HIKOMSYS
How I KnOw My SYStem

Learning About Java Dependencies Through Gamification

Bachelor Thesis

Dominique Rahm
from
Bern BE, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

July 2015

Prof. Dr. Oscar Nierstrasz
Andrea Caracciolo

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

How well do you know the dependencies within your system? Dependencies between methods, classes, interfaces, libraries and even different projects play an important role in today's Java projects.

Our solution, called How I KnOw My SYStem (HIKOMSYS), is essentially a platform for developers to improve their knowledge. HIKOMSYS provides a way to learn more about a project. With the help of gamification, HIKOMSYS engages developers to learn about their projects, in particular about its dependencies, while having fun.

After uploading a project hosted on Github, users are able to select the modules within this project and draw the dependencies between those. As soon as they complete a quiz, HIKOMSYS gives the users helpful feedback, by showing them the dependencies they ignored, those wrongly assumed and of course those they knew about. Furthermore, users working on the same projects are able to compare their knowledge about their system with the help of a ranking board, displaying the best results for each user and each project.

Two different case studies, one quantitative with 23 students of the University of Bern and one qualitative, showed that users are very interested in learning about their projects with the help of our tool. Thanks to the gamification aspect of HIKOMSYS, some of the 23 students started comparing their results to see who did better and who knew more.

Adding new levels with different difficulties would increase the fun users have in solving quizzes and competing against each other, as well as increasing their insight into their systems. Additional levels could also help improving the quality of a project, for example by suggesting possible refactorings, recommending improvements to a user's project and letting them re-upload their project for re-evaluation. Furthermore, letting users guess which dependency is wrong in a given selection of modules or even asking multiple choice questions about properties and relationships existing among classes, methods and dependencies, would be a possibility.

Acknowledgments

I would like to thank Andrea Caracciolo[1] and Mircea Lungu[2] for having a lot of patience with me and this project.

I thank Haidar Osman and Bledar Aga for helping me successfully test HIKOMSYS with 23 computer science students, while keeping track and taking notes of everything those students did. Special thanks to Haidar once more for always having time to test and talk about my project and other funny things.

I would also like to thank Michael Single for providing a bigger project (his bachelors thesis [3]) and time for a case study.

Also I would like to thank Eleonora Windler for proofreading and criticizing my every word while also motivating me to finally finish this project. Thanks to you I greatly improved my English writing skills.

I would like to give my thanks to my family for believing in my work knowing I would not be satisfied with a minor project although it took a very long time. Thank you again for leading me on the great path of computer science and web development starting as a child.

[1] `http://scg.unibe.ch/staff/Caracciolo`
[2] `http://scg.unibe.ch/staff/mircea`
[3] `https://Github.com/simplay/Bachelor-Thesis`

Contents

1 Introduction

In today's larger Java projects, code organization becomes more and more important. It improves the code's quality and maintainability. There are many ways to keep track of all information within a project. Developers write Java documentation, draw UML diagrams and refactor their code to keep the project in a clean and accessible state.

Before going into more detail, let us have a look at the 'SampleServiceImpl[1]' class of the Java project we used for our case study (see Page 6). On lines 4 and 5, we can see that the SampleServiceImpl class depends on 'UserDao' and 'AddDao' and therefore on the 'Dao' module itself. Furthermore, we can tell that the 'saveForm' functions (line 8) throws an 'InvalidUserException' and last but not least, on line 18 we create a new User, coupling this function to the user model. The next step would be to document our findings to let the next developer know what the different dependencies of this class are without looking through it again.

Java documentation may be helpful for describing what a specific class is responsible for, what the different functions do and on what other classes it depends on. Unfortunately there are is a problem with this and the other previously mentioned methods. UML diagrams and documentations become outdated, which means they need to be maintained or they are not read at all. Knowing that, we created a platform, called HIKOMSYS, where one can upload Java projects and test their knowledge about a system, while competing against other users collaborating on the same project.

[1] `https://Github.com/ese-unibe-ch/ese2014-wiki/blob/master/Skeleton/src/main/java/org/sample/controller/service/SampleServiceImpl.java`

```
@Service
public class SampleServiceImpl implements SampleService {

  @Autowired    UserDao userDao;
  @Autowired    AddressDao addDao;

  @Transactional
  public SignupForm saveForm(SignupForm signupForm) throws InvalidUserException{

    String firstName = signupForm.getFirstName();

    if(!StringUtils.isEmpty(firstName) && "ESE".equalsIgnoreCase(firstName))
      throw new InvalidUserException("Sorry, ESE is not a valid name");

    Address address = new Address();
    address.setStreet("TestStreet-foo");

    User user = new User();
    user.setFirstName(signupForm.getFirstName());
    user.setEmail(signupForm.getEmail());
    user.setLastName(signupForm.getLastName());
    user.setAddress(address);

    user = userDao.save(user);   // save object to DB

    // Iterable<Address> addresses = addDao.findAll();  // find all
    // Address anAddress = addDao.findOne((long)3); // find by ID

    signupForm.setId(user.getId());

    return signupForm;
  }
}
```

HIKOMSYS provides a platform for Java developers and teams to improve their knowledge in a fun and competitive way. This is achieved by letting users take quizzes about the dependencies between their selected modules in their project. As soon as a user finishes a quiz, HIKOMSYS tells him how he did and what dependencies he missed or assumed wrong. Additionally, users are able to compare themselves to other users, who took a quiz on the same project, on the ranking board.

With the help of gamification, HIKOMSYS enhances the user's knowledge by supporting the exploration of all the dependencies contained within a given system (possibly also highlighting refactoring opportunities and anomalies, such as tight coupling). As a result, users are able to improve the re-usability of the system's modules.

Our goal was not only to improve our user's knowledge, we also wanted to collect data. HIKOMSYS collects various data across any quiz taken, from saving the position of

multiple elements inside a canvas up to which modules are commonly selected and even what mistakes are regularly made. This may help one to identify common behavioral patterns and implicit assumptions that might help one to understand how developers perceive their software.

2
Related Work

In the words of Gabe Zichermann and Christopher Cunningham: “What do Foursquare, Zynga, Nike+, and Groupon have in common? These and many other brands use gamification to deliver a sticky, viral, and engaging experience to their customers.”[4]. We wanted to achieve the same effect using gamification to create a platform, where software developers can improve their knowledge.

But learning something new can be a time consuming and sometimes even boring process. Gabe Zichermann and Christopher Cunningham claim that “The process of game-thinking and game mechanics to engage users and solve problems”[4] is one reason why “games have begun to influence our lives every day.”[4]. Going even one step further “simulation and digital-based learning are considered to have great potential to extend the learning experience”[1] of its users. But gamification does not only improve the user’s experience, according to Xie Tao, Tillmann Nikolai and de Halleux Jonathan, it provides “automatic grading, intelligent tutoring, problem generation, and plagiarism detection”[3]. We are not only able to grade a single user, as Jonathan Bell said, we are able to “measure the performance of the whole team”[1], for example, with the help of leaderboards.

On one hand, tutors can reach a bigger audience with less effort and time. They can even automate most of the steps as we will see on an example later on. On the other hand, a student can enjoy the “benefits of automated grading of exercises assigned to”[1] him, therefore getting faster feedback on his progress. Also “playful experimentation becomes part of the learning process”[1]. This was not possible with previous learning systems due to user’s lack of time or motivation. They had to wait for feedback or just wanted to be done with an exercise as soon as possible. All this “motivates learners to

take responsibility for their own learning, which leads to intrinsic motivation"[1].

In summary, the following points are considered to improve the learning experience and success of a learning application:

- automated grading and instant feedback
- measuring the performance
- status (experience or reputation)
- responsibility for their own learning
- bigger audience

There are two examples I would like to talk about considering the points made above to show how different approaches can lead to a solid and easy way of learning and helping others:

1. Stack Overflow[1] is one of the best online sources for Q&A related to computer science. Posting questions or answers rewards users with reputation points and badges. Those "badges have been used to provide incentives for Stack Overflow users to ask or answer questions"[3], as well as pointing out to the author how much experience users have, answering his question. Furthermore, there are additional features which are only unlocked after collecting a certain amount of reputation points. Commenting on a question or answer is only possible after gaining 50 reputation points. This makes sure that only users with certain experience comment on possible answers and also motivates other users to earn the needed reputation quickly by posting questions and answers.

2. The other example is "Shaping up with AngularJS"[2] created by Google and Codeschool[3]. Codeschool successfully enhances its courses by adding interactive programming tasks or quizzes in between lectures. For example, AngularJS[4] can be learned through video lectures. In between those, the user either has to answer some multiple choice questions or solve a problem in the "Flatlander store", a sample project created for the user, by implementing or improving AngularJS code. They both are instantly corrected, the user earns points for correct answers and receives immediate feedback. Those points can be spent for tips on a given task or kept for the leaderboard at the end of the lecture.

[1] `http://stackoverflow.com/`
[2] `https://www.codeschool.com/courses/shaping-up-with-angular-js`
[3] `https://www.codeschool.com`
[4] `https://angularjs.org/`

As seen above, “gaming, social dynamics, educational usage and software engineering technologies”[3] are “four common aspects of a typical project on educational software engineering”[3] in “the coming age of Massive Open Online Courses (MOOCs)”[3].

The other thing we have to consider is a way to compare an engineer’s high-level model with an actual implementation of a software system. Murphy,Notkin and Sullivan claim that their reflexion model “helps an engineer use a high-level model of the structure of an existing software system as a lens through which to see a model of that system’s source code”[2].

This is exactly what HIKOMSYS does, we built a tool where a developer “first specified a model he believed, based on his experience, to be characteristic”[2] for the system on hand. Afterwards “a tool then computes a software reflexion model that shows where the engineers high-level model agrees with and where it differs from a model of the source”[2] revealing divergences between the developers point of view and the actual implementation. HIKOMSYS provides a tool for developers “to easily explore structural aspects of a larger software system”[2], focused on its dependencies, while also producing “at low-cost, high-level models that are good enough”[2].

3
The Solution

3.1 Architecture

This chapter focuses on the general architecture of HIKOMSYS and provides a brief overview of the project. Each individual part will be discussed in more detail later in Section 3.3.

Like most web applications, HIKOMSYS consists of a back end and a front end, as shown in Figure 3.1.

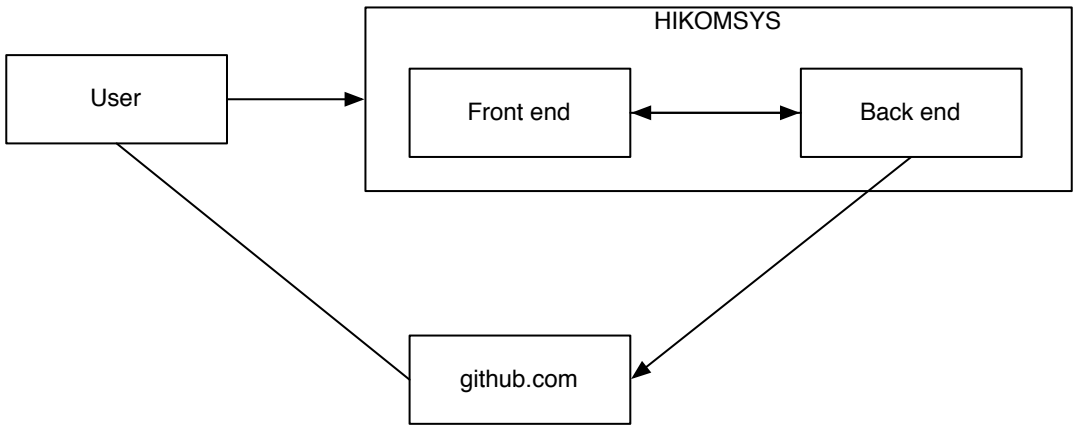


Figure 3.1: General HIKOMSYS overview

Users, having Java projects hosted on Github, interact with the front end by uploading projects, managing their profile or solving quizzes. The back end evaluates user input, manages uploads, displays results and more. As shown in Figure 3.2, the back end relies on a Model-View-Controller (MVC) framework (Laravel) which provides a UMS. The system relies on two databases (MySQL used for user management and MongoDB to store the projects). The front end depends on a number of Javascript and Cascading Style Sheets 3 (CSS3) frameworks & libraries, which are used to support HIKOMSYS multiple functionalities, ranging from drawing on a canvas to populating user input. Additionally, the front end loads some data asynchronously from the databases. We will have an in-depth look at all these features in the next two sections.

