DoodleDebug

A shot-gun marriage between
System.out.println and object inspectors

Bachelor's Thesis of
Cedric Reichenbach

mentored by
Niko Schwarz
Oscar Nierstrasz

Software Composition Group
University of Berne
2013

Developers need effective ways to inspect and explore the run-time state of programs they are developing and debugging. Modern debuggers and object inspectors are powerful tools, but they can only be used to explore specific points in the execution where breakpoints have been set. As a result, developers often resort to inserting “print statements” in code to log the state at multiple points in the execution. Print statements, however are a “poor man’s debugger”, since their output is static and cannot be further explored. We propose to combine the simplicity of print statements with the graphical sophistication and interaction of modern debugging tools. DoodleDebug is a simple API modeled loosely after Java’s `System.out.println`. Objects that are “printed” generate graphical views that can be further explored, and can also be used to navigate back to source code in the IDE. We introduce DoodleDebug and present the results of a usability study that shows that DoodleDebug can be very effective for common debugging tasks.

1 Introduction

To understand and debug a program, developers rely on tools to track its runtime states during execution. One method is to insert print statements like `System.out.println` in Java. This method is quick and allows programmers to compare different states in time of a specific object. However, this output is static and comes with a couple of conceptual restrictions. On the one hand, the level of detail is hardcoded through the textual representations of objects. If a developer decides for a simple and clear way of representation, they will need to rewrite their code for any further inspection and re-run the program after every change, which is particularly a problem in a long-running program. If they initially choose a detailed and verbose object representation, the output will grow and become tedious to read.

Another drawback of textual representation is caused by the simplicity of plain text. Its one-dimensional nature prevents the user from encapsulating multidimensional object representations. In other words, objects using line breaks in their `toString` representation cannot be nested consistently, since switching to a new line is always a final operation.

The other half of the two most widely used debugging tools is the family of debuggers [3]. When utilizing a debugger, a program can be stopped at a specific point of its execution, allowing developers to inspect any detail of this very state. A clear advantage over textual output is the ability to inspect objects on demand. Information is only displayed as soon as the user asks for it, nevertheless available without re-running the program. As McConnell states in Code Complete[6, p. 539], debuggers help to avoid bad practices caused by debugging with `System.out.println`, like scattering print statements randomly throughout a program. The drawbacks of debuggers arise from the fact that their inspector is always bound to a specific point in time and therefore makes it impossible to directly compare different states of the same object.

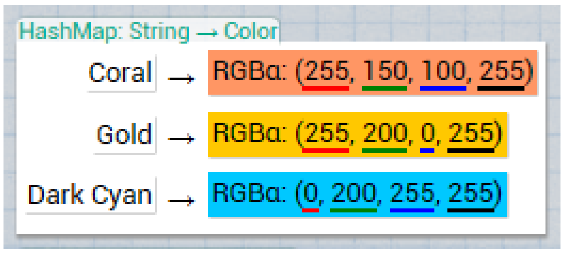
DoodleDebug combines the power of the above mentioned tools, erasing the conceptual problems coming with them. Its output is generated through an API taking its cue from `System.out.println`. A developer simply needs to call `Doo.dle` (object) to doodle any object type. Hence, DoodleDebug's usage, as well as that of `System.out.println`, is orthogonal to the control flow of debuggers; they can be used in combination. A debugger can step over one `Doo.dle` statement at a time and new doodles instantly appear. On the other hand, when a program is held still by a debugger, the user can make it evaluate any custom expression, including `Doo.dle` statements.

For simple customization of object representations, a class can implement the `Doodleable` interface which contains 2 methods, `doodleOn` and `summarizeOn`. In contrast to Java's `toString`, there are two methods, allowing developers to define representations on two levels of detail. This distinction allows inspection of objects directly in the output window.

As output format, DoodleDebug uses the web standards HTML and CSS to enhance formatting possibilities over simple text.

The functionality DoodleDebug offers consists of the following main points:

1. A log of all states of an object when it was doodled in the past
2. Zooming into one particular state in the log for more detail
3. A lean API for doodling objects


HashMap: String → Color

Coral → RGBα: (255, 150, 100, 255)

Gold → RGBα: (255, 200, 0, 255)

Dark Cyan → RGBα: (0, 200, 255, 255)

Figure 1: Doodle of a map from strings to colors with built-in renderings only.

2 Features

This section outlines the major features and properties of DoodleDebug from a user's perspective. Different parts are ordered by importance we gave them during development.

2.1 Lean API

The DoodleDebug API is lean. It features three ways for developers to interact with it, though most will only ever use the first two.

- The `Doo.dle(Object)` method as a drop-in replacement for `System.out.-println`
- The `Doodleable` interface and the associated interface `DoodleCanvas`, in which objects can define simple custom representations
- The `RenderingPlugin` interface, which allows developers to provide powerful custom renderings for any type, based on HTML and CSS; source code access is not required here.

Altogether, DoodleDebug features no more than 10 public methods.

2.2 Configuration-Free

A key question in the design of a user interface is the level of configurability exposed to users. Highly customizable solutions may be better for power users who are very familiar with the software in question. Other users may always remain on default settings, independent of their suitability. We followed the advice of Norman[7, p. 199-200] and Buxton[1, p. 102], which says not to treat everyone as a designer, but rather take away design decisions from users by creating sophisticated defaults. As a consequence, there is no settings dialogue or file for DoodleDebug.

2.2.1 Smart Scrolling

In general, an output console can either move its view port to the bottom when new content is appended or stay where it was before. DoodleDebug implements the scrolling behavior of MUSHClient[1] and mIRC[2]: The viewport will only be moved to the bottom if it already was there before new content was added. If the user doesn't scroll away from the bottom, they will benefit from notifications about new doodles. On the other hand, users can scroll up to an old doodle without being bounced away when new objects are doodled.

2.2.2 Notifications

When new objects are doodled, DoodleDebug autonomously decides whether to set focus to the DoodleDebug output view for user notification or not. The crucial factor for this decision is the elapsed time since the last doodling. Focus is gained if more than 4 seconds have passed and always for the first doodle of a program run.

When debugging a program with many output events per second, like a game, there is no sense in always notifying the user. Either they keep their attention on the output as they see it's rapidly changing, or they work somewhere else in the UI and don't want to be pulled back every time. During our study (section §5), subject Echo manually disabled such notifications of the Eclipse console, complaining about it stealing the focus of the UI part they were working in.

For a program expected to be silent in general, there's no need for suppressing eventual output notifications. One use case could be an unplanned exception that's caught, but doodled in order to inform the programmer about a possible problem.

2.3 Inspection of Doodled Objects

To avoid the trade-off between detail level and compactness, we implemented the principle of *semantic zoom* along with DoodleDebug: Every object features two levels of detail for its rendering. Objects that are nested into others are not graphically scaled down, nor removed, but rendered in their summarized, less detailed version.

