DoodleDebug

Niko Schwarz
University of Bern

Cedric Reichenbach
University of Bern

Oscar Nierstrasz
University of Bern

ABSTRACT

Software developers actively make use of debugging tools in order to find problem sources in code. In relation to a previous paper[1], we designed and implemented a tool representing a new style of debugging: Objects are responsible of their own visual representation, as already seen in classical tools, but with a powerful and yet simple mechanism for graphical visualization rather than text only. Most of its power is gained when taking on conceptually multidimensional objects like lists of matrices.

1. INTRODUCTION

Read and quote all publications assiciated with whyline: http://faculty.washington.edu/ajko/whyline-java.shtml Read: "http://www.amazon.com/Write-Point-Bill-Stott/dp/0231075499"

1.1 Idea

Debugging tools should save time, they help visualizing objects for inspection and thus bug detection. Java's `System.out.println()` allows developers to easily compare many temporal states of an object or many different instances beside each other, which may be a reason for its popularity;On one of the first pages of "programming pearls", the author notes that often it's easier to write "just writing println is often easier". That would fit here. but when it comes to complex problems, like a list of matrices, `System.out.println()` breaks down and becomes either expensive to use or completely useless[1]. A Debugger may handle such a problem better, but it does not allow users to compare two time slices. DoodleDebug combines graphical object representation with a simple API, still approving the simple and widely accepted usage pattern of `System.out.println()`.

For programmers, it's an everyday problem: Some feature of a program is not working correctly, but it's not clear where the problem originating from. The simplest approach, just staring at code and build some mental model to play the program in one's head, is usually time and concentration intensive. For more comfort, tools help to automatically build a visual representation of runtime data, which is provided by debuggers for instance. They allow users to stop at a particular moment in time, the whole program state is inspectable and commonly used data types have some useful textual representation. One big problem here is that a program is only inspectable at one snapshot in time, but two states cannot directly be compared. Excellent. Really good sentence. Maybe move this sentence further up. It's kind of key to the whole thing. But we need it here...

Another widely used solution, printing on a console, eliminates this issue and has other advantages: Because such printing instructions are directly written into code to debug, which might be quicker or more convenient in some cases Restructure. If you're showing these things later, you can say that. You can say: as our study results show, ... refer to study, where some subjects didn't use any debugger. Also, printing to a console allows developers to precisely represent an object with needed data only using its `toString()` method (in Java). Does this make sense now? But still it lacks some features: On the one hand, the only supported output format is one-dimensional text, no colors, images or free spacial positioning. On the other hand, a line that has been printed cannot be edited any more, therefore advanced formatting becomes very complicated. Convinced about downsides? Fighting alternatives should go into the "related work" section. Not the introduction. But yea, it's a lot better. But do move it to related work. This paper thesis:) I'm confused now: I was told to write a paper and derive the bachelor's thesis out of it later... introduces a solution which aims to eliminate the above mentioned disadvantages and includes a small qualitative study for its verification.

Please do read "write to the point", linked above. Among many other things, it explains quite nicely how to break text into paragraphs.

The above comment is still good. Are there any paragraphs left in the paper?

2. RELATED WORK

What further related work is worth mentioning?

Finding the source of problems in program code is an omnipresent issue in computer science and therefore a recurrent research theme. Previous studies about human behaviour when developing and especially debugging code were good

hints to point where DoodleDebug should be going. We filtered relevant data out of them and aligned them with our research efforts, resulting in a separate paper.

2.1 Previous Paper

In a previous paper[1], we analysed generally used patterns for textual representation of open source projects and showed basic conceptual issues preventing useful representations of certain data types. Following that, we outlined a concept automatically including information mostly used in custom textual representations and supporting "difficult" data types that prevent powerful renderings in textual form.

2.1.1 Mining SqueakSource

We had programmed a simple Smalltalk data miner, which fetched all projects from SqueakSource, searched all classes overriding Smalltalk's `printOn:` method, which is roughly equivalent to Java's `toString()` in its meaning to programmers; namely it provides a textual representation of an object in order to print it to a console. In a next step, we manually went through the received `printOn:` code snippets (590) and searched for repeating patterns.

- 44 used ASCII graphics to build two-dimensional structures.
- 137 wrapped their printed instance variables using parentheses.
- 219 called `printOn:` of their super class.
- 452 included their class name.

For DoodleDebug, we tried to consider those patterns and include them as well as possible and eventually include them in every output without the need of user manipulation. If not, they should be easily produced through an API.

2.1.2 Text is not always powerful enough

In the same paper, we argued that there are data structures, where using classical console printing contains clear disadvantages, demonstrated using Arrays of two-dimensional matrices as an example of multi-dimensional structures. Objects with independent textual representation methods can only be listed vertically, which does not allow nesting with a consistent and thus clear layout (figure 1). This nesting problem could be solved, if representation methods would not be restricted to one-dimensional text only (figure 2).

