DuEPublico
Duisburg-Essen Publications online

Computer Aided Assessments and Programming Exercises with JACK

Goedicke, Michael; Striewe, Michael; Balz, Moritz

In: ICB Research Reports - Forschungsberichte des ICB / 2008

This text is provided by DuEPublico, the central repository of the University Duisburg-Essen.

This version of the e-publication may differ from a potential published print or online version.

DOI: https://doi.org/10.17185/duepublico/47108

URN: urn:nbn:de:hbz:464-20180920-073703-9

Link: https://duepublico.uni-duisburg-essen.de/servlets/DocumentServlet?id=47108

License:
As long as not stated otherwise within the content, all rights are reserved by the authors / publishers of the work. Usage only with permission, except applicable rules of german copyright law.

Source: ICB-Research Report No. 28, Dezember 2008

ICB
Institut für Informatik und
Wirtschaftsinformatik

Michael Goedicke
Michael Striewe
Moritz Balz

Computer Aided Assessments and Programming Exercises with JACK

ICB-RESEARCH REPORT 28

UNIVERSITÄT
DUISBURG
ESSEN

ICB-Research Report No. 28

Dezember 2008

Die Forschungsberichte des Instituts für Informatik und Wirtschaftsinformatik dienen der Darstellung vorläufiger Ergebnisse, die i. d. R. noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar.

The ICB Research Reports comprise preliminary results which will usually be revised for subsequent publications. Critical comments would be appreciated by the authors.

Alle Rechte vorbehalten. Insbesondere die der Übersetzung, des Nachdruckes, des Vortrags, der Entnahme von Abbildungen und Tabellen – auch bei nur auszugsweiser Verwertung.

All rights reserved. No part of this report may be reproduced by any means, or translated.

Authors' Address:

Michael Goedicke
Michael Striewe
Moritz Balz

Institut für Informatik und
Wirtschaftsinformatik (ICB)
Universität Duisburg-Essen
Schützenbahn 70
D-45127 Essen

goedicke@s3.uni-due.de
michael.striewe@s3.uni-due.de
moritz.balz@s3.uni-due.de

ICB Research Reports

Edited by:

Prof. Dr. Heimo Adelsberger
Prof. Dr. Peter Chamoni
Prof. Dr. Frank Dorloff
Prof. Dr. Klaus Echtle
Prof. Dr. Stefan Eicker
Prof. Dr. Ulrich Frank
Prof. Dr. Michael Goedicke
Prof. Dr. Tobias Kollmann
Prof. Dr. Bruno Müller-Clostermann
Prof. Dr. Klaus Pohl
Prof. Dr. Erwin P. Rathgeb
Prof. Dr. Albrecht Schmidt
Prof. Dr. Rainer Unland
Prof. Dr. Stephan Zelewski

Contact:

Institut für Informatik und
Wirtschaftsinformatik (ICB)
Universität Duisburg-Essen
Universitätsstr. 9
45141 Essen

Tel.: 0201-183-4041
Fax: 0201-183-4011
Email: icb@uni-duisburg-essen.de

ISSN 1860-2770 (Print)
ISSN 1866-5101 (Online)

Abstract

In this report a system for computer aided assessments and exercises on Java programming is presented and discussed. The report includes a detailed system description, experiences and evaluations from using the system, plans for future development and a brief overview about related work and discussions.

Contents

List of Figures

List of Tables

1 Introduction

Computer based exercises, e-learning and computer aided assessments (CAA) became important topics of research and discussion in recent years. Both increasing numbers of students and steady progress in computer infrastructures made it more and more desirable to offer computer based exercises and examinations. Main goals were increased efficiency, reduced manpower needed for corrections and possibilities to apply various media and modern teaching techniques. Although one might think of automated grading of multiple-choice tests or submitting exercise solutions via e-mail as the first applications of CAA-systems, a program for automated grading of ALGOL programs published in 1965 [FW65] can be considered the oldest CAA-system.

Since then, much progress has been made in making CAA-systems easy to use both for students and teachers, in applying them in different subjects and in analysing their usefulness. This report presents JACK, a web-based system for exercises and examinations on Java programming, which is developed by the research group for Specifications of Software Systems at the University of Duisburg-Essen. Computer aided learning and computer aided assessments are treated as very closely related in this context, because we are convinced that a tool used for grading solutions should be precise enough to explain its decisions in a way that improves the students' learning process and especially increase their ability to actually transfer solution ideas into programming constructs.

1.1 Teaching Java to First-Year Students

Finding an algorithmic solution and an implementation for a given problem is a major challenge for first-year students of computer science. Abstract concepts have to be learned to find mappings from problems to algorithms. Program structures have to be known to derive a corresponding program later on. Teaching both abstract structural concepts and corresponding program structures in one course seems to be an adequate teaching concept. To make sure both theoretical concepts and their practical realizations are well understood, the lectures have to be complemented with numerous accompanying coding exercises [WW05] as well as asking frequently comprehensive questions. Although it is not impossible to learn programming by writing programms on paper, it suggests itself to apply e-learning techniques in this subject at a larger scale and to make use of methods for automated grading of submitted solutions.

Programming exercises from the scenario scetched above are characterized by the fact

that a given problem induces only a small number of principal solutions in most cases. Furthermore, we can expect a set of similar "standard flaws" to occur in the solutions, which can be derived from incorrect usage of the concepts and structures of the related lesson. In our scenario several hundreds of first-year students took part in six exercise sessions, leading to much more than thousand pieces of code having to be corrected. In each case the purpose of the code pieces was well known to the correctors, but nevertheless they had to get an almost full understanding of the actual solutions to give valuable feedback to the students. Even the students who coded a particular solution could not be expected to understand completely what they were doing. It was a major goal of the exercise to raise their capabilities of understanding programs. To reduce the need of manpower for this task we developed a checking system called JACK for partial automation of these corrections by means of static and dynamic tests. Static tests check the program without starting it, e.g. by looking for right syntax and presence or absence of certain program statements. Dynamic tests check the program by running it, e.g. by comparing its output to the output of a sample solution.

We used JACK for the first time in winter term 2006/07 during the programming lecture for first-year students and used a slightly enhanced version one year later. The goal of JACK was not only to test the code in terms of right or wrong, but to give detailed hints on possible flaws and hence to support a better program comprehension both for students and correctors. Static and dynamic checks complement each other to reach this goal, because both can reveal errors not detectable by the other. For example, the dynamic test based on input-output-conformance can reveal that an algorithm with a loop returns the wrong value for a given input, but it cannot determine whether this happens because of a wrong arithmetic statement or a wrong loop counter. The static test based on analysis of the abstract syntax graph can for example reveal whether a counter increment is placed correctly inside this loop or a certain arithmetic statement is present, but it cannot determine whether the output is correct for all given inputs. Both tests together can reveal the error and give valuable feedback to the student.

When talking about automated grading complex of solutions from programming exercises, it is no big step to include automated exercise checking for simple multiple-choice questions into the existing system. Comprehensive questions on programming lessons for example can be formulated very well as multiple-choice questions and both benefits mentioned above apply here, too. The teacher can concentrate on preparing good questions instead of checking hundreds of answers and each possible answer on the multiple-choice forms could be combined with explaining statements, giving feedback to the stu-

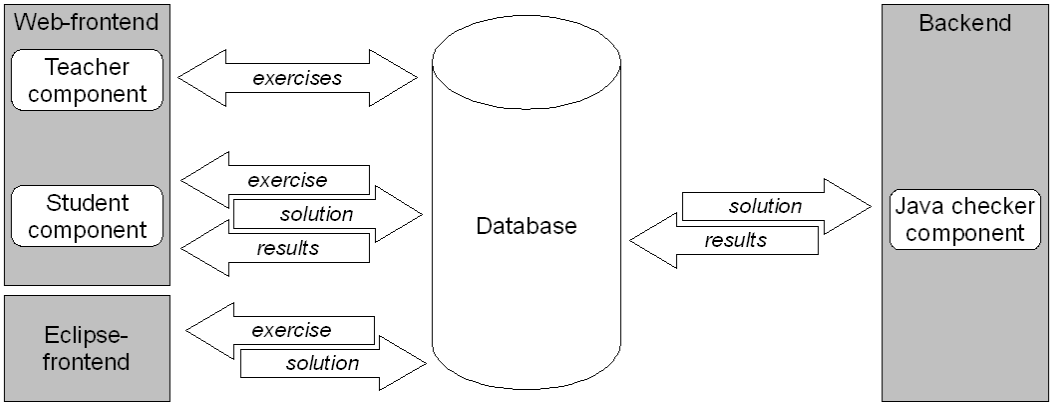

Figure 1.1: Basic architecture of JACK. The backend can be extended by other checker components than the one shown for Java checking. Other rich clients than ECLIPSE can be used at the front end side when they provide appropiate plug-in mechanisms.

dents when revealing right and wrong answers. We added support for multiple-choice questions for the programming exercises in winter term 2007/08.

In the following, we describe the environment of the sketched scenario and the general architecture of the system. Chapter 2 explains the workflow inside the checking component for Java exercises in detail. Both the dynamic and the static checks are explained here. Chapter 3 covers the use of JACK for multiple-choice tests. In chapter 4 we describe how we organized exercises and examinations with JACK and evaluate the system based on our experiences. Our plans for further work are sketched in chapter 6.

1.2 General Concept and System Architecture

In general, the whole assessment system is designed and deployed as packages running on Java Enterprise Edition application servers connected with a separate database system. At the front end side of the system, teachers and students can interact with the system through a web interface. Additionally, students can use a plug-in for the ECLIPSE [Ecl] development environment while attending exams. At the back end side of the system, so-called checker components are running in order to read submitted solutions from the database and mark them. Figure 1.1 gives an overview about this general architecture. The single parts are explained in subsections 1.2.1 to 1.2.5 in more detail.

The system can be deployed as several packages that may run on different servers. The

minimum installation uses only the package containing the core system and the server-side parts of web access front ends used by students and teachers. The web services communicating with the ECLIPSE front end are deployed in a separate package that has to be run on the same server as the first package. Another package contains the backend system and can be run on the same server as well as on a different server. The database server location is also independent and can be connected via network access.

Our JACK system is configured in a way we consider the default design: All packages except the backend system are hosted on the same server as the database. The backend system is separated for security reasons and several security requirements are implemented by using elaborated network and firewall settings. The two servers have been set up as virtual machines, making management and complete system backups easier. Additionally, we were able to provide a copy of these virtual machines to another department of the University of Duisburg-Essen, who were able to use JACK right out of the box this way, with only small local configuration changes for network addresses and server names.

1.2.1 Core System

The core system serves as a broker for all information used in the assessment process, the main tasks being: (1) Management of authentication; (2) import and management of according student data; (3) management of exercise definitions, creation of exams and assignment to examinees; (4) delivery of exams to students; (5) collection of results; (6) delivery of solutions to marking components depending on the type of the tasks and abilities of available components; (7) management of reviews and manual corrections if necessary. Errors occuring during user activities are recorded using logging mechanisms of the application server to make any interaction with the system traceable.