Object doodles are divided into levels, based on their nesting depth. An object provided in the `Doo.dle` call has level 0. Every object directly referenced inside this one renders at level 1, those referenced from level 1 render at level 2 and so on.

In DoodleDebug, objects rendered at nesting levels 0 and 1 are rendered with full detail; objects at nesting level 2 show their summarized version. Clicking on a level 1 object opens a popup window only showing this very object, but with more detail since it's at the new nesting level 0 now. Level 1 objects in this window can again be clicked in order to inspect those. This can be repeated until any end node of the doodling graph is reached, *i.e.* one that has no references. Figure 5a shows a doodled address book (at level 0) containing several contacts (at level 1). Clicking on one of them creates a pop-up window, moving the respective contact object to new level 0 (Figure 5b).

[1] `http://www.gammon.com.au/mushclient/`
[2] `http://www.mirc.com`

Navigation between nesting layers inside a pop-up is aided with bread crumbs [4, p. 76-78], which traces the currently inspected branch of the object graph (figure 2). Any parent doodle in this trace can be clicked to jump to it directly. When zooming out of the graph this way, the just visited branch is still visible in the breadcrumbs area until the user turns into a new path.

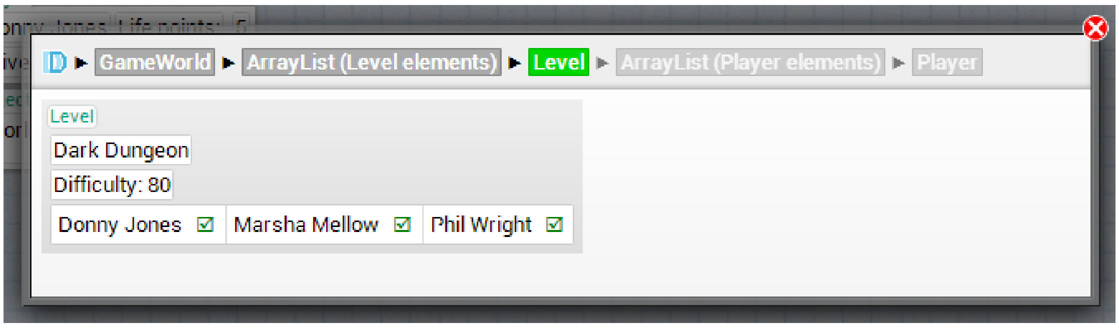


Figure 2: The labels at the top are *bread crumbs*. The developer is currently looking at a `Level` object, which is the member of an ArrayList, which is the member of a `GameWorld`. The object that was doodled was the `GameWorld` object. The developer was previously zoomed in to player but zoomed out again to Level.

2.3.1 Tracking an Object Over Time

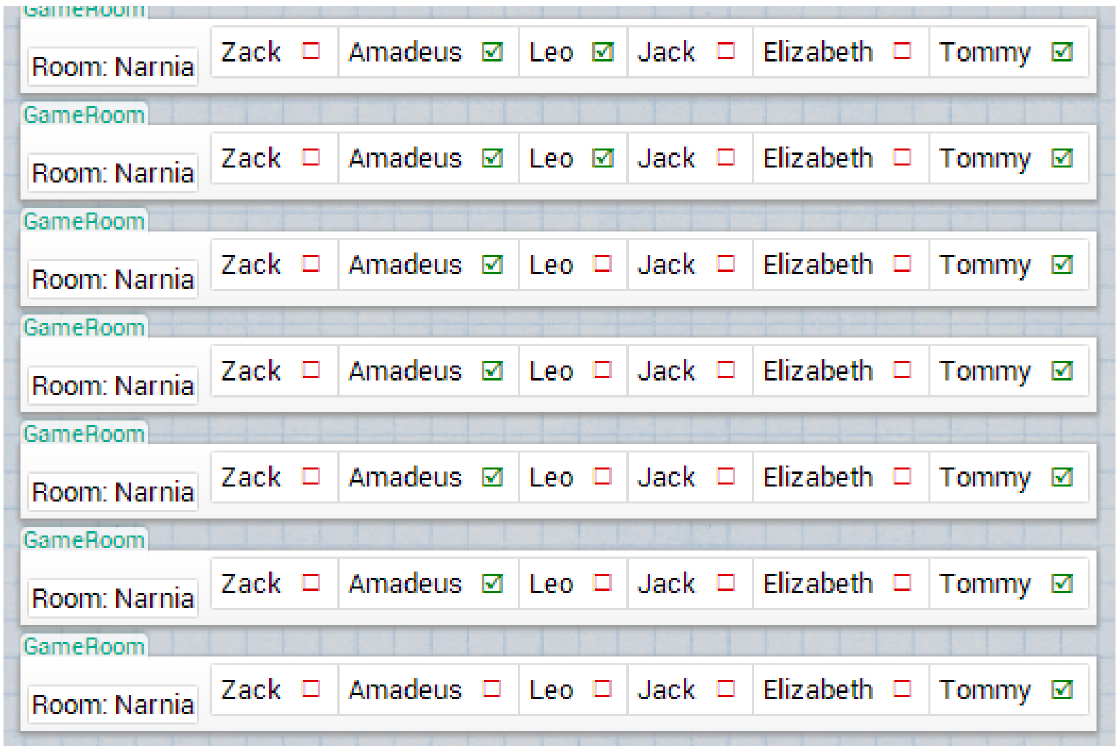
Thanks to DoodleDebug's inspection feature, the summarized representation of an object can be kept rather terse. As an example, the state of a game can be doodled on every update cycle like in figure 3. A developer observing this output may be interested in more detail of one particular state, which can be done by clicking on parts of it, resulting in a pop-up (figure 4).

The program code defining a `Player`'s rendering looks as follows:

```
public class Player implements Doodleable {
        public void doodleOn(DoodleCanvas c) {
                c.draw(name);
                c.newLine();
                c.draw("Alive?");
                c.draw(isAlive);
                c.newColumn();
                c.draw("Life points:");
                c.draw(lifePoints);
        }

        public void summarizeOn(DoodleCanvas c) {
                c.draw(name);
                c.draw(isAlive);
```

```
            }
}
```



