DoodleDebug, Clustered

Morphing DoodleDebug into a clustered setup using fat clients

Master Thesis

Cedric Reichenbach
from
Eriz BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

21. August 2015

Prof. Dr. Oscar Nierstrasz
Niko Schwarz, Boris Spasojević

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Acknowledgements

I want to thank everybody who supported me during the DoodleDebug project. In particular, Oscar Nierstrasz for making this possible, Niko Schwarz, Boris Spasojević, Mirca Lungu and Andrei Chis for their support during my Bachelor's and Master's projects, and finally the SCG, all study participants and everybody who helped me out.

It's been a long journey, but worth every step.

Abstract

Knowing the run-time state of objects is essential for analyzing and resolving errors in a program. Inserting print statements into code for troubleshooting is considered bad practice and the resulting output is completely static. While debuggers give dynamic insight to particular run-time states, they suffer from certain shortcomings. Furthermore, attaching debuggers to live production systems is often not possible, and errors need to be locally reproduced in order to debug them. DoodleDebug has previously been shown to provide advantages over debuggers and print statements by combining the best of both worlds. This work documents how it has been morphed from an Eclipse plugin to an independent framework, to be utilized both, in development and production systems. In particular, its built-in support for HBase and decentralized data management makes it a reasonable logging solution for clustered applications with performance requirements.

Contents

1 Introduction

To understand and debug a program, developers rely on tools to track its run time states during execution. Two common strategies are inserting `System.out.println` statements in source code and using a debugger. Both have exclusive advantages and drawbacks compared to each other, which is the reason why we introduced DoodleDebug, trying to get the best of both worlds.

1.1 What is DoodleDebug?

One method for understanding a program's state is to insert print statements, like `System.out.println` in Java, that output relevant state information. 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 to use 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 for long-running programs. If they initially choose a detailed and verbose object representation, the output will grow and become tedious to read. As McConnell explains in the book, Code Complete [14, p. 539], "put[ting] print statements in the program to find the defect [...] is not adequate", and lists "scatter[ing] print statements randomly throughout a program" in "The Devil's Guide to Debugging".

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.

A different widely adapted approach to understanding program states are debuggers [10]. When utilizing a debugger, a program can be stopped at some specific point of its execution, allowing developers to inspect any detail of state. A clear advantage over textual output is the ability to inspect objects on demand. Any information in the current state can be gathered without having to manipulate the program's code or re-running it. McConnell states that "[m]ost of the defects [...] will be minor oversights and typos, easily found by [...] stepping through the code in a debugger" [14, p. 535].

The drawbacks of classical 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. There have been approaches to this problem by enhancing debuggers to memorize different object states over time, so that users could move back and forth among execution steps. However, those solutions suffer from performance issues and only run reasonably fast when cut down, resulting in loss of relevant information [12].

DoodleDebug attempts to combine the power of the above mentioned tools while avoiding their downsides. Its output is generated through an API taking its cue from `System.out.println`. A developer simply needs to call `Doo.dle(object)`, henceforth referred to as doodling[1], to graphically visualize an object of any 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 a graphical representation, called doodle, of the object provided as argument instantly appears. 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 override its default rendering by implementing 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

Figure 1.1 shows the doodle of a `Map<String, Color>` object containing three entries, using the default rendering for maps. Each key and value object is clickable to inspect it for more detail.

