

# Measurement of modeling abilities

Ira Diethelm<sup>1,2</sup>, Leif Geiger<sup>2</sup>, Christian Schneider<sup>2</sup>, Albert Zündorf<sup>2</sup>

<sup>1</sup>Gaußschule  
Löwenwall 18a  
38100 Braunschweig

<sup>2</sup>Fachgebiet Softwareengineering  
Universität Kassel  
Wilhelmshöher Allee 73  
34121 Kassel

(ira.diethelm | leif.geiger | christian.schneider | albert.zuendorf)@uni-kassel.de  
www.se.e-technik.uni-kassel.de

## Abstract:

This paper discusses the difficulties of measuring modeling abilities within examinations. Modeling abilities are inherently difficult to measure since they imply cognitive processes that may not become evident in the result of a written examination. In addition, for a given problem there exists a wide variety of valid models that may just differ in the employed modeling language, technique, or paradigm. The models may just differ with respect to the aspects of the problem that are covered. Or the models may differ in the level of abstraction that has been chosen, e.g. UML level or code level. Even for a given modeling language and for clearly identified aspects that are to be covered and for a given level of abstraction there are still many possible solutions for a given problem that are difficult to compare and where it is difficult to judge their relative quality. This paper will mainly raise questions related to these problems. However, in addition we will describe a specific solution employed at the University of Kassel for grading the modeling abilities of 3rd term students.

## 1 Introduction

Modeling abilities are of major interest in the discussion of educational standards. Accordingly, measuring techniques for modeling abilities need to be created, evaluated and established. Generally, there are two different basic approaches: measuring the progress during the modeling process or evaluating the result. In the following article we focus on the results.

Many problems in this area are caused by the wide variety of valid solutions for one given problem. Different modeling languages may be used, different levels of abstraction may be chosen and there may actually exist multiple valid solutions for the same problem. In

this paper we do not present a solution in general but we show one possibility we found for measuring special UML models.

In chapter 2 we point out the initial situation and assumptions we made. Then we discuss the difficulties in measuring modeling abilities in general in Chapter 3. Therefore we rely on the measurement of mental models as a basis for information processing and problem solving. We also specify some requirements on measurement techniques that are concluded from discussion in this chapter.

Furthermore, in Chapter 4 we show some difficulties that we identified during our lessons in software engineering at the University of Kassel. In this course, we evaluated the modeling abilities of the students (not their knowledge in UML) on the basis of a homework consisting almost only of UML diagrams. To measure the amount and quality of the functionality modeled within a homework, we used so called *norm activities* that we describe in Chapter 5. Norm activities allow us to measure the size and quality of an UML interaction diagram (story boards and story diagrams).

In chapter 6, we resume and reflect which requirements are covered by our solution and which are not. We conclude with future work to be done.

## 2 Initial Situation and Assumptions

Any discussion of the measurement of modeling abilities requires a sound definition of the term modeling ability, first. In this paper, we consider modeling ability as the ability to capture an existing or described context and to create a mental model illustrating the given context as suitable as possible. Furthermore it requires to describe this mental model with a well-known and suitable modeling technique to make it accessible and assessable to others.

Our interpretation of mental models is based on [LWS96], where the authors prove that human information processing, thinking and concluding often takes place in entire models, which structures resemble the conditions of the given context.

In the area of software engineering, models are frequently implemented using a standard programming language and the size of such a program model is measured with function points or lines of code. From such a size measurement one derives the costs for the production of the model. Obviously, code size is no suitable measure for the students modeling abilities. In our opinion, today's programming languages deal with too many technical details and have a too low level of abstraction. In our opinion, a higher level of abstraction that is closer to the way of human thinking is needed in most aspects of education. Besides, we would like to be able to measure modeling abilities also with tests, which do not require an executable program and do not require too much effort to be suitable.

In addition, the length of an executable program is no measure for its quality. Compared with the size of a sample solution, longer solutions should be considered as worse compared to smart short solutions realizing the same functionality.