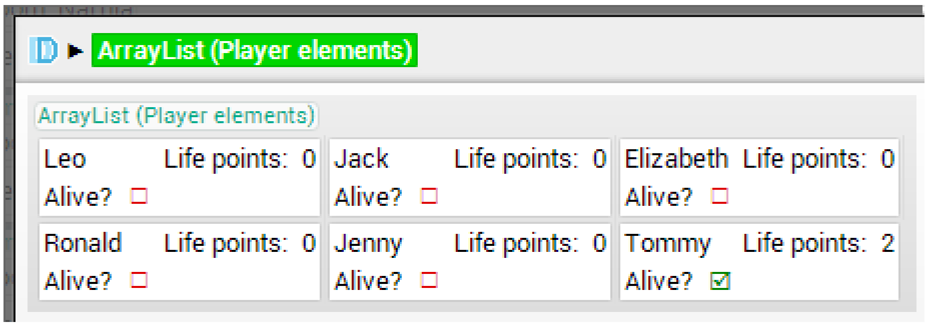
GameRoom
Room: Narnia | Zack ☐ | Amadeus ☑ | Leo ☑ | Jack ☐ | Elizabeth ☐ | Tommy ☑
GameRoom
Room: Narnia | Zack ☐ | Amadeus ☑ | Leo ☑ | Jack ☐ | Elizabeth ☐ | Tommy ☑
GameRoom
Room: Narnia | Zack ☐ | Amadeus ☑ | Leo ☐ | Jack ☐ | Elizabeth ☐ | Tommy ☑
GameRoom
Room: Narnia | Zack ☐ | Amadeus ☑ | Leo ☐ | Jack ☐ | Elizabeth ☐ | Tommy ☑
GameRoom
Room: Narnia | Zack ☐ | Amadeus ☑ | Leo ☐ | Jack ☐ | Elizabeth ☐ | Tommy ☑
GameRoom
Room: Narnia | Zack ☐ | Amadeus ☑ | Leo ☐ | Jack ☐ | Elizabeth ☐ | Tommy ☑
GameRoom
Room: Narnia | Zack ☐ | Amadeus ☐ | Leo ☐ | Jack ☐ | Elizabeth ☐ | Tommy ☑

Figure 3: A DoodleDebug console showing the doodles of `GameRooms`. The boxes beside each are summarized booleans and indicate if the player is alive.

2.4 Built-in Renderings

DoodleDebug comes with predefined renderings for a number of commonly used data types in Java. Those include:

- Primitives
- Arrays (same layout as Collections)
- Booleans (figure 6)
- Classes (figure 8)
- Collections (figure 3 and figure 7)
- Colors (figure 1)
- Images (various types)
- Maps (figure 1)
- Nulls (figure 6)
- Objects (default, figure 9)
- Strings



Leo Life points: 0 Alive? ☐	Jack Life points: 0 Alive? ☐	Elizabeth Life points: 0 Alive? ☐
Ronald Life points: 0 Alive? ☐	Jenny Life points: 0 Alive? ☐	Tommy Life points: 2 Alive? ☑

Figure 4: Details of a `GameRoom` from Figure 3.

- Tables (rectangular two-dimensional arrays and collections, figure 10)
- Trowables (figure 11)

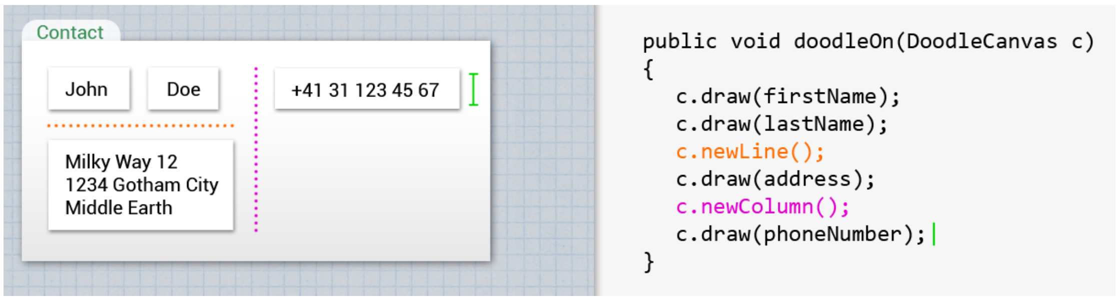
2.5 Customization API

DoodleDebug's API features two API layers for customization, Doodleables and Plugins. `Doodleable` customizations are always preferred over plugins when both are available for a type.

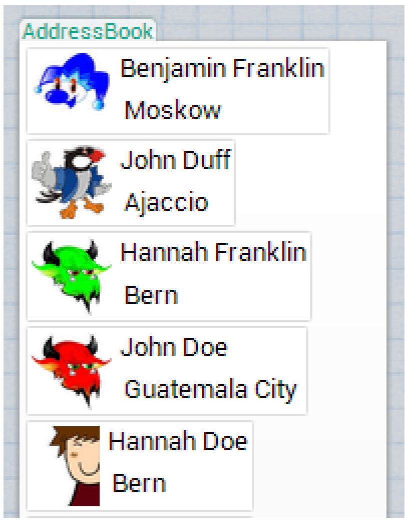
2.5.1 The Doodleable Interface

As a high-level customization layer, `Doodleable` utilizes an approach similar to Java's `toString()` method and targets most use cases since it's a simple and quick solution. The `Doodleable` interface features two methods: `doodleOn(DoodleCanvas)` for a regular representation and `summarizeOn(DoodleCanvas)` for a simplified and compact version. Distinguishing between them enables semantic zoom[9] when inspecting doodles: instead of geometrically scaling items, they gain more detail when zooming in.

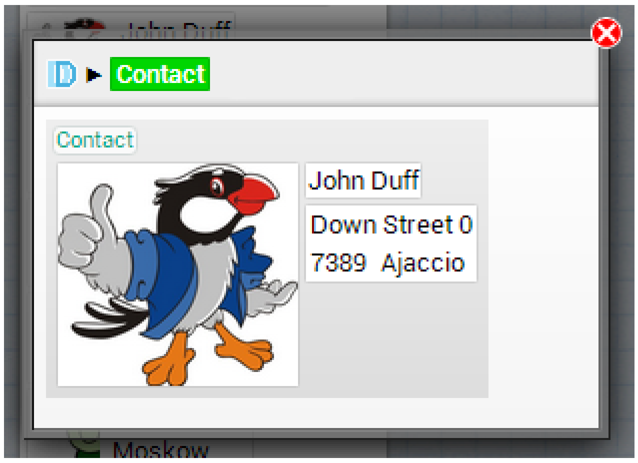
DoodleCanvas Instead of creating a string like in `System.out.println`, both methods receive a `DoodleCanvas` object for drawing contents on.



```
public void doodleOn(DoodleCanvas c)
{
    c.draw(firstName);
    c.draw(lastName);
    c.newLine();
    c.draw(address);
    c.newColumn();
    c.draw(phoneNumber);
}
```



(a) An address book only showing the summaries of its addresses. Clicking reveals the details in figure 5b.



(b) A popup showing the details of an address.

Figure 5: Exposure of a contact object at two different levels of detail.

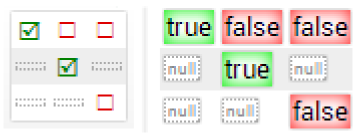


Figure 6: Summarized and detailed renderings of booleans.

