Towards Deriving Programming Competencies from Student Errors

Marc Berges*, Michael Striewe†, Philipp Shah*, Michael Goedicke† and Peter Hubwieser∗

* TUM School of Education
Technical University of Munich
Arcisstr. 21, 80333 München, Germany
Email: {berges,philipp.shah,peter.hubwieser}@tum.de

†The Ruhr Institute for Software Technology
University of Duisburg-Essen
Gerlingstr. 16, 45127 Essen, Germany
Email: {michael.striewe,michael.goedicke}@paluno.uni-due.de

Abstract—Learning outcomes are more and more defined and measured in terms of competencies. Many research projects are conducted that investigate combinations of knowledge and skills that students might learn. Yet, it is also promising to analyze what students might fail to learn, which provides information about the absence of certain competencies. For this purpose, we are evaluating the outcomes of automatic assessment tools that provide automatic feedback to the participating students. In particular, we analyzed the errors of the students that participated in an introductory programming course. The 604 students participating in the course had to solve six tasks during the semester, resulting in a total of 12274 submissions. The error analysis is done by evaluating the data from the automatic assessment tool JACK, which provides automatic feedback on programming tasks. To derive information about prospective competencies, we conducted a qualitative analysis of the different errors the students made in their solutions. The results provide interesting insights into missing competencies. In further research our findings have to be validated by investigating the cognitive processes involved during programming.

I. INTRODUCTION

During the last years, learning outcomes have been more and more defined and measured in terms of competencies. Aiming to define programming competencies based on empirical research, many research projects have been conducted to investigate the combinations of knowledge and skills students might learn. Yet, it seems also promising to analyze what students might fail to learn, which might provide information about the absence of certain competencies. In the presented study we show the first results of a qualitative analysis of the error messages that were gathered from the automatic assessment system JACK. We introduce a categorization process on error messages adapted from the qualitative analysis methodology of Mayring [7]. The resulting groups have to be treated in different ways to extract the underlying competencies. Particularly, the deriving process of the missing competencies is different for some specific error groups. In further research the process has to be automated. Therefore, the treatment of these specific error groups has to be investigated further.

II. BACKGROUND AND RELATED WORK

A. Competence

Unfortunately, the term “competence” has a wide range of definition. Although this is not the place to report all the different understandings, we want to clarify how the term is understood in the context of our study. According to Weinert, a competence in human and social sciences is “a roughly specialized system of abilities, proficiencies, or skills that are necessary to reach a specific goal. This can be applied to individual dispositions or to the distribution of such dispositions within a social group or an institution” [6].

Müthing et al. propose a competence structure for object-oriented programming [7]. The authors describe six dimensions derived from an empirical research of novice programmers’ structural knowledge on object-oriented programming. Two of the six dimensions are further exemplified by difficulty levels based on the SOLO taxonomy.

B. Automated Assessment

Automated assessment is a common feature of introductory courses at universities to cope with the high numbers of participants. For the subject of programming exercises, a wide range of e-assessment systems exist that detect errors by executing test cases and analyzing source code [7]. In general, automated systems are not worse than human teachers in grading and feedback generation [8]. Fine-grained feedback and marking schemes have been proven to be effective in this context as well [7].

Quantitative and qualitative error analysis as presented in this paper have already been performed to find frequent blocking syntax errors [7], derive main test cases [7], and find correlations to perceived exercise difficulty [7].

C. Errors and Misconceptions

A misconception is a conception which is not in line with accepted scientific notions [7], e.g. some students are arguing that the increase of a counter variable of a loop after the loop is done still affects the loop [7]. It is important to mention that not every error is a misconception. If a student just forgets a curly bracket at the end of a loop in Java he might not have a wrong or alternative conception of loops at all.
In computer science education there are a lot of publications about misconceptions, but they are merely collections of misconceptions without a link to competencies or reasons; e.g. Ragonis and Ben-Ari [?] listed about 58 difficulties and misconceptions.

Holland et al. [?] where some of the first authors who collected misconceptions using an object oriented programming language (Smalltalk). Fleury [?] searched for misconceptions using Java as a programming language and referred to[?]. Also Ragonis and Ben-Ari’s [?] research about misconceptions was based among others on Holland et al. [?] and Fleury [?].

