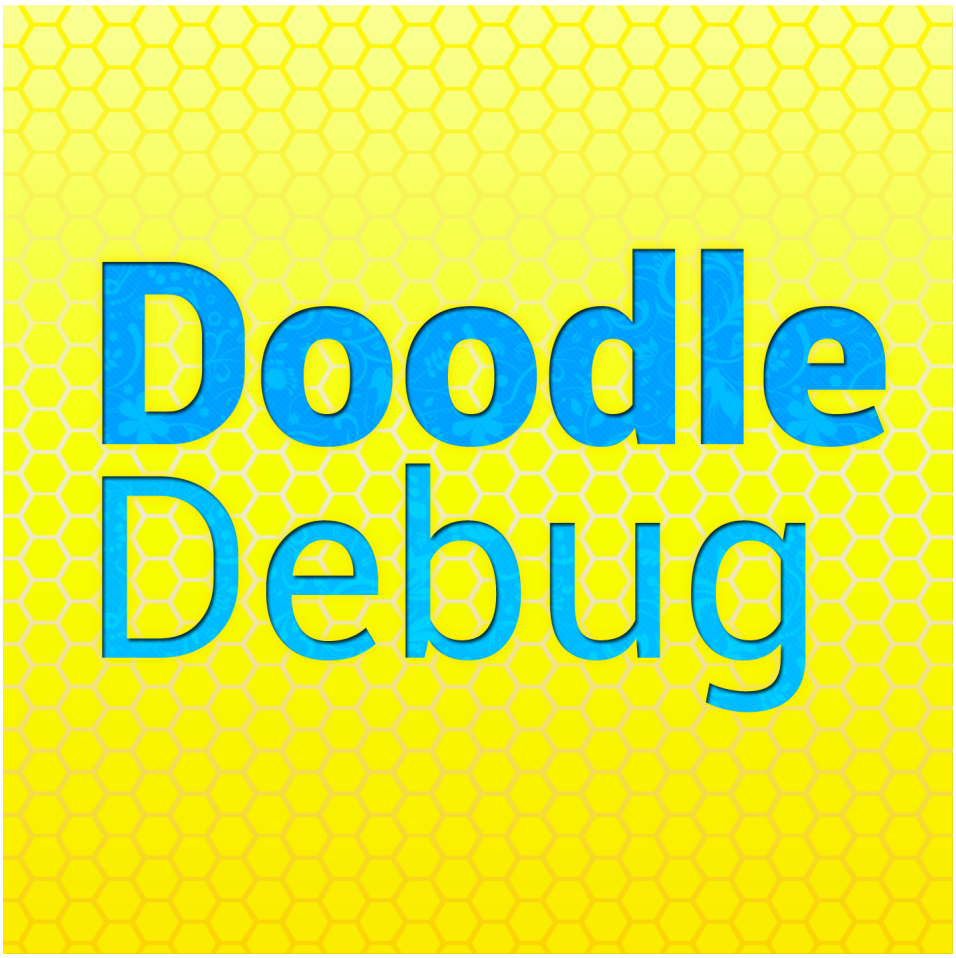
# DoodleDebug


Doodle
Debug

# Abstract

# Introduction

Please don't check the pdf or aux files (except bbl) into the git repo Read and quote all publications assiciated with whyline: http://faculty.washington.edu/ajko/whyline-java.shtml

### Idea

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. In many cases, it's not even clear what is wrong in the runtime data. Start is too slow. You have a list of interesting things to say. Think of the most interesting one, and say it. Don't prefix it. Read: "http://www.amazon.com/Write-Point-Bill-Stott/dp/0231075499" Those problems, commonly called bugs, Too prosaic. are approached by many various Many and various don't go together techniques who a technique isn't a who always have pros and cons, mostly it's a trade off between (manpower) time cost and quality of output. Why would any of that matter? That's just a long useless prefix that has nothing to do with doodledebug. The simplest approach, just staring at code and build some mental model to play the program in one's head, not only demands a lot of concentration and imagination, You're spending many lines trying to convince me that I need help fixing my bugs. I don't think that's a point worth making: everybody agrees. Focus instead on how doodledebug is better than toString. See the DD workshop paper for how to run an introduction but also consumes a huge amount of time if the affected code sample is rather big. More trivialities. In most cases, it's much more comfortable to automatically built build a visual representation of runtime data, which can is provided by debuggers for instance. They allow users to stop at a particular moment in time, specified by a previously defined line of code that is run over. The whole program state is inspectable Finally we're getting to doodledebug! This isn't nearly concrete enough, though. this way and commonly used data types have some improved textual representation. One big problem here is, No comma. English comma placement is a bit funny. Look it up. that a program is only inspectable at one snapshot in time, but two states cannot directly be compared. Now we're getting somewhere. Make this the beginning of the introduction. Another widely used solution, printing on a console, eliminates this issue and has other advantages: Because such printing instructions are directly timed into code to debug, it's much quicker to use instead of setting a break point, opening a debugger and navigating to whatever one is searching for. Nonsense. Have you used smalltalk? Opening up a debugger needn't be difficult at all. Again, look at the doodledebug workshop paper for how to introduce our problem. Don't open silly side battles. Also, printing to a console allows to save space by just printing exactly what is needed and therefore display a bigger amount of useful information using the same space on screen. That makes no sense to me. But still it lack neslacks some features, on one hand on the one hand, because the only supported output format is text, on the other hand, a line that has been printed cannot be edited any more, therefore advanced formatting becomes very complicated.