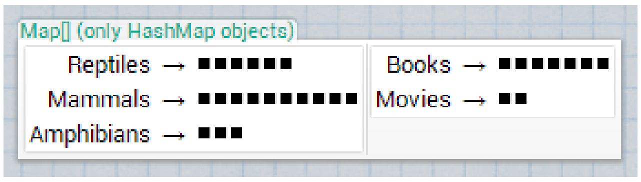
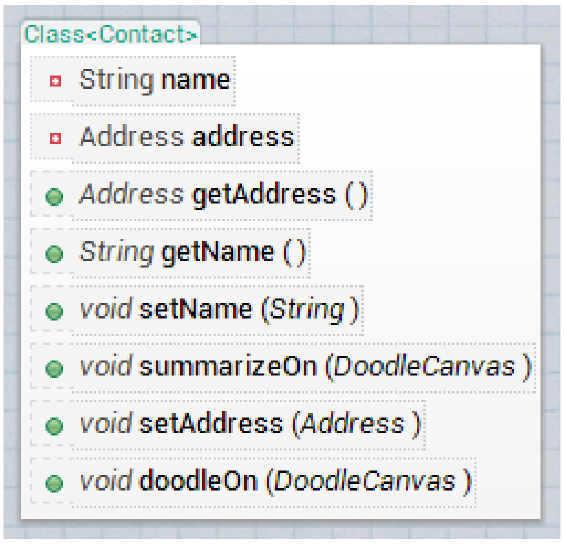


Figure 7: A doodled array of maps from strings to collections. The collections are rendered as summarized.

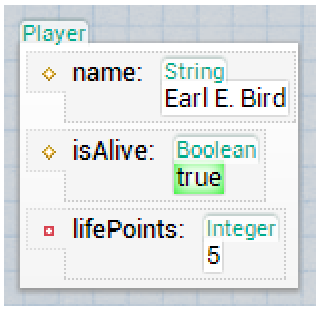
Figure 12: Example of a `Contact` class' `doodleOn()` method. Dotted lines visualize the effect of the structuring methods `newLine()` (orange) and `newColumn()` (purple). The green i-beam indicates the final position of the canvas' imaginary cursor.

Figure 12 illustrates how output is generated from a code sample. The paradigm behind `DoodleCanvas` adopts the formatting of text in terms of lines and columns. A virtual cursor starts at the upper-left corner of the canvas. Drawn objects align one beside each other until a new line is created, which causes the cursor to jump back to the left which one line height offset. The second formatting option is to create new columns, moving the cursor to a position on top, to the right of the right-most previous object. `DoodleCanvas` therefore has three public methods: `draw(Object)`, `newLine()` and `newColumn()`.


Class<Contact>

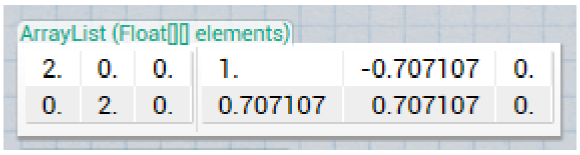
- String name
- Address address
- Address getAddress ()
- String getName ()
- void setName (String)
- void summarizeOn (DoodleCanvas)
- void setAddress (Address)
- void doodleOn (DoodleCanvas)

Figure 8: Rendering of a class object.


Player

- name: String Earl E. Bird
- isAlive: Boolean true
- lifePoints: Integer 5

Figure 9: The standard rendering of an object, visualizing all of its fields.


ArrayList (Float[][] elements)

2.	0.	0.	1.	-0.707107	0.
0.	2.	0.	0.707107	0.707107	0.

Figure 10: DoodleDebug's rendering of the same array as shown in Figure 20

2.5.2 Plugins

There are cases where implementing the `Doodleable` canvas is not a satisfying option for developers. If they don't have access to the source code, there is no (clean) way to add an interface. Also, the `Doodleable` API only provides primitive formatting options.

The second layer hooks in on a lower level by allowing users to provide `RenderingPlugins`, which are also used internally for DoodleDebug's built-in renderings. At the plugin level, the user can directly control the generation of HTML, CSS, and JavaScript.

Implementing Plugins For advanced arrangement or additional features like coloring, DoodleDebug includes the option to provide plugins. They must implement `RenderingPlugin`, which is most easily done by extending the built-in `AbstractPlugin`. Each plugin holds information about the object types it is able to render. Instead of drawing to a virtual canvas, a plugin receives a html `Tag` object and renders its own HTML code

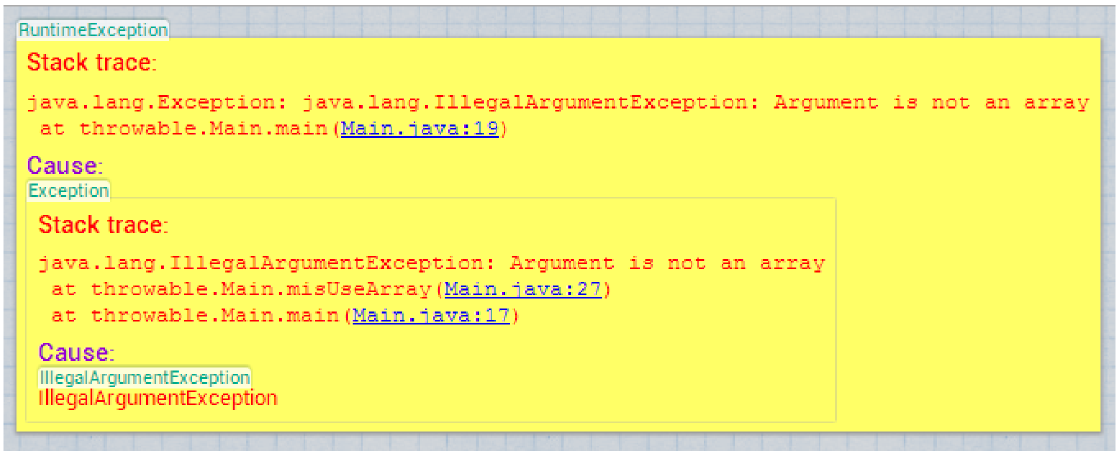

```
RuntimeException
Stack trace:
java.lang.Exception: java.lang.IllegalArgumentException: Argument is not an array
 at throwable.Main.main(Main.java:19)
Cause:
Exception
  Stack trace:
  java.lang.IllegalArgumentException: Argument is not an array
   at throwable.Main.misUseArray(Main.java:27)
   at throwable.Main.main(Main.java:17)
  Cause:
  IllegalArgumentException
  IllegalArgumentException
```

Figure 11: Rendering of an exception.

into this tag. The principle of semantic zoom is retained through two different methods for different detail levels. In addition to HTML code generation, plugins have the option to cleanly provide CSS rules and individually adjust class attributes assigned to object doodles.