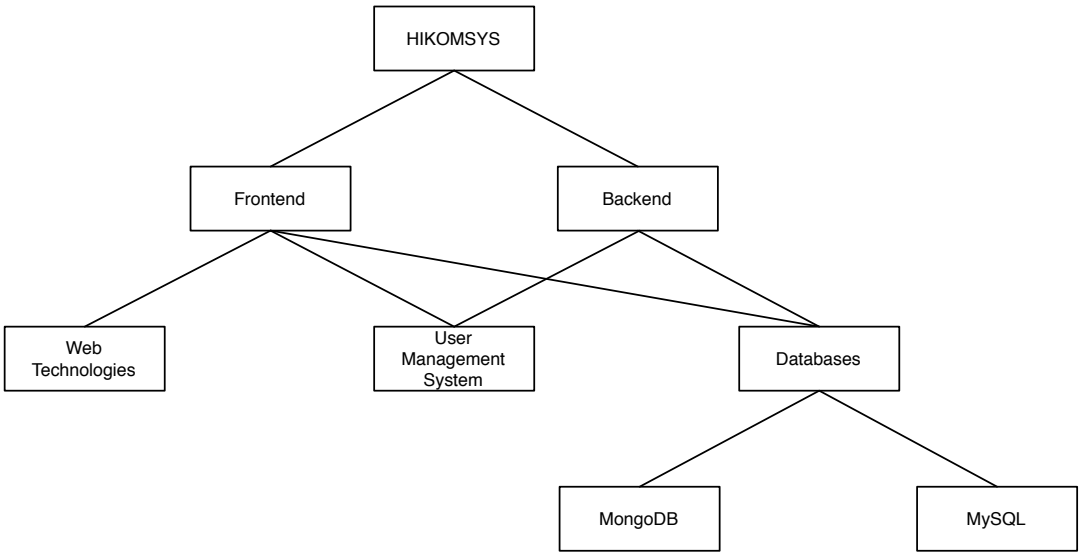


Figure 3.2: The individual parts of HIKOMSYS

3.2 User Experience

Within this section we talk about the user interaction and experience with HIKOMSYS. We describe a typical user scenario, from sign up to starting and solving a quiz.

3.2.1 Dashboard

As soon as a user is registered or signed up, the dashboard is presented to him. If he is a new user, no project is shown and he is asked to upload his own (See Figure 3.3). Otherwise the user sees all projects he uploaded previously and can choose between a number of actions for each of those (e.g. Start new quiz, inspect ranking and solutions), as shown in Figure 3.4.

Figure 3.3: Dashboard: Upload

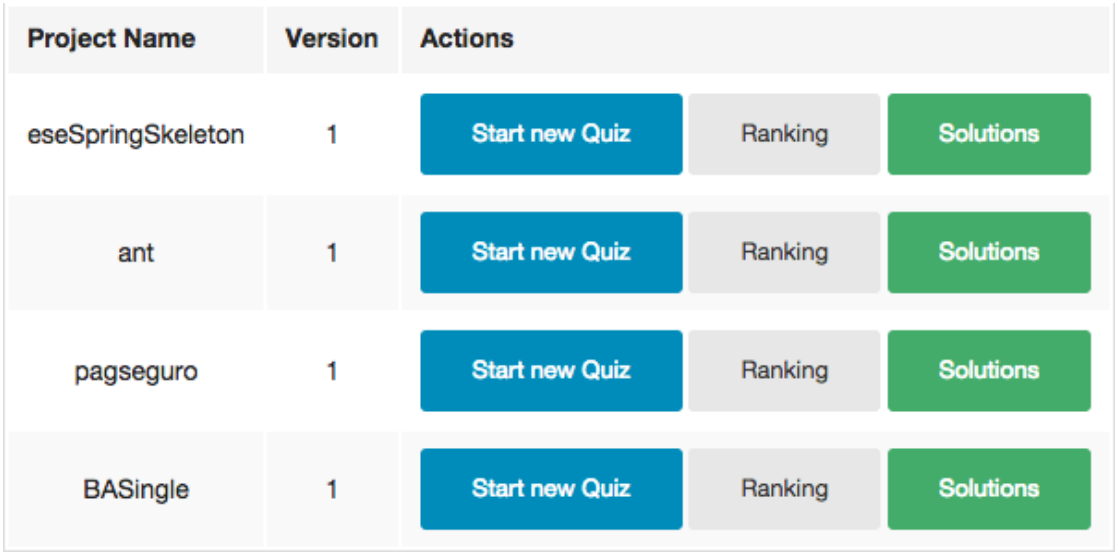


Figure 3.4: Dashboard: Projects overview

As in other UMS, users can manage their profile or inspect profiles of other users. They can also inspect other players' solutions for any completed quiz. Administrators have some additional rights and are capable of deleting users. More user-specific features and rights could be added in the future.

3.2.2 Package Selection

After starting a new quiz, the player has to select the main packages or modules of his system. To do so, HIKOMSYS provides an interactive tree view created out of HyperText Markup Language 5 (HTML 5) lists using the JStree[1] library (See Figure 3.5).

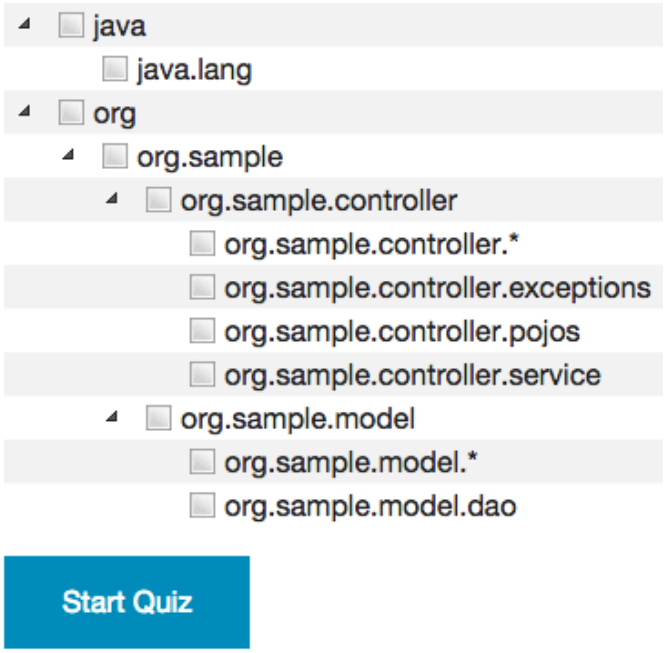


Figure 3.5: Select modules in the tree view

The user can simply click on the node corresponding to the considered module to see the children of a package. Selecting a package for the quiz is done by checking the corresponding checkbox. With a right click on a tree node a custom menu is displayed, see Figure 3.6

[1] http://www.jstree.com/

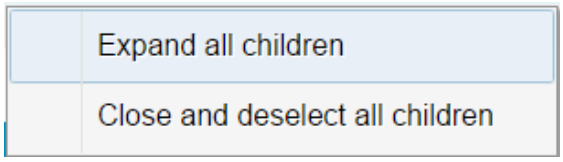


Figure 3.6: modified JSTree menu

The first option fully expands the selected package and all containing subpackages, whereas the second option closes and deselects all children. The modified menu should help the user select or deselect modules and navigate through the project's structure more easily. If the user is satisfied with his selection he can start the quiz by clicking on the "Start Quiz" button.

3.2.3 The Quiz

All the previously selected packages appear on the canvas in the middle of the screen (Figure 3.7). Their starting position is randomly set. The user can either move them around or draw dependencies between them. Modes are switched by interacting with the toggle button positioned at the top of the canvas.

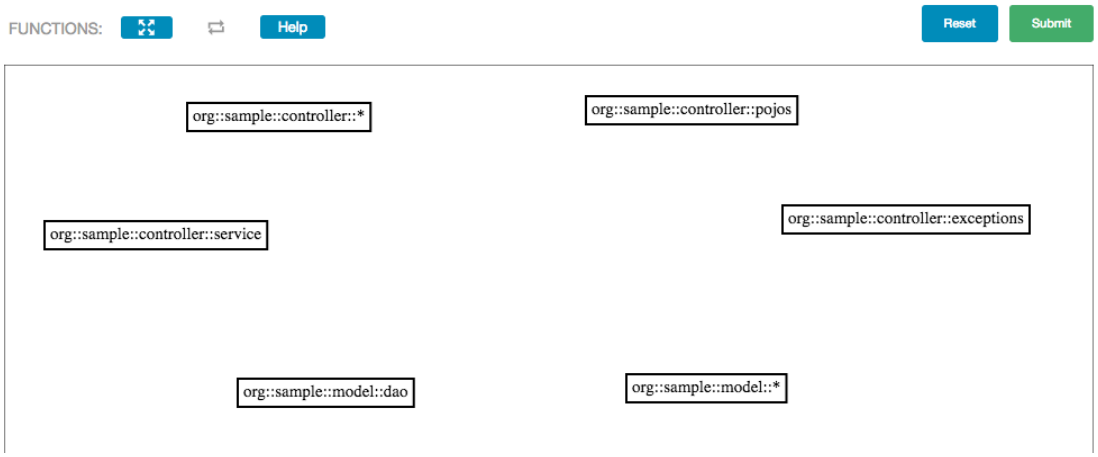


Figure 3.7: Starting the quiz displays the previously selected main modules of the system

After switching out of the moving mode, the user can start drawing a dependency by clicking and dragging an arrow from one package to another. As soon as he starts dragging, the arrow attaches itself to the mouse cursor and follows it along the canvas until the mouse is released. Releasing the mouse over another package attaches the arrow to it. If the cursor is released on the same package or not on a package at all, the arrow disappears and the user has to start over again.

After drawing all the dependencies (see Figure 3.8), the user can submit his solution and see the results.

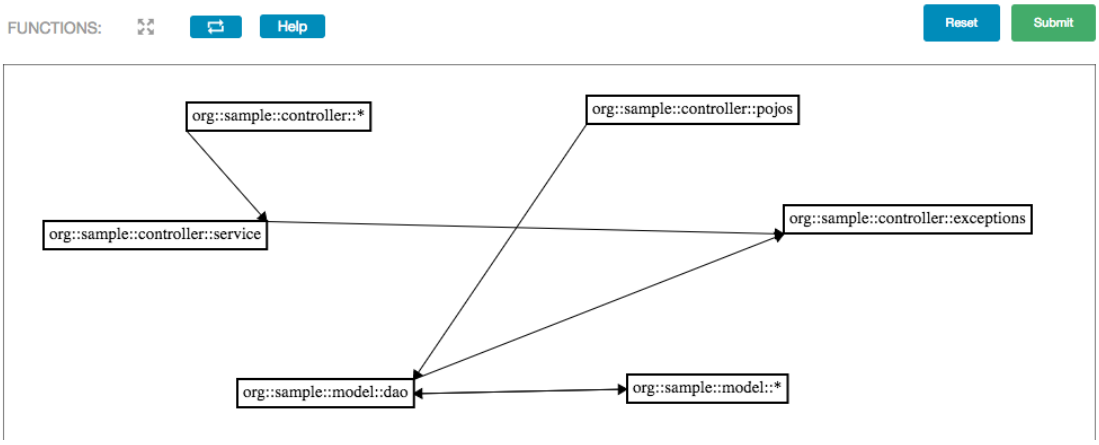


Figure 3.8: Some dependencies

3.2.4 Result View

The results are presented in multiple views. After submitting a solution, the exact same canvas with all the packages selected by the user is rendered on the screen. The first view presented to the user shows the dependencies drawn correctly by him (green arrows in Figure 3.9). The color of each individual package show that either all outgoing dependencies are correct, wrong or missing. If at least one outgoing dependency is missing the package's color is yellow. Analogue the package is colored red if one or more outgoing dependencies are wrong.