2.2 Whyline for Java

Whyline[6] is a Java library that should "answer why and why not questions"[5] about a program's behaviour. In particular, it allows its users to click on a piece of either graphical or textual output and ask questions about the cause of its properties, like fields. Whyline then tracks back the program execution to the line of code where this particular field has been changed for the last time and visually reports it to the user, including the causal chain leading to it.

```
an Array()

an Array(
MatrixTransform2x3(
    2.0 0.0 0.0
    0.0 2.0 0.0
))

an Array(
MatrixTransform2x3(
    2.0 0.0 0.0
    0.0 2.0 0.0
) MatrixTransform2x3(
    0.707107 -0.707107 0.0
    0.707107 0.707107 0.0
))
```

Figure 1: Textual visualization of an array of 2D-matrices in Smalltalk

$$\{\}$$

$$\{\begin{pmatrix} 2. & 0. & 0. \\ 0. & 2. & 0. \\ 0. & 0. & 2. \end{pmatrix}\}$$

$$\{\begin{pmatrix} 2. & 0. & 0. \\ 0. & 2. & 0. \\ 0. & 0. & 2. \end{pmatrix}, \begin{pmatrix} 0.707107 & -0.707107 & 0. \\ 0.707107 & 0.707107 & 0. \\ 0. & 0. & 1. \end{pmatrix}\}$$

Figure 2: Possible visualization of an array of 2D-matrices without the restriction to 1D-text

2.2.1 Difference to DoodleDebug

Whyline aims on connecting properties of output to code snippets. In other words, it answers the "why" question. DoodleDebug visualizes objects and helps to understand what properties objects have and how they evolve over time. It answers the "what" question. By now, Whyline only supports output on `java.awt.Graphics` and `java.io.PrintStream` objects and is developed with closed source only, but from a conceptual point of view, those two technologies could be combined and eventually enhance each other.

3. DEVELOPMENT

The efforts made in analysing classical debugging tools and how programmers behave when interacting with them all aimed on implementing which would actually fill the mentioned problematic gaps. This chapter describes the process of realizing it in the form of an eclipse plugin, including general conceptual patterns as well as technically relevant parts.

3.1 Planning

Instead of immediately starting to implement, we followed several well-proven planning steps from literature to avoid costly redesigning or refactoring later on. Following the principle of "not mixing up the design and engineering processes" seemed to be a good way to avoid losing the initial path[2].

3.1.1 Sketching the Features

As summarized in the Related Work chapter, we had shown that textual output is lacking features in certain mentioned cases on the one hand, and that custom textual object representations tend to exhibit a couple of frequently observable traits. Too biological? Based on this information, we started to gather commonly used data types and sketch them to paper with a simple, meaningful appearance. Beside every sketched candidate rendering, we tried to think of a simple-as-possible but yet modular code snippet, that would imaginary render this image. Over time, while painting more and more candidates, we tried to converge our imaginary API, so we threw away outliers and repainted their object type in a more consistent way. When we had a consistent (still imaginary) API, we would take a step forward and consult third-party programmers for some feedback.

3.1.2 Feedback For Hand-Made Sketches

As soon as we had a consistent imaginary API and enough sample sketches, we stuck them to a wall and eventually asked other programmers to quickly look at those drawings and explain what they believe to see (figure 3). If they immediately anticipated the virtual situation, it was a good example and we kept it. If most people failed to understand what a particular drawing meant, or even wrong conclusions were made out of it, we either threw it away or tried to redesign, based on people's statements and then loop and ask them others about it. Loop back to the book.

Figure 3: Programmers should look at our code snippets and corresponding design sketches for us to evaluate their intuitiveness

3.2 Implementation Details

There should be text under every heading. What kind of text? Well, an introduction paragraph. As a random example, see Katja Schwarz's thesis from 3 weeks ago. Note how below every heading, there's never immediately another heading, but always text.

After we thought to have gathered enough information for a simple API mapping to useful renderings, we were ready to head on selecting appropriate technologies for putting DoodleDebug into practice. Write more

3.2.1 Selecting Technology

The decision of technology is not a question of right and wrong, but it's about weighting factors between different technologies and maximizing the benefit. We thought about implementing DoodleDebug in Smalltalk. This would have had the advantage of simplicity because of the strong consistency of Smalltalk's overall structure. Also, building dynamic output would have been easy, as in Smalltalk, the environment runs the same technology as programs running inside.

Java and Eclipse. Java is one of the most popular programming languages and therefore contains many active software developers which could benefit from our work. After the decision to implement DoodleDebug for Java, we discussed if we should implement it as an independent library, which would make it independent in terms of environment, but the advantages of building it as an Eclipse plugin seemed to be bigger. Within Java developers, Eclipse users build a big community, which again provides a lot of possible users, but also causes a broader community of plugin developers and more information resources about plugin development. Furthermore, interaction can be smoothly integrated in to a user's working environment: Installation is done by Eclipse's plugin installation wizard, library dependencies are automatically added to projects if needed, imports are done on the fly and the output screen integrates consistently with the rest of Eclipse's user interface. Because Eclipse itself is written in Java, there are no barriers for data exchange between client (user) and server (plugin/IDE) side.

Web Technologies. As an output format, we needed technology that meets some requirements:

1. Two-dimensional arrangement Objects must have at least 2 degrees of freedom
2. Consistent nesting Nesting two or more objects doesn't break the structure of the outer one (for reasonable scales)
3. Dynamic change Content can be dynamically manipulated in order to allow user interaction

Based on these properties, we decided on HTML. It can be integrated in Eclipse, which uses the OS' built-in browser. CSS technology allows to tune design and arrangement of output separately from the rendering process and adds an additional layer of abstraction. Dynamic manipulation is done by Javascript, which is pretty straight forward thanks to additional libraries. In addition to above mentioned advantages, HTML is wide-spread and allowed us to add an additional layer of API where users can provide plugins to DoodleDebug, which define HTML representation of selected object types.

3.2.2 Communication between Java Virtual Machines

DoodleDebug renders its output into a view tab inside eclipse, so we implemented it to have the main part running in the same VM as eclipse for convenience. Since runtime information about objects and rendering calls must be provided

from the user's VM to the eclipse VM, a solid communication mechanism is needed. We use SIMON (Simple Invocation of Methods Over Network)[7], a library for object-oriented remote communication in Java. It allows to create a registry on a specified port of localhost and add a Server object to it. Clients can find the server through this registry and send messages to it by calling it's methods with simple objects like Strings as arguments.