Teachers at school have very diverse evaluation criteria. In most cases an exercise is given

requiring a solution method and a result, e.g. the answer to a question. The work on the exercise is documented by the student and evaluated by the teacher. Usually points are assigned to the basic approach, the solution method and the result. The tasks are assigned to certain difficulty ranges and several of such tasks form a test. For example, the tasks of the German „Abitur“ have to be specified in this way (see [KMK04]). However, such an evaluation scheme measures the ability to apply a pre-defined solution method to an appropriate problem.

In our opinion, good modeling abilities will frequently result in a wide variety of solutions for a given problem where each solution has its own value and quality. Thus, an evaluation scheme for modeling abilities should not assume a certain standard solution but it should provide freedom for alternative solution approaches.

### 3 Difficulties in General

As discussed, the common test schema requires a fixed solution strategy and thus is not applicable to computer science. It differs from mathematics and other natural sciences since in computer science, for most modeling problems, there are many solutions to an exercise where each solution may be as good as the other. Thus, the measurement of modeling abilities requires flexible evaluation schemes that are able to deal with a wide variety of different solutions.

Second, a problem may be modeled at different levels of abstraction. One may use a fairly coarse grained UML model, e.g. only a class diagram, or a very fine grained level, e.g. a fully implemented program in a standard programming language. Solutions at such different levels of abstractions are not easily compared or graded with a single evaluation scheme.

Similarly, different schools or different teachers usually teach many different modeling techniques. This creates the problem of comparing and grading solutions to a problem that are described using different modeling languages. To overcome this problem, we need to find a small set of common modeling languages that unifies currently used modeling techniques and that allows to compare different solutions more easily. Ideally, for wide range examinations like a „Zentralabitur“ examination or a PISA test, all students should have similar experiences and skills in the modeling techniques employed in the examinations.

But it doesn't appear neither realistic nor meaningful to demand, that only one or two modeling techniques should be taught nation wide, just in order to be able to establish a simple evaluation scheme. Usually, educational standards should not be effected by evaluation methods, but vice versa. However, common nation wide educational standards would be of great value, anyhow. And if this is achieved, evaluation would benefit from it, too.

As an alternative to a specific modeling language, written or verbal natural language could be used to describe a solution to a given modeling problem. This could be used to seize and evaluate the mental model of students, cf. [HBB00]. However, [HBB00] points out that natural language is only suitable for the measurement of mental models under certain conditions, since some knowledge is difficult to verbalize. In our opinion modeling abili-

ties in the area of computer science raise a similar problem. In addition, the evaluation of a textual description requires an individual interpretation by the examiner. This creates problems for the comparability of grades given by different examiners.

[HBB00] also discusses free graphical representations of information, which they prefer in comparison to textual descriptions. Additionally [HBB00] points out the problems of unrestricted graphical representations: unrestricted graphics have to be interpreted by an examiner also. Still, [HBB00] assumes graphical descriptions to be a comparatively intuitive approach for learning if they have pre-defined meanings for the used symbols.

## 4 Our Modeling technique: Story Driven Modeling

We have developed a tool supported software process called Fujaba Process (FUP) which we teach in our courses at school as well as at the University of Kassel [DGZ04]. The underlying modeling technique is called Story Driven Modeling (SDM). In this paper, we report our experiences with evaluating SDM models, created by our students as a homework for a UML lecture.