All persistent data is stored in a single relational database to avoid different storage locations like separate files in the file system. By using a relational database we can rely on database transaction mechanisms to prevent critical data loss during examinations. Additionally, it is easy to backup all system data from this single storage location. Each submitted solution to an exercise is stored with time stamp, unique identification number of the submitting account and network address of the computer used for submitting the solution, making them traceable even without using the server logs. The business logic itself uses object-relational mappings to represent data and thus facilitates a structured development approach regarding the data model.

1.2.2 Web Access for Students

The web-based front end for students provides two perspectives on the system. The default perspective is the *self-training mode* that can be accessed by students with a personal account. Once logged in, students can work with available self-training exercises and are for this purpose free to submit multiple solutions without any time restrictions. The personal account also allows a review of existing results of self-training exercises as well as exams. Since this user interface is not appropriate for exam situations, we provide a second, simplified perspective. This *exam mode* uses TANs to identify students and allows to attend only one assigned exam. Students can thus neither review results nor choose to attend different exercises. Nevertheless they are allowed to submit multiple solutions.

The general handling of exercises is similar in both perspectives. Multiple-choice questions as a simple type of exercises can be handled directly inside the web browser, so students can directly tick their answers and submit them to the server. More complex types of exercises are handled by offering files as downloads. Students have to know how to handle them properly, e.g. opening them in an appropriate editor. Similarly, files have to be uploaded again to the server in order to submit a solution. Since all expected files of the solutions are known beforehand, the system guides the user through the upload process by specifying all expected resources. This ensures that a solution can only be submitted if all expected files are present.

The result review allows students to examine their submitted data as well as all output from the marking run time and possibly manual teacher comments. All files attached to a solution are offered as downloads. Solutions cannot be changed in any way in this view.

1.2.3 Rich Client Access for Students

Our solution of a plug-in for the widely used ECLIPSE IDE is an example how arbitrary client software for certain purposes can be integrated in the overall system with lean communication layers. Rich clients can be used to offer extended features to the students that assist them in solving their task and to enable the use of more complex exercises at all. The communication between core system and rich clients is realized with SOAP web services. The according server-side communication layer allows to deploy customized adapter components for different types of clients that can be independently enabled and disabled.

(a) (b)

Figure 1.2: Screenshots from teacher's web frontend, showing management of checking options (a) and review of checking results for a single solution (b).

The client plug-in itself uses the provided ECLIPSE platform, especially the Java Development Tools [JDT], to accomplish programming exercises. A dialog guides the student through the login process by requesting a TAN, downloading files and opening a Java project as well as all resources the student is expected to edit. Additionally, the user interface is simplified by closing all elements except a navigational view and a view for console output. To identify source code files and the according compiled binaries our plug-in relies on unambiguous information provided by the platform and avoids asking the user for additional information.

1.2.4 Web Access for Teachers

In contrast to the student user interfaces that are as simple as possible, the administrative access for teachers must provide comprehensive and flexible tools to create, edit and analyze assessment data. User accounts can be created by importing existing user data. Exams can be assembled by using a repository of single exercises. Exercises have for this purpose been defined independently by specification of a description, attached files and an assignment to a checking component (see fig. 1.2a). Questions and answers in multiple-choice exercises can directly be edited in the web-browser, while source code templates for programming exercises have to be provided as uploaded files. The TAN creation process which joins login and exam data relies on standard techniques for generating random strings and explicit checks for duplicates to produce unique values. Reviewing results (see fig. 1.2b) includes the opportunity to view or download the submitted solution code and to override automated results from the marking components. Detailed statistics for every exercise can be exported as spreadsheets for further process-

ing in external tools.

1.2.5 Backend System

The checking components are executed in a run time system that can operate independently from all other parts of the system, thus forming a master-worker architecture. In this way different checking components may be distributed over multiple physical servers for security or performance reasons and thus perform their work in parallel. Since checking and result submission is subject to security concerns, we set up strict network access rules using firewalls to ensure that access to the core server is only possible from valid backend systems.

The backend systems themselves are able to execute multiple checking components on one physical server by providing only an environment and a network connection to the core system which is dynamically configurable. The core system is contacted regularly to access upcoming checking tasks which are then passed to an appropriate component. The result of the check is submitted to the core system including all error messages and hints like the console output from black-box tests for programming exercises. This architecture is to some degree fault tolerant because the core system is not affected by the checking process. Thus checking components or the related servers may be disabled, disconnected or even crash without any consequences for the exam situation. Hence it would also be possible to design checking components connecting to different core systems, but running on one fixed physical machine, for example because of specialized hardware resources not available on all servers.

When checking programming exercises by executing code submitted by examinees, this code is started in a sandbox environment and not inside the marking run time system. This allows to apply security constraints to the sandbox, for example to prevent file and network access, and to catch easily any kind of runtime exception without affecting the checking component itself.

2 Checking Java Exercises

Checking Java exercises is the core feature of JACK and hence most of the program logic and data structures are concerned with it. Each Java exercise consists of a description, some settings and a set of files. Descriptions are used to explain the exercise to the students and to store internal notes from the teachers. Settings determine whether an

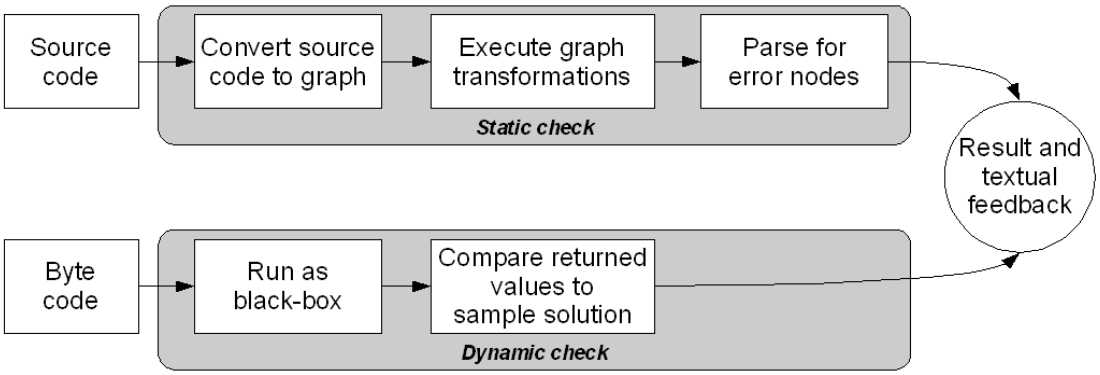

Figure 2.1: Checker workflow for Java exercises.

exercise is visible to the students and checked by static or dynamic checks. Both can be switched on and off separately. Files could be Java source files visible to the students, source files for the dynamic tests or rule files and control scripts for the static test. A special kind of files are placeholders being identified by a file name. They do not provide content, but indicate files that have to be submitted as part of a solution.

Students can access Java exercises either via the web interface or the ECLIPSE plug-in. Both in examination mode and self-training mode, the web interface offers a list of exercises that belong to the examination or are available for training, respectively. Students select an exercise from this list and receive a set of code templates for this exercise which may be copied to a local source code editor. For uploads, the according exercise has to be choosen again from a list and the system asks for all necessary files successively. The ECLIPSE plug-in can be used alternatively in an examination as already described in the introduction.

2.1 Checker Component Workflow

The currently implemented workflow of the checker component is depicted in figure 2.1. It combines several tools, APIs and techniques. A major challenge was the integration of the various data formats in a way which assures a coherent presentation of the source code along the tool chain in order to generate useful hints related to the submitted original source code in the case of an error.

As described in the general architecture, the checker component is running in the back-end. Each solution submitted by a student is inititially marked as “WAITING” in the

database. The backend searches for solutions with this marker and selects the oldest for processing. It is then marked as "PARTLY_PROCESSED". If there are several instances of the backend running, the next one will hence select a different solution for processing. After selecting a solution, the checker component looks for the checking settings of the respective exercise. If no checks are activated for this exercise, the solution is immediately marked as "WAITING_BUT_NOTHING_TO_DO". Whenever the teacher decides to activate checks for this exercise somewhere later, all solutions belonging to this exercise are set back to "WAITING" and will be processed again by the backend. If there are checks activated for the exercise, the according static or dynamic checkers will be invoked. They process the solutions using the tool chain described below and attach an individual result to the solution. One solution can thus have more than one partial result from different checkers which may differ. However, each checker does only provide one partial result for a solution.

After all activated checks are completed, the overall result for the given solution is set. This is either "CORRECT" if all attached results mark the solution as correct or "INCORRECT" if at least one result marks the solution as incorrect. If any kind of internal error occured during the checks, the solution status is set to "INTERNAL_ERROR", indicating that this solution needs manual checking. Independent from this hint, a teacher can allways review the solution manually and attach a manuel result, marking the solution as correct or incorrect. In contrast to automated results, where one incorrect result forces the solution to be marked as incorrect, manual results are allways considered with higher priority. Thus, if a teacher marks a solution as correct manually, it is marked as "CORRECT" in the database, independent from the other results and even if no checks have been performed at all. Analogously, a solution is marked as "INCORRECT" whenever the teacher marks it as failed. The teacher can add textual comments to the result to explaini the decision to the students and point towards errors to increase program understanding.

Depending on the amount of students that are submitting solutions to the server at the same time and on the size and complexity of the solutions, it can take several minutes until the results for a submitted solution are available. Espacially the static analysis can be time-consuming if nested structures have to be analyzed and complex rules take much time when searching for matches. Dynamic checks can take up to 30 seconds in case a solution contains an endless loop and the test execution does not stop until it is aborted by the system.

2.2 Static Analysis

As described in [KG06], the checker component for static analysis is based on a graph transformation engine. Graph transformation techniques as the underlying core of this checker were choosen to provide a stable way for structured representation, verification and modification of the Java source code. We are thereby able to handle the source code structure in a well-defined way as in several other contexts like program refactoring [FBB+00, GS07]. This can be used for static analysis as well as to merge elements and insert additional statements into the code as an initialization before performing dynamic checks.

Graph transformations are realized by applying a series of transformation rules to a host graph [EEPT06]. We use an attributed graph grammar, so every node and edge in the graph has a type and each type has a set of typed attributes. Consequently, the first step of the static analysis is to transform the source code into a graph structure by using a tool named *java2ggx*. It supports the full Java 5 syntax and is designed as a basic toolkit for different kinds of more extensive analyses and transformations. We developed this general-purpose tool to treat Java source code as an abstract syntax graph in the "GGX" file format of the Attributed Graph Grammar (AGG) system [Tae00, AGG]. Support for Java 6 syntax is planned for future relases of JACK, but most concepts taught in programming lectures are already available in Java 5.