3.2.3 Providing Code from Eclipse Plugin to Workspace

What Is Provided to the User Workspace. On the one hand, all API methods must be visible from user projects. On the other hand, users should only see a minimal amount of DoodleDebug's code in order to prevent them from using it in an unintended way and cause bad behaviour. The compromise made between those two requirements was to mainly provide simple, well-documented interfaces (such as `Doodleable` and only show a proxy of fully implemented classes (e.g. `Doo`).

3.2.4 Serialization for the Transport between VMs

SIMON only allows transporting simple data types such as Strings, so every object is serialized before its transportation. For this purpose, we make use of XStream[8], a simple serializing library, originally created to serialize Java objects to XML and back again. Because of it's modularity, it also allows to serialize to JSON and comes with a built-in mapping for this. Because of XML's more verbose syntax and therefore bigger space/time consumption in many cases, we decided to use JSON as first choice, with a fallback to XML if errors occur. This is necessary because standard JSON cannot handle circular references, it lacks the ability to append attributes to entities and therefore cannot index them in order to reference to a parent id in case of a circular reference. Obviously, there are workarounds for this issue, but the fallback to XML does not take as much time that users could even recognize it is happening.

Maybe simple benchmark of XML vs. JSON here

3.2.5 Communication between Java and JavaScript

From Java to JavaScript. DoodleDebug renders its output into a dedicated tab inside the Eclipse UI, using Eclipse's `Browser` class. To append newly rendered objects to the output screen, a naive approach would be to just cache the current code on Java side and in the case of freshly added object renderings, just repaint the whole html page. This has some disadvantages: A refreshed page may always jump back to the top, and even if it does not or it is avoided by jumping back down with JavaScript, it would flicker anyway for the split of a second. Thanks to `Browser`'s method `execute(String script)`, there is a smarter way to update: On Java side, the object's html code is escaped in a way to not collide with some JavaScript properties. Then, it's wrapped into a JavaScript method `addCode(code)`, which simply appends the html code in its argument to the document body.

From JavaScript to Java. Every rendering of Java objects to html is done in Java, so the output entity can only operate as a thin client. In order to make output interoperable, communication from JavaScript to Java is necessary, e.g. if an object is inspected and it's nested objects are not yet rendered. Because the output is rendered as html into a Browser, it is of course encapsulated from it's environment, which is a good thing for general portability of web site, but in this special case, it was a drawback. There is no such thing like a JavaScript-Method like "`sendToBrowser(message)`", so we had to find some workaround to notify the Java instance about occurring events (e.g. lightbox closing) on one hand and be able to provide some arguments (e.g. which object to render) on the other hand. Using AJAX calls on localhost would probably cause some tedious delays, break encapsulation and just be bad style. The solution we finally came up with causes no remarkable delays and stays inside its dedicated context: Eclipse's `Browser` class allows to append event listeners for the case that its `window.location` is about to change. We defined a pseudo-protocol `doodledebug` and append some message to it, maintaining syntax limitation such as no white space. On Java side, a listener handles all location change events; if an event's target location fits the pseudo-protocol's pattern, its "message" is parsed and desired steps executed. The location change itself is cancelled, so the user stays on the same page. For instance, if an object is clicked and should be inspected inside a lightbox, this item's id is determined and the window location set to `doodledebug:<id>`. On Java side, the object to render is determined from this id, rendered and a message with html code sent back to JavaScript again.

4. USER INTERFACE

text

4.1 API

Where should this part go?

DoodleDebug's API can be divided into three parts:

1. The `Doo.dle(object)` method, which uses the same pattern of uses as `System.out.prinln(object)`
2. The `Doodleable` interface, which can be implemented by users to define two methods for simple and quick custom object representations
3. `RenderingPlugin`s as plugins for DoodleDebug, which enable custom renderings of any type without changing its source code

Two different customization APIs represent different implementing costs as well as different patterns. They are not meant to compete, but to be specialized for different use cases.

4.1.1 The Doodleable Interface

In our previous paper, we had proposed a patter where objects should contain information about how to represent them. During the planning phase, we continuously improved a initially rough prototype of such an API until it converged to a form that met our requirements of intuitiveness and

simplicity. For adding rendering information to an object, its class must implement the `Doodleable` interface. `Doodleable` contains two methods, `doodleOn(DoodleCanvas c)` and `SummarizeOn(DoodleCanvas c)` in order to support semantic zoom by having two representations with different levels of detail. The pattern behind the `DoodleCanvas` is a virtual canvas with a virtual cursor behaving similarly as those in common text editors. A user basically has three possibilities of interaction with the canvas:

1. Draw an object to the place where the (virtual) cursor is right now
2. Switch to a new line
3. Switch to a new column

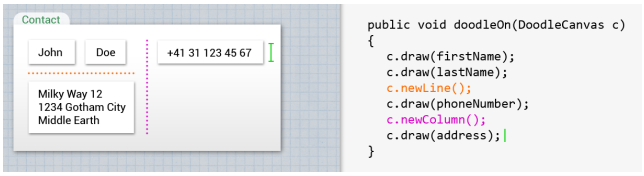

Figure 4: Example of a `Contact` class' `doodleOn()` method. Dotted lines visualize the effect of the corresponding structuring methods and the green i-beam indicates the final position of an imaginary cursor

Can I make this wider?