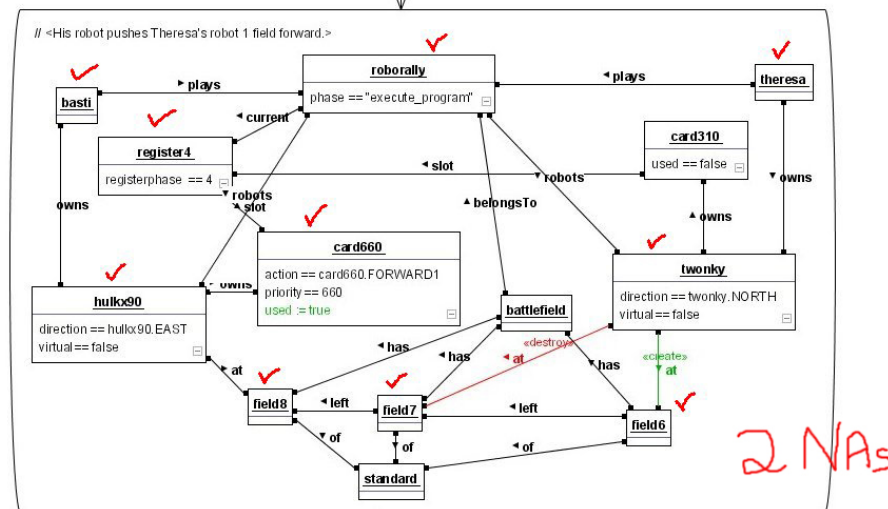
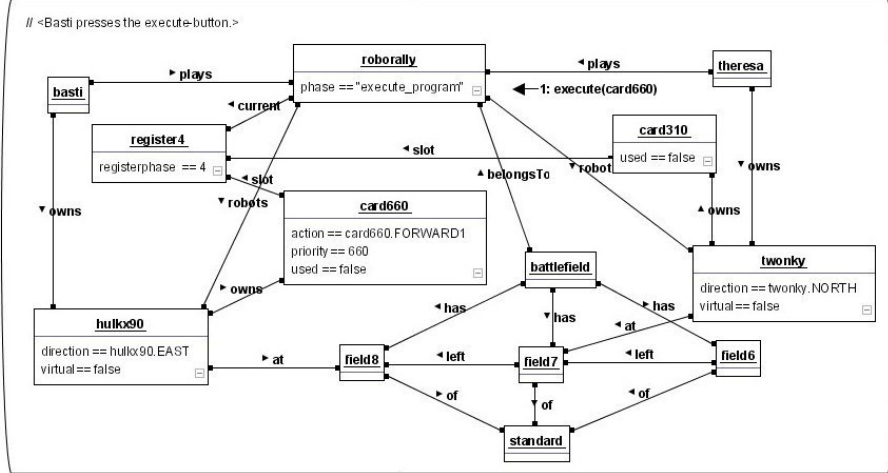
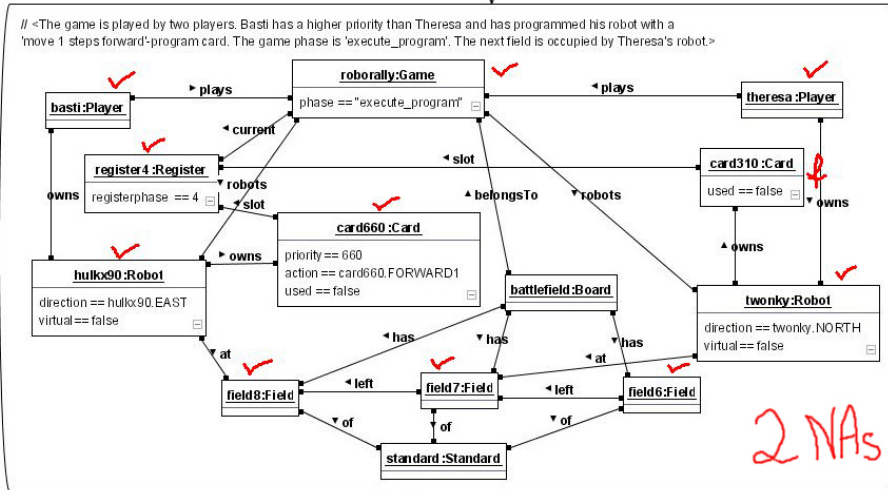
The students had to model the board-game „Roborally“ using our CASE tool Fujaba [Fu02] and the Fujaba Process. Since the Roborally game consist of many rules where some are very complex, the students did not have to model the complete game, but only parts of it. Since the students could freely decide which parts of the system they liked to model and how they wanted to model the functionality, we had to find an approach to compare and evaluate these models.