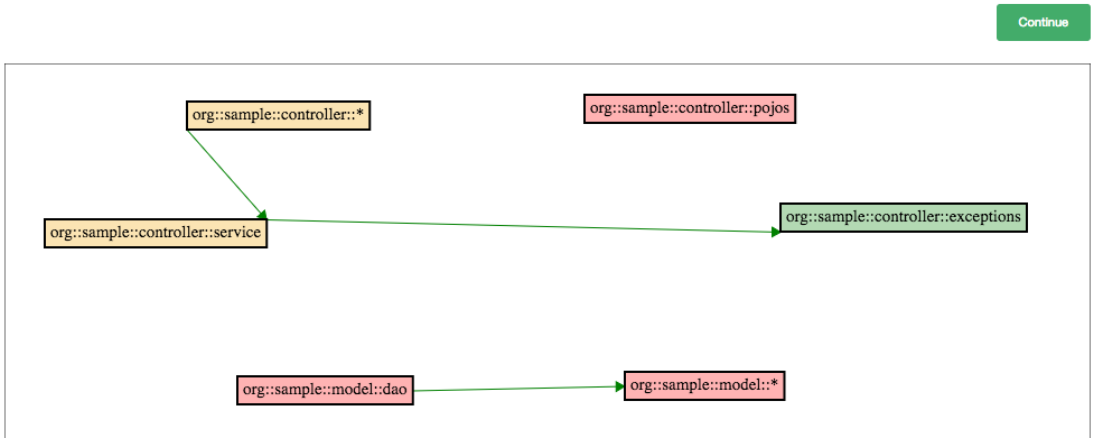


Figure 3.9: Result View 1 - Correct dependencies

Clicking on the “Continue” button leads to the second view, where the user sees the missing dependencies as yellow arrows (See Figure 3.10).

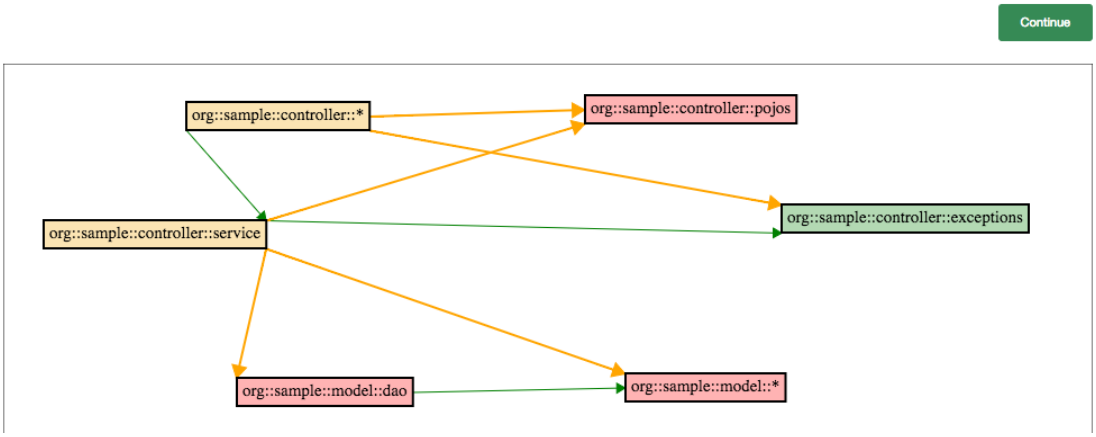


Figure 3.10: Result View 2 - Missing dependencies

Next, the user sees all incorrect dependencies as red arrows (Figure 3.11).

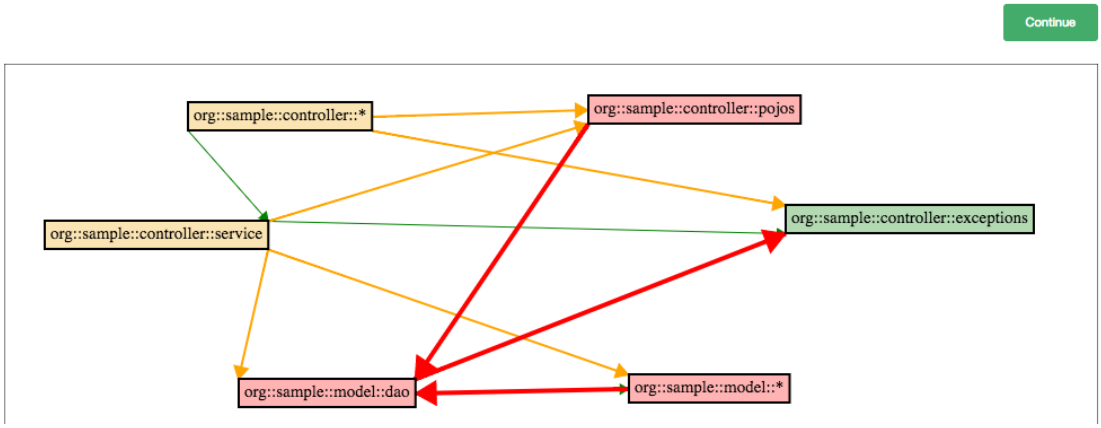


Figure 3.11: Result View 3 - Wrong dependencies

Clicking on the “Continue” button once more presents the user with all the previous information as well as a summary on the right side of the page (Figure 3.12).

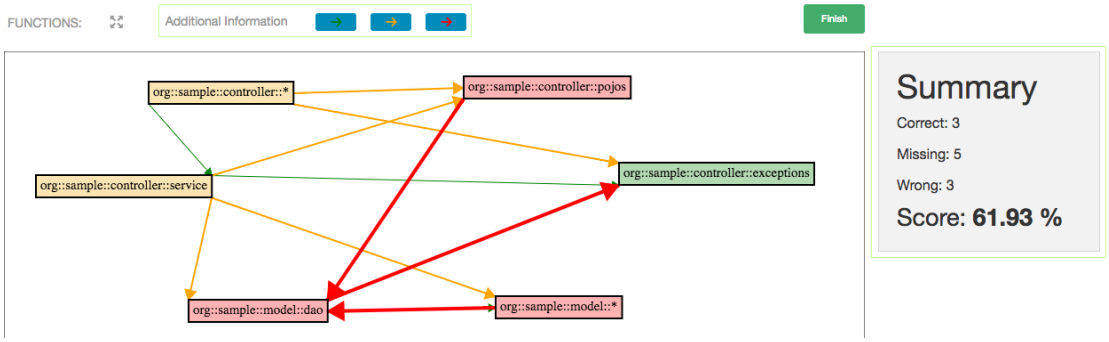


Figure 3.12: Result View - Summary

Furthermore, the user can once more drag around the packages and he can now also enable “Additional Information”. By doing so he can see detailed information about a package’s children, classes and even dependencies by hovering over it.
For example, when expanding the dependency list (Figure 3.13) all outgoing dependencies of the hovered package are shown. Clicking on the “Finish” button leads the player to the ranking board.

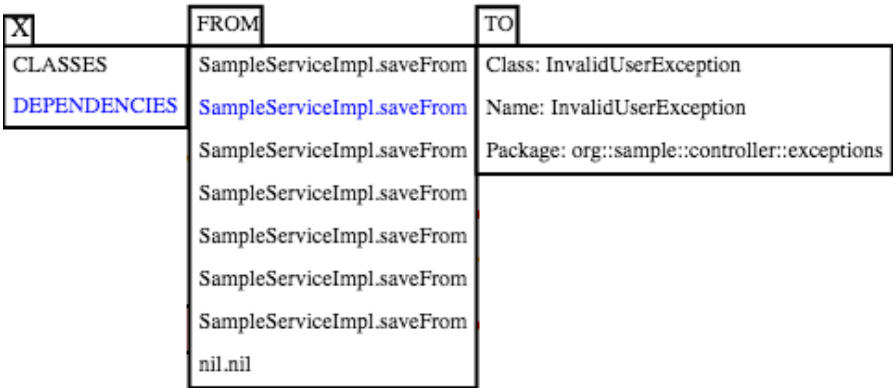


Figure 3.13: Additional Information - Dependencies

3.2.5 Ranking Board

Each project uploaded to HIKOMSYS has a ranking board listing the best players that took the quiz on that project. The individual score is represented by a green bar (Figure 3.14 - green bar) and the scores of all other participants by a blue bar. The list is ordered from the highest achieved score to the lowest. Each submitted solution can be inspected by every user.

Figure 3.14: Ranking Table

Users can also inspect all their previously completed quizzes to review their mistakes or check the existing dependencies. This functionality is available through the dashboard or the navigation menu.

3.3 Technical Implementation

As we saw in section 3.1, HIKOMSYS can be split into front- & back end. Let us go into more details about those two individual parts.

3.3.1 Back end

The interaction with the back end is illustrated in the following sequence diagram (see Figure 3.15).

