heuristic approaches.

3.2 The MeTHOOD Process

The MeTHOOD process (Figure 2) consists of different activities that are executed iteratively by designers to improve a design fragment.

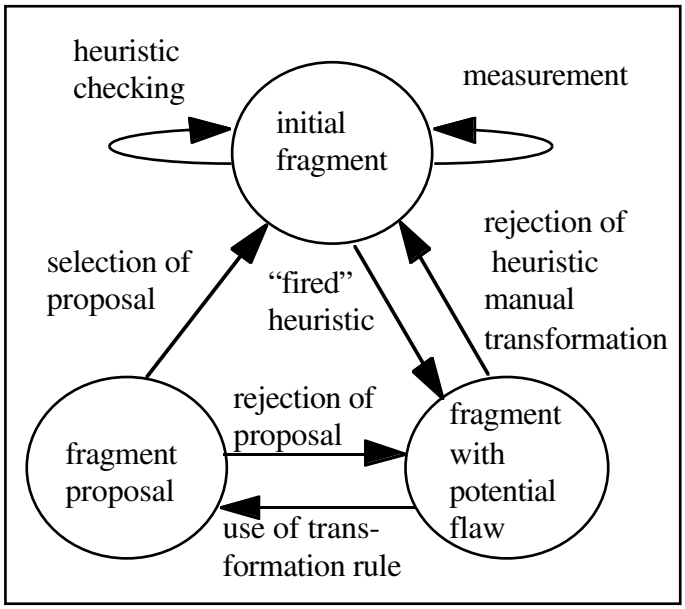

Figure 2: The MeTHOOD process

These activities result in state transitions of the considered fragment. In the *initial fragment* state, the designer can measure values of different features of the fragment, e.g. change resistance or isolability. He may then decide that one of these values has to be improved. To find locations in the fragment where improvements are possible, he can check the fragment for improving heuristics (see Section 3.1) of the corresponding measurement. If such a heuristic "fires", the state of the fragment changes to *fragment with potential design flaw*. This state change also occurs, if the designer performs a check on all available heuristics, and one of them fires.

If a design fragment is in the state *fragment with potential design flaw,* it contains a violation to a heuristic. The designer has different options to resolve this conflict. One option is to reject the heuristic. The rejection disables the heuristic for this fragment. Another option is to remove the violation manually. This means, the designer executes a set of schema transformations, which remove the violation to the heuristic. Both of these options change the state back to *initial fragment* where the designer can observe the effects of the provided transformations on the measured values of the fragment. The third option is to check if the violated heuristic has a

transformation rule. If this is the case, the transformation rule can be used to generate a proposal for a fragment, which does not contain violations to the heuristic.

This changes the state of the fragment to *fragment proposal*. In this state the designer can see how the fragment can be transformed to comply with the violated heuristic. He can reject the proposal which leads back to the "fragment with potential flaw" state. If he selects the proposal, the transformation rule is executed (eliminating the conflict) and the state changes to "initial fragment".

4 Example

Consider a design fragment of a vehicle simulation system as sketched in Figure 3(A). The *car* class contains a *motor* class and a *wheel* class.