When the solution source code is represented as a graph the checker can start to apply rules. Each rule consists of a left hand side (LHS) and a right hand side (RHS) and possibly one or more negative application conditions (NAC). First, the transformation engine tries to find a match for the LHS in the host graph, i.e. a set of nodes and edges that are connected the same way and having the same types and attributes as the LHS of the rule. Rules do not have to specify all attributes, so it is possible to make a match on a node representing a method with an attribute with a certain name, but to neglect the attribute telling whether this method is `public` or `private`. When a match is found, the engine tries to find matches for the NACs if they are specified. Whenever a match for a NAC is found, this rule is discarded and the next one is processed. If no NAC matches or none is specified, the rule will be applied. In this case, all nodes and edges that are present in the LHS are replaced by elements from the RHS if there is a replacement defined for them. If no replacement is defined, they are removed from the host graph. If the RHS contains elements without origins on the LHS, they are inserted into the host graph. Inside JACK this is done by using the AGG API and our own control script handler *RuleControl*. The script handler allows to apply rules in a certain order

and to apply rules only if other rules matched before. Typically, checking rules can be designed in one of the two following ways (see section 2.4 for an example):

- Optimistic rules assume that the solution is correct unless a undesirable structure is present. In this case, the erroneous structure is placed on the LHS of the rule. The RHS contains the same structure, because we don't want to change the code, and an additional "error node", which is a node of a fixed type that does not appear in normal syntax graphs and that contains a message describing the detected error. So whenever the erroneous structure is present in a solution, this rule matches and inserts the error node into the graph. A NAC containg the error node assures that the rule is not applied twice.

- Pessimistic rules assume that the solution is wrong unless a certain structure is present. In this case, the LHS of a rule is empty and correct structures are added as NACs. The RHS contains only an appropiate error node. Because of the empty LHS, this rule can be applied always, except when one of the correct structures is present. Again an additional NAC assures that the rule is not applied twice.

It is possible to apply other transformation rules before the actual checking rules are applied. For example, `i += 1;` is a semantically equivalent, but syntactically different short version of the longer `i = i + 1;`. To avoid two versions of each rule dealing with assignments like this, a general transformation can be applied before, replacing all long statements with an according short version. These changes are only performed internally for checking the solution, but not written back into the database, so the original code submitted by the student remains unchanged. More sophisticated rule sets can use manipulations to add auxiliary nodes or edges with additonal information, for example counters for recursive method calls, that can be used to perform more complex checks. Another good and often used example is to connect all literals of a logical statements with new edges to an additional auxiliary node. This enables easy access by other rules, for example, when checking the correct use of complex termination conditions in a loop. Additionally to NACs, rules can be parameterized by using attribute conditions. For example, a check may be performed for field declaration that are either `private` or `protected`, but not for those that are `public`. Of course this could be done by two versions of the respective rule, each using the appropriate attribute. A more convenient way is to define a rule variable for this attribute and add an attribute condition, forcing this variable to contain either "`private`" or "`protected`". Rule variables can also be used to record attribute values from the LHS when finding the match and to reuse them on the RHS when placing an error node.

After processing all rules the resulting graph is parsed for the inserted error nodes. Their content is stored in the database as a report of the static test and helps to get a better understanding of the source structure and possible mistakes. In detail, the static test can have one of the following four results:

- **Correct**: A result is marked as correct, if the static analysis could be performed completely and no error nodes where produced. In this case, the error record is empty.
- **Failed**: A result is marked as failed, if the static analysis could be performed completely, but at least one error node was produced. In other words, the submitted source code has no syntactical errors, but contains semantically wrong structures or needed structures are missing. The error record contains the contents of all error nodes produced during the analysis.
- **Error**: A result is marked as an error, if the static analysis was terminated by an exception. In contrast to the former result, in this case the tested code has no valid syntax at all. This error can occur because of wrong character encondings, too. The error record contains one message explaining the exception.
- **Internal error**: A result is marked as an internal error, if the checker component itself failed to perform the static analysis or collect the error nodes. This is expected to be a serious error and involves human inspection of the occured problems. In most cases, this will not help to gain any deeper understanding of the solution.

2.3 Dynamic Analysis

The dynamic analysis currently consists of a simple black-box test of a compiled piece of code, using the tests submitted by the teacher. If all tests are passed, the source code is marked as dynamically correct. Otherwise, error messages are generated and stored in the database that explain the faults to make the program behaviour quickly understandable for students and correctors. For example, a sequence of tests may reveal that an algorithm ignores one of its input parameters.

The black-box tests take a byte code version of the exercise solution to be tested and try to execute it with a set of pre-defined inputs. The methods to be called and the inputs to be used are defined by the teacher and stored in the database. Typically it is sufficient to write a single class file containing all the testing code, but additional class files may

be added. The tests are designed in a similar way as unit tests by calling methods on the tested code and comparing the returned value to a specified reference value [Lin03]. If the expected value is not met, the test method may write a detailed error message to the standard error output. The effectiveness of the system for the purpose of program comprehension depends directly on the quality of these error messages. Hence the tests may execute several method calls before generating an error message as well as terminate the test after the first unexcepted output from the tested code. When terminating the test method either after having performed all tests or as a consequence of a failed test, the method returns a value indicating the overall outcome of the test.

To handle possible exceptions that may occur during the test execution, a wrapper class encapsulates all tests for one piece of code. This way general error handling can happen independently from the actual tests. Nevertheless the stack traces and messages of exceptions can be read out and used to create further detailed error messages to improve program comprehension and error analysis. The wrapper class is called as a seperate process on a second Java Virtual Machine either by using the command `Runtime.exec()` or by executing a shell script. This seperate process is necessary for several reasons. First, the process can easily be terminated after a fixed amount of time when it runs e.g. into an endless loop. This can be done automatically and is also important during the mass validation without manual interaction. We used JACK with a time limit of 30 seconds, which turned out to be sufficient to check all long running but terminating solutions for our exercises. Second, the new VM may have a special environment defined by its commandline parameters. This allows to set memory limits independent from the settings of the testing system itself. Third, the new VM can make use of SUN's security concept by involving the `java.security.manager` and `java.security.policy` options [Gon]. This prevents malicious code from tampering with the testing system. Furthermore, it is a comfortable way for checking the absence of certain code structures (e.g. access to input or output streams or network sockets) without running static tests for them.

As already explained, the testing process contains the wrapper class that transforms exceptions into helpful error messages. To get more precise and comfortable messages, the tests are split up into an action phase and a verification phase. Exceptions thrown during the action phase hint at errors in the tested code while exceptions from the verification phase hint at errors in the testing process itself, e.g. a missing test on a returned null value. In addition to a possible exception, the wrapper class takes the return values of the tests and sets the exit state for the testing process. In particular, black-box tests may

have five different results:

- **Correct**: A result is marked as correct if the testing process terminated, no exceptions where thrown and all tests returned the expected value. In this case, the error output is empty.
- **Failed**: A result is marked as failed, if the testing process terminated, no exceptions were thrown but at least one test did not return the expected value. In other words, the tested code works basically, but not correctly. The error output contains at least one message with hints at the reason of the failure. As explained above, a single failure needs not to terminate the whole testing process, so the error output may contain more hints on different failures.
- **Error**: A result is marked as an error if the testing process was terminated by an exception. In contrast to the former result this indicates that the tested code does not work at all. The error output does in this case contain a message explaining the exception.
- **Cancelled**: A result is marked as cancelled if the testing process did not terminate before a given point of time. It is most likely that the code contained an infinite loop in these cases, although it might be possible that the code would have returned the correct result later. As far as an acceptable execution time may be part of a desired behaviour, a cancelled test can be assumed to be failed. If the tested code passed lines of code writing to standard or error output so far, this can help to get a better program understanding regarding the position of the erroneous loop, but more hints can not be given using black box testing methods.
- **Internal error**: A result is marked as an internal error if the checker component itself failed to execute the test or to verify the returned results. Similar as in static checks, this is expected to be a serious error. It involves human inspection of the occured problems and will not help to gain any understanding of the tested code.

2.4 Example

To illustrate static and dynamic checks on Java programs, we present an example from the first attestation session of winter term 2007/08. In this attestation, students were asked to implement a `for` loop for calculating the faculty of a given input, add some constant and test whether the result is a prime number. Both static and dynamic checks

```
public class Testat1abc {
   public boolean berechnung(int zahl) {
      //Loesung eintragen
   }

   public static void main(String[] args) {
      Testat1abc t = new Testat1abc();
      System.out.println(t.berechnung(5));
   }
}
```

Listing 2.1: Code template for a Java exercise. If students change the signature of the method "berechnung", the static check from figure 2.2 will fail. The `main`-method is irrelevant for automated checks.

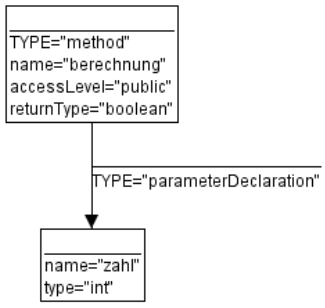

Figure 2.2: A NAC for a static check. This NAC ensures that the checked solutions contains a method named "berechung" that takes an Integer as parameter and returns a boolean value. The code template shown in listing 2.1 provides this structure and students were expected to leave it unchanged.

were activated for this attestation. The code template provided to the students is shown in listing 2.1.

Since students where not allowed to change the basic structure of the template the first rule in all our static checks was concerned with the general program structure as prescribed by this template. These checks can be easily implemented as pessimistic rule, marking each solution as wrong that does not contain the given methods. Figure 2.2 shows the structure placed in the NAC of this rule. Note that static tests like these are important hints for dynamic checks, too, because those checks will fail if an expected method is not present.

After checking the structure, some common transformations are applied to align different semantically equivalent statements, to mark empty block statements and to add auxiliary nodes and edges. These auxiliary nodes and edges are used directly when

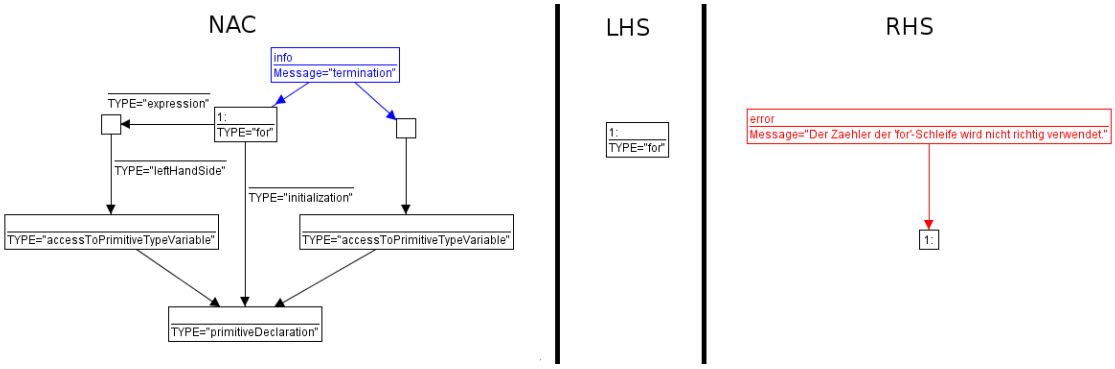

Figure 2.3: A pessimistic rule for checking the correct use of a `for`-loop. The complete rule contains five more NACs that catch variations of the one shown.

checking the structure of the `for` loop. One of the pessimistic rules used here is shown in figure 2.3. Note that his is a pessimistic rule although it has a node on the LHS, assuring that this rule is only applied to source code that contains a `for` loop. Code where this node is missing is handled by a different rule. The check whether the structure of the loop is right or wrong is done via the NACs. We show only one NAC here, but there are five more, which are very similar. In general, they look for a `for` loop that refers to a primitive variable in its init statement (center), its termination condition (right) and its update statement (left). The NACs differ regarding the update statement where the variable may be "leftHandSide" or "leftOperator" in several cases or "rightOperator" of an increment expression. Additionally, the variable may not only be declared in the init statement, but also be declared beforehand and only referenced there.