Danielsiek et al. [?], who referenced to Holland et al. [?], and Ragonis and Ben-Ari [?], investigated misconceptions on complex data structures like heaps and binary search trees.

Gal-Ezer and Zur [?] and later Shah et al. [?] succeeded in connecting intuitive rules, which had effect in science and mathematics, to misconceptions in computer science. They discovered ([?]) and verified ([?]) among others the basic misconceptions “two programs containing the same statements (even if in different order) are equally efficient” following the intuitive rule “same A, same B” [?], which was discovered in science education by Stavy and Tirosh.

A new study on novices’ conceptions of objects and classes is conducted by Xinogalos [2]. Apart from her investigation of the conception she provides a broad overview on the current state of the investigation of misconceptions related to objects and classes.

III. DATA COLLECTION

A. Course Design

All data used in this study was collected in a programming lecture in the winter term 2014/15. The lecture is designed to be an introductory course to object-oriented programming for first year students. The programming language used throughout the course is Java. The course is designed for students that have no prior knowledge in programming. The course follows the “objects first” principle and thus in particular covers the following topics in the given order: (1) Classes, fields and methods; (2) Primitive data types and expressions; (3) Control structures; (4) Simple data structures; (5) Inheritance.

The course consists of 4 hours of lecture plus 2 hours of exercise per week. Additional tutoring sessions are offered for voluntary participation. During the term, students must pass at least three out of six tests to be allowed to take part in the final exam. For each test, there is one mandatory homework exercise and also a set of voluntary exercises. The six mandatory homework exercises are the ones used in this study.

B. JACK

The e-assessment-system JACK provides automated feedback to exercise solutions using different means of static and dynamic program analysis [?]. In particular, it is able to (1) execute the standard Java compiler and pass all compiler messages to the student; (2) perform teacher-defined rule-based checks on the syntax graph of a solution and provide arbitrary feedback on unwanted or missing code structures; (3) execute teacher-defined test cases and provide both arbitrary feedback on test results and generic feedback on Java runtime exceptions.

While test cases focus on functional aspects of a solution, rule-based checks on the syntax graph can be used for a variety of aspects: Standard rules that are independent from the actual task can provide feedback e.g. on style issues such as variable names starting with capital letters or on control structure issues such as an conditional statement that has two empty branches. Task specific rules can be added by a teacher for example to enforce using a particular loop statement in a method or to check that a given method signature has not been changed by the student.

JACK can execute the test cases as white-box-tests and record all steps and variable values occurred during execution. From these data it can also detect unused code in terms of lines of code that are not reached by any test case and it can create visualizations of data structures. However, since these means of feedback do not contribute to the grade computed for a solution and do not involve plain text feedback statements, they are not considered in more detail in this study. The same goes for any software metrics for submissions computed by JACK, as these has not been displayed to the students but just used as an experimental feature for teachers so far.

C. Data Preparation

In adaption of the qualitative content analysis methodology of Mayring [?] we prepared our data to find groups of student errors. Mayring proposed a step-by-step model of summarizing content analysis. Step 1 is the “determination of the units of analysis”. In our case, this is the error message provided by the JACK system. The “paraphrasing of the content-bearing text passages” is step 2. Here, all unnecessary information of the original message, like a html link that points to a trace file, is dropped. In step 3 the “envisaged level of abstraction” is determined and the paraphrases are generalized. For this step, we generalized the error messages by substituting specific values with general parameters. The steps 4 to 6 deal with the transition of the generalized paraphrases into a category system.

The following sample explains the steps in more detail. All the error messages are translated from German to English by the authors.

Unit determination

The unit determination is conducted on the complete error output of the JACK system. Besides the position of the error which is not of interest for deriving competencies, and a html link which points to a trace file of the test case, the error message is conducted.