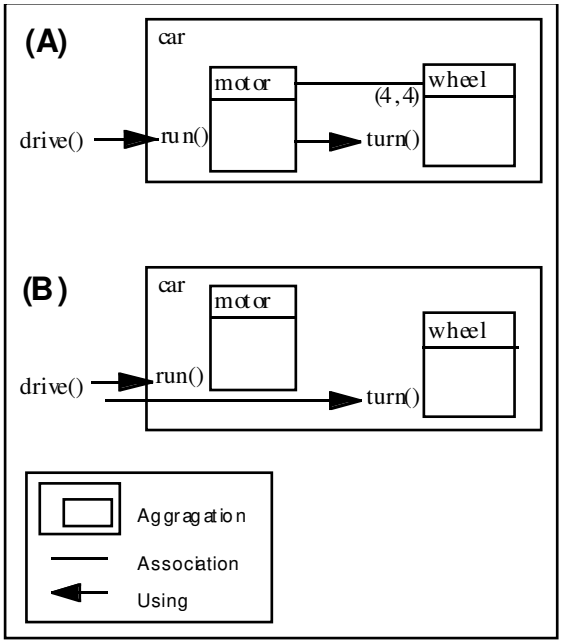

Figure 3: Car example

It has a method *drive()* which uses the method *run()* of *motor*. The method *run()* uses the method *turn()* of the *wheel* class. In this example, the designer plans to use the contained parts of the class *car* also for other vehicles like aeroplanes and boats. He should therefore reduce the effort to take a contained class out and use it in another class. A possible approach for this task is to reduce the strength of dependencies of contained classes on other classes. The strength of the dependencies from a fragment to other classes is measured using the measurement WDS from the MeTHOOD knowledge catalogue:

Measurement 3: Weighted Dependency Strength of a fragment *WDS(f)*
Motivation:
Changes in classes outside of a given fragment can lead to changes within the fragment. Changes within a fragment can lead to subsequent change. Designers need to know the degree to which a given fragment is influenced by changes outside and how changes effect other classes. This degree is influenced by the strengths of dependencies from the considered frag-

ment to others. Existing dependency measurements do not take the strength implied by different relationships into account.

Assessment/prediction:

Assessment.

Kind (Product/Process/Resource), Granularity (Class, Method): Description of measure.

Product measure of fragment.

Direct measurement/indirect measurement

Indirect measurement of external attribute

Required level of detail:

Classes relationships (messages, associations, and inheritance) between classes.

Definitions:

Fragment *f* is given. *I(f)* is a set of relationships representing dependencies between *f* and other classes. *R* is the set of all relationships in a conceptual model. *SD: R->R* is a measure of the strength of a relationship in R.

Formula:

$$WDS(f) = \sum_{i \in I(f)} SD(i)$$

High values of WDS indicate poor isolability. For the sake of simplicity, we assume the same strength (1) for all relationships *I(f)* in our example to assess SD. In (A) the fragment motor has the isolability 2 (the uses relationship *turn()* and the association to *wheel*), and *wheel* has the isolability 0 (no dependencies on other classes). The designer now checks all heuristics that improve the values of isolability of a fragment measured by WDS. One of the heuristics with a positive effect on WDS is the following:

Heuristic 11: A class in a containment hierarchy should only depend on its child classes.

Description: A class should not depend on its container. A class should not depend on a sibling in a container. A class should not depend on classes contained in its siblings.

Definitions: ...

Rationale: *Reuse of abstraction*: If a contained class depends on its container or its siblings there are two different possible actions to make it usable in another target context:

- it has to be isolated (the dependencies have to be eliminated), or
- the depending classes have to be provided in the target context.

Both options are undesired and require additional development effort. The first option results in code changes and a complete new version of the class. The second action results in additional development effort in the target context.

Complexity reduction: ...

Priority in Life Cycle: ...

Granularity: ...

Example: ...

Subsuming/subsumed heuristic:

Checking query: Find all items with uses relationship or association relationship to siblings:

```
select   x, y, b.from, a
from     x, y in type, a, b in Relationship
where    a.myMetarel. name="sendsMessage"
and      b.myMetarel. name="hasParts"
and      a.from = x and y in a.to
and      x in b.to and y in b.to
```

Transformation rule: Move the dependencies to the common container class.

```
insert b in b.from, y;
remove b from x;
```

Violated heuristics: ...

Justifications for violating the heuristic: ...

Influenced measures: *isolability (WDS): positive*

This heuristic is violated. The checking query of this heuristic would detect[3] that *motor* has a uses relationship with *wheel* and both are contained in *car*. The *motor* class can (without modifications) only be (re-) used in vehicles where the motor is used to turn the wheels. The transformation rule in this heuristic proposes to remove the uses relationship from *motor* to *wheel* and insert a new uses relationship between *car* and *wheel*. The result (Figure 3 (B)) is presented to the designer who can accept or reject it. The isolability of *motor* in the new fragment is 0.