Besides the presence of a general structure and certain detail elements, the absence of other structures has to be checked, too. In the given example, the students where not allowed to use arrays. Figure 2.4 shows the two rules checking this constraint. Both are optimistic rules, adding an error node only if they find an array declaration. Nevertheless the second rule has a NAC excluding the array named `args`, because this array is part of the `main`-method in Java and hence the only accepted exception. Note that both rules use a rule variable named "type" to record the type from the node on the LHS and insert it into the error message created on the RHS.

Listing 2.2 shows the method implementation of a student who failed the exam. While the first `for` loop is correct, the second is obviously wrong. In fact, there are several errors in there, but one of them is the incorrect use of the counter variable `i`, which is initialized, but used neither in the termination condition nor in the update expression. Consequently, the static check results in an error message indicating the incorrect use of

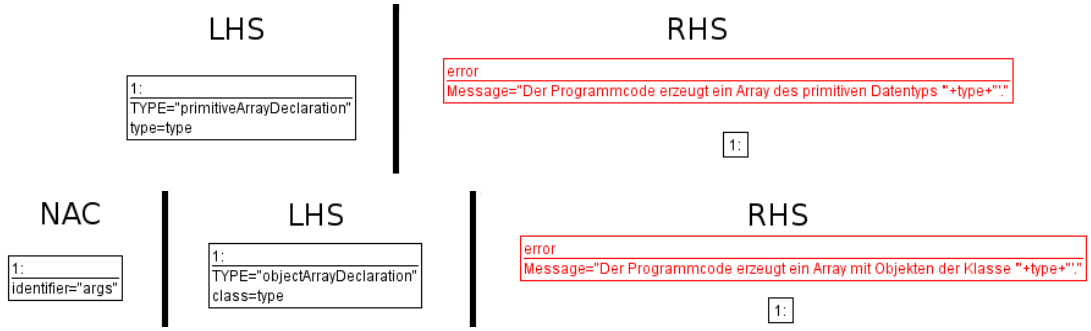

Figure 2.4: Optimistic rules for checking the presence of accesses to arrays.

```
public boolean berechnung(int zahl) {
   int fak = 1;
   for(int i=zahl; i>0 ;i--) {
      fak = fak * i;
   }

   int erg2 = fak + 5;

   boolean prim = true;
   for (int i = erg2; erg2<0; erg2++) {
      if (erg2%i==0) {
         prim = false;
      } else if (erg2%i==1) {
         prim = true;
      }
   }
   return prim;
}
```

Listing 2.2: A student's solution for the presented example. The second `for`-loop is not correct, which was detected during the static checks.

the counter variable and marking the result als incorrect.

Nevertheless, the dynamic check is executed as well, because it could possibly add more useful information on faults in this solution. The dynamic checks in this exam were quite lazy because the method was only called two times with parameters 3 and 5, expecting "true" as a result in the first case and "false" in the second. However, in case of the solution listing discussed above, this was sufficient to produce another error message, because the wrong loop was never executed and hence the solution always returned "true".

3 Multiple-Choice Exercises

The system component for multiple-choice questions was introduced into the JACK system one year after the initial development. It differs much from the components for Java checking and required some new features in the system architecture. The support for multiple-choice questions was intended to be only a secondary feature and can thus only be used in a limited range of cases in the current version. However, it will become a more prominent and better integrated feature in future releases of JACK, as we are going to show in chapter 6.

In general, this component supports two kinds of multiple-choice questions: questions where one of several suggested answers has to be choosen (1-out-of-n) and questions where many of several suggestions may be choosen (m-out-of-n). Similar to Java exercises, where one exercise is the top-level element that may cover several source files, in multiple-choice exercises there are questionnaires as the top-level element, that may contain an arbitrary number of questions. Questions cannot be managed separately, i.e. assigned to different questionnaires at the same time. An arbitrary number of possible answers can be assigned to each question and the teacher must mark each answer as either right or wrong. The system does not require to have at least one right or wrong answer per question, thus it is as flexible as possible. Both the ordering of questions in a questionnaire and the order of answers in a question have to be maintained by the teacher and JACK will not change this order for display.

The ECLIPSE plug-in for JACK is not capable of displaying questionnaires, so they can only be used through the web interface. Students can select them the same way than Java exercises from a list of available questionnaires. The web interface displays a form containing all questions of the selected questionnaire. Students can make their choices by ticking the according checkboxes or radio buttons as usual in web-interfaces. The choice which marks are ticked of the various questions can be changed until the entire questionaire is submitted to the server.

3.1 Questionnaire Checking

In contrast to Java Exercises, the checker for questionnaires is not running on the backend, but integrated into the frontend. Hence each solution is checked immediately after submission and the result is available instantly. The main reason for this is that questionnaire checking is a very easy process and will not slow down the frontend significantly.

At the moment the system does not support complex checking rules for questionnairs, so there are three simple possible results:

- **Correct**: A result is marked as correct if in each question each answer that is marked as right was ticked by the student and none of the answers that are marked as wrong was ticked by the student as well.

- **Failed**: A result is marked as failed, if either at least one answer that is marked as right was not ticked by the student or at least one answer that is marked as wrong was ticked by the student at any of the questions of a questionnaire.

- **Internal error**: A result is marked as an internal error if the checker component failed to verify the submission.

Similar to Java exercises, the teacher may add a manual result with a textual comment to a solution and override the automatic result. When reviewing a result, the student can see his own choices and a notice at each answer indicating whether this answer was expected to be selected or not. The possibility to display an explaination for each question would raise the teaching effect of the questionnaires, but is not implemented yet.

4 Organizing Examinations and Exercises

JACK can be used for self-training by students as well as for attestations or examinations. In general there are no differences in preparing Java exercises for one of these purposes, but attestations require additional steps for creating TANs. Questionnaires were only used for self-training in winter term 2007/08, so we can only report some basic experiences. In any case it is necessary for students to have an account for JACK in order to participate in exercises and examinations. We connected JACK to the login module of our general website, so students could use their personal login from this page for JACK. In another case, JACK used an XML file to identify admininstrators and allowed students to use the self-training mode without password, while the usual TAN-based login was used for attestations.

4.1 Java Self-training Mode

For self-training, Java exercises were set up in the system as described in chapter 2. An exercise sheet with detailed information was available on the website for the lecture, so it was sufficient to place a hyperlink in the exercise description presented to the students. Usually there where two or three different exercises available at the same time. Students where free to access the JACK server whenever they wanted and from anywhere to download the code templates. No restrictions where made on which tools to use for coding, although the use of ECLIPSE was recommended. Not surprisingly, students organized themselfs in internet forums to discuss the exercises and exchange accepted solutions. In fact, this was no bad replacement to an official support desk, because false negatives in error recordes where explained there quite quickly and animated students to submit slight variants of their solutions. It only took some minutes per day, if any, for a teacher to review the submitted solutions and set manual results or explain results to students who did not understand them.

4.2 Java Attestations

In winter term 2006/07 we used nearly the same steps mentioned above for attestations, because the ECLIPSE plug-in and the TAN login were not yet available in that year. The setup differed in just two points: First, the server was only reachable from our computer pools to avoid manipulated uploads from anywhere else. Students could use their personal logins for JACK as known from self-training mode, but got dedicated logins for the computers. Second, there was only one exercise available at the same time, so students could not make any choice.

Students where divided into groups, each with a time slot of one hour. Each student got an exercise sheet upon entering the room that explained the task and gave short instructions on using JACK. The sheet had to be signed and handed back to the supervisors to document that the student was present and got instructions. To avoid the effects of communication between earlier and later groups, we used several slight variants of one task in an attestation and switched them manually between the groups by hiding one exercise and publishing a different one. The computer pools had to be prepared very carefully to ensure that web browser, Java compiler and at least one Java editor where running without problems on each computer. Despite the exercise sheet mentioned before, several students encountered problems in dealing with the different tools and a significant amount of time was consumed by solving those problems instead of working on the ex-

ercise. Some students even lost their personal login and the supervisors had quickly to create new accounts for them during the attestation session.

To avoid these complications, TAN login and ECLIPSE plug-in where invented for winter term 2007/08. We prepared a dedicated version of ECLIPSE just for the attestations, including the plug-in for connecting to the JACK server. This version was distributed in the computer pool and was made accessible for the accounts used for attestations. These accounts could in turn be even more restricted than in the year before to avoid attempts of deception by using other tools or accessing external resources. Problems in dealing with the tools could be minimized this way and students could spend more time on solving their tasks.

The TAN login required additonal steps in preparation of the exercise sheets. A list of registered students was provided to JACK in order to generate an unique TAN for each of them. These TANs where used to print personalized exercise sheets with a conventional mail merge function of a wordprocessor. Both the import of the student list and the export of TANs was done by manual copy-and-paste, which was easy enough to defer the implementation of a proper import and export functions for common office applications. Another benefit of the TAN login was a more flexible assignment of task variations to students, so that tasks could not only be changed between time slots but also be mixed in the groups.

In both years, students were allowed to submit more than one solutions and all of them were checked. To pass the attestation it was sufficient to submit at least on correct solution, which had not necessarily to be the latest one. In average every student submitted two solutions for each attestation. Students were allowed to review all their results a few days after the attestation. They could use their personal login again and got thereby access to all their solutions and the lists of error messages produced by the checkers. Please refer to the next chapter for an evaluation of the expressiveness of these messages.

4.3 Multiple-choice questions

We used multiple-choice questions only for self-training in winter term 2007/08. Several questionnaires where made available during the term in loose succession. Due to the simple structure of the questionnaire module no further preparations or reviews where necessary. Results where reviewed by the teacher mainly to get some statistics or to learn which topics have been well understood by the students and which ones needed further explainations or exercises.

In the current version of JACK, the use of multiple-choice questions in attestations would require the use of the web interface because the ECLIPSE plug-in is not able to display questionnaires, yet. Please note that TAN login is not restricted to the ECLIPSE plug-in, but also available through the web-interface.