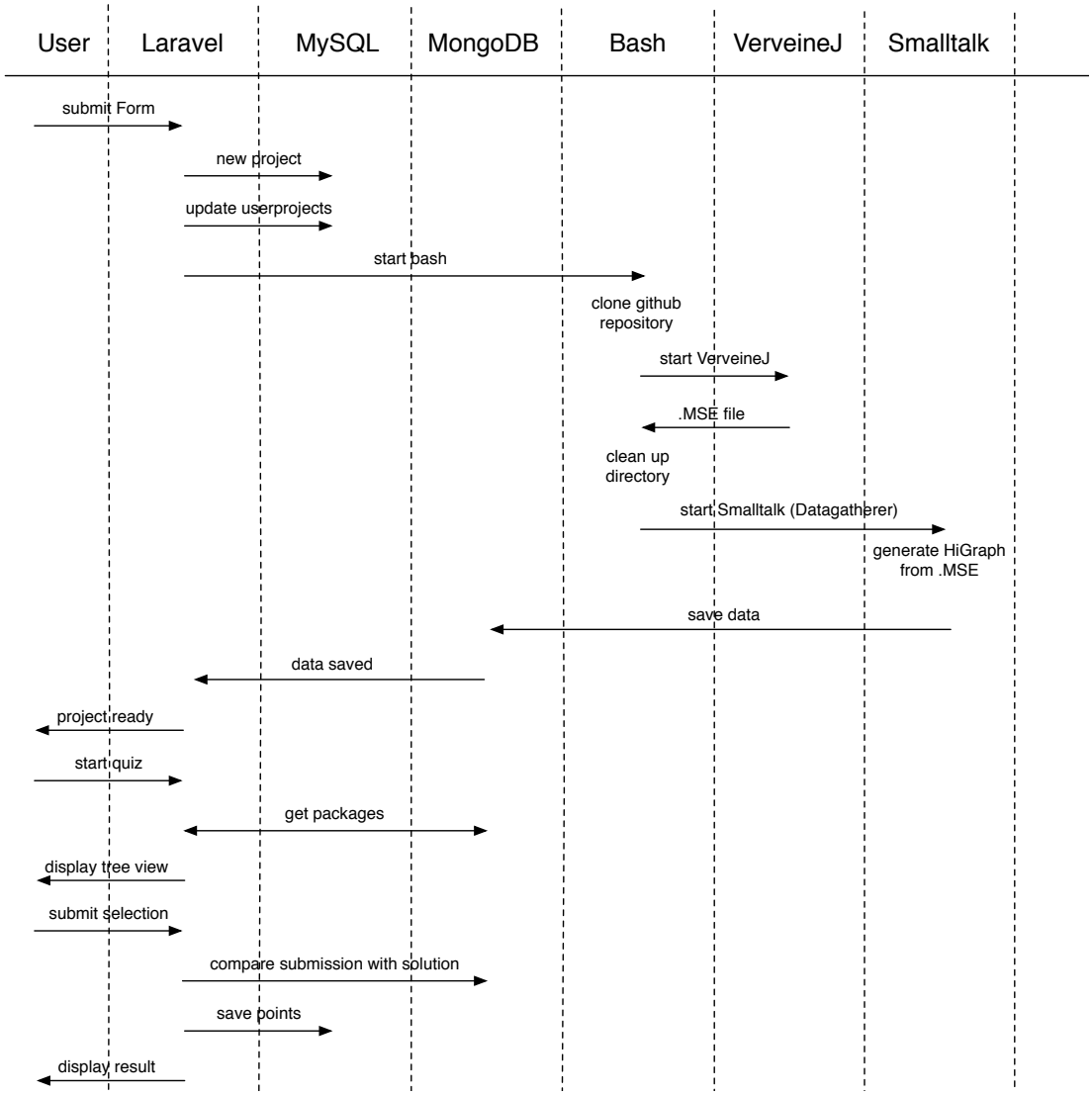


Figure 3.15: back end sequence diagram

As soon as the user submits a project, the user data (`id`) and the actual input (Github `URL` and `project name`) is passed to Laravel, i.e. the project controller. There are two possible options. If the project has already been uploaded and the user already has this project within his project table, nothing happens. If the user does not have the current project yet, his user ID is added to the table. A project exists if and only if the URL and the hash value[2] of the current master branch are equal to one of the projects within the Database. Although the hash value is not a perfect unique value, it is highly unlikely to have the same hash value within the same project twice. As a result, the same project can be uploaded to HIKOMSYS as long as the hash value of the projects is not the same. To be more precise, different stages (branches or versions) of a project are treated as individual projects. The path to create a new project, is far more complex, therefore it is split up into its individual components, which are:

- Bash
- VerveineJ
- Datagatherer
- Databases
  - MongoDB
  - MySQL

3.3.1.1 Individual Components

A bash script is responsible for orchestrating all the different tools used to download, parse and import the project data as well as cloning the git repository into the *gitRepos* directory.

Parsing The Code

VerveineJ[3] and inFamix[4] are two parsers for creating model representations of Java projects which can be imported into Moose. Those model files contain all the important data about a Java project (like all the class, method, package and variable names, all the dependencies and many more). Although inFamix does a better job at parsing Java Code than VerveineJ, HIKOMSYS uses VerveineJ due to limited Random-access memory (RAM) availability on the server it runs on. In fact, inFamix did typically not manage to completely parse larger Java projects and therefore we decided to opt for the other parser. Either of those two parsers can be easily invoked from the command line:

[2] Git creates a unique SHA-1 hash value for each branch and commit.
[3] `https://gforge.inria.fr/projects/verveinej`
[4] `http://www.intooitus.com/products/infamix`

```
./verveinej.sh -Xmx2000m gitRepos/$name/src
```

Only two arguments are needed for VerveineJ to run on the server. The first one tells it to use 2 gigabytes of RAM for parsing the project and the second argument is the path to the *gitRepos* directory where the project is currently being uploaded.

As soon as VerveineJ is finished parsing the Java code and the model file is created, the directory is cleaned up by deleting everything besides the needed file.

Creating the Database Entries

After creating the model file, Pharo is called with the *runDataGatherer* command and the current project folder as argument. This will start the data gatherer tool, implemented in Pharo, which will collect all the required data and save it to the Mongo database. Pharo[5] is a pure open source implementation of Smalltalk[6]. It is not only a programming language but also a powerful environment. Moose[7] is a platform for software analysis based on Pharo.

Quicksilver[8], is a library for Moose capable of creating a HiGraph of a given project. HiGraphs are graphs made of nodes, each representing a package. This tree contains all information about classes, packages and the dependencies existing among them. By recursively traversing a HiGraph, starting with all the nodes without parent element (the root packages of a project), HIKOMSYS collects all the important information it needs. This includes package names, parent packages, classes and outgoing and incoming dependencies.

As soon as the DataGatherer is finished with its task, all obtained information is stored to the HIKOMSYS Mongo database together with the current project ID as the name of the document[9]. HIKOMSYS uses MongoDB because other alternative Database management systems (DBMS) only benefit from limited support in Pharo.

3.3.1.2 Laravel and UMS

Laravel [10] is a PHP framework that provides an UMS for users to sign up/register, manage their profile and inspect other users. Laravel simplifies the creation of a lightweight web applications by binding models and their database tables. As a result no queries

[5] http://pharo.org/

[6] http://www.smalltalk.org/main/

[7] http://www.moosetechnology.org/g/

[8] http://scg.unibe.ch/research/quicksilver

[9] MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs: { ”item”: ”pencil”, ”qty”: 500, ”type”: ”no.2” }

[10] http://laravel.com/

are needed. Furthermore with the help of RESTful[11] services Laravel provides, it is possible to inhibit access to specific URLs (routes) from people which are not signed in. Additionally, different user roles can be created (for example an administrator can change the user profiles of all the users and he can delete users). All the tables currently used for HIKOMSYS are listed in Figure 3.16. There is another table (i.e. "Migrations") which was not represented because it is always present in Laravel projects. Migrations are used to set up the database and, later on, even extend or change tables.

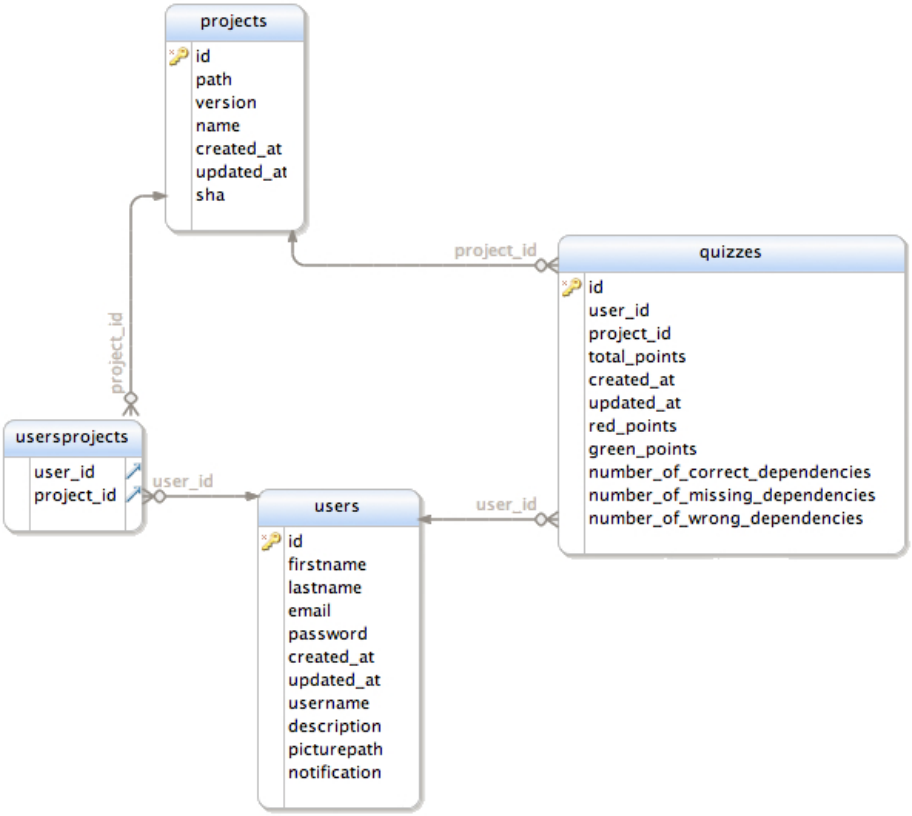


Figure 3.16: Database schema

[11] Representational state transfer: `http://en.wikipedia.org/wiki/Representational_state_transfer`

3.3.1.3 Result

As soon as the user/player submits his solution for the quiz, the result is created. For this purpose three tables are used:

- User Submission[12]
- Solution[13]
- Result[14]

Each project has its own solution table which is created only once, at the time the project is uploaded to HIKOMSYS. As we will see later, the solution table is copied every time a user takes a quiz. Therefore we would need to either query the table for the current quiz and only copy the result or copy the full table. Although it would be possible to have all the projects in one table, we chose to have an individual table for each project, containing all packages and dependencies. As soon as a quiz is completed, user defined dependencies are compared to the actual model extracted from the source code which is saved in the solution table. The outcome of the comparison is stored in the result table.

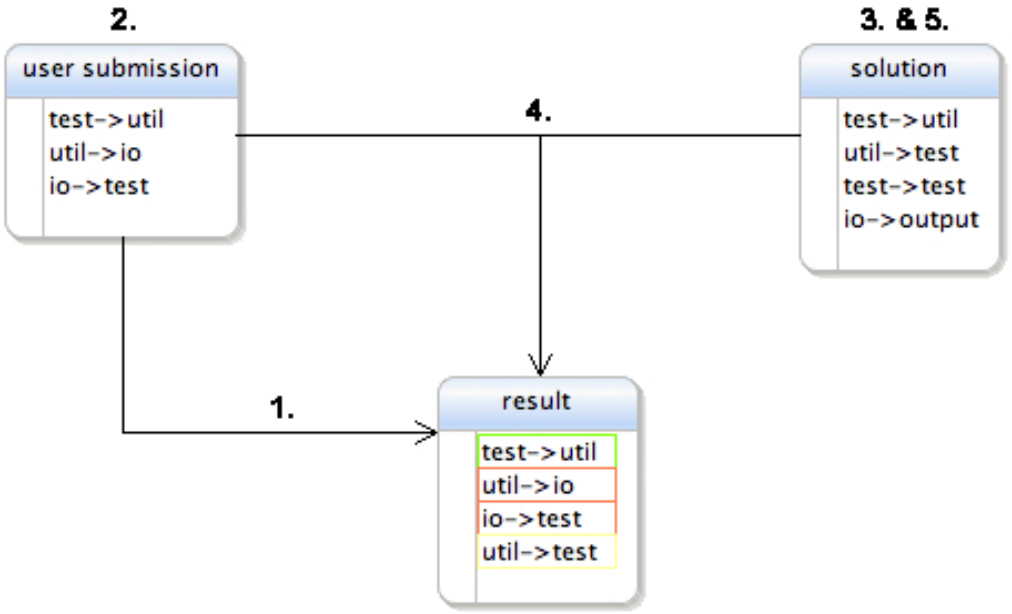


Figure 3.17: Steps to create the result table for a quiz.

[12] US_quizID
[13] S_quizID
[14] RES_quizID

All the steps needed to create the results table are shown in Figure 3.17 and explained in more detail in the following.