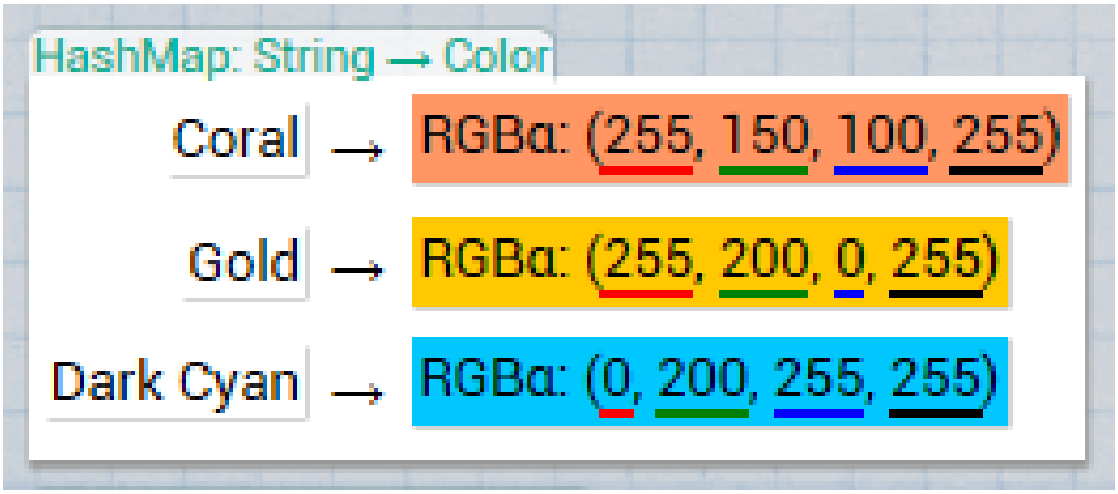


Figure 1.1: Doodle of a map from strings to colors with built-in renderings only.

[1] Buffy: "What is this?" Willow: "A doodle. I do doodle. You too. You do doodle too." – *from Buffy the Vampire Slayer*

1.2 Extending DoodleDebug to a Clustered Environment

The initial goal of the DoodleDebug project was to create a new tool for simple, quick, but powerful debugging during development. The first iteration focused on local usage inside an IDE, making the tool obsolete in production mode. However, some applications, like web servers, still require ways to visualize their state during run time. Developers usually rely on text-based logging libraries that work similar to `System.out.println`, but write to a file instead of directly to a console. Since DoodleDebug utilizes a similar workflow to `System.out.println`, but introduces more features, it suggests itself to also be used for run-time state visualization. Considering that there are applications running many instances simultaneously, e.g. clustered web servers, we aimed on a setup that would scale well in this case.

In order to realize those goals, an extensive refactoring of DoodleDebug's architectural structure was required. On the one hand, all dependencies to Eclipse needed to be eradicated and, where necessary, features re-implemented in a more environment-agnostic way, so that DoodleDebug could be used in virtually any application, during development and production. We achieved this by integrating a standalone web server for serving the output and utilizing websocket communication between web front end and said server for dynamic user interaction.

On the other hand, the new setup should be able to handle a great amount of doodling traffic, possibly from multiple applications logging to the same database. We considered various architectural options, discussed their individual assets and drawbacks, and finally decided for fat client setup. Each application contains the full DoodleDebug rendering stack and sends processed data to a database, which by default is an HBase and possibly shared with other applications.

We carried out performance benchmarks, comparing various possible DoodleDebug setups with related other technologies. Furthermore, we integrated DoodleDebug into an open source application to evaluate the difficulty of such a process.

For more clarity, we'll henceforth refer to the first release as DoodleDebug 1, and analogously DoodleDebug 2 for the new one introduced here.

2 Related Work

As DoodleDebug affects debugging during development as well as in production mode, and works for single applications as well as clustered systems, there are several areas of related work to consider. We'll omit the part of local, development debugging tools like `System.out.println` or classical debuggers since they've already been discussed in previous work related to DoodleDebug [16].

2.1 Precompiling and Run Time

This section covers related work that operates before and during the logging process, i.e. influences the content to be logged.

2.1.1 Apache log4j

For classical logging in Java, log4j is one of the most widely used libraries available. It provides basic differentiation into 7 levels of importance for logging calls, which can then be filtered for output generation [9]. The output can be written to any target using appenders [9]; which makes it possible to let several instances of a clustered application to write into one log, e.g. a clustered HBase.

However, log4j is completely text-based without any further formatting options or features for graphical representations. Furthermore, output is static, meaning that abstraction is limited to predefined importance levels and requires many carefully organized logging statements in user code.

2.1.2 LogEnhancer

Since the only information about an application's inner state in production mode is usually a simple log file, diagnosis in case of an error is generally difficult. LogEnhancer [22] aims to improve post-failure debugging of such systems by detecting log calls at compile time and extending them with more related variables to be printed into a separate log file. That way, the original log works on as before, but if required, developers may fall back to the extended log file for more detailed information.

2.2 Post Mortem

This section lists work about static analysis on applications or code sequences after finishing their execution, utilizing log output and source code.

2.2.1 Log Filtering and Interpretation for Root Cause Analysis

Zawawy, Kontogiannis, and Mylopoulos [23] state that one problem of error diagnosis in large applications is the vast amount of logged data. They propose a framework which normalizes log data and reduces it based on user-defined goal models.