To illustrate our evaluation criterions, we will take a closer look at the FUP and the UML diagrams used in this process:

The FUP starts with the identification of usecases within the problem domain. For every usecase one or more textual scenarios (descriptions of example runs) are written. Then the developer has to translate these textual scenarios into so called story boards. A story board is a sequence of object diagrams, which shows the evaluation of the object structure comic-strip alike. Figure 1 shows a story board for a scenario where a robot moves and pushes another robot. The example used here is originally taken from a student group of our Roborally project.

The embedded object diagram in the first activity in figure 1 models the start situation of this scenario. Note, the corresponding textual description is automatically copied as comment into this activity. Our students have modeled the initial object structure using an object *roborally* of the class *Game*, which represents the functionality of the game. Two *Player* objects *basti* and *theresa* are linked to the game via a *plays* link. The robots are located next to each other on the fields *field8* and *field7*. The robot *hulkx90* has a card which tells him to move one field forward (modeled by the action attribute of object *card660*).

move\_forward1\_push1





destruction of links / objects using `<<create>>` and `<<destroy>>` markers. The card object *card660* is also marked as used by setting the *marked* attribute to *true*.

In the next activity the pushing robot *hulkx90* is also moved one field forward to field *field7*. It is not obvious here, why our students have chosen to make this a sole step. This action could as well have been executed in the activity above.

The last activity always models the result situation which has to be reached if the scenario is successfully executed. Here, the robots *hulkx90* and *twonky* have both moved one field forward and the card *card660* has been marked as used.

From the story boards, the main class diagram may be derived, automatically. The developer may refine the class diagram by adding inheritances, changing cardinalities etc.

We also suggest a systematic approach how to derive the behavior specification (here: the method bodies) from the story boards (see [DGZ02, Zü01]). Method body specifications are modeled using so called story diagrams. A story diagram is a UML activity diagram with UML collaboration diagrams embedded into the activities. The activity diagram specifies the control flow whereas the collaboration diagrams model the changes done to the object structure. Figure 2 shows such a story diagram.

The story diagram of figure 2 models the behavior of the method *doIHaveToPush()* of class *Robot*. This method is a helper method needed by our students to implement the pushing of robots as specified in the story board of figure 1. The method returns *true* if there is a robot which must be pushed in front of the robot on which the method was called, and *false* otherwise.

The first activity checks whether the virtual attribute of the object of class *Robot* on which the method has been called (called *this* in Java and Fujaba) is set. If this check succeeds the activity is left via the *[success]* transition. In this case, the method is left and returns *false* since virtual robots do not interact with other robots in the Roborally game. Otherwise the activity is left via the *[failure]* transition.

The collaboration diagram in the second activity tries to identify the specified object structure. The matching is started at the *this* object. From there the *robots* link is followed. If an object of class *Game* is found, it is called *roborally*. From the *roborally* object a robot is searched using the *robots* edge, which has an *at* link to the object *nextField*. This robot is then called *robotX*. Note, the object *nextField* is already known to the system since it has been passed as parameter. In Fujaba known objects are visualized by omitting the class name after the object name. Such objects are called „bound“.

If this object structure analysis fails, the activity is left via the *[failure]* transition and again *false* is returned. Otherwise the *[success]* transition is taken to the third activity. Here it is checked if the attribute *virtual* of the bound object *robotX* (known from the object search in the previous activity) has the value *true*. If yes the method returns *false* and *true* otherwise.