Don't use double backslash. Unconvincing. "only supports text" is a property, not a downside. Why would that be bad? Conscious of all those problems, we tried to model a new way of debugging, joining all advantages of classical tools and eliminate all discomfort Are we overselling a little bit?. We considered console printing as a good starting point because it already brings a lot of desirable properties, It doesn't matter what you consider. The goal is to convince the reader that DoodleDebug solves a problem. so our tool should be able to be called directly from code and render its output into some console-like device. To get away from article. text-only format and also to support standards as well as portability, we decided to use html as output format. A Bachelor's thesis isn't the story of how you implemented your project. It's a scientific text, that leads from a problem to a solution. It's convincing, more than documenting.

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

# Development

## Planning

**Verifying Our Idea.** Refer to DoodleDebug paper here

**Sketching the Features.** Refer to DoodleDebug paper here (probably) Before we could start with building a system, Again: the goal is not to trace back how we invented it. A thesis is like a long paper. See the DD workshop paper. Does it talk about how we started building the system? we needed to find out what information programmers mostly need for effective debugging. We built a script for data mining on SqueakSource, which mainly searches for all classes overriding Smalltalk's `printOn()` method and shows their source code. Based on review of this data, we expanded the script to numerically check our observations. One widely used pattern was that the object's textual representation contained the its own class name, so we decided to always include the class name of an object in it's rendering. Is this really better than the workshop paper? You can stay pretty close to it, here.

**Confronting People With Hand Drawings.** Please read and reference "sketching user experiences". It contains all the motivation needed to run a design phase up front. Don't try and think up the argument by yourself. It's a long book, so maybe only read the first 50 pages or so. That'll do. Instead of blindly implementing some GUI based on our assumptions, we decided to first sketch candidate renderings of commonly used data structure with pen and paper with their respective producing code and then present them to other programmers in order to see if they intuitively understand the meaning of those visual representations.

## Programming

There should be text under every heading.

**Providing Code from Eclipse Plugin to Workspace.**

*Lack of Documentation.* 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. Doesn't matter One may say that everything is well-documented in the Eclipse documentation Too prosaic. Also: doesn't really matter. This is science., 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. Blank line before sections, please.

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

*What Is Provided to the User Workspace?* No question marks in headings. On the one hand, all API methods must be visible from user projects. On the other hand, users should only see the very minimal amount of DoodleDebug's code in order to prevent them from using it in an unintended way, which could cause unexpected behaviour I'd prefer American spelling: behavior and is much less future-proof, i.e. when DoodleDebug is internally changing. Try and be briefer. 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`).

**Communication between Java Virtual Machines.** Why would this come second? This is the actual trick, the part that makes stuff work. The Eclipse plugin nonsense is accidental complexity: stuff that doesn't matter. Here's the trick, the part that makes DD work. This needs to come first. And it needs to be well-motivated. Because a user program is running in a different Virtual Machine (VM) than Eclipse itself, we had to find a way to somehow communicate between those two in order to display the DoodleDebug rendering in an Eclipse tab. As always: don't talk about how you solved it. Talk about problem and solution. Also: motivate. As a first attempt, we tried to use Java's built-in Remote Method Invocation (RMI) run-on sentence what resulted in frustration as some Java Security Manager always put obstacles in our way. Doesn't matter. What matters is: you try and keep objects kind of alive in an old state. To do that, you serialize them and ship them off to a background service. Which framework you used isn't really important. You can mention it, but not at this length. Looking for alternatives, we found SIMON (Simple Invocation of Methods Over Network)[1], a more comfortable That's a bit vague alternative, which allows to create a registry on a specified port of localhost and add a Server object to it. The Server class implements an Interface and the

client, knowing the server's Interface, can then search for this server name and send messages to it, including simple data types such as Strings. In small o Order to transport any kind of Java objects, a serialization util is used.