2.2.2 Problem Detection by Mining Log Text

Xu, Huang, Fox, Patterson, and Jordan [19] have introduced an approach for finding large-scale system problems based on source code and console logs. First, textual patterns to be expected in the output are derived from source code and converted into regular expressions. Then, vast amounts of log text are parsed for said expressions and possible operational problems detected using a machine learning algorithm.

This method conceptually differs from DoodleDebug in the sense that it assumes large amounts of information to be logged by default, and mines relevant information from that. User interaction is only needed at the very end to sort out false positives. DoodleDebug, on the other hand, encourages users to log few high-level object, which can be expected later. That way, the log remains concise while implicitly holding more information.

2.2.3 SherLog

Based on source code and log output, SherLog [20] infers control flow and data information to help programmers debug errors that cannot be reproduced locally. First, it reconstructs the execution path a program has taken before a certain error (path inference). Based on the execution path, it further tracks down the value-flow of a certain variable (value inference).

2.2.4 Automated Detection of Failure Causes in System Logs

Another log parser has been presented by Mariani and Pastore [13]. It identifies dependencies between events and values for legal behavior and generates models from them. In case of a failure, those models are compared to the actual log and differences used to track down the root cause.

3 DoodleDebug in a Nutshell

A more detailed description of DoodleDebug can be found in previous work [16].

DoodleDebug's visual design rationale is that layouts must be class-specific and scannable. Being scannable means that a printed item is never much bigger than a few lines of text, so that many objects simultaneously fit on screen, and outliers can easily be spotted. All provided visualizations attempt to use screen-estate wisely.

DoodleDebug features a simple API that offers programmers the ability to display the state of objects at various points during execution of a program. The API offers the simplicity of `System.out.println`, with the graphical sophistication and interaction of an object inspector. In this section, we show how DoodleDebug works from the point of view of the developer who is using it. We see how DoodleDebug can be used to display and interact with an Address Book application and a simple role-playing game.

3.1 Canvas API

Developers can choose how objects of a class are to be doodled on a virtual canvas by implementing the `Doodleable` interface. It requires the method `doodleOn(Canvas)`. The canvas lets the developer choose the rough layout of the doodle. It is designed to be low-friction and useful for debugging.

The Canvas API features a paradigm for simple object alignment inspired by the flow of text on a sheet of paper. A virtual cursor is spawned in the upper left corner and ends up at the right side of each newly printed object. Repeatedly printing objects therefore outputs them in a sequence, just like words on the same line. Users may use two formatting methods, `newLine` and `newColumn`, where lines are nested into columns as seen in Figure 3.1.

3.2 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.

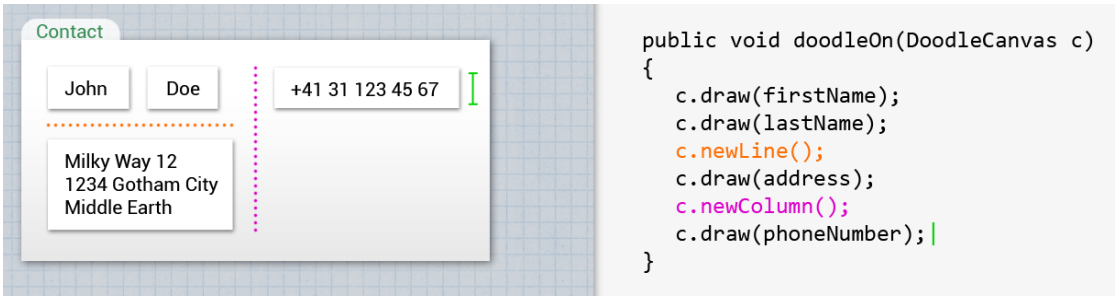


Figure 3.1: Example of a `Contact` class' `doodleOn()` method. Dotted lines visualize the effect of the structuring methods `newLine()` (orange) and `newColumn()` (purple). The green I-beam indicates the final position of the canvas' imaginary cursor.

Object doodles are divided into levels, based on their nesting depth. An object provided in the `Doodle` 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 3.2a 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 3.2b).

Navigation between nesting layers inside a pop-up is aided with bread crumbs [11, p. 76-78], which traces the currently inspected branch of the object graph, as seen in the top area of Figure 3.3. 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.