Error in miniproject 1 - Circle.containsPoint(): A circle with center (3.5,000000) and radius (4.000000) does not contain the point (-1.000000, 1.000000) but your method returns 'true'. <p><a>
Paraphrasing
As mentioned above the paraphrase of an error in the program code is an error message from the JACK system. In the provided example the error message looks like:

```
Error in method Circle.containsPoint(): A circle with center (3,500000) and radius (4,000000) does not contain the point (-1,000000) - 1,000000 but your method returns 'true'.
```

Generalization
In the generalization step all specific values are replaced by generic parameters. The example error from above changes into the following:

```
Error in method Circle.containsPoint(): A circle with center (CENTER) and radius (RADIUS) does not contain the point (POINT) but your method returns BOOLEAN.
```

Categorization
The group of the error in the given example is calc as there is a wrong calculation of the boolean return value in the program code.

In order to conduct further investigations towards competencies the program code that is related to the error is needed. So, as an explanation in the sense of Mayring [?] the program code the participants handed in is added to the dataset. The following program code is related to the example above. The error message occurs in line number 2.

```
1 public boolean containsPoint(Point point) {
2     if (point.getDistance(point)<=radius) {
3         return true;
4     }
5     //if (location>=getLocation(point)) {
6         //return true;
7     }
8     else return false;
11 }
```

IV. RESULTS
In total 31476 error messages were conducted by the JACK system. 1332 of these were excluded because they were either no errors but warnings, or the error message was incomplete and because of that without any sense, or the error occurred because of restrictions made by the task (e.g. no packages were allowed, predefined signatures must not be changed). The remaining messages can be reduced to 1891 unique error messages which can nonetheless contain different underlying errors and because of that different missing competencies. The generalization led to 221 generic error messages. After generalizing, the remaining errors were grouped together in 35 groups according to similarities in the messages. Each group must be treated in its own way for deriving the underlying missing competencies. This is especially important for future automatic analyses. The following tables present the groups and a generic message example. Table I includes all groups where no further analysis steps are needed; the missing competence can be directly examined without interpretation of either the error messages or the program code. In contrast, Table II displays all the error groups that need further analysis and interpretation of the program code for examining the underlying missing competencies. Additionally, in both tables the absolute frequency of occurrence is presented.

### Table I. Error Groups Representing the “CLEAR” Cases with Sample Generalized Messages

<table>
<thead>
<tr>
<th>group</th>
<th>anchor example</th>
<th>abs. freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax</td>
<td>Syntax error on token TOKEN, TOKEN expected</td>
<td>1177</td>
</tr>
<tr>
<td>resolve 1</td>
<td>NAME cannot be resolved to a variable</td>
<td>491</td>
</tr>
<tr>
<td>type</td>
<td>Type mismatch: cannot convert from TYPE1 to TYPE2</td>
<td>177</td>
</tr>
<tr>
<td>resolve 3</td>
<td>NAME cannot be resolved to a type</td>
<td>136</td>
</tr>
<tr>
<td>resolve 2</td>
<td>NAME cannot be resolved or is not a field</td>
<td>95</td>
</tr>
<tr>
<td>resolve 0</td>
<td>NAME cannot be resolved</td>
<td>59</td>
</tr>
<tr>
<td>static</td>
<td>Cannot make a static reference to the non-static field</td>
<td>26</td>
</tr>
<tr>
<td>initialization</td>
<td>The local variable VARIABLE may not have been initialized</td>
<td>21</td>
</tr>
<tr>
<td>operator</td>
<td>The operator OPERATOR is undefined for the argument type(s) ARGTYPES</td>
<td>20</td>
</tr>
<tr>
<td>assignment</td>
<td>The left-hand side of an assignment must be a variable</td>
<td>19</td>
</tr>
<tr>
<td>duplicate</td>
<td>Duplicate local variable VARIABLE</td>
<td>16</td>
</tr>
<tr>
<td>class</td>
<td>The public type TYPE must be defined in its own file</td>
<td>7</td>
</tr>
<tr>
<td>import</td>
<td>The import IMPORT cannot be resolved</td>
<td>6</td>
</tr>
<tr>
<td>modifier</td>
<td>Illegal modifier for parameter PARAMETER; only final is permitted</td>
<td>6</td>
</tr>
<tr>
<td>unreachable</td>
<td>Unreachable code</td>
<td>6</td>
</tr>
<tr>
<td>void</td>
<td>void is an invalid type for the variable VARIABLE</td>
<td>5</td>
</tr>
<tr>
<td>abstract</td>
<td>The type TYPE must implement the abstract method METHOD</td>
<td>3</td>
</tr>
<tr>
<td>constant</td>
<td>Invalid character constant</td>
<td>2</td>
</tr>
<tr>
<td>invoke</td>
<td>Cannot invoke METHOD on the array type TYPE</td>
<td>2</td>
</tr>
<tr>
<td>cond.state.</td>
<td>In the file FILE the conditional statement in line LINE does not contain any code in both branches and because of that it is useless.</td>
<td>1</td>
</tr>
<tr>
<td>string</td>
<td>String literal is not properly closed by a double-quote</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table II. Error Groups Representing the “UNCLEAR” Cases with Sample Generalized Messages