4.1.2 Providing Plugins to DoodleDebug

write this

4.2 Output

text

4.2.1 Output format and rendering engine

For displaying and handling HTML code and related web technologies, Eclipse provides the package `org.eclipse.swt.brows` which among others includes a `Browser` class to render and integrate into the Eclipse UI. However, this `Browser` class is not completely platform independent, it uses the default browser rendering engine of its current host operating system (and not the standard browser). When running on Windows for instance, Internet Explorer's rendering engine is used, even though Firefox is set as the System's standard browser. This fact forced us to be even more careful with the usage of HTML5 features, because their support varies between Browsers.

4.2.2 Semantic Zoom

In order to save space and keep information available to users, DoodleDebug uses the concept of "Semantic Zoom"[4]. Yup, a reference would be nice. Make sure to read the referenced paper, too. In part, because you need to get a feeling for scientific style. Semantic Zoom is from a book, right? See the DoodleDebug workshop paper. It, too, quotes Semantic Zoom. This is a design thing, and should go into the design chapter. Implementation things should go into the implementation chapter. Split differently. Currently, there's a chapter "programming" Make the new structure: 1. Introduction 2. Related work 3. Design (this one should include *[illegible]* sketches. Both accepted and denied sketches.) 4. Implementation 5. Study 6. Discussion 7. Future work (this one is optional. The others aren't.) 8. Conclusions and then never mix them. I think I see... - The sections "Planning" and "Output" would both mainly go into "Design", right? - Section "Implementation Details" will become a chapter plus some parts from "User Interface" Object visualizations are divided in levels of nesting, where level 0 represents outermost objects, referenced in `Doo.dle(object)`, level 1 objects are (semantically) nested ones inside level 0 etcetera. Saving space is achieved by only completely rendering level 0 and 1 objects, and use a smaller representation for a objects of level 2. Level 1 objects are clickable, which will cause them to be repainted as new virtual level 0 objects, so previous level 2 objects will move to level 1. This pattern allows to arbitrarily explore an objects nesting tree, similar to a debugger.

Illustrate. This will probably take several sketches.

Something like that?

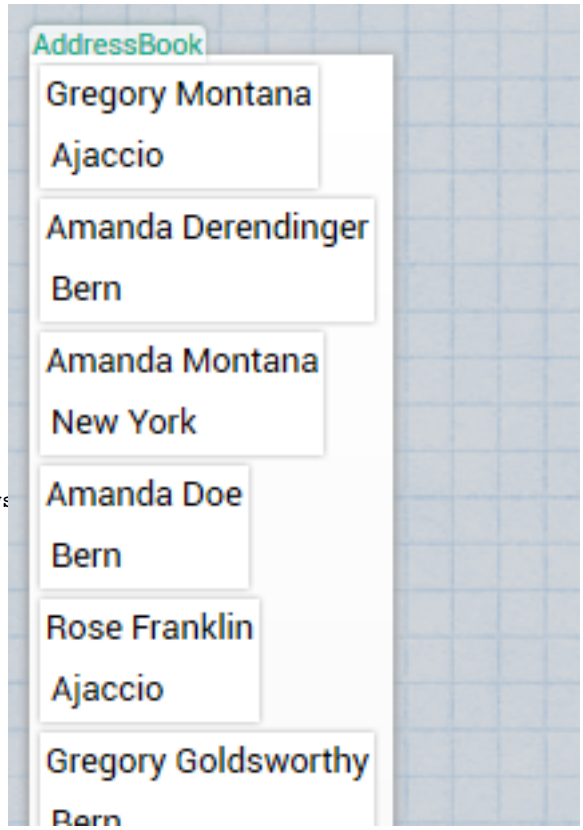

Figure 5: The representation of an AddressBook instance lists its contacts with few information.

4.2.3 Smart Behaviour versus Configuration Hell

When designing user interfaces, one basic decision must be taken: How much configuration options should it contain? Either a lot of settings are left to the user let them take part on the design process. Or the default design is made to fit everyone as good as possible. We decided not to treat

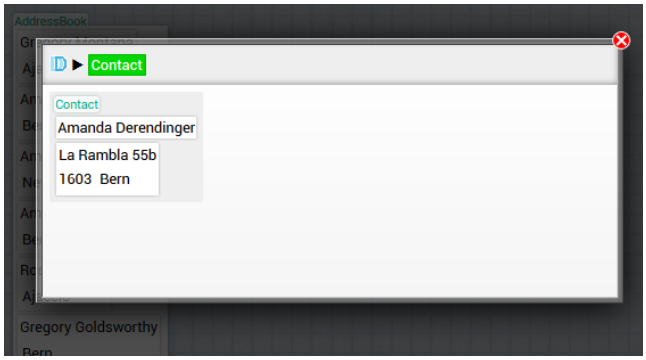

Figure 6: When clicking on an element in the list, the selected contact is visualized with more detail and clickable inner objects.

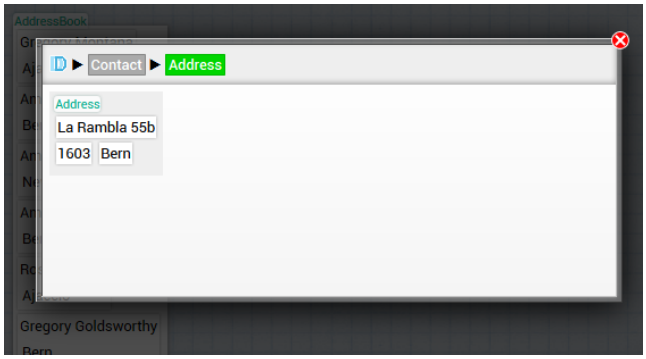

Figure 7: Clicking on the address area inside contact opens a more detailed view of this `Address` object. Since its content only consists of `String` and primitive objects, there is no further detail revealed.

everyone as designer[2]p. 95+[3]p. 155 but take away design decisions from them by creating sophisticated defaults. As a result, there is no settings dialogue or file for DoodleDebug.