3 Design

For the design of DoodleDebug's output, we consulted literature to carefully plan the different project cycles. As suggested by Buxton[1, p. 73-76], we decided to start with an initial design phase, which later fades into implementation as soon as a reasonable concept is available. In particular, we started off by sketching parts of the future user interface with pen and paper and confronted possible users with them. They were not told what a specific picture would represent, but directly had to state their intuition of what they believed it to be (figure 13). Based on their feedback, we had an estimation of every sketch's quality in terms of intelligibility and could either keep, improve or dismiss it. This cycle was repeated until our sketches reached a form where they were easily understood by people and we could start implementing.

Since the pen and paper sketches were rough and drafted quickly, the implementation phase initially revealed minor usability problems that had not been obvious in the design phase. Those were eliminated dynamically, still following Buxtons idea of gradually reducing design while implementation slowly starts.

4 Implementation

This section outlines the architecture and mechanisms DoodleDebug uses in order to perceive data, process them and display the results.

Figure 13: Programmer reacting to our sketches.

4.1 Data Transport

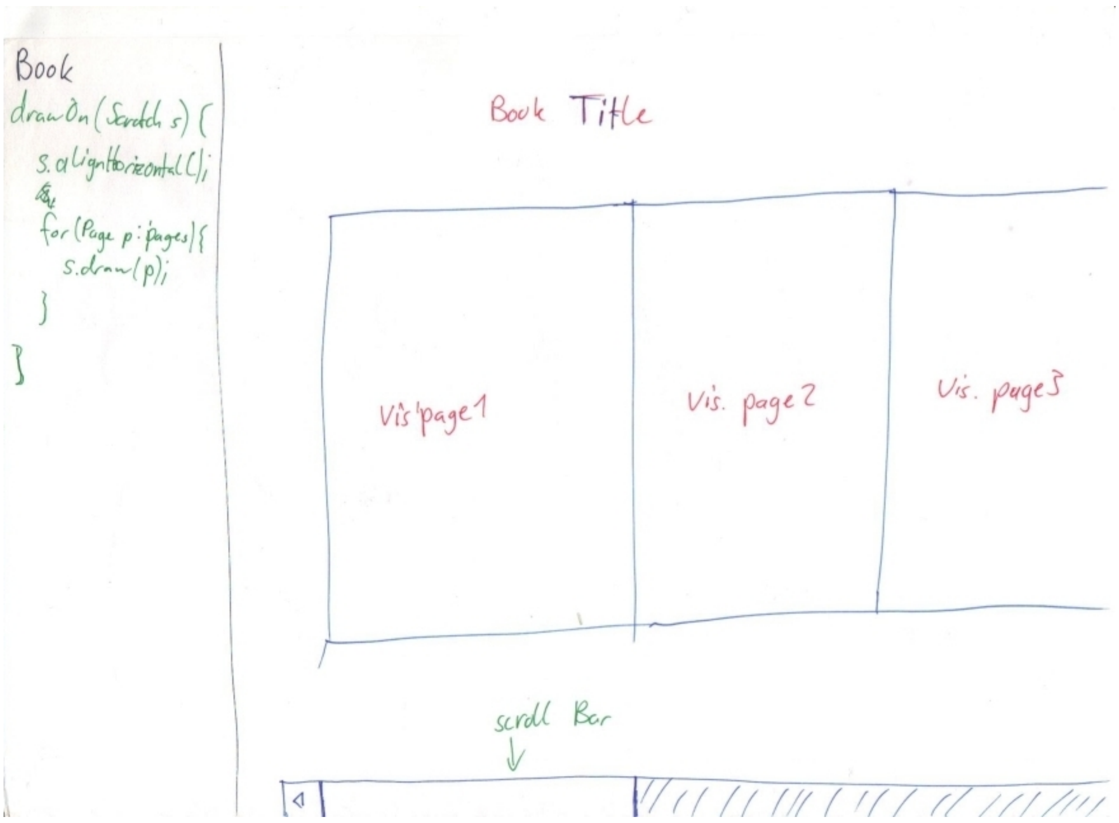
Since DoodleDebug is an Eclipse plugin, it's always running in a different Java VM than the project to be debugged itself. As a consequence, object data needs to be transported after each `Doo.dle` call. DoodleDebug uses a third-party library[3] to serialize doodled objects to XML, then transports the result as a string over a connection on localhost using SIMON[4].

4.2 Rendering

After a request for doodling an object has been received, DoodleDebug analyzes its type and searches for a fitting rendering in the different customization layers. If none is available, a default rendering is used.

[3]XStream, XML serializing library: `http://xstream.codehaus.org/`

[4]SIMON, Simple Invocation of Methods Over Network: `http://dev.root1.de/projects/simon`



```
Book
drawOn(Scratch s) {
  s.alignHorizontal();
  for(Page p : pages){
    s.draw(p);
  }
}
```

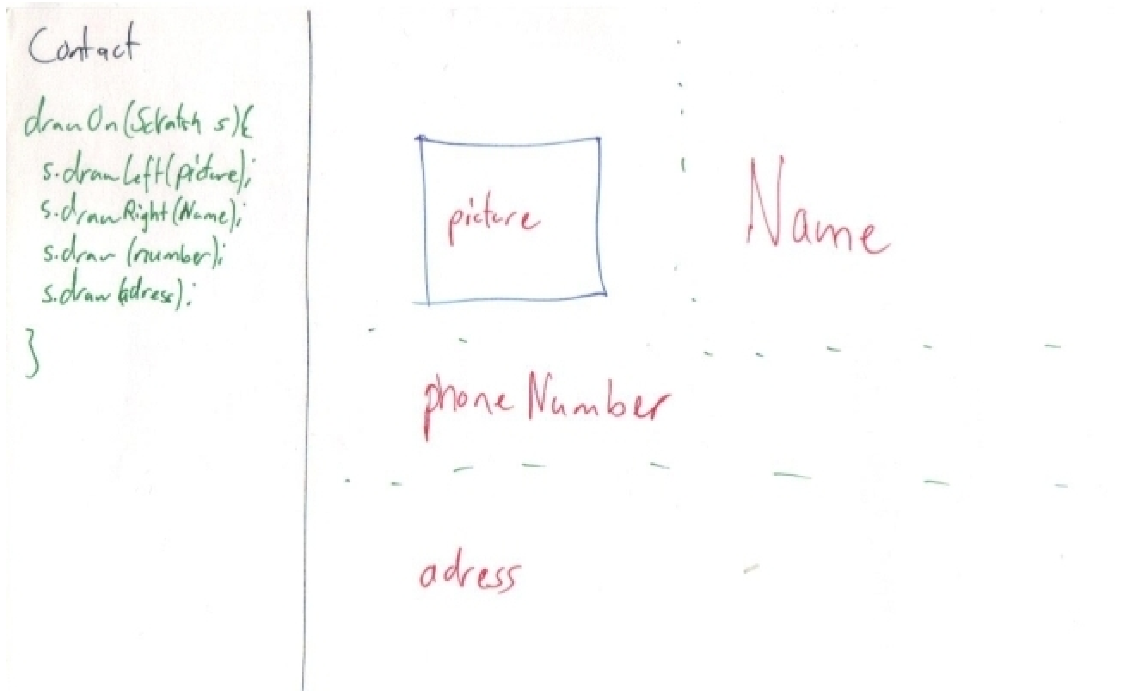
Figure 14: Inspired by AWT's FlowLayout, all objects are aligned horizontally.