5 MEx: A design support system for MeTHOOD

We have observed that it is hard to do a complete "design check" on a complex conceptual design schema "manually". Therefore, we develop a design support system called Mex[4] that provides automatic support for design monitoring, heuristic checking and fragment transformation using the design knowledge from the MeTHOOD catalogue. The main components of MEx are illustrated in Figure 4.

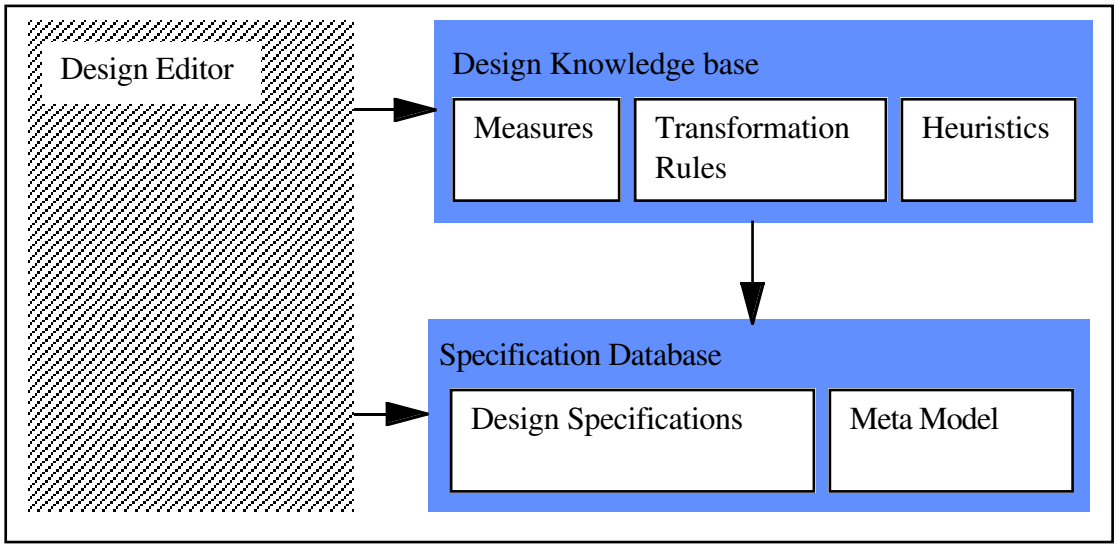

Figure 4: Main components of MEx

5.1 Design Editor

The design editor provides a graphical view on a conceptual design schema. The primary task of the design editor is to edit the conceptual schema. The design editor of MEx is a commercially available design tool. It provides different design notations and source code generators for various programming languages and object-oriented database management systems.

5.2 Specification Database and Meta Model

The design editor uses a *meta model* to store the conceptual design schema in the MEx specification database. The meta model contains information about the allowed modelling constructs (e.g. classes, attributes, methods, ...) and their relationships (e.g. a class has attributes and methods, a class inherits from other classes). Our prototype provides a dynamic meta model

[3] We will show how automatic detection of potential design flaws and design transformation is implemented in the next section.

[4] **M**eTHOOD **Ex**pert

that allows users to modify the modelling concepts dynamically. This makes it possible to adapt the meta model to support different object-oriented design methodologies.

Figure 5 shows how the meta model (upper grey box) is related to a conceptual design schema (lower grey box).