5 Evaluation

As mentioned in the introduction, JACK was first used in winter term 2006/07 to assist in the lecture on programming for first-year students. In winter term 2007/08 it was used in the same lecture again. In both years, students had to take part in six attestation sessions per term, each preceded by a self-training session for the same topics. The years before no mandatory attestations were offered to the students because of the huge amount of time this would consume without automated support. In winter term 2001/02 students had to implement a short program and demonstrate it in an oral attestation. About 500 students took part and it took two weeks until all attestations were finished. In winter term 2005/06 optional attestations were offered that were corrected manually.

Looking at these experiences, JACK and its components for Java exercises can be evaluated with respect to organizational, technical and didactical aspects. The organizational evaluation includes the question whether the use of JACK can reduce the needed manpower for preparing, executing and correcting exercises or examinations in the expected way. The technical evaluation asks for the precision of the different checkers and the amount of rules and tests that have to be written to reach this precision. The didactical evaluation examines whether JACK gives valuable feedback to the students that is at least sufficient to understand their errors and explain a failed attestation.

Since the component for multiple-choice questions was only used for self-training exercises in winter term 2007/08, we don't give a separate evaluation for this component here. It can be assumed that it influenced the didactical use of the whole system. Hence it will be discussed in the respective section. For experiences and evaluations regarding the organization of examinations, please refer to the previous chapter.

5.1 Organizational Evaluation

In both winter terms 2006/07 and 2007/08 students had to take part in six Java attestations during the term and to pass at least three to be admitted to the final examination.

session	winter term 2006/07		winter term 2007/08	
	training	attestation	training	attestation
1	-	399	508	626
2	286	496	774	622
3	180	319	484	651
4	-	241	455	463
5	-	176	187	354
6	-	106	199	217
Total	466	1737	2607	2933

Table 5.1: Solutions submitted to Java exercises in winter term 2006/07 and winter term 2007/08.

In winter term 2006/07 the self-training mode was only used as an experimantal feature for two sessions, but one year later each attestation was accompanied by a similar, more extensive exercise in self-training mode. Not surprisingly, the first attestations and exercises had more participants than the later ones, because students who passed already three attestations or failed in four tended to avoid the work for further exercises. In winter term 2006/2007 a total amount of 1737 solutions for Java attestations had to be corrected. The highest number of results produced in one attestation session was 496. In winter term 2007/08 the numbers where even higher with a total amount of 2933 solutions and up to 651 solutions in one attestation session. On the self-training server there were 2607 solutions submitted in winter term 2007/08 and up to 774 solutions for one exercise. All figures based on submitted solutions can be seen in table 5.1.

Preperations of the attestations, i.e. writing dynamic tests and rules for static tests, took at most one day for one teacher for each session. Running both static and dynamic tests on the results was an over-night job for our server. Reviewing all solutions marked as incorrect and random reviewing of correct solutions took never more than one day for one teacher. In summary it took up to two days for one teacher to process a complete attestation session, excluding the time needed to write the exercises themselves, the time spent for attestation supervision as well as time needed for postprocessing of statistics and the announcement of results. In comparison, typical written examination or attestations on paper take about 5 minutes per solution, which would result in 50 hours of work for one teacher to correct 600 solutions. Again time for common pre- and postprocessing is excluded here. Hence it can clearly be seen that the use of JACK resulted in a massiv reduce of manpower and time for the accomplishment of attestations.

In chapter 4 we already referred to the very little amount of time that was needed to

review exercises for the self-training mode. Preperations took less then one day per task, too, because on the one hand the tasks were more complex and hence needed more complex checker rules, but on the other hand no variations had to be produced. So we can summerize here, too, that JACK reduced the needed manpower significantly. Additionally, it has to be taken into account that with server-based self-training there is no need to organize computer pools and avoid collision of exercise sessions with other events, because students can work on the exercises from everywhere and whenever they want.

In summary we can point out that the service for the students was extended to a level which was impossible by using only manual means. Thus the automation has helped clearly to enhance the service for the students. In addition we should mention here that 26 hours of contact time per week were also available where students could get personal explanation and help for their programming problems.

5.2 Technical Evaluation

The technical evaluation has to consider the amount of rules and tests that were needed for checking, the test coverage and precisison of this rules and the quality of error records that were produced by them. The latter can only be done informally based on our experiences and feedback of students based on our teaching evaluations. When students were allowed to review their results, most misunderstandings regarding the correctness of a solution could be cleared quickly and often by carefully reading the generated error messages without additional explainations by the teacher. So in most cases the error messages generated by JACK where sufficient to explain the critical part of the program behaviour.

Nevertheless it turned out that the quality of results was unsteady. Sample figures from attestations in winter term 2007/08 are shown in table 5.2. Due to the checker workflow explained in chapter 2, a checker result may not correspond to the overall result and thus give wrong hints. A result is considered as false positive if it marks a solution as correct while the overall result marks the solution as incorrect because of an related automated or manual result. A result is considered as false negative if it marks a solution as incorrect and is overridden by a manual result marking this solution as correct. Hence, totals of false positives count the intersection of false positives in static and dynamic checks. Totals in false negatives count the union of false negatives in static and dynamic checks. In both cases there might be an estimated number of unreported case, because not all

session	solutions	false positive		false negative	
		static	dynamic	static	dynamic
1	626	99 (16%)	4 (1%)	86 (14%)	0 (0%)
2	622	65 (10%)	3 (0%)	15 (2%)	0 (0%)
3	651	106 (16%)	16 (2%)	38 (6%)	1 (0%)
4	463	92 (20%)	9 (2%)	1 (0%)	0 (0%)
5	354	5 (1%)	4 (1%)	99 (28%)	0 (0%)
6	217	37 (17%)	1 (0%)	2 (1%)	8 (4%)

Table 5.2: Precision of checker results for Java attestations in winter term 2007/08.

results where reviewed manually. We validate, however, all solutions manually which are marked by the checkers as failed.

Up to 28% of the results needed manual correction by the teacher, in most cases because of false negatives. Nevertheless there were very satisfactory sessions where only 2% of the automatic results needed to be overridden by the teacher. The main reason for manual changes were false reports in static tests. Between 12% and 30% of the results from static tests did not correspond to the overall result, either by being too lazy (false positive) or too rigorous (false negative). It seemed nearly impossible to cover all potential correct code structures by means of checking rules even in large rule sets with both optimistic and pessimistic rules. Often the rules pointed to an error that wasn't present. In this case the choosen implementation was valid but not covered by the rules. As a consequence, rule sets were either too rigorous and producing false negatives or rules known as too inprecise were left out by the teacher, resulting in more lazy checks with false positives. As a minor variation of too rigorous checks, an error record could contain misleading information when correct messages were mixed with false reports. At the same time, dynamic checks produced a very low fraction of wrong results, eleminating at least most of the false positive results. Passing the dynamic test became thereby the main criterion for passing an exercise. The output from these tests was correspondingly important for understanding program behaviour and explaining results. False negatives in dynamic tests origin from tasks where less precise results of calculations were accpeted, e.g. by use of the constant `3.14` instead of `Math.PI`. When tolerance margins were not initially included in the tests, the automated results where treated as false negatives and overridden by manual results.

On the one hand we can summarize that dynamic checks are more precise and less misleading. On the other hand it must clearly be stated that there are various errors

that can not be detected by dynamic checks at all, e.g. the correct (required) use of inheritance structures. Additionally, there is a need of interpreting the error record of a dynamic check i.e. to trace it back from an observed wrong output to the wrong statement. In this case the static checks are very helpful since they point directly to a wrong statement. So we can summarize that static and dynamic checks complement each other in the expected way, both in detecting errors and in explaining them.

As an absolute figure, 28% of false reports seem to be not very convincing at the first glance. However several points have to be taken into account to set this figure into perspective. First, in the average case it is both easier and faster to confirm or reject reports made by the system than finding errors without automated guidance. Second, JACK has never any symptoms of fatigue as human correctors would have. If a human corrector has to check large amounts of solutions that look all very similar, it is most likely to have some oversights in small details, that make the difference between a right and a wrong solution. This cannot happen with JACK. Third, if a checking rule is known to be misleading, it can be corrected and the automated checks can be restarted, at least for all solutions that were rated as incorrect. If similar would happen in manual checks, it would be much more expensive to check all solutions again. Finally, even if 28% of all solutions need additional manual supervision, this is nevertheless a reduction of time by more than 70% of the a complete manual processing. An additional aspect is here, if the marking process is farmed out to several human teachers. As experience shows this causes variations in the marking based on individual differences in (subjective) judgement. Since some of this subjectivity cannot be ruled out by using JACK it can be, however, decreased considerably since due to the great saving in assessment time only a small number – in our case one – of persons has to be coordinated / synchronized in the manual supervision task.

According to table 5.2 the rules and tests were best in the second attestation session with only 12% false reports in static tests, only 2% total false negatives and no false positives. Only 15 results out of 622 had to be overridden manually. In contrast to this, the fifth session had an extraordinary high number of false negative in static tests. Table 5.3 compares and inspects the rules and tests used in this sessions in more detail.

Already a few rules and tests seem to be sufficient to get a good coverage of possible flaws. As can be expected in case of static checks, an increased number of tests leads to a higher amount of detected programming flaws. The same is not valid for dynamic tests, where less flaws are detected although the total number of incorrect solutions increased from session 2 to session 5. Hence, beside testing with different input values, one

	session 2	session 5
Total number of solutions	622	354
Currect solutions	470	138
Incorrect solutions	152	216
Rules used in static checks	4	9
Total number of flaws detected in static checks	93	739
Total number false negatives in static checks	15	196
Solutions marked as failed in static checks	102	310
Tests used in dynamic checks	3	5
Total number of flaws detected in dynamic checks	217	114
Total number false negatives in dynamic checks	0	0
Solutions marked as failed in dynamic checks	149	212

Table 5.3: Details for checker results of Java attestations in winter term 2007/08. Solutions marked as failed may be higher than the number of flaws detected by rules or tests, because encountered syntax errors, endless loops and similar are counted seperately.

key benefit of dynamic checks is the fact that submitted solutions are actually executed leading to the discovery of exceptions, potential endless loops or similar errors.

For static checks there seems to be a trade-off between few rules, resulting in few false negatives but many overseen flaws, and many rules, resulting in many false negatives. It has to be kept into account that fewer rules imply smaller error records giving less feedback to the students. As already mentioned in the organizational evaluation, it turned out to be acceptable to manually check all solutions marked as failed in order to eliminate false negatives. Still, more precise rules will help to reduce the number of false negatives from the start.

5.3 Didactical Evaluation

All technical benefits and achievements of JACK and especially its Java checkers would be worthless if many exercises with automated results would not lead to better learning success than less exercises with personal feedback from human tutors. The effect can be considered by analyzing the results from the final examinations over several years. The relevant data is summarized in table 5.4.