In the first step (Figure 3.17 number 1), all submitted packages with their name and position on the screen are added to the result table (e.g name: util, position: [x,y])

Secondly, the dependencies need to be checked. Hypertext Preprocessor (PHP) iterates over every dependency in the submission table and checks if that dependency is also contained in the solution table. If a dependency matches another, it is added to the result table with the color green, otherwise with the color red. (e.g test - util is in both, therefore add it to results with the color green, see number 2). Each matching dependency is removed from the solution table because it cannot be a missing dependency (Figure 3.17 number 3). This is the reason why a copy of the solution table is required.

Next, all the dependencies that the user forgot to specify are added to the result table (see number 4). This is done easily because all the matching dependencies were deleted in the previous step and the dependencies left within the solution table are either those missed before or those from different packages.

Finally, the solution table is deleted and replaced by its previously created copy (step number 5). This is a safe process because each individual quiz has its own copy of the solution table and therefore even if two different users are taking the same quiz and reach this point of the calculation at the same time, different tables are deleted.

3.3.2 Point Calculation

HIKOMSYS' main goal is to let users know how well they know their system. For each submitted quiz, HIKOMSYS calculates how well the user did ranging from 0% to 100%. Any given quiz can be in one of four of the following distinguishable states:

1. If the user only draws dependencies which are not in his system, he should get 0% on the quiz.

2. If the user draws all the dependencies which really are within his system without mistakes, his quiz is 100% correct.

3. If the user draws nothing at all, he should receive a better score compared to other users drawing almost only wrong dependencies. In fact, knowing nothing about your system should be considered better than having incorrect assumptions.

4. If the user draws every possible dependency, he has, as a logical conclusion, all the correct (100%) and all the wrong dependencies (0%). We figured it would make sense to award this case with 50%.

Taking those four states into consideration we need to figure out a way to rate a quiz based on the number of correct and wrong dependencies. First of all, we start by defining some variables:

- $dep_{correct}$ is the number of correct dependencies
- Let `P` be the packages selected by the user, then $|P|$ is the number of selected packages.
- `t = |P|*(|P|-1)` the number of all possible dependencies
- `t -` $dep_{correct}$ the number of all wrong dependencies
- $|userdep_{correct}|$, $|userdep_{missing}|$ and $|userdep_{wrong}|$ are the number of correct, missing or wrong dependencies submitted by the user.

Moving forward, we will think about percentages as points ranging from 0 to 100, therefore $p_{correct}$ are the points awarded for a correct dependency. Similarly, p_{wrong} are the points for a wrong dependency. I would like to specially point out not to confuse p_{wrong} with $|userdep_{wrong}|$. The former being the points for each wrong dependency and the latter being the number of the submitted dependencies which are wrong (the same is said for $|userdep_{correct}|$). With all this in mind, we are able to formulate the following equations to find out how to rank a quiz.

$$dep_{correct} = |userdep_{missing}| + |userdep_{correct}| \tag{3.1}$$

$$100 = dep_{correct} * p_{correct} \Rightarrow p_{correct} = 100/dep_{correct} \tag{3.2}$$

$$-100 = (t - dep_{correct}) * p_{wrong} \Rightarrow p_{wrong} = -100/(t - dep_{correct}) \tag{3.3}$$

In words: A user receives 100% if he has all the correct dependencies, meaning that the number of correct dependencies is equal to the number of submitted dependencies. Therefore one correct dependency should be worth a fraction of 100%, to be more precise, 100% divided by the number of actual dependencies within the system ($dep_{correct}$). As a result, the more dependencies are within a system the less one correctly submitted dependency is worth and vice versa.

Furthermore, the user is awarded with -100% if he submitted all wrong dependencies, which as declared above, are all possible dependencies minus all the correct dependencies. By calculating $p_{correct}$ and p_{wrong} for each individual quiz, based on the number of selected packages and therefore based on the number of correct and the number of possible dependencies, we are now able to calculate the points a user receives for the correctly ($\texttt{points}_{correct}$) and wrongly ($\texttt{points}_{wrong}$) drawn dependencies:

$$points_{wrong} = p_{wrong} * |userdep_{wrong}| >= -100 \tag{3.4}$$

$$points_{correct} = p_{correct} * |userdep_{correct}| <= 100 \tag{3.5}$$

As a result, the total number of points for a quiz is $points_{wrong}$ added together with $points_{correct}$:

$$points_{total} = points_{wrong} + points_{correct} \tag{3.6}$$

Looking back at our four distinguishable states (section 3.3.2), especially number 4, submitting all possible dependencies should award 50%. But using the formulas just created, we can see that it would be awarded with 0%:

$$points_{total} = -100 + 100 = 0 \tag{3.7}$$

Therefore the formula had to be changed to match this condition as well:

$$points_{wrong} = (p_{wrong} * |userdep_{wrong}| + 50)/2 \tag{3.8}$$

$$points_{correct} = (p_{correct} * |userdep_{correct}| + 50)/2 \tag{3.9}$$

As a result, we have the following two cases:

- If there are no wrong dependencies, 25% of the quiz is correct

$$points_{wrong} = (0 + 50)/2 = 25 \tag{3.10}$$

- If there are all correct dependencies, 75% of the quiz is correct

$$points_{correct} = (100 + 50)/2 = 75 \tag{3.11}$$

Let us have a look at a concrete case using the example shown in Section 3.2 (Figure 3.12). If there are 3 correct ($|userdep_{correct}|$) and 3 wrong ($|userdep_{wrong}|$) dependencies, a total of 6 packages ($|P|$) and a total of 8 dependencies between those 6 packages ($dep_{correct} = 8$) the calculation would look like this:

$$t = 6 * (6 - 1) = 30 \tag{3.12}$$

$$p_{correct} = 100/8 = 12.5 \tag{3.13}$$

$$-100 = (30 - 8) * p_{wrong} \Rightarrow p_{wrong} = -100/22 = -4.\overline{54} \tag{3.14}$$

$$points_{red} = (-4.\overline{54} * 3 + 50)/2 = 18.\overline{18} \tag{3.15}$$

$$points_{green} = (12.5 * 3 + 50)/2 = 43.75 \tag{3.16}$$

$$points_{total} = 18.\overline{18} + 43.75 = 61.93\overline{18} \tag{3.17}$$

As we could see, this user would receive 61.93% for his quiz.

The point distribution is one part which may benefit from rework and rethinking. But our main goal was to leave people with a positive feeling about their quiz and this is why we opted to mainly have results above 50%.

3.3.3 Front end

Only the latest web technologies were used to build HIKOMSYS: HTML 5, CSS3 and JavaScript. HIKOMSYS heavily utilizes the mobile first CSS3 web framework Foundation[15]. Foundation makes it very easy to create fully styled and responsive navigation menus and forms[16]. Foundation made the development of an HTML 5 compatible website fast and easy. As seen in the previous section 3.3.1, Laravel and its blade templating made it simple to create page templates and inject only the changing parts. Together with the help of Foundation we created a basic layout all the different pages use, containing a header, a navigation and some basic HTML 5 structure.

JavaScript is the most important front end technology used to build HIKOMSYS . JavaScript and three of its libraries (e.g. jQuery, JStree and KineticJS), was used to build the treeview of the modules, the dependency drawing tool and some AJAX requests.

JQuery[17] was used to simplify the access to Document Object Model (DOM) Objects. Furthermore, HIKOMSYS needs to interact with the database as seen in Figure 3.13. This is done with the assistance of asynchronous requests (AJAX), which are used to retrieve the points for the current quiz or the information about selected packages and drawn dependencies.

[15] `http://foundation.zurb.com/`

[16] On a side note:HIKOMSYS is accessible on all devices although users are not able to take quizzes on mobile devices because drag and drop is not supported on touch devices yet. Users are still able to checkout previous results, the ranking board or user profiles.

[17] `http://jquery.com/`

Drawing Packages and Edges

KineticJS[18] makes it really easy to draw anything on an HTML 5 canvas. Packages are rendered as rectangles; Arrows, representing dependencies, are visualized as a combination of a line and a triangle.

In KineticJS a triangle always is inserted with one of its corners pointing upwards. As we can see in Figure 3.18, the green arrowhead on the right side needs to be rotated according to the corresponding vector's direction (green line). This is achieved by calculating β with the help of the scalar product.

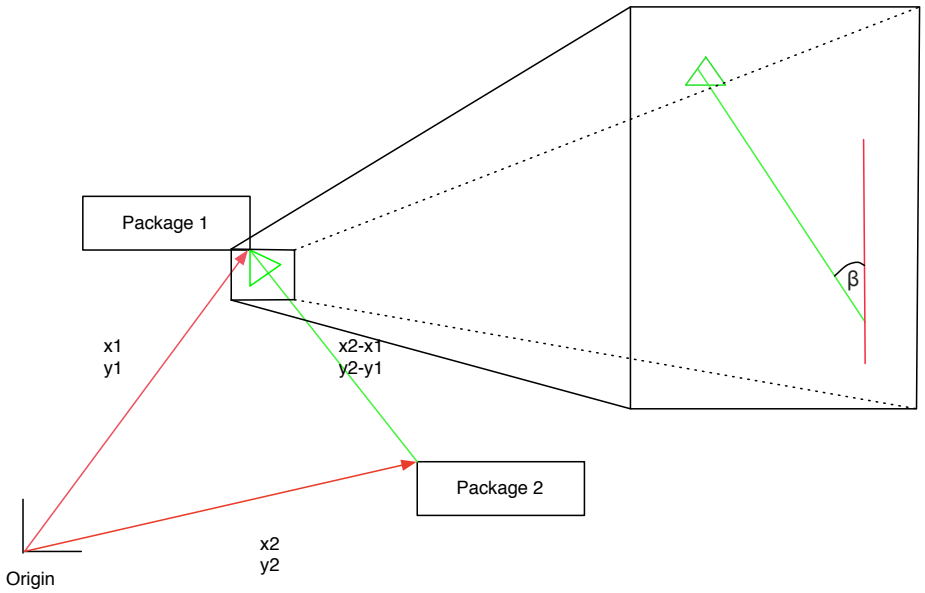


Figure 3.18: Drawing a dependency arrow with vectors

Anchor Points and the Shortest Path Between Packages

It is not enough to just point from the center of a package to the center of the other package. This would quickly lead to a lot of arrows and lines pointing to or starting from the center of a package, making the name of the package unreadable.

To draw dependencies and minimize cluttering, each package has eight anchor points, as shown in the following figure 3.19, to which a dependency arrow can be attached.

[18] https://Github.com/ericdrowell/KineticJS

Figure 3.19: Anchor points

For each dependency arrow, the shortest path between two given packages and their eight anchor points is calculated. To do so, both centers as well as the height and width of one package are needed. We are able to calculate the y offset of one package to another, distinguishing the corner to which the dependency is connected to with the help of the following functions:

```
function yOffset(center1, center2, center1_height) {
  if (isBellow(center1, center2, center1_height)) {
    return center1_height / 2;
  }
  if (isAbove(center1, center2, center1_height)) {
    return -center1_height / 2;
  }
  return 0;
}

function isAbove(center1, center2, center1_height) {
  return center2.y < (center1.y - center1_height / 2);
}

function isBellow(center1, center2, center1_height) {
  return center2.y > (center1.y + center1_height / 2);
}
```

The y offset, depends on the package being above or below the other package. If it is, we know that the offset is either 1 or -1 times half the height of the packages, meaning it is either one of the two top or the two bottom outer anchors. Otherwise, it is either the top or bottom center anchor. We are able to determine the x offset with the help of analogous functions used in a similar way.

As a result, we can think about the position of a package, relatively to another package, as of the eight sections shown in Figure 3.20.