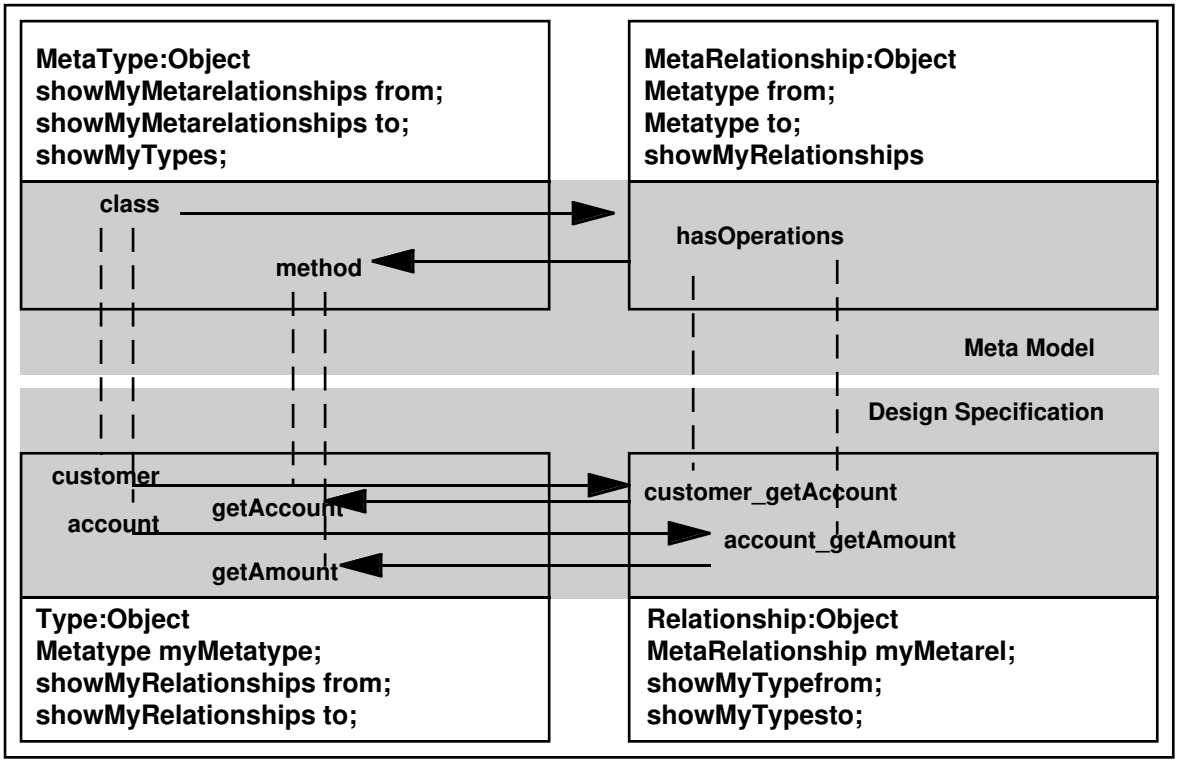

Figure 5: MEx Specification Database

MetaType, *MetaRelationship*, *Type*, and *Relationship* are the main classes of the specification database. The instances of the classes *MetaType* and *MetaRelationship* represent the meta model. *MetaType* represents the valid modelling concepts; *MetaRelationship* represents the relationships between them. In Figure 5, only the modelling concepts "class" and "method" and the relationship "hasOperations" between them are shown as instance of *MetaType* and Meta *Relationship,* respectively.
The classes *Type* and *Relationship* represent the conceptual specification level. In the example, the instances "motor" and "wheel" with their methods "run" respectively "turn" from Figure 3 (B) are shown. "motor" and "wheel" are classes in the conceptual design schema. Therefore, they have the type "class" (denoted by dashed lines), "run" and "turn" have the type method. "run" is a method of "motor", "turn" is a method of "wheel". This information is represented as instances of the class *Relationship*. In the meta level, "hasOperations" is an instance of *MetaRelationship*. This meta relationship associates "class" and "method" (denoted by arrows). The concrete relationships are captured as the *Relationship* instances "motor_run" and "wheel_turn".

The most interesting features of this kind of representation of a conceptual design schema are that it allows to

- analyse a design specification by means of queries, and
- update the conceptual design schema and the meta model dynamically (at run time).

In fact, the query

```
select * from x in Type where x.myMetatype = "class"
```

delivers the result "motor" and "wheel", i.e. all elements of type with the metatype "class". The update

```
insert "wheel" w into Type;
w.myMetatype="class";
```

defines the new class "wheel".