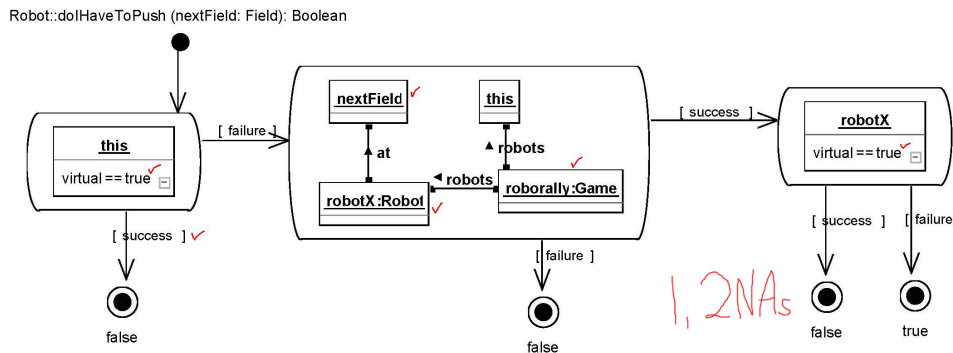


Figure 2: Story diagram for method *doIHaveToPush()*

From the class diagrams and the story diagrams the Fujaba CASE tool automatically generates executable Java code, which may be compiled and then tested using our object browser DOBS.

From the story boards the Fujaba CASE tool generates also automatically JUnit test specifications ([Gei04]). This generated tests check whether or not the implementation (modeled by story diagrams) covers the scenario specified by the corresponding story board. Using this test generation enables us to verify easily which scenario has been completely modeled and which not. But we still have no information about the complexity of a particular scenario and of the quality of the model.

At the end of the Roborally project, we had to evaluate the results presented to us by our students. This brought up the following questions:

- How should we evaluate story boards and story diagrams? Just giving points for every object, link, attribute condition etc. would just measure the size of the diagrams and would not take the real complexity of the modeled system and of the quality of the model into account.
- How can we measure functionality of systems that do not have a user interface and are therefore not testable by humans?
- How can we evaluate pure models that do not have an implementation?
- How can we measure the complexity of a scenario / method specification?

## 5 Our Solution: Norm Activities

We introduced the term of a „norm activity“ (NA) to evaluate the modeling abilities of our students. An NA is a group of five objects within a story board or story diagram which are involved in a non-trivial change to the object structure. Such changes may be changes to

attribute values, creation or destruction of links or objects and messages sent to objects. Objects not needed for these changes as well as symmetries do not count for NAs.

For passing the Roborally project, each of our students had to model 10 NAs with story boards and another 10 NAs with story diagrams within roughly 2 weeks of work. To be able to count NAs of story boards and story diagrams easily and repeatably, we set up a list containing criteria on how to count and how to identify symmetries:

**Criteria list for story boards:**

- If one step of a story board has about 5 objects that are needed for a sound realization the step counts as 1 NA.
- For a multiple of 5 objects the steps counts more NAs according to the factor (10 needed objects → 2 NAs).
- If a story board has no significant new elements, it is not or only partially counted. This criterium is needed to identify symmetries between story boards.
- Invocation and result situation are not counted. This is because the invocation is mostly symmetrical to the start situation and the result situation contains the same objects as the previous steps.
- If one could describe multiple steps in one step, the NAs are counted only once. This again identifies symmetries.
- Trivial story boards count less or even no NAs (e.g. only one changed attribute). In this case, the scenario was too simple and does not fulfill the requirements for containing NAs.
- For start activities we count all objects that are necessary for the subsequent steps. Objects that are not counted in any of the subsequent steps are not counted in the start situation either. This criterium facilitates to identify the part of the start situation that models only the context of the scenario but that is not used later on.
- Parts of a story board which do not fulfill the requirements description do not count. Here, the correctness of the model affects the counting of NAs.