**Serialization for the Transport between VMs.** SIMON only allows transporting simple data types, such as Strings, so every objects grammar is serialized before its transportation. For this purpose, we make use of XStream, a fast and simple No advertisement talk serializing library, originally created to serialize Java objects to XML and back again. Because of it's modularity, it 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

**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.

# User Interface

## Output

**Output format and rendering engine.** Why talk about portability? In science, you need to write for a hostile audience that disagrees with everything that could possibly be wrong. Avoid risky sentences. Especially if they aren't really important. You're in the business of convincing the reader. You try and convince him that (a) there was a problem and that (b) you solved it. How does talk about portability help an argument? In favor of portability and simplicity, DoodleDebug uses html as its output format. For its handling, Eclipse provides the package `org.eclipse.swt.browser`, 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 a lot of Eclipse-using Java developers are running Windows.

**Semantic Zoom.** In order to save space and keep information available to users, DoodleDebug uses the concept of "Semantic Zoom"[link/reference]. 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. 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.

**Smart Behaviour versus Configuration Hell.** When designing user interfaces, most people tend to be unsure about best behaviour their UI could provide Do you have data to back that up? No? Then you can't say it., or at least think that users will know better what they want. As a result , especially open source products come bundled with a lot of configuration possibilities, which is a good thing if some power users really know what they are doing and have lots of experience with this particular software. But most user will barely ever touch them, mostly because it's too expensive to find that specific check box they are searching for. General rambling. Stay focused on DoodleDebug. If you must talk about design and simplicity, do so by staying close to the literature. Then, quote the aforementioned "sketching user experiences", and "`http://www.amazon.de/Design-Everyday-Things-Donald-Norman/dp/0465067107/ref=sr_1_1?ie=UTF8&qid=1355486165&sr=8-1`" Make sure to have read both. Don't voice your own opinion, unless you can back it up. Instead, it's mostly Why mostly? more convenient to have

a smart default behaviour fitting the vast majority of people. We made using this mindset as one of the goals for DoodleDebug's user interface, which is why there is no properties dialogue or file. No talk of goals. Talk about what was achieved.

*Smart Scrolling.* I'd like to bet that Apple has a patent on this. Find and quote. 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.* You ran user studies, right? Talking about these things without mentioning the user studies is just sad. This section should stay as close as possible to the user studies. All that can't be backed up using data from them should go away. 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. 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 ...

# Qualitative Study on Beta Version

Write the introduction from the point of view of qualitative psychological experiments. Read and reference "Introduction to Research Methods and Data Analysis in Psychology" One of DoodleDebug's big purposes No talk of purposes. is to compete with classical debugging mechanisms, so we decided to do a study with a handful of Java developers in order to proof this statement and to find bugs as well as usability problems. Nope. Reference

### 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.
No double backslash. 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.
What's the point of not letting them use doodle debug? Do you see the difference between a qualitative and a quantitative study? There's plenty studies on how developers use the currently available toolset. 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

### 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 modelled I'd prefer american spelling: 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).

## 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.

**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
is this true? Ph.D. Student in Computer Science I don't think he's got his PhD already.

*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

*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

**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().* Most Quantify. of the subjects made use of this mechanism to visualize runtime data. It's quick and most of them 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.* Few subjects used the eclipse debugger to inspect objects, only one (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 problem 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.

## Sessions

Watch videos to check text and eventually add some numbers

**Alpha.** Nope. Don't group by candidate, group by problem, and then list the candidates that support that problem. If you need to list the work log per candidate, do so in the appendix. On how to write up a user study like this, see "http://research.microsoft.com/en-us/um/redmond/groups/hip/papers/ko2007bugfixing.pdf" 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 to open a debugger and start thinking, better just print out whatever could be affected. Do you have data to back up that he's LAZY?! Just stick to what you know.

*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.

**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.*

**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.*

# References

(1) SIMON: http://dev.root1.de/projects/simon
(2) XStream: http://xstream.codehaus.org