5.3 Design knowledge base

The design knowledge base is the core of MEx. It represents design knowledge in the form of measures, heuristics and transformation rules. It is important to note that measures, checking queries (of heuristics) and transformation rules are stored as queries respectively update specifications. The description of heuristic 11 contains the following checking query[5]:

```
select   x, y, b.from, a
from     x, y in type, a, b in Relationship
where    a.myMetarel. name="sendsMessage"
and      b.myMetarel. name="hasParts"
and      a.from = x and y in a.to
and      x in b.to and y in b.to
```

In MeTHOOD, such a query is specified in OQL (Object Query Language) [18]. It is applied to a selected design fragment. It delivers a non-empty result if the corresponding heuristic is violated in the fragment. The result is a description of all occurrences of violations of the heuristic. The specific format of the result is different for different heuristics. It is used to display violations of the heuristic to the designer, and to provide necessary information to transform the violating fragments. If a violation to a heuristic is found, it can be removed using a transformation rule. A corresponding transformation rule to the checking query above is implemented by the update

```
insert b in b.from, y;
remove b from x;
```

which removes the detected violation of the heuristic from the fragment.

Like the information in the specification database, the information in the design knowledge database can be updated. This means that users of MEx are able to modify the design knowledge included in MEx and provide their own measures, heuristics and transformation rules.

6 Related Work

The MeTHOOD method approach is related to various other quality approaches in software engineering. These include

- design patterns,

[5] Note that the meta model for this query is more complex than the meta model shown in Figure 5.

- heuristic collections,
- object-oriented methodologies, and
- design quality approaches.

6.1 Design patterns

A pattern is a [proven][recurring] solution in a context. Patterns attempt to capture (and distil!) design experience and best praxis from developers. The MeTHOOD integration schema also describes a relationship between a context, a problem and a solution. The context is described by heuristics, the problem is described by a set of measurements and the solution is described by transformation rules. However, MeTHOOD is different to *design* patterns as described by [19-21]. In MeTHOOD the structure of the solution (found in all design patterns) is not explicitly described. Instead of this we take the point of view that different solutions for a problem exist which often cannot be adequately described by a single general design pattern. We describe the solution *implicitly* as a process that marks bad locations in a design, and allows removing recognised misfit. In MeTHOOD, the process of "marking misfit" is supported by explicit, formal descriptions of the context. The process of finding locations in a design where a specific design pattern could/should be applied is not very strongly supported in current design pattern literature. In general, patterns should be better suited to support the "creation" part of design whereas MeTHOOD better supports the "refinement" part. Patterns are a high level language for building; MeTHOOD provides a high level language for systematic improvement and repair.

6.2 Heuristic collections

The most important collection of design heuristic is contained in [17]. The notion of heuristics and *transformation patterns* in [17] are similar to heuristics and transformation rules in MeTHOOD. The main differences between [17] and MeTHOOD are twofold:

- [17] uses *transformation patterns* (consisting of a "bad" and "good" pattern pair) to support correcting potential design flaws. MeTHOOD does not include an explicit description of the structure of the bad and the good pattern. Instead of this it uses a heuristic and its related transformation rules. This makes the conflict resolution concepts of MeTHOOD simpler because beside heuristics only transformation rules describing a process are needed. Furthermore they are more flexible because many different problem patterns can be recognised by a single heuristic and many different solution patterns can be generated.
- [17] suggests that the designer has to resolve conflicting advice of different heuristics and proposes only limited support for these kinds of decisions. MeTHOOD uses measurements to help during conflict resolution.

Other heuristic collection can be found in [22] and [23]: [22] includes a collection of nine design "principles": These "principles" include design ideals, organisational rules and checkable heuristics. Unlike MeTHOOD this collection does not include the notion of transformation rules. Some of the heuristics are supported by measures. [23] describes a prototypical tool called TOAD (Teaching Object-oriented Analysis and Design). This tool is able to examine static class properties captured in C++-header files based on a set of heuristics.

6.3 Object Oriented Methodologies

Most current methodologies, e.g. [5, 10, 24, 25], contain very useful example designs and implicit design advice e.g., "Avoid traversing associations that are not connected to the current class." [5]. Some recent methodologies like [26] contain collections of heuristics explicitly. The heuristics found in these methodologies are often on a high level and hard to check objectively, like "Build a system that mirrors the real world; and keep it that way." Descriptions of *how* a given design can be improved using the heuristics are not given. Other methodologies, e.g. [8], contain a collection of metrics or measurements. In most cases these metric collections in methodologies are similar to [27].