4.2.1 Traversing Object Types

Renderings are iteratively searched for all types and supertypes of an object, starting at the innermost type, defined through the object's class name. As long as no rendering has been found, the algorithm traverses the inheritance tree in a layer-wise manner, always preferring the class type over interface types inside a layer. In other words, this algorithm starts searching on the object's direct class and interface types, then goes on for the class' and interface's direct ancestors and repeats until a match was found or all leaves were reached. The only type excluded from this search is the `Object` type, since it might be reached before some interface types.

4.2.2 Output

As output format, DoodleDebug uses HTML. Eclipse provides the package `org.eclipse.swt.browser`, which includes a web browser in the form of an Eclipse UI component. The rendering used for this browser's content is always the one from the OS's built-in browser and cannot be changed. As a consequence, we had to be careful when creating HTML output and always test in different browsers to prevent layout disparities on different systems. All major changes or extensions affecting the output were tested in



```
Contact
drawOn(Scratch s){
  s.drawLeft(picture);
  s.drawRight(Name);
  s.draw(number);
  s.draw(adress);
}
```

Figure 15: A proposed canvas API that traverses the canvas from top to bottom, dividing it into virtual lines.

four Browsers with different rendering engines: Gecko (tested in Firefox), Presto (tested in Opera), Webkit (tested in Chrome) and Trident (tested in Internet Explorer). On Windows systems, for instance, Eclipse uses the Internet Explorer rendering engine. As a consequence, a `meta` tag needed to be set in order to suppress quirks or any other compatibility mode[5] and make sure the newest installed rendering engine is used for best-possible support, especially for CSS3 features.

5 Study

We carried out a usability study [4] to validate the design of DoodleDebug. Usability studies are qualitative rather than quantitative, offering deeper semantic insights [5, pp. 13–15] than a quantitative study while simultaneously being easier to set up. We posed three different problems to each test subject to be alternately solved with or without DoodleDebug, recording with screen capture videos and think aloud protocol. [4, 5] We searched for problematic situations experienced with one tool to see if they'd been solved more easily using other tools. Seven developers solved 3 tasks each.

[5]MSDN documentation about legacy document modes for Internet Explorer: `http://msdn.microsoft.com/en-us/library/jj676915%28v=vs.85%29.aspx`

5.1 Study Session Setup

A fully functional release candidate of DoodleDebug (version 0.9.0 for Alpha and Bravo, 0.9.1 for others) was used. It ran inside Eclipse 4.2 (Classic edition), using a ThinkPad T410 with Windows 7 (x64) and an external mouse. 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, both on DoodleDebug and standard tools. However, since System.out.println is a fairly basic tool, no question regarding its use came up.

For the actual session, there were 3 different small programs containing some manually inserted bug, which the subjects had to find and eliminate. For one or two of those problems, subjects were allowed to use DoodleDebug and for the other one or two respectively, they had to fall back to classical tools (table in 5.4). The reason for letting some subjects only use classical debugging tools was to have a reference of behavior 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.

Subjects worked on a problem until it was completely solved, none of them needing more than 30 minutes for any particular problem.

5.2 Subjects

The subjects were convenience-sampled; programmers from our university and work environment were asked for participation if they had free time available. Further, they were assured of their anonymity. We informed them about the purpose of our study beforehand, and neither promised nor gave any reward for participating. The participants are enumerated in order of their participation.

Alias	Degree	Current activity
Alpha	BSc in Mathematics	Master student in CS
Bravo	BSc in CS	Master student in CS
Charlie	MSc in CS	PhD student in CS
Delta	BSc in CS	Master student in CS
Echo*	MSc in CS	Industry (1 year)
Foxtrot	BSc in CS	Master student in CS
Golf	Lic.rer.pol. in Economics	Industry (15 years)

*Experience with Eclipse plugin development.
CS: Computer Science

5.3 Posed Problems

In the first two tasks, the participants are given a failing unit-like test. They are informed that the test is correct, and asked to fix the bug and thus have the test pass. Subjects have access to all of the source code and are allowed to manipulate it. While the posed problems might seem biased towards DoodleDebug, that doesn't distract from their demonstration that DoodleDebug *can* vastly outperform `System.out.println`. DoodleDebug, being a generalization of `System.out.println`, at worst performs on par with `System.out.println`. Furthermore, not claiming that those problems are certainly best representatives of usual programming tasks, we still assume they're realistic enough to potentially occur once in a while during a programmers life.

5.3.1 Sorting

A couple of gray 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 in a unit test to a hand-built `List` which initially has the expected order.

Bug and Solution In the comparator, completely black colors are wrongly treated as complete white. It's an in-line case distinction that has to be adjusted.

5.3.2 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. In a unit test, comparison of a contact object before and after serialization fails.

Bug and Solution In the `SerializingUtil`, every field of type `long` is cast into an `integer` before serialization and back into a `long` afterwards. This causes a field called `phoneNumber` of `Address` to be changed into a negative value. The solution is to remove those casting operations from the code.

5.3.3 Decimal Alignment

In the third and last task, a class `DatabaseUtil` is given without source code, as a black box simulating access to an imaginary database by returning a two-dimensional array of `floats` when calling its only method `getData()`. Subjects are informed that in the returned table there are duplicated tuples and have to identify them. Finding the bug is not part of this task.

5.4 Observed behavior

For four study subjects, `toString()` was used as the default rendering. Two of them, Bravo and Delta, were allowed to use DoodleDebug for solving the serialization problem and both implemented `Doodleable` with `Contact` and `Address`. For three subjects, the ObjectDoodler (which shows all instance variables) was the standard rendering.

	Sorting	**Serialization**	**Table**
Alpha	DD	Classic	-
Bravo	Classic	DD	-
Charlie	DD	Classic	DD
Delta	Classic	DD	Classic
Echo	DD	Classic	DD
Foxtrot	Classic	DD	Classic
Golf	DD	Classic	DD

5.4.1 Observations using System.out.println

5 out of 7 subjects (all except Delta and Echo) made use of this mechanism to visualize runtime data. Alpha brought forward the argument of laziness to open a debugger or to stare at foreign code. When quizzed, Alpha said 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 are not directly possible with a classical debugger like the one built into Eclipse classic.