**Criteria list for story diagrams:**

- Each object that is needed for a sound realization counts 0.2 NAs. This does again lead to 1 NA for a group of five objects.
- Branches, Loops and For-Each-Activities count 0.2 NAs. Since control-flow plays an important role when modeling behavior, constructs creating control-flow are counted as well.

- Methods that do not work (no green JUnit bar) are not counted.  
If a method has not passed the tests automatically generated from the story boards, it does not cover this specific scenario and so this method is erroneous within the small part of the modeled system.
- Unusable parts of methods count nothing.  
Our students tend to model branches that are never reached. Of course such parts of a model are not counted.
- Redundant structures of any type are not counted.  
This criterium should avoid symmetrical parts to be counted more than once.
- If one can model multiple story pattern in a single one, only one story pattern is counted.  
This again identifies symmetry.
- Case differentiations with very small differences are counted only once.  
Some of our students modeled huge switch-case constructs with only little differences between the different cases (e.g. move by one, move by two, move by three fields). Of course, such constructs are only counted once.
- The *this* object does not count.  
This is because the *this* object is needed in most cases to start an object structure analysis and does not stand for any modeling effort.

Applying the criteria above to the story board of figure 1 leads to a total of 4 NAs. We start counting NAs with the first step which is the third activity in Figure 1. As mentioned in chapter 4, the first and the second step (activities three and four) of the story board can be combined. So, according to the fifth criterium of the list above, every object is only counted once. Combining the two steps would result in figure 3.

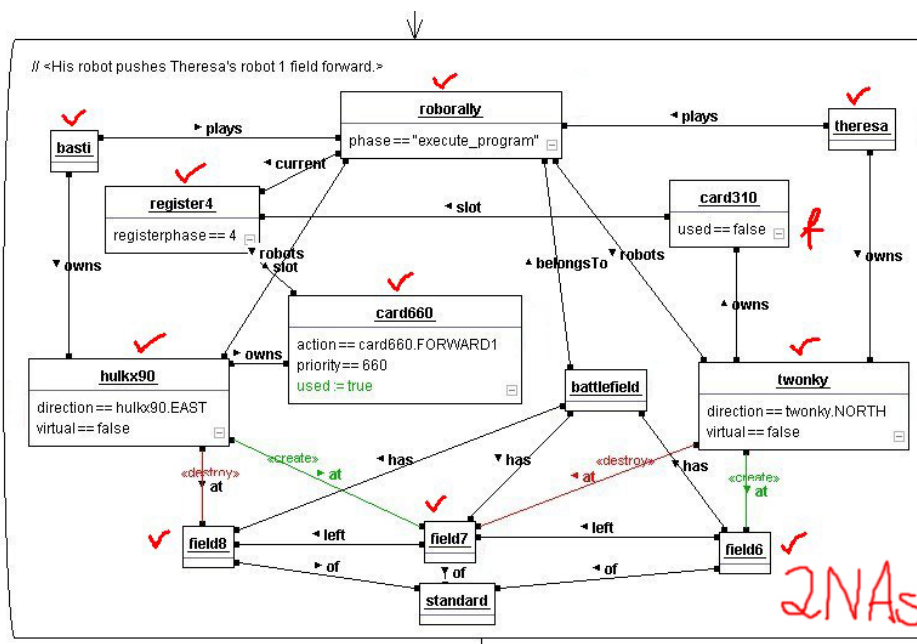


Figure 3: Combination of step 1 and 2 of figure 1

Every object with a red check mark is part of an NA. The object *battlefield* does not count since it does not take part / is not important for the changes to the object structure. The object *card310* might be important for choosing who is next because according to the Roborally rules, the card with the highest priority is played next. But because the students have not modeled a priority of *card310* (using the priority attribute) this object is useless here and therefore does not count. The object *standard* is not part of an NA as well. Of course, the information on the type of field on front of a robot, if it contains walls or holes is important for moving. But here the students have modeled an *instance of* relationship explicitly as a link. Inheritance would have been the better choice here and that is why the *standard* object is not needed here and does not count for NAs. So this step contains 10 objects counting for NAs. Since we have four changes to links and one attribute value assignment, these changes are non-trivial even for 2 NAs and therefore this step counts the whole of 2 NAs. Invocation and result situation do not count for NAs, so since this step is now the only one, only the start situation still has to be counted. According to criterium seven of the above list, we count the elements of the start situation that are employed in later steps. In this case this also sums up to 2 NAs. We end up with a total of 4 NAs for the whole story board. This means that two story boards of this complexity and one simpler one would be sufficient to pass the project concerning story boards.