Both the absolute number of students who passed the final examination as well as the rate of successful participants was higher in years where JACK was used than in the years before. This cannot prove that many oral attestations or exercises would not have lead to even better results, but when those are not possible due to the great number of

term	total participants	successful participants	success rate
2002/03	262	89	33.97%
2003/04	311	139	44.69%
2004/05	188	99	52.66%
2005/06	355	100	28.17%
2006/07	210	167	79.52%
2007/08	228	158	69.30%

Table 5.4: Success rates in final examinations from 2002/03 to 2007/08. Attestations with JACK where used in 2006/07 and 2007/08.

students, automated exercises can substitute them with measurable success. So it can be stated that automated exercises and attestations lead to better learning success.

Still it has to be considered critically how students used JACK in self-training mode. In attestations they had no possibility to get feedback from the server in between and had to check their solutions on their own as in the final examination with pen and paper. In self-training, they could submit solutions, wait for results and use the error messages to incrementally correct their solution. Although it is the obvious intention of an exercise to point out errors and missunderstandings and allow students to correct them, it was not the intention to save the students from the burden of testing their solutions carefully. Sometimes it could be observed that single students submitted solutions in rapid succession in order to explore the checks made by JACK and provide a solution that satisfies these checks afterwards. When dynamic checks defined by the teacher where not covering the complete desired behavior, this leads to solutions that where assumed to be correct although they still contained errors. Fortunately, this combatative programming against JACK got reduced quickly when checking a solution on the server took more than a few seconds and students realized that their own testing would be faster in the end. Of course, a similar effect is not unlikely in manual attestations, when a teacher starts to oversee errors.

Another important point of discussion is the question whether students should be allowed to submit more then one solution in attestations. Similar to exploring the checks in self-training mode, this could theoretically inspire immature students to make random changes to their solutions and to hope that one of them will be a success. This behaviour could only be observed in very few cases in winter term 2007/08. In nearly all of these cases students had marked their variations by code comments in order to explain what they were trying to do. Attempts like these are also well known from written examinations on paper, so this phenomenon is neither introduced nor avoided by the use of a

technical system. Nevertheless, it could be avoided to some degree by only accepting one specific submitted solution, e.g. the latest, if this turns out to be a real problem.

5.4 Feedback from Students

In winter term 2007/08 questions about JACK were included in the regular teaching evaluation for this term. Although the feedback in this survey counted only 37 participants and hence the sample was quite small, a positive response and general acceptance of the system among the students can be seen clearly.

Students were asked whether they considered the answers from JACK to be helpful or not. On a spectrum from 0 ("very helpful") to 4 ("never helpful"), the average result was 1,68 and less than a fourth of the students voted for 3 or 4. Additionally, students were asked how often they used JACK for self-training. On a spectrum from 0 ("often") to 4 ("never"), the average result was 0,89, indicating that most students accepted JACK as a good opportunity for self-training. Finally, students were asked whether they would like to see more opportunities for e-learning in the lecture. 89% of the students gave an positive answer in this question.

As part of the questionnaire free comments were possible. Several of these comments named JACK as a positive feature of the lecture and expressed satisfaction with this kind of e-learning and testation. Some students asked explicitly for more questionnaires in JACK. Additionally, some students passed constructive criticism on JACK that will influence the further development. In winter term 2006/07 no special questions about JACK were asked in the regular evaluation, but textual comments from students support the findings above. Consequently, several ideas for enhancements of JACK were already realized for the following year.

6 Future Work

Because of the good experiences we had with JACK and the increasing interest in automated and computer-based exercises and examinations, the system will be extended in several ways. The existing types of exercises will be improved by refactoring the underlying architecture to add more checkers to Java exercises and to enable more complex multiple-choice questions. Additionally, the system should be made available for other types of exercises in the long run, e.g. design exercises using UML diagrams or mathe-

matical exercises for lectures on formal methods.

6.1 Advanced Java Checking

Black-box tests turned out to be not sufficient in all cases, even if they are done fully automatically as described above. The main reason is that black-box tests are not intended to learn and tell anything about the internal behavior of the checked piece of code. For example, the given task could be to compute whether a given integer is a prime number. A solution contains at least a loop for testing several factors, a modulo division and a conditional return statement referring to the result of the division. If the black-box test for a solution fails, we just know that we have a false positive or false negative result. We do not know whether the loop failed to test all necessary factors or the division was used with the wrong divisor or the conditional statement was wrong. The use of static tests is limited in this case, too, because they would need to check for at least three types of loops (`for`, `while` and `do-while`), which might be counting upwards or downwards, using a `break` statement, a `return` statement or a boolean flag to terminate and so on. It would be at least hard, if not impossible, to cover all possible solutions by static tests. As a consequence, some additional ideas will be integrated into JACK in further releases.

6.1.1 Model Checking

Runtime model checking can be of great benefit because it will not only return a single result, but a complete trace of program steps if a checked program fails. This can add valuable information for the program comprehension task. To perform runtime model checking, the teacher has to design a model for the desired solution and derive assertions and invariants from it. If a student's solution conforms to this model, it will be possible to find locations inside the Java code where these assertions can be checked and they have to hold for every valid input. This can be checked automatically by a runtime model checker like JAVA PATHFINDER [Pat].

Consequently, runtime model checking needs some more preparation. The teacher has to define the assertions similar to rules for static checks in the teacher component of the system. When a solution is processed they have to be inserted automatically into the Java code. For this purpose, the graph representation created for static checks is used again. Graph transformation rules allow, for example, to replace any write operation to a variable by a sequence of the original write call followed by an assertion statement for the variable value or to place assignments for invariants at the end of each loop. In

addition, any input operation can be replaced with calls to the API of the model checker PATHFINDER including the input range that shall be checked. For the prime number example depicted above, the input range could be the same as used in the black-box tests and an assertion would be placed before the return statement to ensure the the returned value is true for all prime numbers and false otherwise. After applying these modifications the graph structure is re-transformed into source code by *ggx2java*, the reverse tool to *java2ggx*. Then the new extended source code is compiled and executed in a separate thread similar to the black-box tests described above. Again a wrapper class can be used to encapsulate program execution and transform possible exceptions into useful hints for program comprehension. As far as the runtime model checker test is executed after the black-box test, there is no need to catch all possible exceptions. More precisely, only exceptions thrown by the model checker have to be handled because only solutions that were marked as "correct" or "failed" are handed over to this test. In the first case, the additional test may reveal errors that were not covered by the tests written by the teacher. In the second case the test may add more detailed information to the already known failure reasons. The three other results mentioned above indicate serious problems that have to be fixed, before runtime model checking could be performed. Otherwise no helpful results could be expected. The overall result of a runtime model checker test is one of the following options:

- **Correct**: A result is marked as correct if the runtime model checking process terminated, no exceptions where thrown and all assertions and constraints held.
- **Failed**: A result is marked as failed if the runtime model checking process terminated, no exceptions where thrown but at least one assertion or constraint did not hold. The error output contains at least one trace of a failed execution. Comparing this trace to a default trace from correct execution can generate hints to the reason of the failure. Depending on the content of the trace, it is also possible to trigger more specialized static tests for deeper analyses of certain parts of code.
- **Internal error**: A result is marked as an internal error if the runtime model checking process was terminated by an exception. As far as this exception may be expected to be raised by misplaced assertions this involves manual inspection of the occured problems. It will not add any useful information for program comprehension.

The greatest difficulty in using a runtime model checker for mass validation of different solutions is to find appropriate locations inside the source code where assertions should be added. As described above, every writing operation on a variable could be replaced,

but this general approach may not be sufficient in any case. Too many assertions would slow down the checking process significantly and in some cases useful constraints could not be generated or guessed automatically. Careful static checks are needed to distinguish e.g. between variables relevant for the known main purpose of the program and auxiliary variables that were invented by the programmer. Additionally, the resulting traces are related to the manipulated source code and not to the original student's solution. Hence another postprocessing step is necessary to rearrange the traces, e.g. by changing line numbers and omitting statements that were introduced by the manipulation. Since these statements are only used for model checking this would not change the meaning of the trace.

6.1.2 Online Test Generation

In some situations it can happen that errors in submitted solutions can only be revealed by a certain sequence of method calls in the dynamic checks. Generating extensive test cases that are combining method calls in every relevant order is possible, but costly. Runtime model checking can reduce the time needed for preparations but not the time needed for checking all sequences. Another alternative would be to apply online test generation. To realize this, both a set of method calls for testing and a valid sample solution have to be provided by the teacher. A dynamic checker for this kind of testing can then execute these method calls both on a student's solution and the sample solution and compare the results. This can be done either in random order or with respect to some coverage criteria. The test will stop if outputs from student solution and sample solution differ or if it has executed a pre-defined number of calls or all coverage criteria are met. Possible results are mainly the same as in conventional dynamic checks, but the "internal error" result has to include the option that the method call to the sample solution has failed with an exception.

Besides the good trade-off between effort for preperation and test accuracy, other benefits can be expected from this kind of testing. The time needed for testing will be shorter in comparison to tests generated offline, because tests can react on the behavior of particular solutions and thus omit method calls that are without relevance for the testing result, e.g. because of an error detected previously or necessary logical conclusions from other results. Additionally, students can less easily prepare solutions that match exactly the dynamic checks performed by JACK because it becomes much harder to explore them. In fact, students would have to submit complete solutions necessarily, although dynamic checks will only test them by using a well selected subset of possible inputs.

Nevertheless these checks have some limitations when used in examinations if they are not prepared carefully. When using a random order for method calls or weak coverage criteria, a solution may pass one check and fail in another one. Similar, if two different solutions contain the same error, one may pass the check and the other one may fail. To avoid judicial complications arising from this, it might be a better choice to use this kind of testing for generating additional output for the error records thus supporting an easier understanding by the student but not for deciding whether a solution is correct or incorrect.

6.1.3 Dynamic White-Box and Graphical Feedback

Teaching algorithms for the manipulation of data structures is often easier when the main concepts are explained graphically. Even without an explicit introduction into visual modelling, students often understand concepts like linked lists or trees very fast when presented and explained in an appropriate graphical notation. It is thus not far-fetched to include graphical representations of data structures into the error records of dynamic checks.

Primarily this can add a lot of valuable information towards easier program understanding. Secondary this can serve as a basis for finding additional detailed white-box checks as well. Dynamic black-box checks through method calls can only analyse the returned values or objects and their accessible public fields. When displaying a graphical representation of a data structure, it might be necessary to connect to the debug interface of the Java Virtual Machine [Sun] and examine all objects on the Java heap. This allows to check objects not visible in the black-box view and compare the produced data structure with the expected one.

6.2 Advanced Multiple-Choice Questions

Multiple-choice questionnaires can be used for much more than simple questions with a set of possible answers. For example, puzzles where sentences have to be completed by inserting the right words out of a set of suggestions can be seen as specialized forms of multiple-choice. The same applies to questions where answers are not only wrong or right but where a combination of selected answers matters. Questions like this can be made possible by both extending the user interface for displaying more complex question layouts and input fields as well as extending the checkers by implementing answer validation mechanisms based on logical expressions.