Smart Scrolling. I'd like to bet that Apple has a patent on this. Find and quote. Can't find it. Try googling something with apple and patent, and you will get a million results... Google patent search did not reveal it as well. Going to stackoverflow... One Example is the behaviour of the scroll position in the console when new objects are rendered into it. As default (with no user interaction), the window keeps scrolling along with rendered objects, always jumping to the newest one. As soon as the user scrolls away from the bottom, this mechanism is stopped and the scroll position remains equal while new objects are appended on the bottom of the output silently. However, if the user decides to go to the bottom of the output again, the auto-scrolling mechanism is obtained again.

Focus When Likely Desired. Similar thoughts were made on focus handling of DoodleDebug's view in the space of Eclipse's whole UI. Eclipse's built-in console receives focus on any output event per default, but provides a button to deactivate this. For a programmer, it's likely desirable to generally be notified if something happened, but it can be annoying for many outputs over a long time, e.g. if they're working inside another view of Eclipse. In this case, they would probably want to be notified of sporadically occurring outputs. In our study, we observed this problem as well, one subject even turned off console notifications (see 5.4.3). Based on these thoughts, we defined an algorithm to check time since the last output event and only gain focus if the previous event is longer than 4 seconds ago. For example some loop of a game which prints out it's calculation time for the last frame every time will only gain focus on the DoodleDebug view once, but a quiet application like a web server that only generates update when a user visits a site will usually regain focus on such an event. refer to design book here And read ... I've never seen such a technique described before, do you know something?

5. QUALITATIVE STUDY ON BETA VERSION

On how to write up a user study like this, see "http://research.microsoft.com/en-us/um/redmond/groups/hip/papers/ko2007bugfixing.pdf"

Write the introduction from the point of view of qualitative psychological experiments. Read and reference "Introduction to Research Methods and Data Analysis in Psychology"

As soon as we had a stable prototype of DoodleDebug, we made a small qualitative Study in order to determine how users behave when they use DoodleDebug in comparison to their behaviour using classical debugging tools. The study should consider problems considered to occur frequently in similar forms as well as special cases, where the choice of debugging tools may be more crucial than in others.

5.1 Study Session Setup

A fully functional version of DoodleDebug was used, most probably the equivalent to a "release candidate". It was run inside Eclipse 4.2 (Classic edition), using a ThinkPad T410 with Windows 7 (x64) and an additional mouse (2 buttons + wheel, standard size). The screen was captured during the whole session and one instructor sitting beside the test subject for problem explanation and protocol. Before the actual testing, the user had 15 - 30 minutes to work through a tutorial and play around with DoodleDebug inside a sandbox. At this time, the instructor was allowed to answer questions and support the subject.
For the actual session, there were 3 different small programs containing some manually inserted bug, which they had to find and eliminate. For one or two of them, they were allowed to use DoodleDebug and for the other one or two respectively, they had to fall back to classical tools. The permission to use DoodleDebug on a particular problem changed with every study session, i.e. if subject 1 was allowed to use DoodleDebug on problem A, then subject 2 would not be allowed, but subject 3 would. The reason for letting some candidates only use classical debugging tools was to have a reference of behaviour in order to show that they are not trivial and detect what particular sub-problems they pose in detail, so we would see if DoodleDebug enables better approaches to solve them. Obviously we could not let people solve the same problem in both modes, or they would have

been prejudiced by the solution they found before.
A subject was always working on a problem until it was completely solved, none of them needed more than 30 minutes for all problems together.

5.2 Posed Problems

Sorting. A couple of grey scale `Color` objects are put into a `List` and then sorted using a custom `ColorComparator`, which should sort by brightness. The result then is compared to a hand-built `List` which initially has the expected order. This test fails and it's the subject's task to find out what is ordered wrong, i.e. if there is a clear pattern, and to fix this bug. Subjects have access to all of the source code and are allowed to manipulate it.
Solution: In the comparator, completely black colors are wrongly treated as complete white.

Serialization. Phone book contacts are modeled using `Contact` and `Address` objects. They should be serialized using a `SerializingUtil` (simulated serialization only) and de-serialized afterwards. Those mechanisms are executed with example data, but comparison of a contact object before and after serialization fails. The subject's task here is to find out what parts of the contact object were broken and why, i.e. fix the bug. Subjects have access to all of the source code and are allowed to manipulate it.
Solution: In the `SerializingUtil`, every field of type `long` is casted into an `integer` before serialization and back into a `long` afterwards. This causes a field called `phoneNumber` of `Address` to be changed into some negative value.

Decimal Alignment. A class `DatabaseUtil` is a black box simulating access to an imaginary database by returning a two-dimensional array of `float` when calling it's only method `getData()`. Subject know that in the returned table, there are duplicated tuples and have to name them. Because they have no access to source code, they need to rely on received data only and also cannot fix the bug.
Solution: There are two pairs of fitting rows (3 & 8, 6 & 9).

5.3 Candidates

To respect privacy, the real names of our test subjects have been replaced by character names of the usual Radio Spelling Alphabet, enumerated in order of their participation.