For understanding the evaluation of story diagrams, we have a look at figure 2. By negating the attribute assertions, the three different branches can be combined to one. This would result in the story diagram in figure 4, which has obviously the same behavior as the one of figure 2. So, we use this diagram for counting NAs.

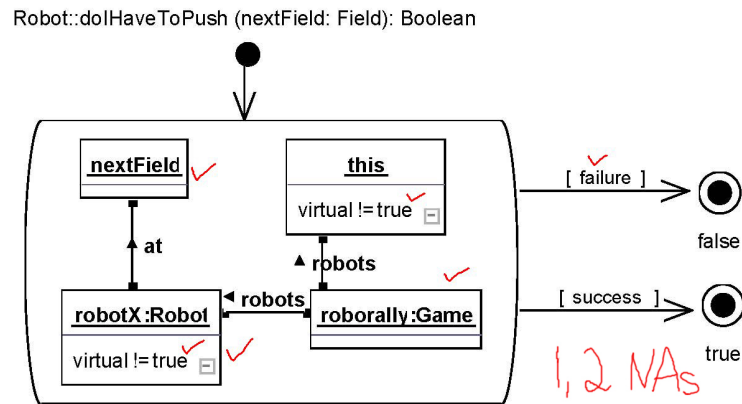


Figure 4: Simplified version of figure 2

Every object except the *this* object and every attribute assertion counts 0.2 NAs. The branching condition again counts 0.2 NAs. So this diagram would have a total of 1.2 NAs.

Using the so counted NAs enables us to measure the size of the model created by our students while ignoring symmetries and useless parts of their models. To also measure the quality of their models, we made the following considerations:

The size of a standard solution to some modeling task may be considered as a measurement for the complexity of that task. This allows to measure the complexity of the modeling tasks addressed by a homework and enables a comparison of the quality of the homework with respect to the standard solution.

Concluding, standard solutions provide

- a measure for the complexity of the task on the one hand
- a measure for the weight of errors or unworked task on the other hand

The comparison of size (in NAs but also in pages or LOC) of the standard solution with the solution of our students gives a hint of the quality of the presented solution. Comparison with the standard solution can easily answer the question: Was a long winded model chosen that needs much more and more complex branches and contains much „Cut-Copy-Paste” code?

Concluding, we found a way to measure complexity of models created with our modeling technique. We also made some progress in measuring the quality of such models. We found out that the intuitively indicated rating by the teacher and our rating by norm activities gave similar results. Another interesting observation was that for our modeling technique models of static contexts, e.g. class diagrams, are less interesting and of less effect to the measurement than models of dynamic processes.

Our approach also allows the evaluation of partly modeled systems which do not need to be runnable. It allows the evaluation of pure models which do not need to be created using a CASE tool. Such models also may only be specified e.g. on paper.

Concluding, we think, that using NAs is a step forward to achieve objective measurement of models, but lot of work still has to be done.

## 6 Summary and future work

As discussed, there are many open problems for the measurement of the modeling abilities of students. In our opinion, the following steps should be executed in order to improve this situation:

- modeling abilities should be measured at a higher level of abstraction as provided by e.g. programming languages. Today's programming languages deal with too many tiny technical problems that are not related to modeling abilities. In addition, the creation of an executable model for an even modest real world problem in an usual programming language requires just too much effort to be applicable for common examinations.
- we need to measure modeling abilities and not modeling language skills. Thus, we need an intuitively usable modeling language that does not require several months of training and that does not restrict the solutions to a given problem.
- we need to be able to measure the complexity of a modeling problem in order to be able to classify examination tasks into categories of difficulties.
- equipped with such an ideal high level modeling language, we still need a simple evaluation scheme that allows to grade students independent from the examiner's personal interpretation of the provided solution.