6.4 Design quality approaches

A large amount of approaches have been proposed in the areas of object-oriented metrics and quality models including [13, 28-30] and [14, 31-35]. Most of them use measurements and relationship models between measurements (quality models) to define what software quality is. The main difference to these approaches is that MeTHOOD not only includes a definition of quality and how it can be measured, it also describes explicitly how to improve it (using heuristics and transformation rules). There are some other quality approaches providing this kind of "active" design support, e.g. [15, 36-38]: [36] describes a rule for object oriented programming which is known as the "Law of Demeter". This rule is a restriction on the set of possible message receiver objects in a method. It is a typical example of a very powerful heuristic. [15] describes a set of important metrics that is used to measure the quality of an object-oriented design. The key idea is to measure the stability of class categories and to reduce dependencies from non-stable categories. [37] has implemented a tool that integrates static analysis and dynamic visualisation and checks for a variety of design principles like coding styles and architectural rules. The tool is used to verify that principles used for design are not obscured in the implementation, e.g. that a mediator pattern in design is a mediator pattern in implementation. [38] has extended the SQM (Software Quality Metrics) [31] model to a quality model including design rules. His model is called GRCM (Goal Rule Checklist Metric) model. GRCM provides assistance for developers and design inspectors. According to a checklist, they can evaluate the quality of a design and improve it. The focus of GCRM is on training and design inspection. GCRM does not include design heuristics. Furthermore, GCRM is focused more on organisational issues of quality design than MeTHOOD, e.g. the process also includes inspections by "external" inspectors.

7 Summary, Conclusions

We have described MeTHOOD, a new extension for object-oriented design methodologies. The core of this extension is the MeTHOOD integration schema and the MeTHOOD process. The MeTHOOD integration schema is used to integrate measures, heuristics, and transformation rules.

The benefits of this integration schema are the following: It provides effective support for design refinement and improvement. It reduces one of the most important weaknesses of pure measures, their passiveness. It provides concrete design advice helping to improve the value of a measure. It solves two major problems with pure design heuristics: It provides advice on how a design fragment can be transformed if a design heuristic is violated and gives clear base for decisions if two design heuristics with different interests collide.

The MeTHOOD integration schema is the foundation for the MeTHOOD process. This process supports designers monitoring their conceptual design schemata continuously, checking their conceptual design schema for design flaws and transforming flawed specifications.

Using the MeTHOOD process, designers

- detect more potential design flaws early in the design phase, and
- base design decisions on a clear rationale.

Besides the MeTHOOD integration schema and process, we have also described the MEx system, which is currently under development. MEx will be integrated in a commercially available design tool. It supports the design monitoring, checking and transformation activities of the MeTHOOD process. MEx is based on a continuously growing design knowledge catalogue

containing measures, heuristics and transformation rules. The description of this catalogue is too large to include it in this paper, however it is available at our web site.

We expect that usage of MEx and MeTHOOD leads to

- a higher learning curve for object-oriented designers,
- reduced costs and effects of design reviews, and
- better quality of resulting software in terms of detectable design flaws.

Currently, we are conducting a controlled experiment with four different commercial projects (two banking projects, an insurance project and a telecommunication project) to support this claim.