5.3.1 Education and Experience of Each Candidate

Alpha

- B.Sc. in Mathematics, Minor Computer Science 60 ECTS
- Master Student in Computer Science

Bravo

- B.Sc. in Computer Science
- Master Student in Computer Science

Charlie

- M.Sc. in Computer Science
- Ph.D. Student in Computer Science

Delta

- B.Sc. in Computer Science
- Master Student in Computer Science

Echo

- M.Sc. in Computer Science
- Working as Software Engineer, 1 year of experience
- Minor experience in Eclipse plugin development (master thesis)

Foxtrot

- B.Sc. in Computer Science
- Master Student in Computer Science

Golf

- Lic.rer.pol. in Economics, Minor Computer Science 60 ECTS
- Working as Software Engineer, 15 years of experience

5.3.2 Different Problem Approaches

Depending on study session with our candidates, we could observe different patterns of approaching a problem with classical tools.

System.out.println(). 5 out of 7 subjects (all except Delta and Echo) made use of this mechanism to visualize runtime data. It's quick and Alpha for example argued with laziness to open a debugger or to stare at foreign code. Also, they can compare things, either two different objects as posed in the sorting problem or the same object at different points in time, as in the Serialization problem. Both is not directly possible with a classical debugger like the one coming built-in with Eclipse classic.

Debugger. Four subjects (Bravo, Delta, Echo and Foxtrott) used the eclipse debugger to inspect objects, only Delta and Echo used it exclusively. The argumentation for this usage was that debuggers are more powerful in comparison to System.out.println(), because they allow to inspect objects dynamically and additionally provide simple improvements of standard textual representations (e.g. arrays are represented in the form of `[objectA, objectB, ...]` instead of `[Ljava.lang.Object;@4cb162d5`. But Echo also missed the feature to compare two objects, even at the same point in time they could not manage to do so.

Source Code Staring. As mentioned before, debugging tools were mainly developed to avoid the need of staring at code, and most people found this the most annoying part, especially because it was code they had not written on their own. Nevertheless, subject Golf solved the Serialization almost only by using this method. They tried to comprehend the logic of the problem's `SerializingUtil` and thus, in contrast to others, found the problem source at the same time as the semantic problem itself. To be exact, they located where the problem was (long casted to int) and used System.out.println() only to check what it resulted in.

5.4 Problems With Classical Tools

5.4.1 Using System.out.println()

Homogenous Output. We previously stated that purely textual output is poor in terms of formatting and therefore makes it harder for users to classify different parts as they always look similar in terms of size, color, alignment etc. When solving the sorting problem, subjects were slowed down due to this fact. Beta for instance firstly iterated over the wrongly sorted color list to print each element and then stared at the (unaligned) numerical values of red, green and blue color components. After they had found out that a black element was at the end instead of the beginning, they could go on searching what had caused the problem. Subjects using DoodleDebug already had built-in renderings for `Collection` and `Color`, which enabled them to visualize it by only using one call. Charlie Doodled the wrongly sorted color list and the correct one that was used as ground truth for comparison. They instantly noticed their similarity and pointed out that black is on the wrong side of the list.

Uninspectable and Useless Output. The standard implementation of `System.out.println()` prints class name and object hash for non-primitive objects. Output printed to a console is static and can not be inspected. If an object's representation lacks a particular piece of information, the programmer will need to go back into the code and either change the `toString()` method of its class or manually gather information from outside and print it. In our study, Charlie was the only subject to override `toString()` methods after firstly using their standard representation. They implemented it by printing out all fields of an object: `Contact` was represented by

```
"name: " + name + ", address: " + address,
```

whereat `Address` was represented by

```
"street: " + street + ", phoneNumber: " + phoneNumber + ", city: " + city
```

This representation recursively breaks down a `Contact` object into primitive types, which can be easily represented by text. Alpha produced a very similar output, but instead of overriding `toString()`, they extracted all fields from outside using getter methods directly inside the System.out.println() method. Golf only used System.out.println() as help for code inspection, they printed one particular primitive field at a time without considering its containing object.
Another approach to solve insufficient output was to switch from System.out.println() to the debugger, observed on Bravo and Foxtrot.
DoodleDebug includes an objects fields in it's standard rendering (if there are not too many) while still labelling its class name. All 3 subjects (Bravo, Delta, Foxtrot) using DoodleDebug for the serialization problem managed to find the changed field instantly after calling `Doo.dle()` once before and once after the de-/serialization step.

5.4.2 Using A Debugger

Comparison Between Objects. The built-in eclipse debugger only allows to inspect one object at one point in time. As the serialization problem consists of two objects unexpectedly being unequal, part of the debugging process was somehow comparing them in order to find their difference. Every subject except Golf did this, Golf only tried to understand the serialization and de-serialization process to find out, where the implementation has mistakes. Echo never used `System.out.println()` so they also attempted to compare objects before and after serialization using the debugger. Even though the debugger supported simple and fast inspection to any point inside the object, Echo explicitly pointed out that they miss the feature to inspect two objects simultaneously instead of memorizing small pieces and going to the other one for comparison.