<table>
<thead>
<tr>
<th>group</th>
<th>anchor example</th>
<th>abs. freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculation</td>
<td>Error in method Circle.containsPoint(): A circle with center (CENTER) and radius (RADIUS) does not contain the point (POINT) but your method returns BOOLEAN.</td>
<td>21885</td>
</tr>
<tr>
<td>reference</td>
<td>In class CLASS in method METHOD in line LINE you tried to access an attribute or method of an object that does not exist. Instead of pointing on the desired object the specific variable point on ‘null’.</td>
<td>1733</td>
</tr>
<tr>
<td>field</td>
<td>You tried to access an array index in class CLASS in method METHOD in line LINE that is not available. In an n-elements array the smallest possible index is 0 and the biggest possible index is n-1.</td>
<td>1123</td>
</tr>
<tr>
<td>getter null</td>
<td>Your implementation of GETTER returns null.</td>
<td>880</td>
</tr>
<tr>
<td>constructor</td>
<td>The call of your “student-constructor” initializes the following field in a wrong way: FIELD</td>
<td>695</td>
</tr>
<tr>
<td>loop</td>
<td>Your program calls itself in the class CLASS in method METHOD in line LINE with the same parameter values and because of that causes an endless loop.</td>
<td>364</td>
</tr>
<tr>
<td>getter</td>
<td>The method GETTER returns a wrong value (VALUE)</td>
<td>360</td>
</tr>
<tr>
<td>setter</td>
<td>The method SETTER sets the field ARRAY in a wrong way. After the call of SETTER the field ARRAY has the value VALUE</td>
<td>317</td>
</tr>
<tr>
<td>return</td>
<td>This method must return a result of type TYPE</td>
<td>252</td>
</tr>
<tr>
<td>method</td>
<td>The method METHOD in the type TYPE is not applicable for the arguments ARGTYPES.</td>
<td>184</td>
</tr>
<tr>
<td>exception</td>
<td>The call of the method Train.boardPassengers() on an empty train causes an EXCEPTION.</td>
<td>34</td>
</tr>
<tr>
<td>sort</td>
<td>The array of the method METHOD does not correctly sort the men depending on their rank.</td>
<td>20</td>
</tr>
<tr>
<td>structure</td>
<td>A left successor is missing.</td>
<td>19</td>
</tr>
<tr>
<td>div 0</td>
<td>The division '/' by 0 and the modulo calculation '%', at the division by 0 are not allowed mathematical operations.</td>
<td>2</td>
</tr>
</tbody>
</table>
The basic idea behind deriving competencies from student errors is that we assume missing competencies as a reason for most errors. Because of that we had a closer look on the original errors that lead to the previously described error groups. Here, a few examples for the examination process of the competencies are presented. The steps from the source code to the error message and finally to missing competences are shown for different error groups. To provide an overview we selected "clear" as well as "unclear" groups with regard to foster our idea of deriving the competencies from the error in the program code.

A. Sample: “syntax”

Program code:
```java
1 public void setLocation(Point newLocation) {
2     location = newLocation;
3 }
```

Generalized Error Message: IDENT cannot be resolved to a variable.

Underlying Error: The assigned parameter is written in lower case (line 2)

Possible missing competency: The students are able to implement variables, attributes and parameters with attention to the case-sensitivity of Java.