3.2.1 Tracking an Object Over Time

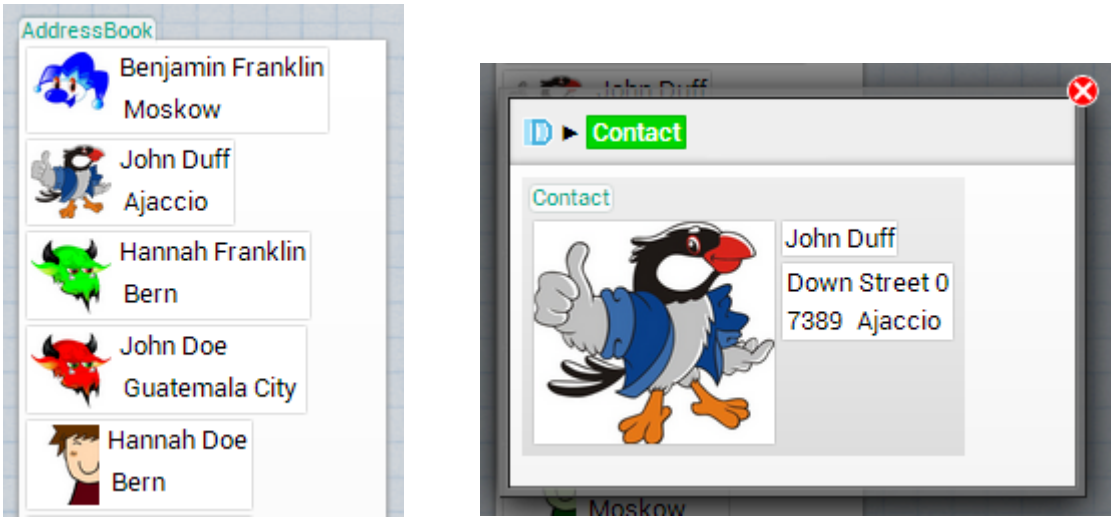
Thanks to DoodleDebug's inspection feature, the summarized representation of an object can be kept rather terse. As an example, the state of a game can be doodled on every update cycle like in Figure 3.5. A developer observing this output may be interested in more detail of one particular state, which can be obtained by clicking on areas within the doodle. For instance, clicking a player list generates a popup with the state of this object at the time of doodling (Figure 3.6).

In this example, the `GameRoom` and `Player` (Figure 3.4) classes define a custom rendering. Other types, like the player list or booleans representing the alive/dead state are rendered by DoodleDebug's defaults.

3.3 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.



(a) An address book only showing the summaries of its addresses. Clicking reveals the details in Figure 3.2b.

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

Figure 3.2: Exposure of a contact object at two different levels of detail.

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 an HTML `Tag` object and renders its own HTML code 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.

For instance, someone may be particularly interested in quotation segments inside text strings. They can override DoodleDebug's default rendering for `String` type objects by creating a `RenderingPlugin` as in Figure 3.7. In our example, quotations are defined as text snippets starting and ending with quotation marks, thus, we simply replace wrap such occurrences with an HTML `q` tag by using regular expressions[1]. After styling `q` elements in an eye-catching way with simple CSS rules and doodling a test string, quotation snippets can be easier found in running text (Figure 3.7).

3.4 Built-in Renderings

Not all classes implement the `Doodleable` interface. For those that don't, DoodleDebug defaults to built-in renderings. Currently, there are built-in renderings for arrays, primitives, booleans (Figure 3.9), classes (Figure 3.11), collections (Figure 3.5 and Figure 3.10), colors (Figure 1.1), images (various types), `Maps` (Figure 1.1), null values (Figure 3.9), objects (Figure 3.12), strings, tables (rectangular two-dimensional arrays and collections, Figure 3.13) as well as `Throwables` (Figure 3.14). For every supported type, two default renderings are needed; one for the detailed view and one for the summarized view.

We know from previous work that 77 % of custom `toString()` methods print out the class name [17]. We therefore tried to find renderings that always show the class name, but do so in a space-efficient way.