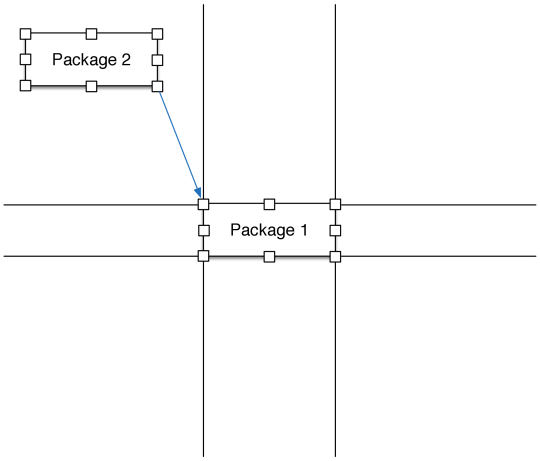
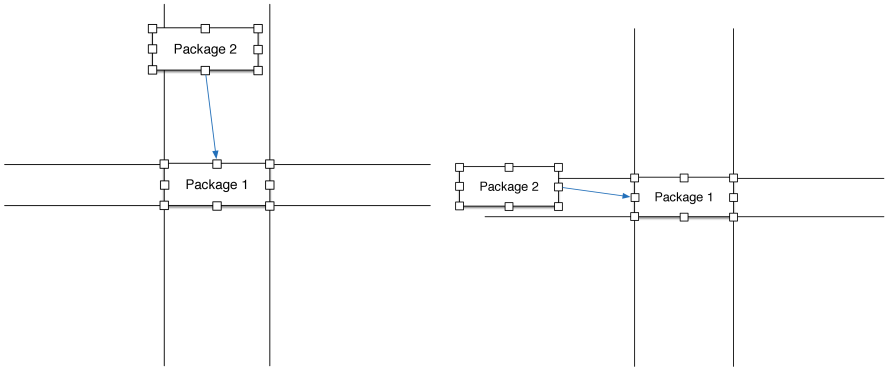


Figure 3.20: Sections

For example, in Figure 3.20, package 2 is in the top left section of package 1. Therefore, the shortest path from one package to the other is from the bottom right corner of package 2 to the top left corner of the other package.

KineticJS allows every object to be draggable and whenever a package is dragged around, the shortest path of each of its outgoing and incoming dependencies has to be recalculated. This is done by deleting and redrawing each dependency arrow as long as it is dragged around, always repeating the steps mentioned above.

Figures 3.21(a) and 3.21(b) show two different situations for the positioning of a package and how the shortest path would change from one section to another.



(a) Center moves above Package 1 (b) Center moves to the left of Package 1

Figure 3.21: Additional shortest path examples

Drawing and Saving with KinetiJS IDs

KineticJS uses classes and IDs to identify objects on the canvas. This allows HIKOMSYS, with the help of Javascript, to select all the dependencies by their ID and save these connections within the Mongo database.

As we saw in Section 3.2, the user is able to change between two different modes: moving and drawing. As soon as the user switches into the drawing mode, he is no longer able to drag and move packages around. Instead, if he starts dragging a package, an arrow connects to the mouse pointer and the package, following it as long as the mouse button is pressed. If the mouse is released over another package the arrow is detached from the mouse and attached to the given package.

Figure 3.22: Start drawing with drag and drop

This feature is implemented with the same logic used for the moving packages. As long as the mouse button is pressed, the mouse basically acts as a package that is dragged around. After releasing the mouse over another package, KineticJS hands us the ID (`name`) of the package and an arrow from the starting to the ending package is drawn. The arrow gets an ID as well[19]. This identifies the dependencies to store them in the database later.

[19] Package1.name_Package2.name

4
The Validation

Within this chapter we discuss two case studies. One study was done with a single person and the other with a group of 23 students and the help of multiple facilitators. The main focus of those case studies was to analyze how users react to the gamification of dependencies within their system. We also wanted to see how users interact with HIKOMSYS and to determine if there are any major usability issues.

4.1 Quantitative Case Study

A case study has been done with 23 software engineering students at the University of Bern. Most of them were in the third semester of their bachelor's degree at that time. They were all taking part in the same lecture, during which they were asked to develop a Java application within smaller groups. All the projects are derived from the same skeleton project (ESE-Skeleton on Github[1]). For the quiz, we decided to test all the students on the skeleton rather than on their current implementation of the project. The skeleton consists of a basic implementation of a web application using the Java framework Spring MVC[2]. It contains several classes (sample controller, model, service, exception, pojos and a data access object), implemented by Andrea Caracciolo[3], to help the students speed up their learning process. As a result, the students only had to identify

[1] `https://Github.com/ese-unibe-ch/ese2014-wiki/tree/master/Skeleton`

[2] `http://projects.spring.io/spring-framework/`

[3] PhD Student at the SCG Group, University of Bern `http://scg.unibe.ch/staff/Caracciolo`

the dependencies between six packages defined by the author before the experiment.

Before the actual experiment, we gave a brief introduction to the general concept of dependencies between modules. Then each student was asked to take an actual quiz. We decided to not let them choose the packages they are interested in but rather tell them which packages they had to select. Our goal was to have a number of quizzes we could compare based on the knowledge about the dependencies rather than focusing on the selection of packages by the users. The experimenters tried not to interfere with the process after this point. They only answered quiz related questions, like how to delete a dependency or how to draw one, when those were asked by the quiz takers. Afterwards the experimenters discussed what they noticed during the study.

Observations During the experiment, we made the following observations:

- Overall, most of the students really liked how they got to see their system and even wondered if they could do this for other projects.
- The thinking process of some students was accurate and some of them even explained in detail how the different classes interact with each other.
- One of the students drew every single dependency in the wrong direction. As a conclusion, it would be helpful to provide more information about how dependencies are represented within HIKOMSYS.
- Some students wondered about the points they received. We plan to add an explanation about the ranking system used in HIKOMSYS.
- We could clearly see which students had worked with their system and which had not. This conclusion was reached based on the number of points those students earned and how they interacted with the system and the experimenters. The students even tried to explain some of the dependencies while drawing them because we required them to think out loud. Some students went as far as remembering the exact method call from one package to another.

4.1.1 Collected Data

In this Section, we present the data collected during those 23 experiments. Out of a total of 8 correct and 22 wrong dependencies, the students had 50% correct and only 14% wrong dependencies on average. The accumulated points ranged from 40.34% to 95.45%.

Figure 4.1 shows the quizzes of the 23 students, ordered by their result (rounded down). We see that the more a student knew about his system, meaning the more correct dependencies he drew, the fewer dependencies he assumed wrong. Because most

students only had between 1 and 3 wrong dependencies, we think that they had a general understanding about how dependencies work. As we saw in Section 3.3.2 the amount of missing dependencies does not influence the final score, this is clearly reflected in the diagram. The more a student knew (correct dependencies), the less he forgot.

In Figure 4.1 the first bar presents a student that reached 40%, he told us after the quiz, that he assumed the arrows to work the other way around. Considering that, he would have had 1 wrong and 7 correct dependencies, scoring 91%, which would be above average.

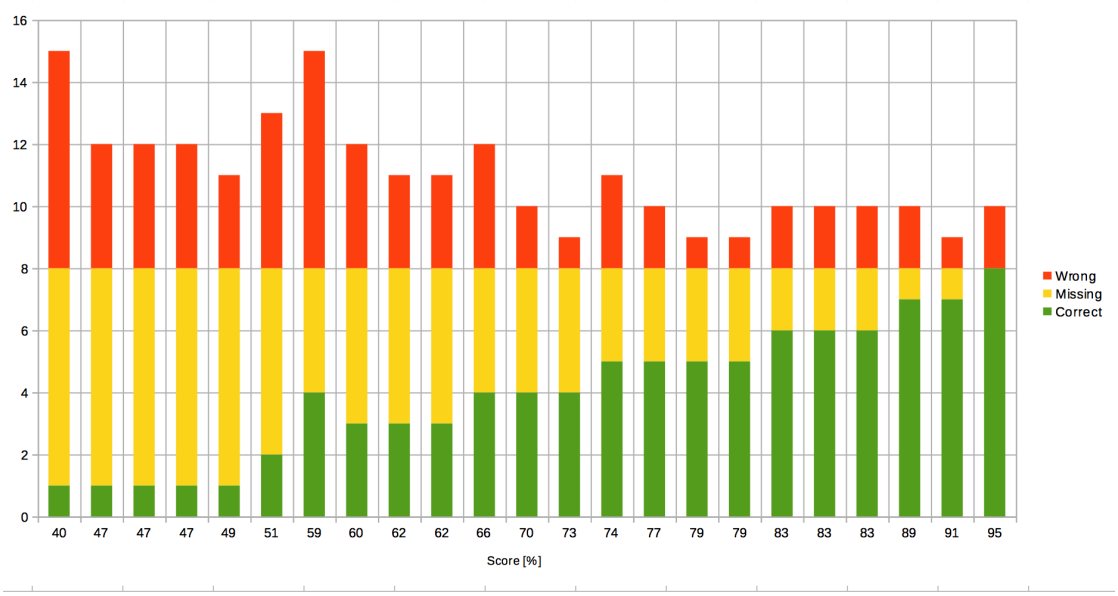


Figure 4.1: Chart: 23 quizzes ordered by their points (ascending)

The following two diagrams (Figure 4.2 and 4.3) show the distribution of correct or wrong dependencies between the 23 quizzes. In the first chart (Figure 4.2) we can see that only one person had 100% of the correct dependencies and nobody had 0% correct dependencies. Furthermore, the graph displays a typical Gaussian distribution, with most people having around 50% - 62.50% and fewer people having extremely good or bad scores. There is only one exception to that: five people had only 12.50% of the correct dependencies. That could either be due to the usability flaws stated above or because they did not invest much time into the group project.

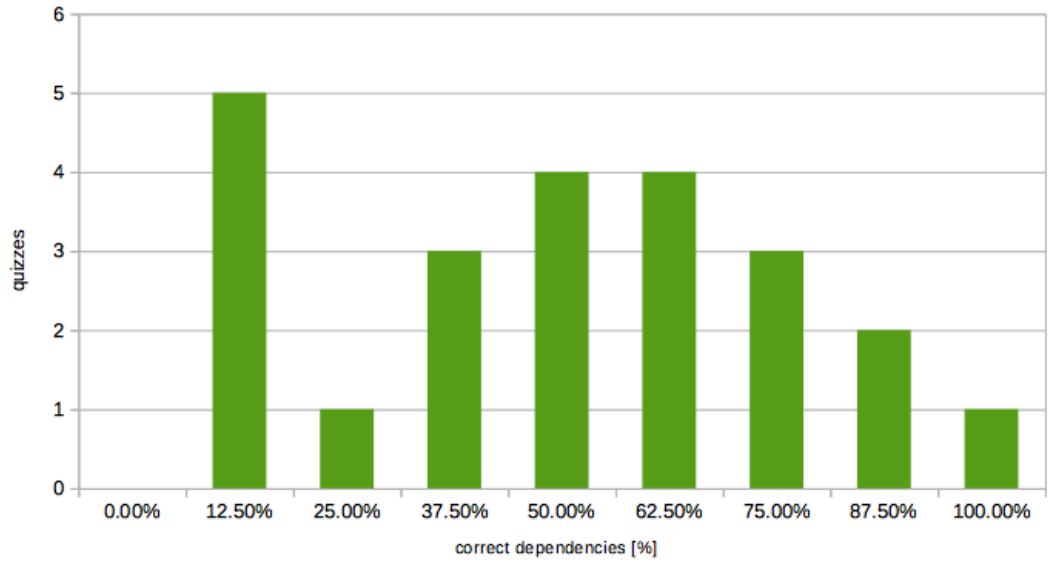


Figure 4.2: Chart: Distribution of correct dependencies among the quizzes

Looking at the second chart (Figure 4.3) we can see that every student made at least one mistake. Luckily most people only had between 4.55% and 18.18% wrong. Because nobody had more than 31.82% wrong, we can assume that nobody randomly guessed all the dependencies.