Homogenous Output Apart from Charlie, none of the subjects spent time customizing their ad-hoc textual prints by overriding `toString` on objects to be visualized. If any, they utilized in-line string concatenation for formatting purposes, like inserting delimiters between different objects. Beta printed each element of the wrongly sorted color list and then stared at the (unaligned) numerical values of red, green and blue color components. Once he found out that a black element was at the end instead of the beginning, he went on to investigate the cause, leading to the problem.

In contrast, subjects using DoodleDebug already had built-in renderings for `Collection` and `Color`, making the problem immediately obvious.

Uninspectable and Useless Output The standard implementation of `System.out.println` prints only the class name and object hash, for example:

```
ch.unibe.scg.spacepirates.Player@ece88d2
```

Bravo and Golf resorted to a compile-run cycle, where they would explore the object graph by re-running the program, adapting the print statement to explore the parts of the object graph they were interested in.

Charlie was the only subject to override `toString` methods, after being dissatisfied with the standard printout of class name plus hash. He implemented `toString` to print out all fields of an object, as seen in Figure 16. He argued that the difference between two states of an object should be visible when seeing all its field, recursively including fields of referenced objects.

Alpha produced a very similar output, but instead of overriding `toString()`, he extracted all fields from outside using getter methods directly inside the `System.out.println` method (figure 17).

```
public class Contact { ...
  public String toString() {
        return "name: " + name
                + ", address: " + address;
  }
}
public class Address { ...
  public String toString() {
        return "street: " + street
          + ", phoneNumber: " + phoneNumber
          + ", city: " + city;
  }
}
```

Figure 16: Charlie overwrote `toString()` to aid debugging.

```
System.out.println("name: " + contact.getName()
                + ", street: "
                + contact.getAddress().getStreet()
                + ", phoneNumber: "
                + contact.getAddress().getPhoneNumber()
                + ", city: "
                + contact.getAddress().getCity();
```

Figure 17: Alpha used `System.out.println` to aid debugging.

Another approach to solve insufficient output was to switch from `System.out.-println` to the debugger, observed with Bravo and Foxtrot.

5.4.2 Observations using a Debugger

Four subjects (Bravo, Delta, Echo and Foxtrot) used the Eclipse debugger to inspect objects, only Delta and Echo using it exclusively. Echo's argumentation for this usage was that debuggers are more powerful in comparison to `System.out.println`, because they allow one 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`. Echo mentioned that they missed the feature to compare two objects from different points in time.

Comparison Between Objects The built-in Eclipse debugger only allows you 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`, but 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 he missed the feature to inspect two objects simultaneously.

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. 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. In particular, programmers only interacting with an object over some super type may have a much simpler mental model of what it is than represented in the debugger on the basis of its runtime type. We observed this in 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 18), but instantly gave up and switched to writing a `for` loop which sequentially prints out all elements using `System.out.println`.

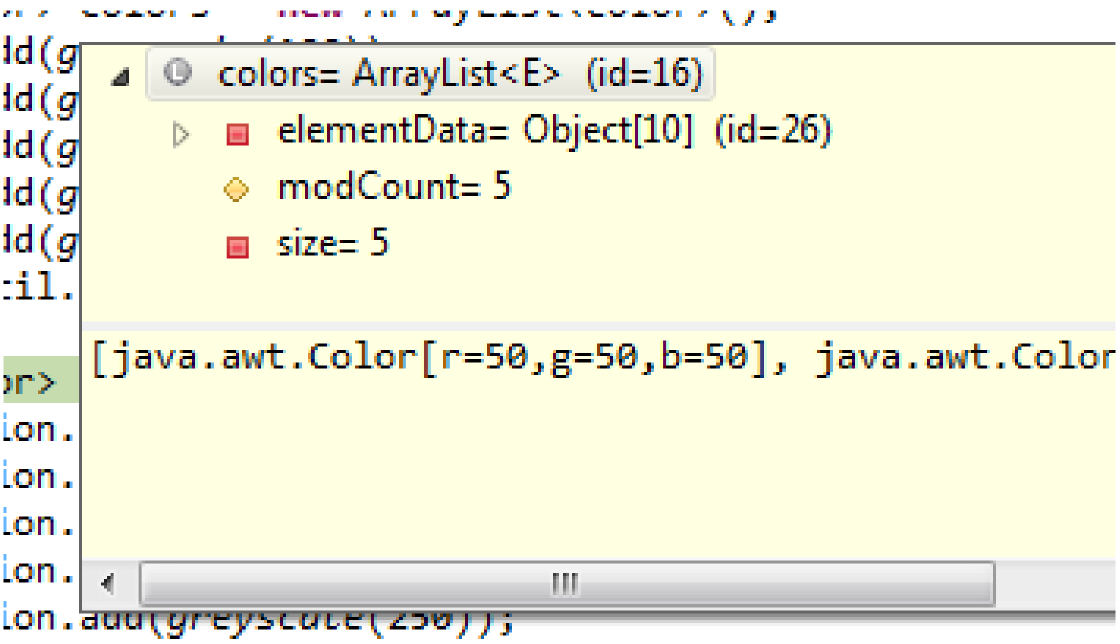

```
colors= ArrayList<E>  (id=16)
    elementData= Object[10]  (id=26)
    modCount= 5
    size= 5
[java.awt.Color[r=50,g=50,b=50], java.awt.Color
```

Figure 18: Eclipse's debugger visualizes all fields of an object's runtime type (ArrayList)

5.4.3 Observations using DoodleDebug

When allowed (but not forced) to use DoodleDebug, every subject solved the problem without making use of any other tool. Only two subjects ever implemented the `Doodleable` interface for solving a problem. Based on their implementation and `toString` implementations of other subjects, we changed the default object rendering from using `toString` to listing all instance variables. As a consequence, subjects found relevant information directly after doodling an object, making any customization unnecessary.

Two methods in `Doodleable` interface Echo was confused about the difference between the two methods in the `Doodleable` interface. After going back to the documentation of the interface, he understood. Alpha did not show any reaction when implementing the `Doodleable` interface, but misunderstood the meaning of its two methods: Instead of reducing the number of objects on the canvas, he simply delegated the zoom mechanism to each one by using the `Canvas.drawSmall` method (figure 19).

Since none of the other subjects ever made used of `Canvas.drawSmall` and Alpha misunderstood its purpose, we removed this method from the API.

```
public void doodleOn(Canvas c) {
        c.draw(name);
        c.newLine();
        c.draw(phoneNumber);
        c.newColumn();
        c.draw(address);
}
public void summarizeOn(Canvas c) {
        c.drawSmall(name);
        c.newLine();
        c.drawSmall(phoneNumber);
        c.newColumn();
        c.drawSmall(address);
}
```

Figure 19: Alpha mistakenly delegated the zooming mechanism to each printed object.

Console Keeps Stealing Focus When certain output is printed onto the Eclipse console, it gains the UI's focus by default. Due to the problem setup, every program initially threw an exception at the end of its execution, signaling the problem had 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, while the other subjects just switched back to the DoodleDebug view tab after a few seconds.