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`

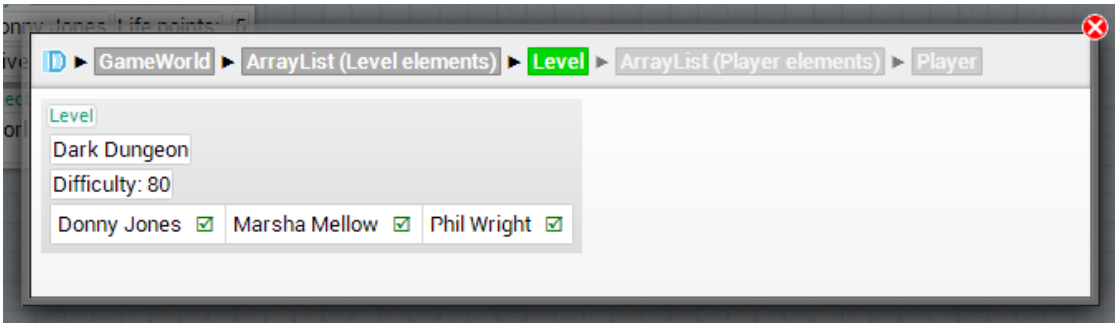


Figure 3.3: The labels at the top are *bread crumbs*. The developer is currently looking at a `Level` object, which is the member of an ArrayList, which is the member of a `GameWorld`. The object that was doodled was the `GameWorld` object. The developer was previously zoomed in to player but zoomed out again to Level.

```
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);
   }
}
```

Figure 3.4: Program code defining a `Player`'s rendering. The `Doodleable` interface enforces two methods referring to different levels of detail.

Furthermore, all plugins attempt to use screen estate wisely and not blatantly waste space in comparison to `System.out.println`.

3.5 Features

In this section, we list some of the conceptual characteristics of DoodleDebug, both the ones that were there in 1.0 and new ones introduced in this iteration.

3.5.1 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`

Figure 3.5: A DoodleDebug console showing the doodles of `GameRooms`. The boxes beside each are summarized booleans and indicate if the player is alive.

- 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.

3.5.2 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 [15, p. 199-200] and Buxton [3, 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.

3.5.2.1 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[2] and mIRC[3]:

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

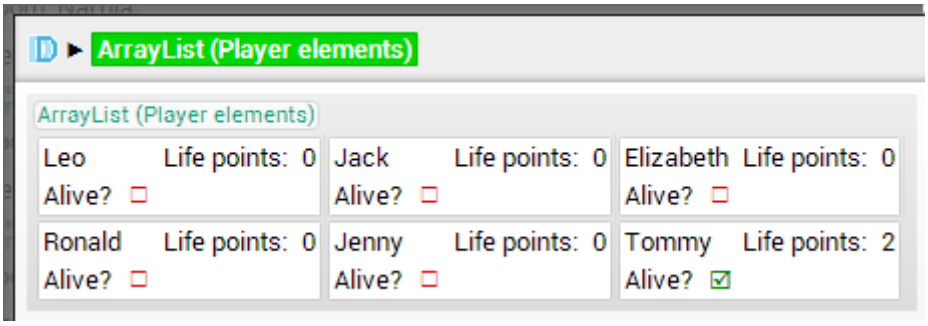


Figure 3.6: Details of a `GameRoom` from Figure 3.5.

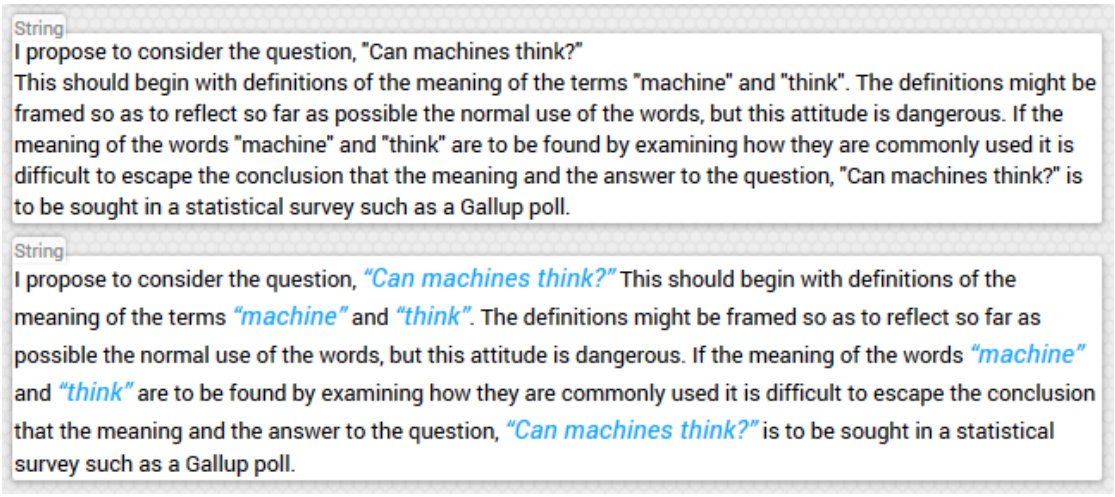


Figure 3.7: A simple string with its default representation (top) and a custom one created through a simple `RenderingPlugin` (Figure 3.8).