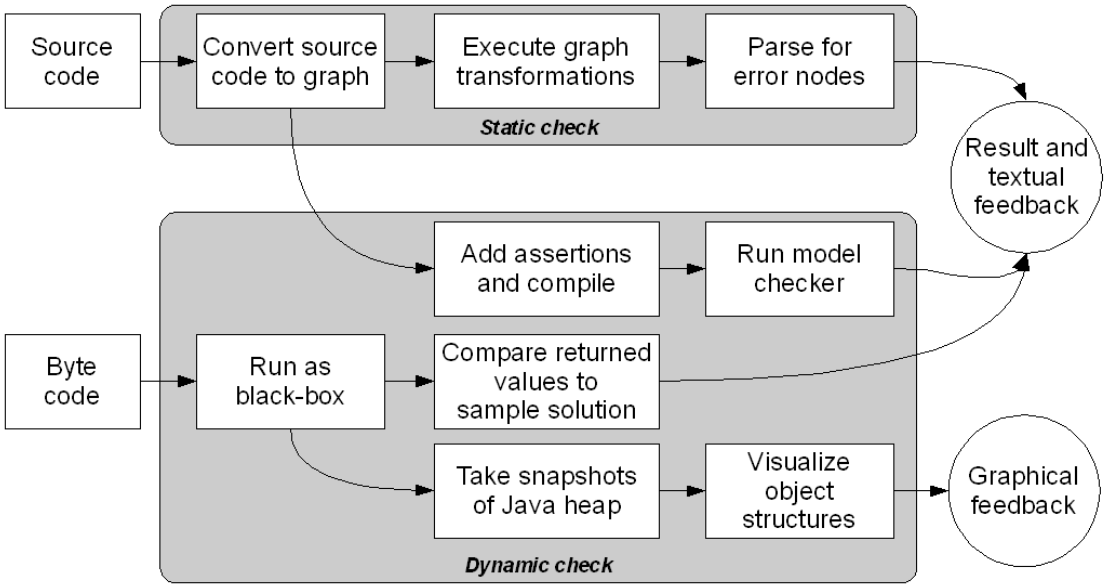

Figure 6.1: Extended checker workflow for Java exercises, including the currently implemented features (see figure 2.1) as well as model checking and graphical feedback.

Since the use of multiple-choice questions is possible in virtually any subject, this will extend JACK to a multi-use tool for automated exercises and examinations. Existing guidelines for high quality multiple-choice exams could be adopted so that JACK will give more assistance in creating exams. Additional functionality for presenting possible answers in random order, presenting questions in random order or selecting a fixed number of questions out of a larger pool are other ideas to extend the multiple-choice module of JACK.

6.3 New Types of Exercises

The concepts used for static checks on Java code can be used for other domains as well since the underlying graph transformation techniques are not only applicable on syntax graphs but any graph-based representation. So it is possible to check exercises for graph-based modelling tasks quite directly. Additionally, any textual exercises from a domain with well defined syntax can be handled similar to Java exercises by parsing the input into a syntax tree and applying checking rules. Similar to multiple-choice questions, these ideas are not limited to the field of computer science and programming languages.

6.4 Tracing Student's Activities

For several reasons it can be interesting to trace student's activities while they are trying to solve an exercise. The first argument is security, to avoid lost data when the used client crashes during an exam and before data has been submitted to the server. If there is a trace of a student's activities available on the server that covers at least the major part of the work done so far, students can resume their exam with only minor disturbance. A second argument are judicial reasons. As in any exam, students may start legal actions against an exam result and claim that they did a certain action or that some other action was not available. If precise tracing data is available in a scenario like this, a student's actions can be reconstructed and possible unrightful claims can be revealed. Obviously, tracing data has to be much more complex in this scenario in comparison to security against data loss. In addition, traces from student's activities can be used as a study object for further investigations about the impact of computer aided assessment environments on the behavior and problem solving strategies of students.

The current implementation of JACK offers only minimal tracing capabilities. When using web access, tracing of student activities is very limited since web browsers submit data only if requested by the user. In the future, we will also explore interactive web sites with JavaScript to fulfill such requirements, but do not expect the related tracking and tracing to be reliably recorded since web browsers allow end users to disable such interaction.

If it is desirable to log mouse movements, clicks or cancelled submission attempts it is hence necessary to use a rich client platform providing the according capabilities. Some of these features can be built on top of the history mechanisms of ECLIPSE, which will be the most obvious starting point. In fact, this is a strong reason to develop rich client plug-ins for any kind of exercises, even for those that are in general simple enough to be handled in the web based front end.

7 Conclusions

In this report we presented a system for automated checking of exercises and examinations that is able to check both programming exercises in Java and multiple-choice questions. We explained the server-based system architecture and the techniques used for checking the solutions. We evaluated the system based on our experiences from using it for two years. The results were positive and motivating to invest more work into

the further development of JACK. Several ideas were presented in this report as well for such a further development towards a versatile tool for automated examinations.

7.1 Bibliographic Remarks and Related Work

Two main topics are related to the content of this report. On the one hand one has to question the use computer aided assessments independent from the limited domain of computer science. On the other hand one must consider techniques being useful especially in this subject or a certain other subject.

Most similar to our system is the web-based tool PRAKTOMAT [KSZ02, Pra] offering almost the same features, including the use of plug-ins for ECLIPSE. It is capable of checking code in Java, C++ and Haskell with static checks and dynamic checks from an external test driver. The system is available as an open source project and under continuous development since 2001, but written in Python and thus limited to UNIX server environments. Thus it is less platform independent than JACK and integration of external tools is more complicated. For example, a direct integration of AGG via its API in order to make static checks more flexible would be impossible.

Another almost similar project is DUESIE [HQW08] which is capable of checking Java and SML exercises as well as UML diagramms. The user interface is limited to a web frontend without plug-ins for ECLIPSE. The core system is realized in PHP5, using external compilers, interpreters and build tools. The technological gap between the scripting language PHP5 and external tools written in other languages is obvious and might cause problems regarding interoperability, security and comfort that are solved in a much more convenient way in the JACK architecture.

A general classification of exercises in the subject of technical computer science is presented by [HvdH05], together with a Java applet that is capable of checking multiple-choice answers, single values, formulas, certain types of diagramms and small assembler programs. Different functional programming languages can be checked with the LLSCHECKER [RAP05] by comparing results from students solutions with results from a sample solution for given inputs. The tool EXORCISER [Tsc04] can be used for self-training without a server and offers exercises and automated verification for various topics in theoretic computer schience, i.e. languages, grammars and markov algorithms. First introduced in 1993, the system TRAKLA and its successor TRAKLA2 [KMS03] is one of the oldest systems for automated grading of exercises for algorithms and data structures. The web-based system SKA [SGB06] can be used for exercises on proposi-

tional and first-order logic. Exercises in mathematics and logic can be considered as very well structured and hence very convenient for automatic checking with formal methods, as also described in [MR06]. But also in weakly structured domains like law, intelligent systems for detecting weaknesses in answers from students are used [PAAL06]. Nevertheless, all of these systems are limited to their special subject and are either not designed to be used both for self-training and assessment, or not prepared to be extendable for virtually any kind of exercise.

The general application of automated and computer-based examinations using the commercial tool LPLUS [LPL] is described in detail by [Ree06]. This system mainly supports multiple-choice questions and questions with single-valued answers which are easy to check. The same applies to another system presented in [GSS+05] where espcially the feedback from students regarding the use of computers in examinations is discussed. Automated testing and assessment without limitation to certain subjects is also offered by the web-based learning management system ILIAS [ILI] having its own conference series with publications on various topics related to e-learning and automated examinations. These tools might be used in virtually any subject of an academic course and be ready for use in self-training and examination, but are strongly limited in the number of different types of exercises they can handle.

Automated static checks on programming exercises using the abstract syntax tree are no new concept, but already described in [TRB04]. Combining static and dynamic checks is a quite common technique in software testing. An example is the check of security properties based on OCL constraints for the architecture against event traces from runtime without the need of code instrumentation [AT06].

7.2 Acknowledgements

The authors would like to thank all students who used JACK and gave feedback that could be used to enhance the system. Dominik Tappe and Holger Schmitt from the University of Duisburg-Essen provided several comments from the teacher's point of view regarding usage of JACK and organization of exercises. Stefan Dissmann and Daniel Maliga from the Technical University of Dortmund provided interesting information about manual attestations at a larger scale. In addition, the authors would like to thank the members of the statewide working group on online examinations "Arbeitsgemeinschaft Online-Klausuren NRW" for many interesting and inspiring discussions about various ways of organizing computer aided assessments.

8 References

[AGG] AGG website. `http://tfs.cs.tu-berlin.de/agg/`.

[AT06] Azin Ashkan and Ladan Tahvildari. A hybrid analysis framework to evaluate runtime behavior of oo systems. In Andy Zaidman, Abdelwahab Hamou-Lhadj, and Orla Greevy, editors, *Proceedings of the 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06)*, pages 1–5, 2006.

[Ecl] ECLIPSE website. `http://www.eclipse.org/`.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformations*. Springer, 2006.

[FBB+00] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[FW65] George E. Forsythe and Niklaus Wirth. Automatic grading programs. *Communications of the ACM*, 8(5):275–278, May 1965.

[Gon] Li Gong. Java™2 platform security architecture. `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html`.

[GS07] Michael Goedicke and Michael Striewe. Dependency analysis and manipulation using abstract syntax graphs. In *Joint Astrenet/Sosornet Workshop on Source Code Analsysis and Software Services*, King's College London, October 2007.

[GSS+05] Ulrich Glowalla, Stefan Schneider, Maria Siegert, Martin Gotthardt, and Jan Koolman. Einsatz wissensdiagnostischer Module in elektronischen Prüfungen. In Haake et al. [HLT05], pages 283–294.

[HLT05] Jörg M. Haake, Ulrike Lucke, and Djamshid Tavangarian, editors. *DeLFI 2005: 3. Deutsche e-Learning Fachtagung Informatik, der Gesellschaft für Informatik e.V. (GI) 13.-16. September 2005 in Rostock*, volume 66 of *LNI*. GI, 2005.

[HQW08] Andreas Hoffmann, Alexander Quast, and Roland Wismüller. Online-Übungssystem für die Programmierausbildung zur Einführung in die Informatik. In Seehusen et al. [SLF08], pages 173–184.

[HvdH05] Norman Hendrich and Klaus von der Heide. Automatische Überprüfung von Übungsaufgaben. In Haake et al. [HLT05], pages 295–305.

[ILI] ILIAS website. http://www.ilias.de/.

[JDT] Eclipse java development tools. http://www.eclipse.org/jdt/.