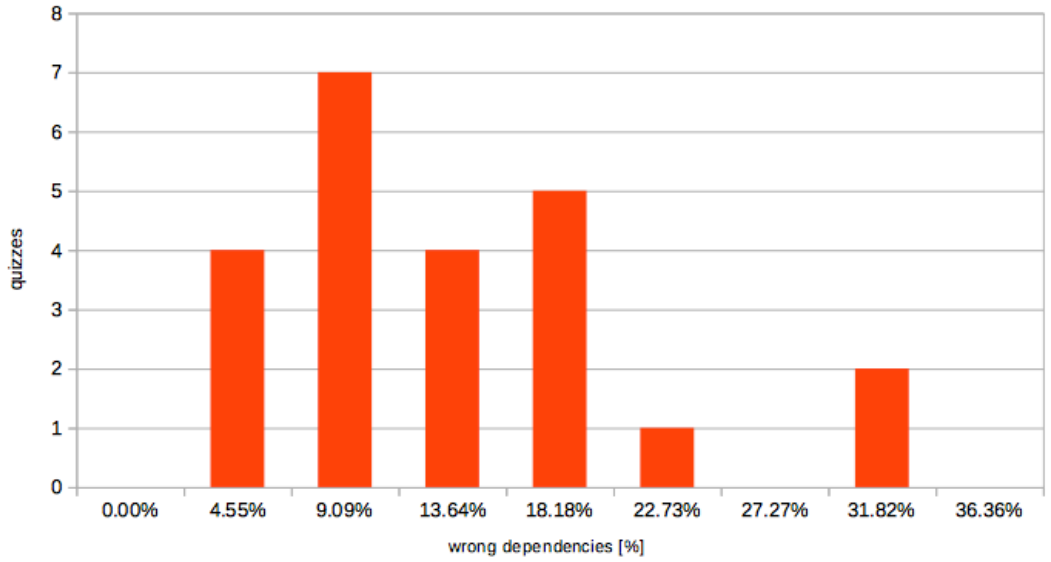


Figure 4.3: Chart: Distribution of wrong dependencies among the quizzes

As a conclusion, even though we only tested HIKOMSYS on a relatively small group, we still got a bell curve. Moving on, it would be interesting to see how those points are spread out for larger groups, for example a development team in a big company. The main challenge is to find a sufficiently large number of developers working on the same project, willing to participate in a similar experiment.

4.2 Qualitative Case Study

Michael Single, a master student at the University of Bern, tested HIKOMSYS using his bachelors thesis project[4] as input. In each individual task he tried to explain what he did and why. Selecting the important packages of his system went smoothly and Michael clearly knew how his system was organized and what the important parts were. Only the project hierarchy was a bit confusing because he expected it to be less flat. This problem did not originate from the way VerveineJ parses a Java project as he realized later. The current version on Github was built with Maven which changes the project structure while building. In the quiz itself, Michael first asked how the arrows have to be interpreted. It was not clear in which way they should point. After a brief explanation he quickly drew some dependencies while always explaining precisely how the dependency is implemented within his system. At some point Michael tried to delete a dependency he had already drawn but could not figure out how to do so without help. After going through each package twice he moved on and submitted his results, scoring 72%. At the end, while looking at the results, he was wondering about each missing dependency and wanted to check if those were correct. After explaining the "Additional Information" feature, Michael saw how those packages are connected and even verified it within his Github repository.

As a conclusion, Michael is a very good example of a developer with a good knowledge of his system. On one hand, each and every dependency he drew (with one exception) was correct and he even explained them perfectly. This makes sense given the fact that this was his own project, which he built single-handedly from scratch. On the other hand, he did forget almost as many dependencies as he knew (5). As a result, we can assume that if a developer is testing his own system, it is easier to forget some dependencies than to draw wrong ones.

Most detected design and usability flaws have already been reported by the participants of the first case study. More information needs to be provided to the user for him to understand how to delete dependencies. Possibly some examples of dependencies existing within a system could also help to eliminate initial doubts.

[4]Diffraction Shaders: `https://Github.com/simplay/Bachelor-Thesis`

5
Future Work

We built a platform for users to learn about dependencies, with the help of gamification, while having fun and competing against others. HIKOMSYS is a good starting point for a platform which might change how we learn about code in general and there are many more things we could add to improve it:

Gamification

First of all, the gamification aspect of HIKOMSYS could be improved. For example, the selection of the modules is only based on the current user's point of view at that time. A user could take two different quizzes, one based on the selection he made and another based on the packages most commonly selected by the users who took the quiz before him. This would lead to a better understanding of what exactly the core modules of a system are. Furthermore, new levels could be added, where for example HIKOMSYS tells the user how to improve his system, letting him re-upload it at a later point. If the user did improve the system in the given way or in another way, which he would have to explain, he would get points. Letting the user remove dependency cycles, for example, would be a good start for this new level.

Social Aspects

Not only new levels could be added to HIKOMSYS, but also the UMS could be linked to Github or other social media accounts. The former would make it possible to link to Github and provide even more information about the individual packages, for example

with a link to the actual implementation. This would allow users to click on a given dependency or package and gain direct access to the source code on Github. The latter lets the user share his success on different social medias and inspire other people to try out HIKOMSYS as well.

Another major improvement for HIKOMSYS would be some kind of status and leveling system where users can earn experience points, levels and badges and as a result unlock additional features or permissions. As we saw in Chapter 2, a good example for this system is StackOverflow, where only experienced users can comment on answers and where one gains medals for answering questions within a given topic. StackOverflow clearly shows that "status drives much of our actions, and it forms a critical part of how we understand ourselves in a context and relation to others"[4]

Back end

As observed during the case studies, the current point system HIKOMSYS uses, could be improved to be more transparent. Users should at least have some hints about how the points are computed.

Furthermore, there might be a better way to calculate the points a player receives and alternative scoring strategies should be explored in the future.

Getting rid of one of the two different databases has always been something desirable for the project. This is due to the fact that some data is duplicated. A project, for example, exists within both systems. On one hand, it contains all dependencies, packages and more in the Mongo databases. On the other hand, a project containing project-ID, user-ID, SHA etc. is saved in MySQL. At the beginning, opting for a Mongo Database was a reasonable choice because the MySQL library for Pharo has not been updated for almost a year whereas the library for Mongo was up to date. Later on MySQL was needed because of Laravel and the UMS. We think it would be possible to get rid of MongoDB and only use MySQL. As a result, a part of the Smalltalk and the Laravel implementation would have to be changed.

User-Interaction

Last but certainly not least, some of the usability flaws shown within the case studies would have to be fixed. On one hand, a small tutorial could be added in which the basic idea of dependencies is explained. Therefore the user would be guided through a sample quiz to learn what dependencies are and how to take a quiz. As a result the user could earn some free experience points. On the other hand, more help could be provided within a quiz by explaining the different user actions and his task.

As we saw, HIKOMSYS is only the first milestone. Adding levels and experience as well as linking projects to their Github implementation, are a few features which could and should be added to improve the platform. Furthermore, HIKOMSYS could be used

as a solid foundation for future bachelor projects or master theses and of course for any gamification or e-learning project.

6 Conclusion

Within this Chapter, I would like to talk about what we learned working on this project and also summarize the main goal of HIKOMSYS and our results.

6.1 Lessons Learned

I learned a lot while working on my biggest project so far. Starting from scratch, I had to think about different technologies and their interaction with each other. Not only based on a single point of view but on two aspects: Front end (design, functionality and usability) and back end. For example: How was it possible to let a user fill out a form and pass that data from the front end to Pharo. As a result, it has been challenging and interesting to figure out all the different interactions, while sometimes taking a step back or starting all over again with a different approach.

On the subject of technologies, I also realized that it can be difficult from time to time to work with Open Source projects if you do not intend to fix their bugs yourself. Getting in contact with the people responsible for VerveineJ was not possible. We had to add some dependencies for our case study manually because VerveineJ had some difficulties recognizing exceptions and some other dependencies like Enums.

One of the most important things I learned, was keeping track of a project's backlog. We started working on HIKOMSYS with some ideas and those became more concrete over time. Furthermore, every time somebody had an idea or some sort of improvement, it was added to the backlog. Although I worked iteratively and with the help of a backlog, which both are useful development techniques, having set a clear goal would have been

helpful and motivating. Moreover, it has been very satisfactory increasing my experience by having a look at new frameworks like Laravel and KineticJS, as well as at other programming languages like Pharo.

All in all, I really liked working on this project and with my supervisors and I hope this project is picked up by someone and will be taken good care of.

6.2 Final Words

In summary, we saw how the HIKOMSYS back end handles project uploads with the help of different tools like Moose and VerveineJ. Moreover, we had a look at how the front end is responsible for the actual user interaction: From displaying packages within a treeview to drawing on a canvas with the help of KineticJS and vector operations.

Chapter 5 shows that this project is far more than just quizzes about dependencies. HIKOMSYS is only the first step to build a web platform where users are able to improve their knowledge about their systems while also gathering a lot of useful data for further research. We have also shown that our users liked the gamification aspect of HIKOMSYS and that they were interested in their systems' dependencies.

Bibliography

[1] *GAS '12: Proceedings of the Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques*, Piscataway, NJ, USA, 2012. IEEE Press. IEEE Catalog Number: CFP1290S-ART.

[2] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20(4):18–28, October 1995.

[3] Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Educational software engineering: Where software engineering, education, and gaming meet. In *Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change*, GAS '13, pages 36–39, Piscataway, NJ, USA, 2013. IEEE Press.

[4] Gabe Zichermann and Christopher Cunningham. *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps*. O'Reilly Media, Inc., 1st edition, 2011.

Appendices

A

Appendix A: DataGatherer

Data Gatherer
Pharo Data Gatherer for HIKOMSYS

Implementation

Supplementary documentation to the Bachelors thesis

Dominique Rahm
from
Bern BE, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

July 2015

Prof. Dr. Oscar Nierstrasz
Andrea Caracciolo

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Within this documentation we have a look at the implementation of the *DataGatherer* used for How I KnOw My SYStem (HIKOMSYS). The *DataGatherer* is written in Pharo and is used to import a model file as well as to parse and to save all necessary model data into a Mongo database.

We will see how Pharo can be called with additional arguments from the terminal or a bash script. A brief introduction into Quicksilver[1] is given and it is shown how the data from a model file is prepared to be used by a quiz in HIKOMSYS.

[1] `http://scg.unibe.ch/research/quicksilver`

Contents

1 Accessing and Executing Pharo's DataGatherer

In this chapter, we will explain how Pharo can be used from the terminal and how it is possible to pass along arguments, all this using HIKOMSYS as example.

1.1 Idea

HIKOMSYS utilizes the *DataGatherer* to create a Mongo database document for any uploaded project. Therefore, we need a way to start Pharo and the *DataGatherer* after creating the model file with VerveineJ. Additionally, the *DataGatherer* is told which file should be used to create the necessary data.

There is a more detailed description about this specific topic called "Zero Configuration Scripts and Command-Line Handlers"[1] in the "Pharo by Example II" book. With the help of this tutorial, we were able to build a *CommandLineHandler* for HIKOMSYS.