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.

3.5.2.2 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 [16].

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.

```
public class StringPlugin extends AbstractPlugin {

  @Override
  public Set<Class<?>> getDrawableClasses() {
    Set<Class<?>> hs = new HashSet<>();
    hs.add(String.class);
    return hs;
  }

  @Override
  public void render(Object object, Tag tag) throws DoodleRenderException {
    String string = (String) object;
    string = string.replaceAll("\"([a-zA-Z1-9.,;!? ]*)\"", "<q>$1</q>");
    renderSimplified(string, tag);
  }

  @Override
  public void renderSimplified(Object object, Tag tag)
      throws DoodleRenderException {
    tag.add(object);
  }

  @Override
  public String getCSS() {
    return "." + this.getClassAttribute() + " q {"
        + "font-size: 120%; color: #0af;}";
  }
}
```

Figure 3.8: A simple plugin for custom string representation, resulting in Figure 3.7. `AbstractPlugin` partially implements `RenderingPlugin` for convenience. The two methods `render` and `renderSimplified` originate from DoodleDebug's semantic zoom feature and should both represent the same object in HTML, once in a detailed version and once reduced to most essential information to spare space.

3.5.3 Output Grouping

In favor of overview and clarity while reading, DoodleDebug virtually separates logs of different applications using a key, called *application name*, for each log. By default, this is the canonical name of the main method's class, e.g. `ch.unibe.scg.example.MyApplicationMain`. In order to merge logs of multiple applications, or to split one up in to multiple others, developers may override the currently used log by calling `DoodleDebug.setApplicationName(String)`.

Figure 3.9: Summarized and detailed renderings of booleans.

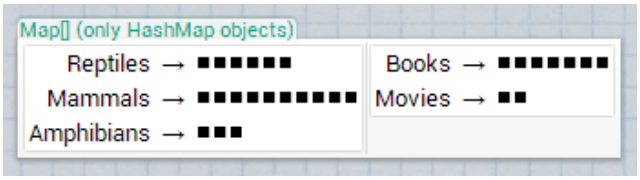


Figure 3.10: A doodled array of maps from strings to collections. The collections are rendered as summarized.

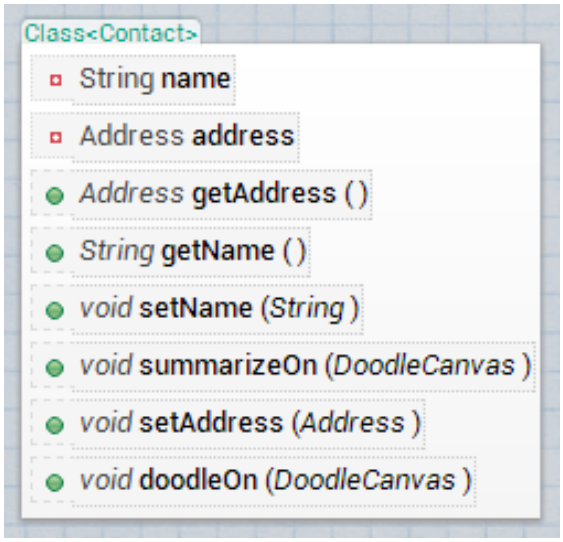


Figure 3.11: Rendering of a class object.

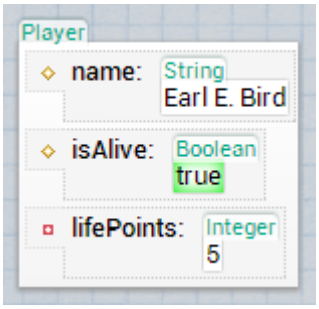


Figure 3.12: The standard rendering of an object, visualizing all of its fields.