Problem immediately obvious Charlie doodled both the wrongly sorted color list and the correctly sorted one. Since they were both on-screen immediately, he instantly noticed both their similarity and the difference, which was that the black color was on the wrong side of one list.

On the serialization problem, all three subjects (Bravo, Delta, Foxtrot) using DoodleDebug managed to find the changed field instantly after calling `Doo.dle()` once before and once after the de-/serialization step. That's because DoodleDebug's `ObjectPlugin` shows all instance variables, which was enough to immediately see the problem.

Default doodles were useless When Alpha used the default visualization without implementing the `Doodleable` interface, DoodleDebug simply defaulted to printing the output of the `toString()` method. This was entirely useless, as it only printed the name of class plus an object hash. Thus, Alpha implemented the `Doodleable` interface, and included all instance variables of the questioned object for its representation.

On sessions without DoodleDebug, Golf iteratively re-ran the program, each time printing another field of the same object and Bravo wrote a custom textual representation, printing all fields labelled with their names.

Based on these findings, we modified DoodleDebug's standard rendering: Instead of falling back to the object's `toString()` method, the new default rendering prints all fields, including field name, as seen in Figure 9.

5.4.4 Observations Without Any Debugging Tool

Subject Golf was the only one to solve a problem without using any debugging tool. On the serialization problem, he tried to comprehend the logic of the problem's `SerializingUtil`. Unlike others, he found the problem source at the same time as the nature of the problem itself. He fixed the bug and verified the result using `System.out.println`.

6 Future Work

Highlighting Object Diffs When tracking an object over time, the most important information is located where properties of an object have changed. Therefore, a mechanism comparing objects when they are doodled multiple times could be a powerful feature.

Clickable Doodles One problem of using `System.out.println` over a long period of time is that it's indeed clear which object is printed, but not where the `System.out.println` call has been made. To find it, some tedious text search over the whole project is needed in the worst case. As a solution in DoodleDebug, doodles could include a link pointing to the line in source code where the `Doo.dle` call was triggered, similar to the way Throwables are tracked.

Debugger Integration Since the eclipse debugger only uses textual representations and doesn't allow any customization, a way to enhance it could be to include doodles. A user would be allowed to switch from the standard textual mode to DoodleDebug mode, where an object would be inspected using DoodleDebug's rendering. Or a button beside the textual representation would create a popup with the doodled version of it.

7 Related Work

7.1 Existing Debugging Tools

In the world of programming languages, there are two classes of widely used debugging tools: Textual output like Java's `System.out.println` and Debuggers.

7.1.1 Textual Output

Java's `System.out.println` provides a default textual representation for any object. Primitives and Strings are rendered in a trivial way; all other objects, including arrays, are represented with their class name and object hash. Any object's representation can be changed by overriding its `toString` method.

Debugging by directly writing into source code may be easier and quicker than having to equip other debugging tools for a session[2, p. 119]. Also, objects can be tracked over time by printing them at different points of execution and comparing the resulting outputs.

Best Practice For Textual Output To understand in detail how programmers use textual output for debugging, we wrote a script that searches for such methods in open source projects. It analyzed SqueakSource, a hosting service for Smalltalk projects, searching for `printOn` methods, Smalltalk's equivalent to Java's `System.out.println`. Achieved results[8] gave us hints on how to design default renderings on the one hand and showed shortcomings of text-only output.

Missing Features On Textual Output To achieve two-dimensional renderings in text, the developer only has the newline character as an option. Since a line can have only one new-line character, putting two 2-dimensional structures next to each other requires coordination. In practice, developers instead refrain from printing 2-dimensional structures next to each other [8], and instead print them below each other, as illustrated in Figure 20. Furthermore, the resulting output is purely static, and offers no possibility to navigate to related objects or back to the source code.

7.1.2 Debuggers

Debuggers allow users to stop a program's execution at any line of source code. When stopped, any detail of the current state is inspectable. Unlike `System.out.println`, debuggers allow their users to put additional or remove existing break points at runtime, which makes changing the target piece of code to debug quicker, especially because the current program state doesn't need to be re-created like on a new run.

```
an Array(
MatrixTransform2x3(
        2.0 0.0 0.0
        0.0 2.0 0.0
) MatrixTransform2x3(
        1 -0.707107 0.0
        0.707107 0.707107 0.0
))
```

Figure 20: Textual visualization of an array of 2D-matrices, without DoodleDebug.

Drawbacks Of Using A Debugger Since a debugger only shows the program's state at one point in time, comparison of two time slices is hardly possible. Important details may only be memorized and mentally compared to the ones at a later point in time.

Eclipse's built-in Java debugger brings no options to customize its output. When inspecting an object, all its fields are listed by name, together with a textual representation. An object containing many fields may be costly to inspect if only few of them are relevant to the current user.

8 Conclusion

DoodleDebug is a valuable drop-in replacement for Java's `System.out.println`. It introduces techniques to enhance debugging output by adopting well-proven mechanisms from debuggers and console printing on the one hand and introducing simple new ones on the other hand. Getting started is rather easy since one single API method already enables the core features of DoodleDebug. Previous `System.out.println` statements can be directly replaced by `Doo.dle` calls without losing any ground.

Since the output is held in HTML and DoodleDebug is bound to be an Eclipse plugin, the possibilities for additional features are infinite. The output view with JavaScript running in it is Turing-complete and supports any graphical output a monitor can display. Any IDE-related actions can be implemented because a stable communication between output and plugin code is already running.

The DoodleDebug Eclipse plugin, all source code and a demo video are available at `http://scg.unibe.ch/wiki/projects/DoodleDebug`.

9 Acknowledgements

I'm not good with speeches. But I can truly say that the following people spared no effort while accompanying me on this journey, thank you: Niko Schwarz, Oscar Nierstrasz, Mircea Lungu, Andrei Chis, the SCG, and all study participants.

References

[1] Bill Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design (Interactive Technologies).* Morgan Kaufmann, 1 edition, April 2007.

[2] Brian W. Kernighan and Rob Pike. *The Practice of Programming (Addison-Wesley Professional Computing Series).* Addison-Wesley, 1 edition, February 1999.

[3] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[4] Steve Krug. *Don't make me think! A Common Sense Approach to Web Usability.* New Riders Publishing, Indiana, United States, 2000.

[5] Darren Langdridge and Gareth Hagger-johnson. *Introduction to Research Methods and Data Analysis in Psychology.* Prentice Hall, 2 ill edition, June 2009.

[6] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition.* Microsoft Press, 2nd edition, July 2004.

[7] Donald A. Norman. *The Design of Everyday Things.* The MIT Press, 1988.

[8] Niko Schwarz. DoodleDebug, objects should sketch themselves for code understanding. In *Proceedings of the TOOLS 2011, 5th Workshop on Dynamic Languages and Applications (DYLA'11).*, 2011.

[9] 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.