References

[1] J. Rumbaugh, "What is a method?," *Journal of object-oriented programming*, vol. 8 (6), 1995.
[2] B. Henderson-Sellers, "Who needs an object-oriented methodology anyway?," *Journal of object-oriented programming*, vol. 8 (6), 1995.
[3] P. Arnold, S. Bodoff, D. Coleman, H. Gilchrist, and F. Hayes, "An evaluation of five object-oriented development methods," *JOOP*, 1992.
[4] D. Champeaux, L. D., and P. Faure, "The Process of Object-Oriented Design," presented at Proc. of OOPSLA, 1992.
[5] J. Rumbaugh and e. al., *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
[6] P. Coad and E. Yourdon, *Object-Oriented Design*: Yourdon Press, Prentice Hall, Englewood Cliffs, 1990.
[7] P. Coad and E. Yourdon, *Object-Oriented Analysis 2nd Edition*: Yourdon Press, Prentice Hall, Englewood Cliffs, 1992.
[8] I. Graham, *Object-Oriented Methods*: Addison Wesley , Workingham, England, 1993.
[9] B. Henderson-Sellers and J. M. Edwards, "The Object-Oriented Systems Life Cycle," *Communications of the ACM*, vol. 33 (9), pp. 142-159, 1990.
[10] G. Booch, *Object-Oriented Design with Applications 2nd Edition*: Benjamin/Cummings, Menlo Park, CA, 1993.
[11] P. Haynes, "Detection an Prevention of Software Cancer in OO Systems," presented at OOPSLA 1996 Metrics Workshop, San Jose, 1996.
[12] A. Riel, "Design Heuristics and their relationship to patterns," presented at OOPSLA 1995, Austin, Texas, 1995.
[13] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions of software engineering*, vol. 20, 1994.
[14] N. Fenton, R. Whitty, and Y. Iizuka, *Software Quality Assurance and Measurement - A Worldwide Perspective*: International Thomson Computer Press, 1995.
[15] R. Martin, "Object-Oriented Design Quality Metrics: An Analysis of Dependencys," *Report on Object Analysis&Design*, vol. 2 (3), 1995.
[16] N. Fenton, "Software Measurement: Why A Formal Approach?," in *Formal Aspects of Measurement*, *Workshop in Computing*, T. Denvir, R. Herman, and R. W. Whitty, Eds.: Springer, 1991.
[17] A. Riel, *Object-oriented Design Heuristics*: Addison-Wesley, 1996.
[18] R. G. G. Cattell, *The Object Database Standard: ODMG-93*: Morgan Kaufmann Publishers, San Mateo, CA, 1994.
[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison Wesley, 1994.

[20] J. O. Coplien and D. C. Schmidt, *Pattern Languages of Program Design*: Addison-Wesley, 1995.
[21] J. O. Coplien and D. C. Schmidt, *Pattern Languages of Program Design 2*: Addison-Wesley, 1996.
[22] R. C. Martin, *Designing Object Oriented C++ Applications using the Booch Method*: Prentice Hall, 1995.
[23] C. Gibbon and C. Higgins, "Teaching Object-Oriented Design with Heuristics," *SIGPlan Notices*, vol. 31, 1996.
[24] I. Graham, *Migrating to Object Technology*. Wokingham UK: Addison-Wesley, 1995.
[25] D. W. Embley, B. D. Kurtz, and S. N. Woodfield, *Object-oriented Systems Analysis: A Model-driven Approach*: Prentice-Hall, 1992.
[26] D. F. D´Souza and A. C. Wills, "Component-Based Development using Catalysis," 1997.
[27] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, 1994.
[28] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*. New York: Prentice-Hall, 1994.
[29] D. H. Abbott, T. D. Korson, and J. D. McGregor, "A proposed design complexity measure for object-oriented development," Clemson University TR 94-105, 1994.
[30] W. Frakes and C. Terry, "Software Reuse: Metrics and Models," *ACM Computing SUrveys*, vol. 28, 1996.
[31] B. W. Boehm, *Characteristics of Software QUality*. Amsterdam: North-Holland, 1978.
[32] R. Harrison, "Assessing the Quality of Object-Oriented Programs," presented at OOPSLA Metrics Workshop, 1993.
[33] B. Kitchenham and S. L. Pfleeger, "Software Quality: The elusive target," *IEEE Software*, 1996.
[34] IEEE, "Standard for a Software Quality Metrics Methodology," IEEE, New York P-1061/D20, 1989.
[35] ISO, "Software Product Evaluation - Quality Characteristics and Guidelines for their Use," ISO, New York ISO/IEC DIS 9126, 1990.
[36] K. J. Lieberherr and I. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software Magazine*, 1989.
[37] M. Sefika, A. Sane, and R. H. Campbell, "Monitoring Compliance of a Software System With Its High-Level Design Models," presented at ICSE 96, 1996.
[38] I. Tervonen, "Support for Quality-Based Design and Inspection," *IEEE Software*, vol. January, 1996.