[1] `http://pharobooks.gforge.inria.fr/PharoByExampleTwo-Eng/latest/ZeroConf.pdf`

1.2 Implementation

According to the "PharoByExample" book, we have to create a new class (*DataGathererCommandLineHandler*) with *CommandLineHandler* as its superclass:

```
1 CommandLineHandler subclass: #DataGathererCommandLineHandler
```

In the next step, we have to define a method called *activate*. The *activate* method will be executed, as soon as we call *DataGatherer* from Bash or the command line.

```
1 activate
2   self evaluateArguments.
3   (self hasOption: 'save')
4     ifTrue: [Smalltalk snapshot: true andQuit: true]
5     ifFalse: [self quit]
```

As we can see on the third line, *self hasOption* is how we access the arguments passed along, in this case *–save*. The other arguments are handled in the *evaluateArguments* method:

```
1 evaluateArguments
2   | t1 |
3   t1 := DataGatherer new.
4   self stdout nextPutAll: 'starting DataGatherer'.
5   t1
6     systemName: (self optionAt: 'projectName').
7   t1 run
```

In this step, we create a new instance of the *DataGatherer* (line 3) and print a comment on the terminal (line 4). Furthermore, we set the *systemName* to be equal to the *–projectName* argument. Last but not least, we call *run* on our *DataGatherer* to start processing the model data as we will see in Chapter 2.

1.2.1 Using DataGatherer from the Command Line

Now, HIKOMSYS can run Pharo and the *CommandLineHandler* with the help of the following three tools:

1. Virtual Machine[2]
2. Pharo Image
3. Command Line Handler

With those three tools, a bash script with *name* as its argument is executed to start Pharo in so called headless-mode:

```
cog="pharo-vm-nox"
headless=""

moose="pathToPharoImage"

smalltalk="runDataGatherer --projectName="$name

$cog $headless $moose $smalltalk
```

[2]Download at `http://pharo.org/download`

2 DataGatherer

In this chapter, we will introduce the *DataGatherer* implemented in Pharo and Moose. To do so, we will have a brief look at Quicksilver.

2.1 Implementation of DataGatherer

As seen in 1.2, the *CommandLineHandler* will call the *DataGatherer* run method:

```
1  run
2    self assert: systemName ~= ''.
3    self createModel.
4    self createHG.
5    packages := Dictionary new.
6    (hg level: 0)
7      do: [:t3 | self createPackages: t3].
8    self propagate.
9    self createNamespace.
10   self createMongoDB
```

Listing 1: The backbone of the DataGatherer

This is the backbone of the *DataGatherer* and before we go into more detail we will have a brief overlook at what this method does. First, we make sure that the system *name* provided earlier is not an empty string. The second step is to load the model file into Moose and create a new hash map named *packages* to store our data in. Afterwards, we are able to use Quicksilver and create a "HighGraph" from the imported model. The

next step is to create and then propagate the packages. Then we create the so called "Namespaces" and in the last step the data is saved into a new Mongo database document.

Model

The first step is to load a new model into Moose, to do so we need to provide a file stream to read the model file created with VerveineJ and a *name*:

```
1  createModel
2    | t1 t2 |
3    ^ (MooseModel root entityNamed: systemName)
4      ifNil: [t1 := MooseModel new.
5        t2 := MultiByteFileStream
6          newFrom: (FileStream readOnlyFileNamed:
7            pathToModelFile).
8        t1 importFromMSEStream: t2.
9        t1 name: systemName.
10       t1 size > 0
11         ifTrue: [^ t1 install]]
```

For this example, the path for the *FileStream* (line 6 and 7) was replaced with "pathToModelFile". This path is specific for the server HIKOMSYS runs on. The model files and the directory it is saved in are named *systemName* and the path would look something like this:

```
1 hikomsys/public/gitRepos/', systemName , '/' , systemName , '.mse'
```

HIKOMSYS uses Quicksilver's "HighGraphs" to prepare the data of a model file.

2.1.1 Quicksilver & HighGraphs

Quicksilver is a tool to explore large data sets. Quicksilver uses hierarchical graphs[1], also called "HighGraphs", to store data in nodes and leafs. Those nodes and leafs are connected with relationships which are propagated up in the hierarchy.

For HIKOMSYS we need to know the dependencies between Java packages (may be nested). As a result, we use Quicksilver's "HighGraphs" to build a hierarchical graph from which we can get the data we are interested in later. The following code example shows how a "HighGraphs" is created within Pharo:

[1] http://scg.unibe.ch/research/quicksilver/HierarchicalGraph?_s=ph4kS4W_mdgNAnv9&_k=42iOctPV&_n&17

```
1  createHG
2    | t1 |
3    t1 := MooseModel root entityStorage at: systemName.
4    hg := MalHierarchicalGraph with: t1
5      allMethods asOrderedCollection name: systemName.
6    hg bottomUp: {FAMIXNamedEntity -> #belongsTo}.
7    hg
8      edges: t1 allSureInvocations
9      from: #from
10     to: [:t2 | t2 to first].
11   hg propagateEdges.
12   ^ hg
```

The graph contains all methods and is created from the bottom up. This means we start at a leaf and from there on we search for all its parents until we reach the root. As we can see in line 8, the edges are the relations between nodes. They are a collection of all the dependencies from and to the current node found by Moose. With *sureInvocations* we make sure that we only get those dependencies which have only one candidate. One candidate means there is only one possible class it belongs to.

In the next step, we iterate through the "HighGraph" we received previously. With the data stored within the graph we create *packages*. A *package* is a dictionary or hash map with the following keys:

- a name
- a parent package
- a list of classes
- a list of dependencies

This is done with the following recursive method *createPackages*:

```
1  createPackages: aMalHgNode
2    | t1 |
3    t1 := helper stripNamespace: aMalHgNode model asString.
4    (t1 = '' or: t1 = '<Default Package>')
5      ifFalse: [packages
6          at: t1
7          ifAbsent: [| t4 |
8            t4 := DGPackage new.
9            t4 name: t1;
10             parentPackage:
11               (helper stripNamespace:
12                 aMalHgNode model parent asString);
13             classes: aMalHgNode classes;
14             addDependency: aMalHgNode outgoing.
15           packages at: t1 put: t4]].
16   aMalHgNode children
17     do: [:t3 | t3 model class = FAMIXNamespace
18         ifTrue: [self createPackages: t3]]
```

First, we check if it is empty or a so called "<Default Package>". The latter is the default name Moose gives to packages which cannot be matched to any given package due to a parsing error or something else. A quiz from HIKOMSYS does not take those packages into account because we would have to explain to the user why some packages are named this way. Secondly, we add all the parameters to the collection and in line 15 we save the package in the collection (*packages*) created earlier. The last step is to check if the current node has any children and if it has we send them the message *createPackages*.

2.1.2 Propagating Dependencies

If we think of a Java project we have a nested directory hierarchy and to provide some more feedback on the results page in HIKOMSYS we wanted to provide a way to inspect the classes within any given package, not just the leaf packages. To do so, we had to add the children of a subpackage to all its parent packages as you can see in the code example 2:

```
propagateChildren
  packages
    do: [:t1 | packages
        at: t1 parentPackage
        ifPresent: [:this | this children: t1 name]]
```

Listing 2: Adding child classes to each parent package

Besides that, in HIKOMSYS any dependency of a package is automatically also a dependency of all its parent packages. For example: let *core::util* have a dependency to *java*, as a result *core* also has a dependency to *java*. This makes sense because if a user selects *core* as one of the packages he is interested in, he should also know that one of its subpackages has a dependency to *java*.

This is achieved with the help of an recursive method (see Listing 3). Starting with the root packages (any package without a parent package) we check if it has any children. If there are, we call *propagateDependencies* on all of the children. If no child packages are present, we return the current packages dependencies adding it to the caller's dependencies.

```
propagateDependencies: package
  Transcript show: package name;
     cr.
  package children isEmpty
    ifTrue: [^ package outgoingDependencies]
    ifFalse: [package children
        do: [:t1 | package outgoingDependencies
            addAll: (self
                propagateDependencies: (packages at: t1))].
      ^ package outgoingDependencies]
```

Listing 3: Propagating outgoing dependencies

As we can see in the *run* method (see 1) the following method is called from there. The *propagate* method calls the two previously discussed methods.

```
1 propagate
2   self propagateChildren.
3   packages
4     do: [:t1 | packages
5         at: t1 parentPackage
6         ifAbsent: [self propagateDependencies: t1]]
```

2.1.3 Namespaces

We figured out that there are some cases where we are interested only in the dependencies from the classes directly within a package and not in all the dependencies of its subpackages. Therefore, we had to search for packages containing classes and subpackages, duplicate those and give them a new, distinguishable name. We gave them the same name the Eclipse IDE[2] uses, which is a * sign at the end of the name, for example "org::model::*". In the following method you can see how this is done. We iterate over all the packages found in line 4, copy those without the children and then only add the dependencies from those classes without propagating the dependencies from the subpackages. As a result, we have a package containing only the dependencies of the classes directly inside this package without taking its subpackages into account.

```
1  createNamespace
2    | t1 t2 |
3    t1 := MooseModel root entityNamed: systemName.
4    t2 := t1 allNamespaces
5          select: [:ns | ns children size > 0
6              and: [ns classes size > 0]].
7    t2
8      do: [:namespace |
9        | name classes classNames package |
10       package := DGPackage new.
11       classNames := Set new.
12       name := helper stripNamespace: namespace asString.
13       package name: name , '::*';
14          parentPackage: name.
15       classes := namespace classes.
16       classes
17         do: [:class |
18           | invocations dependencies |
19           dependencies := OrderedCollection new.
20           classNames add: class name.
```

[2] https://eclipse.org/

```
21          invocations := class queryAllOutgoingInvocations.
22          invocations
23            do: [:invocation |
24              | dep |
25              dep := self createDependencyFrom: name to: invocation.
26              dep class = DGDependency
27                ifTrue: [dependencies add: dep]].
28          package outgoingDependencies: dependencies].
29      package classes: classNames.
30      packages at: name , '::*' put: package.
31      (packages at: name)
32        children: name , '::*']
```

The figure 2.1 shows the *controller* package of the ESE-Skeleton on Github[3]. As you can see, there are three subpackages and one class *IndexController.java*. Therefore, HIKOMSYS would create a new package called *controller::** which would contain only the *IndexController* class and its dependencies.

Figure 2.1: ESE-Skeleton org::controller with subpackages and class

[3] https://Github.com/ese-unibe-ch/ese2014-wiki/tree/master/Skeleton

2.2 Mongo Database

The next step, after parsing and preparing the data, is to export it into a Mongo database document. This is done with the help of the "Mongo class" provided by Pharo. As we will see in the following code snippet, Mongo needs to listen to a server and a port, here "localost" and port 27017. Then we create a new collection with the name of our project as its name (see line 7). The last step is to iterate over every package created previously. In this process we convert each package into a dictionary and add those to the Mongo database collection. A Mongo document is nothing else then a binary JSON file with key and value pairs.

```
1  createMongoDB
2    | t1 t2 t3 |
3    mongoDB := Mongo host: 'localhost' port: 27017.
4    mongoDB isOpen
5      ifFalse: [mongoDB open].
6    t1 := mongoDB databaseNamed: 'hikomsys'.
7    t1 addCollection: systemName.
8    t2 := t1 collectionAt: systemName.
9    packages
10     do: [:t5 |
11       t3 := t5 asDictionary.
12       t2 add: t3].
13   mongoDB close.
```

3
Final words

We saw how Pharo can be accessed from the terminal and how the *DataGatherer* receives, prepares and stores data from a model file.

I would like to point out, that we started to write test for the *DataGatherer* but had to stop because we had to focus on the other parts of HIKOMSYS. So this would be something which could be done in the future. Furthermore, as pointed out in the thesis itself it would be good to switch from using a Mongo database to using MySQL.