Non-Selective Output. Eclipse's Java debugger simply lists all contents of an object, since there is no information about relevance of its respective parts. In particular, interfaces may specify a imaginary concept and be used in in the declaration context for better overview, but for creation of an instance, a full class implementing this interface is needed. The debugger is working at runtime and therefore only knows the instantiated type of an object, so it will visualize all properties and contents of this, potentially resulting in a overly verbose output. We observed this on the color list problem, where `ArrayList` was used as implementation for the interface `List`. Subject Bravo initially tried to perceive the structure of a list right after sorting by pausing the program using the debugger and inspecting the mentioned list (figure 8), but instantly gave up and switched to writing a `for` loop which sequentially prints out all elements using `System.out.println()`. If the standard output of DoodlDebug does not fit a users needs, it can be adjusted through of the two described methods, the simpler Doodleable interface was used in this context by Alpha. Screenshot of Doodleable usage

Are there more examples? (Good ones)

5.4.3 General Problems

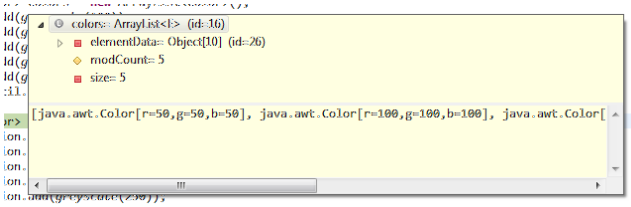

Figure 8: Eclipse's debugger visualizes all fields of an object's runtime type (ArrayList)

Console Keeps Stealing Focus. When some output is printed onto the console, it gains the UI's focus by default. Due to the problem setup, every program initially used to throw an exception at the end of it's execution, signalling the problem has not been solved yet. Every user experienced the following problem at least once: They were using DoodleDebug and therefore had this view tab opened when the exception was thrown and eclipse switched to the console. Only Echo managed to disable its focus-on-change setting, the other subjects just switched back to the DoodleDebug view tab after a few seconds.

6. REFERENCES

[1] Niko Schwarz. *DoodleDebug, Objects Should Sketch Themselves For Code Understanding.* `http://scg.unibe.ch/archive/papers/Schw11bDoodleDebug.pdf`

[2] Bill Buxton. *Sketching User Experiences.* Elsevier, 2007.

[3] Donald A. Norman. *The Design of Everyday Things.* The MIT Press, 1998.

[4] Allison Woodruff, James Landay, and Michael Stonebraker. *Goal-directed zoom.* In CHI 98 conference summary on Human factors in computing systems, CHI '98, pages 305-306, New York, NY, USA, 1998. ACM.

[5] Ko, A.J. and Myers, B.A. *Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior.* `http://faculty.washington.edu/ajko/papers/Ko2008JavaWhyline.pdf`

[6] Whyline: `http://faculty.washington.edu/ajko/whyline-java.shtml`

[7] SIMON: `http://dev.root1.de/projects/simon`

[8] XStream: `http://xstream.codehaus.org`

7. FUTURE WORK

Besides proving things right, our study revealed some problems as well. As they only show that a problem exists but don't provide exact information about its cause, further research might invest in analysing them and propose accurate solutions.

7.1 API

Text

7.1.1 Split The Doodleable Interface

Problem. As already described, the `Doodleable` interface contains two methods for object representation, one for a detailed and another one for a summarized variant. This structure greatly allows integration of semantic zoom because it forces user-provided information about content importance. However, study subjects eventually were unsure about the semantic meaning of those methods. Echo, for instance, verbally communicated his confusion about their difference; after going back to the documentation, everything was clear. Alpha did not show any special reaction when implementing the Doodleable interface for the first time, but simply "delegated" the zoom mechanism by replacing `c.draw(object)` from their `doodleOn(Canvas c)` by `c.drawSmall(object)` in their `doodleSimplifiedOn(Canvas c)`.

Why is this future work?! Just solve it, no? It's not proven. Would this REALLY be better?

Possible Solution. An approach for resolving this issue would be to split `Doodleable` into two interfaces containing one single method each, let's say `Doodleable` and `SmallDoodleable`. For support of semantic zoom, users could simply implement both interfaces; if not, they would likely only use the standard version method. People implementing both interfaces would possibly make sure to first inform themselves of the respective semantic meanings and not run into previously mentioned slips. A user study comparing two groups, one using a single interface like before and another one using the proposed setup, could help to determine if it's an actual improvement. Anyhow, this solution would introduce new conceptual problems: What should happen if a user only implements `SmallDoodleable`? Should DoodleDebug try to guess a way for semantic zoom of objects only implementing one interface or just ignore it and always use the same method for rendering?

7.2 User Interface

Text

7.2.1 Include Variable Name in Visualization

Problem. When people are debugging parts of their code and use object visualizations, it is obviously essential for them to map a visualized element to its corresponding object in code for simple use cases, the object's class name is already good hint and widely used for custom textual representations as well[**?**], but it may become unclear if one is working with many different instances of the same class. Echo, who was only using the debugger beside DoodleDebug, stated it would be practical to see which object has which name in order to directly associate them with code snippets.

Possible Solution. The very first step in this issue should be to clarify if it's technically possible to gather information about variable names during runtime. If so, the next question would be which context is relevant, since objects can be referenced to through different names from different contexts; probably the context where `Doo.dle(object)` was called would be the only reasonable choice. For objects with no names in this context, e.g. inline objects, a meaningful replacement would need to be found to avoid user confusion.

A good solution for UI integration might be achieved by pen and paper work; draw some sketches, show it to programmers and ask about their intuition.

7.2.2 Link Visualization to Source Code

Problem. Mapping a visualized entity to its triggering line (`System.out.prinln(object)` or `Doo.dle(object)`) in code is not a problem if this line has just been written. It might become one if either if it has been written a long time ago or by another programmer, such that the current user does not know where the call is originating. Class name and content of an object may not always be helpful to find it, so full text search must be used to find the right code sequence. This can be costly as well if there is a big amount of code and/or a big amount of other calls for object visualization. As the projects in our study were only small and used one per user, we couldn't prove the mentioned problem as a fact, so a small study or survey could reveal how prevalent it actually is.