B. Sample: “resolve 1”

Program code:
```java
1 public void addValue(int value) {
2     if (i == data2.length - 1) {
3         data2[i+1]=value;
4     } else {
5         data2[i]=data[i];
6         if (i<data2.length - 1) {
7             data2[i+1]=value;
8         }
9     }
10 }
11 data=data2;
12 }
```

Generalized Error Message: In class CLASS in method METHOD in LINE you tried to access an array index that is not available. In an array with n elements the smallest possible index is 0 and the biggest possible index is n-1.

Underlying Error: Attempt to access the value with the index n in an array with length n. (line 10)

Possible missing competency: The students are able to implement the access elements in an array.

C. Sample: “getter”

Program code:
```java
1 @Override
2 public List<Student> getStudentsAttendingCourse(Course course) {
3     // Aufgab
4     List<Student> studentsTmp = new LinkedList<>();
5     for (Student stu : students)
6         studentsTmp.add(stu);
7     return studentsTmp;
8 }
```

Generalized Error Message: Your implementation of the method METHOD returns students that did not attend 'course'.

Underlying Error: Missing conditional statement that only adds those students which did attend 'course' (line 6)

Possible missing competency: The students are able to implement a conditional statement.

D. Sample: “calculation”

Program code:
```java
1 public Train uncoupleWaggons(int index) {
2     if (index<0||this.getSize()==0){
3         return null;
4     } else{
5         Train newTrain=new Train();
6         Wagon node=this.head;
7         Wagon next=node.getNext();
8         node.setNext(null);
9         newTrain.head=next;
10         return newTrain;
11     }
12 }
```

Generalized Error Message: Your method METHOD behaves in a wrong way when uncoupling cars from the middle of the train.

Underlying Error: Missing loop to find the right position in the linked list (lines 5-10)

Possible missing competency: The students are able to apply a loop on a linked list.

E. Sample: “field”

Program code:
```java
1 public void addValue(int value) {
2     int[] data2=new int[data.length+1];
3     for(int i=0; i<data2.length; i++)
4         {
5             data2[i]=data[i];
6             if (i<data2.length - 1) {
7                 data2[i+1]=value;
8         }
9     }
10 data=data2;
11 }
```

Generalized Error Message: In class CLASS in method METHOD in LINE you tried to access an array index that is not available. In an array with n elements the smallest possible index is 0 and the biggest possible index is n-1.

Underlying Error: Attempt to access the value with the index n in an array with length n. (line 10)

Possible missing competency: The students are able to implement the access elements in an array.

V. Threats to Validity

A problem that occurred during the error analysis was the missing distinction of the JACK system between errors that were caused by a wrong application of some concepts and errors that were caused by not answering at all. This has to be fixed for the automatic analysis that is planned.

Additionally, a serious problem is caused by the Java compiler. On the one hand, the syntax error messages are “clear” with regard to the underlying competence. Nevertheless, on the other hand, an indicated syntax error can be arisen just from a previous line of code that contains an error.

Another problem we have to face is the heterogeneity of some error groups we found. For example the group *calc* includes several different errors. Here, an automatic analysis is difficult to find.
VI. Conclusion and Future Work

The presented results exemplarily show the methodology to derive programming competencies from student errors gathered from an automatic assessment tool. We were able to find several errors in the investigated program code that let us assume missing competencies. In adapting the qualitative content analysis of Mayring on the errors and error messages we formed a system of error groups. With the analysis of the specific underlying error found in the program code we were able to derive the programming competency that is assigned to that error.

In future investigations we have to find a way to automatize the processes. For that reason we have to analyze the commonalities of the error groups in a more detailed way. In particular, for the “unclear” groups a revision has to be undertaken. With this we are able to investigate the huge amount of error messages an automatic assessment system like JACK provides. In the end this results in a list of missing competencies which then can be structured to develop a competence structure model for programming. As an implication of our work, we are able to provide individual feedback based on competencies. Hence, we can improve recommendations for further learning in an online learning environment. A further step in developing a competence model for programming is the elaboration of the potentially present competence dimensions and the validation through an measurement tool for the derived competencies. All in all, the systematic evaluation of student errors in big datasets and the development of a competence structure model enables the precise measurement of students’ abilities in programming.