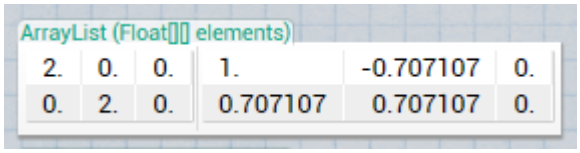


Figure 3.13: DoodleDebug's rendering of an array containing numbers, featuring decimal point alignment

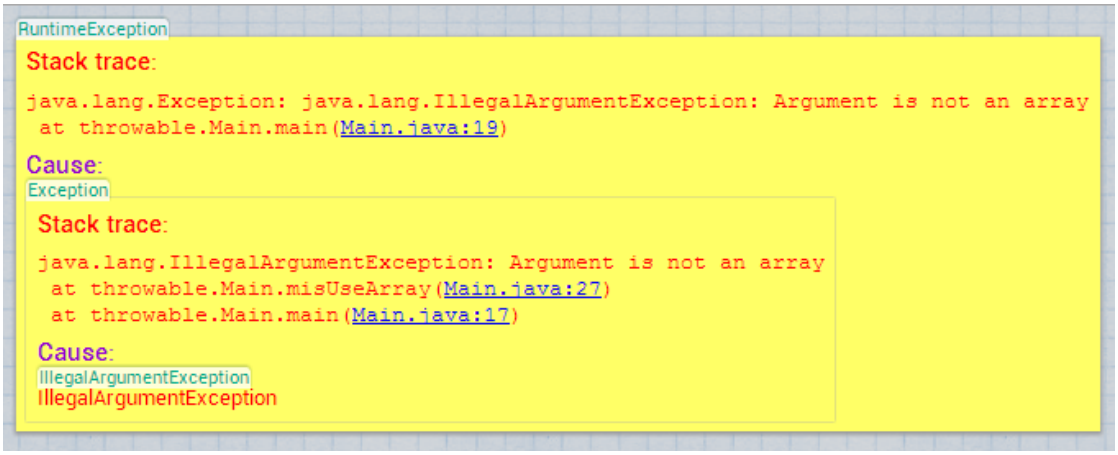


Figure 3.14: Rendering of an exception.

4
Leveraged Problem Solving

There are two major problems with current production software environments we're trying to tackle with DoodleDebug. Debugging is difficult and time-consuming [1], and monitoring a system in production mode is inconvenient, since in case of a failure, log messages are usually the only run-time information available to developers [21].

4.1 Debugging

Debugging in production environments can be tough. Attaching a remote debugger requires adequate security measures, like SSL tunneling, to prevent third parties from reading sensitive data. As an alternative, static text-based output may not always be verbose enough to close in on a certain bug [21].

One possible approach is to locally duplicate or imitate the production environment and try to reproduce the error. However, this is often not a realistic option due to privacy concerns or high costs for exact in-house replication of the live system [21]. Systematic problem solving through classical debugging becomes impractical, and alternatives need to be found for inspecting the dynamic state of the live program when the error occurs.

Since DoodleDebug captures the full run-time state of an object when it's doodled, developers can still acquire further information afterward without having to re-run the program.

In addition, this eases the resolving of so-called heisenbugs, which seem to change their behavior when being examined [8]. For instance, a bug caused by a race condition might disappear when a debugger pauses one thread at a critical point. On the other hand, DoodleDebug doesn't delay program execution. The order of doodles in the output can be used to derive execution order of tasks and detecting racing conditions.

4.1.1 Example: NullPointerException

Let's sketch an imaginary, but likely to occur situation illustrating the problems above. Assume we have a Java web application rendering content, e.g. from a database, to HTML. One particular page should list all entries of an address book from said database. However, when opening it in a browser, all we see is an

internal server error message (code 500). Our first problem handling step is to open the application log in order to check it for errors and warnings, where we find the lines in Figure 4.1.

```
Going to render AddressBook: contact.AddressBook@5b7b9223
Exception in thread "main" java.lang.NullPointerException
  at website.WebsiteRenderer.renderAddressBook(WebsiteRenderer.java:12)
  at website.AddressbookRenderExample.main(AddressbookRenderExample.java:17)
```

Figure 4.1: Sample console output after logging an `AddressBook` object and a thrown `NullPointerException`.