Possible Solution. By utilizing Java's stack trace, the call origin could be determined and integrated in the rendering. As it is an Eclipse plugin, some mechanism could be created to directly create a clickable link to the corresponding source code line of a visualized object, similarly to the mechanism used for printed stack traces of Throwables. A good UI integration could again be achieved with pen and paper interviews.

7.2.3 Simultaneous Usage of Multiple Eclipse Instances

Problem. The communication between DoodleDebug's client (API) and server (plugin) side happens through a Socket on a special port of localhost without any further identification between the two. The main problem when trying to achieve a correct mapping between running client and server instances is that clients have no safe way of gathering information about the Eclipse instance they're running inside. This is of course intended by IDE developers, as the running program should be completely independent, so working solutions might break in special cases or future IDE versions.

Possible Solution. If the client side has no ability to autonomously get information about the Eclipse instance, it has to be provided from outside. One way would be to put a file containing a unique identifier of the running eclipse instance into the client's classpath from server side. One disadvantage of this solution is that it won't work when multiple instances try to run the same program.

8. APPENDIX

8.0.4 Lack of Documentation

Probably throw this into appendix. In order to provide API functionality such as `Doo.dle()` or the `Doodleable` interface to users, we had to find some way of providing Java code automatically from our plugin to the current eclipse workspace. Don't talk about "had to find some way." In its way, this section has the least priority. It should probably be called "Implementation." Because it isn't very important, try and be brief: what were the implemtation challenges, what were the solutions? You're kind of saying that, just say it briefer. You haven't come around to this, have you? Please do. One may say that everything is well-documented in the Eclipse documentation Too prosaic. Also: doesn't really matter. This is science., Still true. Please address all comments. Potentially by answering them, rather than following them. I'm skipping this chapter, since it seems to have been addressed already. but the challenge was to find the right part in the documentation. Eclipse developer forums neither could help, but finally a hint was given by some Stackoverflow user: In order to provide code to workspace Java projects, a plugin needs to use the extension point `org.eclipse.jdt.ui.classpathContainerPage`. From then on, another big help was the source code of JUnit, which uses the same technique, otherwise, it would probably have taken several more weeks until everything was working finely. Rewrite, away from what you did, towards a problem-solution focus.

8.1 Study Sessions

For every candidate, a screen capture video was taken as well as hand notes. The main setup and action plots of each subject is documented here. Add rest of documentation

8.1.1 Alpha

The main purpose of our very first session was to test our study setup and its synthetic programming problems. At this point, we only had the first two mentioned problems, namely "Sorting" and "Serialization", the third one, "Decimal Alignment", was added afterwards. Aware of those unequal conditions, we include those results nevertheless, because this study only has a qualitative background and intended to deliver statistically relevant results.

Problem	DoodleDebug	Classical
Sorting	✓	
Serialization		✓

Table 1: Problems for Alpha to be solved using DoodleDebug

Allowed Tools

General Behaviour. Alpha is a typical System.out.println() user: Too "lazy" (according to themselves) to open a debugger and start thinking, better just print out whatever could be affected.

Successes. After creating a dummy project while learning from the Tutorial, they immediately were able to understand how to use the `Doo.dle(object)` method, even used the auto-completion template `dd` and never typed the whole method name again.

Errors. When working through the tutorial, Alpha had to face the situation where they had to use DoodleDebug's `Doodleable` interface. This interface contains two methods

for generating a visual representation supporting semantic zoom, one for its normal representation and one for reduced size. In that version of DoodleDebug, they were called `drawOn(Canvas c)` and `drawSmallOn(Canvas c)`, and the Canvas object received as parameter had two methods to draw objects onto it, also supporting semantic zoom, even though `Canvas.drawSmall(Object o)` is intended to be used only in special cases. Instead of adapting the idea of semantic zoom and reducing an objects representation in `drawSmallOn(c)` method to as few as possible parts, Alpha basically drew the same fields as in `drawOn(c)`, but used `c.drawSmall(o)`. Conceptually, this is a bad idea, because repeating this behaviour in nested objects can withdraw any size reduction when it's always delegated to inner objects. At best, this would cause all objects of the normal representation still to be rendered, but maybe smaller (depending of the innermost objects) and thus be reduced to something similar to geometrical zoom.

Discussion. They argued that it's probably too much work to implement an Interface like `Doodleable`, even if there are two methods to implement. They argued that one would suffice in most cases. From our point of view, one method is enough when a problem context only contains few information, just like in our study problems. As software grows bigger, structure will become nested deeper and deeper, which forces either very careful selection of what one want to see on their screen (which must be done before writing print statements into their code) or some technique like semantic zoom.

Consequences. After this session, a couple of API methods were renamed in order to enhance intuitiveness and semantic distinctiveness of them.

8.1.2 Bravo

Problem	DoodleDebug	Classical
Sorting		✓
Serialization	✓	
Decimal Alignment		✓

Table 2: Problems for Bravo to be solved using DoodleDebug

Allowed Tools

General Behaviour. In contrast to any other subject, Bravo used Enums during their tutorial walk-through. We had not considered any rendering for Enums, because only very few programmers actually use them with Java, so there was no meaningful output for them and we had to find and supply some visual representation for them afterwards.
For the sorting problem, Bravo started using Eclipse' debugger, but gave it up again quickly due to problems handling it technically, so they switched to System.out.println() usage.

Successes

Errors

Discussion

Consequences

8.1.3 Subject

Problem	DoodleDebug	Classical
Sorting		
Serialization		
Decimal Alignment		

Table 3: Problems for ??? to be solved using DoodleDebug

Allowed Tools

General Behaviour

Successes

Errors

Discussion

Consequences.