Currently, we deal with these problems in our courses at the Gaußschule Braunschweig, a secondary school, and at the University of Kassel using an adapted cut-out of the UML modeling language. This language provides a reasonable level of abstraction, however it still deals with too many technical details. In general our language still requires about two months of learning, thus it still needs to be simplified. Our first experiences in measuring the size of sample solutions in order to judge the complexity of an examination task are promising, but this needs further research. Our evaluation scheme for student projects still lacks repeatability and simplicity. However, the measurement results reflected our individual quality impressions of the solutions and the evaluation scheme was systematic enough to be understood by our students and to create the impression of a fair and reliable scheme independent from personal interpretations. Still, this scheme needs to be facilitated.

## References

- [Ba98] Helmut Balzert: Lehrbuch der Softwaretechnik 1, 2. Aufl., Spektrum Verlag, Heidelberg, 1998
- [DGZ02] I. Diethelm, L. Geiger, A. Zündorf: UML im Unterricht: Systematische objektorientierte Problemlösung mit Hilfe von Szenarien am Beispiel der Türme von Hanoi; in Forschungsbeiträge zur "Didaktik der Informatik" - Theorie, Praxis und Evaluation; GI-Lecture Notes, pp. 33-42 (2002)
- [DGZ04] I. Diethelm, L. Geiger, A. Zündorf: Systematic Story Driven Modeling, a case study; Workshop on Scenarios and state machines: models, algorithms, and tools, ICSE 2004, Edinburgh, Scotland, 2004.
- [Gei04] L. Geiger: Automatische JUnit Testgenerierung aus UML-Szenarien mit Fujaba, Diplomarbeit vorgelegt bei Albert Zündorf, Universität Kassel, 2004
- [Fu02] Fujaba Homepage, Universität Paderborn, <http://www.fujaba.de/>.
- [GI00] GI - Fachausschuss „Informatische Bildung in Schulen“: Empfehlungen für ein Gesamtkonzept zur informatischen Bildung an allgemein bildenden Schulen; Gesellschaft für Informatik, 2000
- [HBB00] S. Hillen, K. Behrendes, K- Breuer: Systemdynamische Modellierung als Werkzeug zur Visualisierung, Modellierung und Diagnose von Wissenstrukturen; in H. Mandl, F. Fischer (Hrsg.) Wissen sichtbar machen - Wissensmanagement mit Mapping-Techniken, Hogrefe Verlag, Göttingen 2000
- [Hu00] P. Hubwieser: Didaktik der Informatik - Grundlagen, Konzepte, Beispiele, Springer Verlag, Berlin, 2000.
- [KMK04] Kultusministerkonferenz: Einheitliche Prüfungsanforderungen in der Abiturprüfung Informatik, Beschluss vom 01.12.1989 i.d.F. vom 05.02.2004
- [LWS96] G. Lier, S. Werner, U. Sass: Repräsentation analogen Wissens im Gedächtnis; in D. Dörner, E. van der Meer (Hrsg.) Das Gedächtnis, Hogrefe Verlag, Göttingen, pp. 75-125 (1996).
- [SN02] C. Schulte, J. Niere: Thinking in Object Structures: Teaching Modelling in Secondary Schools; in Sixth Workshop on Pedagogies and Tools for Learning Object Oriented Concepts, ECOOP, Malaga, Spanien, 2002.
- [Zü01] A. Zündorf: Rigorous Object Oriented Software Development, Habilitation Thesis, University of Paderborn, 2001.