[KG06] Carsten Köllmann and Michael Goedicke. Automation of java code analysis for programming exercises. In *Proceedings of the Third International Workshop on Graph Based Tools*, volume 1 of *Electronic Communications of the EASST*, 2006.

[KMS03] Ari Korhonen, Lauri Malmi, and Panu Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of Kolin Kolistelut / Koli Calling – Third Annual Baltic Conference on Computer Science Education.*, pages 48–56, Joensuu, Finland, 2003.

[KSZ02] Jens Krinke, Maximilian Störzer, and Andreas Zeller. Web-basierte Programmierpraktika mit Praktomat. In *Workshop Neue Medien in der Informatik-Lehre*, pages 48–56, Dortmund, Germany, 2002.

[Lin03] Johannes Link. *Unit Testing in Java*. Morgan Kaufmann, 2003.

[LPL] LPLUS website. http://www.lplus.de/.

[MR06] Ingrid Mengersen and Peter Riegler. Computergestützte Mathematiktests im Informatikstudium. In Mühlhäuser et al. [MRS06], pages 63–74.

[MRS06] Max Mühlhäuser, Guido Rößling, and Ralf Steinmetz, editors. *DeLFI 2006, 4. e-Learning Fachtagung Informatik, 11.-14. September 2006, Darmstadt, Germany*, volume 87 of *LNI*. GI, 2006.

[PAAL06] Niels Pinkwart, Vincent Aleven, Kevin D. Ashley, and Collin Lynch. Schwachstellenermittlung und Rückmeldungsprinzipen in einem intelligenten Tutorensystem für juristische Argumentation. In Mühlhäuser et al. [MRS06], pages 75–86.

[Pat] JAVA PATHFINDER website. http://javapathfinder.sourceforge.net/.

[Pra] PRAKTOMAT website. http://www.fim.uni-passau.de/de/fim/fakultaet/lehrstuehle/softwaresysteme/forschung/praktomat.html.

[RAP05] Dietmar Rösner, Mario Amelung, and Michael Piotrowski. LlsChecker, ein CAA-System für die Lehre im Bereich Programmiersprachen. In Haake et al. [HLT05], pages 307–318.

[Ree06] Jan-Armin Reepmeyer. LPLUS-Integration – Entwicklung eines Rahmens für den Einsatz eines computergestützten Prüfungssystems. Technical report, Universität Münster, Münster, 2006.

[SGB06] Immo Schulz-Gerlach and Christoph Beierle. Ein erweiterbares interaktives Online-Übungssystem mit Aufgaben zu Aussagen- und Prädikatenlogik. In Mühlhäuser et al. [MRS06], pages 243–254.

[SLF08] Silke Seehusen, Ulrike Lucke, and Stefan Fischer, editors. *DeLFI 2008, 6. e-Learning Fachtagung Informatik, 7.-11. September 2008, Lübeck, Germany*, volume 132 of *LNI*. GI, 2008.

[Sun] Sun Microsystems, Inc. Java™Platform Debugging Architecture API. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[Tae00] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In M. Nagel, A. Schürr, and M. Münch, editors, *Application of Graph Transformation with Industrial Relevance: International Workshop, AGTIVE'99*, volume 1779 of *Lecture Notes on Computer Science*, pages 481–488, Kerkrade, The Netherlands, 2000. Springer.

[TRB04] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' java programs. In Raymond Lister and Alison L. Young, editors, *Sixth Australasian Computing Education Conference (ACE2004)*, pages 317–325, Dunedin, New Zealand, 2004.

[Tsc04] Vincent Tscherter. *Exorciser: Automatic Generation and Interactive Grading of Structured Excercises in the Theory of Computation.* PhD thesis, Swiss Federal Institute of Technology Zurich, Switzerland, 2004. Dissertation Nr. 15654.

[WW05] Nicole Weicker and Karsten Weicker. Didaktische Anmerkungen zur Unterstützung der Programmierlehre durch E-Learning. In Haake et al. [HLT05], pages 435–446.

Previously published ICB - Research Reports

2008

No 27 (December 2008)
Schauer, Carola: "Größe und Ausrichtung der Disziplin Wirtschaftsinformatik an Universitäten im deutschsprachigen Raum - Aktueller Status und Entwicklung seit 1992"

No 26 (September 2008)
Milen, Tilev; Bruno Müller-Clostermann: " CapSys: A Tool for Macroscopic Capacity Planning"

No 25 (August 2008)
Eicker, Stefan; Spies, Thorsten; Tschersich, Markus: "Einsatz von Multi-Touch beim Softwaredesign am Beispiel der CRC Card-Methode"

No 24 (August 2008)
Frank, Ulrich: "The MEMO Meta Modelling Language (MML) and Language Architecture – Revised Version"

No 23 (January 2008)
Sprenger, Jonas; Jung, Jürgen: "Enterprise Modelling in the Context of Manufacturing – Outline of an Approach Supporting Production Planning"

No 22 (January 2008)
Heymans, Patrick; Kang, Kyo-Chul; Metzger, Andreas, Pohl, Klaus (Eds.): "Second International Workshop on Variability Modelling of Software-intensive Systems"

2007

No 21 (September 2007)
Eicker, Stefan; Annett Nagel; Peter M. Schuler: "Flexibilität im Geschäftsprozess-management-Kreislauf"

No 20 (August 2007)
Blau, Holger; Eicker, Stefan; Spies, Thorsten: "Reifegradüberwachung von Software"

No 19 (June 2007)
Schauer, Carola: "Relevance and Success of IS Teaching and Research: An Analysis of the ‚Relevance Debate'

No 18 (May 2007)
Schauer, Carola: "Rekonstruktion der historischen Entwicklung der Wirtschaftsinformatik: Schritte der Institutionalisierung, Diskussion zum Status, Rahmenempfehlungen für die Lehre"

No 17 (May 2007)
Schauer, Carola; Schmeing, Tobias: "Development of IS Teaching in North-America: An Analysis of Model Curricula"

No 16 (May 2007)
Müller-Clostermann, Bruno; Tilev, Milen: "Using G/G/m-Models for Multi-Server and Mainframe Capacity Planning"

No 15 (April 2007)
Heise, David; Schauer, Carola; Strecker, Stefan: "Informationsquellen für IT-Professionals – Analyse und Bewertung der Fachpresse aus Sicht der Wirtschaftsinformatik"

No 14 (March 2007)
Eicker, Stefan; Hegmanns, Christian; Malich, Stefan: "Auswahl von Bewertungsmethoden für Softwarearchitekturen"

No 13 (February 2007)
Eicker, Stefan; Spies, Thorsten; Kahl, Christian: "Softwarevisualisierung im Kontext serviceorientierter Architekturen"

No 12 (February 2007)
Brenner, Freimut: "Cumulative Measures of Absorbing Joint Markov Chains and an Application to Markovian Process Algebras"

No 11 (February 2007)
Kirchner, Lutz: "Entwurf einer Modellierungssprache zur Unterstützung der Aufgaben des IT-Managements – Grundlagen, Anforderungen und Metamodell"

No 10 (February 2007)
Schauer, Carola; Strecker, Stefan: "Vergleichende Literaturstudie aktueller einführender Lehrbücher der Wirtschaftsinformatik: Bezugsrahmen und Auswertung"

No 9 (February 2007)
Strecker, Stefan; Kuckertz, Andreas; Pawlowski, Jan M.: "Überlegungen zur Qualifizierung des wissenschaftlichen Nachwuchses: Ein Diskussionsbeitrag zur (kumulativen) Habilitation"

No 8 (February 2007)
Frank, Ulrich; Strecker, Stefan; Koch, Stefan: "Open Model - Ein Vorschlag für ein Forschungsprogramm der Wirtschaftsinformatik (Langfassung)"

2006

No 7 (December 2006)
Frank, Ulrich: "Towards a Pluralistic Conception of Research Methods in Information Systems Research"

No 6 (April 2006)
Frank, Ulrich: "Evaluation von Forschung und Lehre an Universitäten – Ein Diskussionsbeitrag"

No 5 (April 2006)
Jung, Jürgen: "Supply Chains in the Context of Resource Modelling"

No 4 (February 2006)
Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part III – Results Wirtschaftsinformatik Discipline"

2005

No 3 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part II – Results Information Systems Discipline"

No 2 (December 2005)

Lange, Carola: "Development and status of the Information Systems / Wirtschaftsinformatik discipline: An interpretive evaluation of interviews with renowned researchers, Part I – Research Objectives and Method"

No 1 (August 2005)

Lange, Carola: „Ein Bezugsrahmen zur Beschreibung von Forschungsgegenständen und -methoden in Wirtschaftsinformatik und Information Systems"

Research Group	Core Research Topics
Prof. Dr. H. H. Adelsberger Information Systems for Production and Operations Management	E-Learning, Knowledge Management, Skill-Management, Simulation, Artificial Intelligence
Prof. Dr. P. Chamoni MIS and Management Science / Operations Research	Information Systems and Operations Research, Business Intelligence, Data Warehousing
Prof. Dr. F.-D. Dorloff Procurement, Logistics and Information Management	E-Business, E-Procurement, E-Government
Prof. Dr. K. Echtle Dependability of Computing Systems	Dependability of Computing Systems
Prof. Dr. S. Eicker Information Systems and Software Engineering	Process Models, Software-Architectures
Prof. Dr. U. Frank Information Systems and Enterprise Modelling	Enterprise Modelling, Enterprise Application Integration, IT Management, Knowledge Management
Prof. Dr. M. Goedicke Specification of Software Systems	Distributed Systems, Software Components, CSCW
Prof. Dr. R. Jung Information Systems and Enterprise Communication Systems	Process, Data and Integration Management, Customer Relationship Management
Prof. Dr. T. Kollmann E-Business and E-Entrepreneurship	E-Business and Information Management,E-Entrepreneurship/ E-Venture, Virtual Marketplaces and Mobile Commerce, Online-Marketing
Prof. Dr. B. Müller-Clostermann Systems Modelling	Performance Evaluation of Computer and Communication Systems, Modelling and Simulation
Prof. Dr. K. Pohl Software Systems Engineering	Requirements Engineering, Software Quality Assurance, Software-Architectures, Evaluation of COTS/Open Source-Components
Prof. Dr.-Ing. E. Rathgeb Computer Networking Technology	Computer Networking Technology
Prof. Dr. A. Schmidt Pervasive Computing	Pervasive Computing, Uniquitous Computing, Automotive User Interfaces, Novel Interaction Technologies, Context-Aware Computing
Prof. Dr. R. Unland Data Management Systems and Knowledge Representation	Data Management, Artificial Intelligence, Software Engineering, Internet Based Teaching
Prof. Dr. S. Zelewski Institute of Production and Industrial Information Mangement	Industrial Business Processes, Innovation Management, Information Management, Economic Analyses

For more information visit us on the Web: **http://www.icb.uni-due.de**

ISSN 1860-2770 (Print)
ISSN 1866-5101 (Online)