As we see, our application attempts to render an instance of `AddressBook` before a `NullPointerException` is thrown. Though we can see on which line of code the exception originates, we still have little semantic information about what is wrong, i.e. which object reference is null, whether it's related to missing data and if so, which data is missing. As a next step, we could open the `WebsiteRenderer` class on line 12 and try to guess which variable is null. However, we still wouldn't know the context at the moment before that exception is raised, and thus, we'd probably attach a debugger, set a breakpoint and start exploring the program's state right then. Notice that if our application is running on a production server, enabling remote debugging on the JVM may be unacceptable due to security concerns, so we'd need to run it locally with a local copy of the online database as well.

Now let's look at the same situation, but with logging statements replaced with `Doo.dle` ones and exceptions being doodled as well instead of just printed to the log. Instead of checking the textual log, we now open DoodleDebug's output page and see the doodles in Figure 4.2.

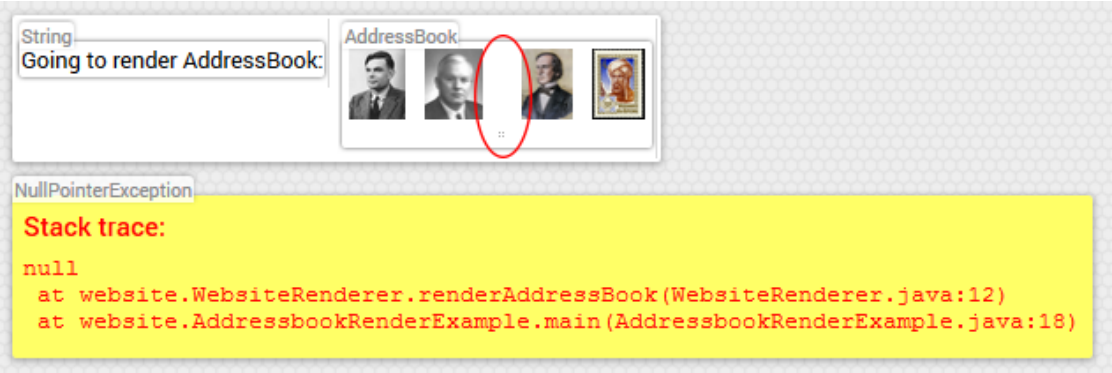


Figure 4.2: The same `AddressBook` object and `NullPointerException` as in Figure 4.1 visualized using DoodleDebug. The circled third entry is a indicates a missing object, i.e. a null reference.

We see there's something wrong with the third entry (circled in red) in our AddressBook object, since it shows no image. To inspect it further, we click on the address book, which opens the lightbox popup in Figure 4.3, revealing more details.

Thanks to semantic zooming, we've gained some verbosity and now see that the third contact in our address book is null, being likely to have caused above NullPointerException. We've found the semantic root cause of an erroneous web page with just a few clicks and neither had to open a log file nor read any code or do any debugging.

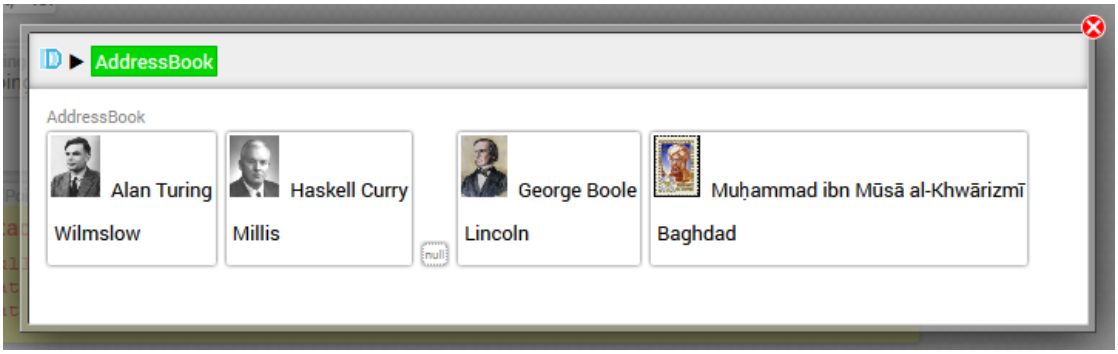


Figure 4.3: Detailed `AddressBook` rendering seen after clicking it in Figure 4.2.

4.2 Checking System Health

Maintainers of a production environment are interested in keeping it healthy and detect problems as quickly as possible. Such problems usually cause warnings or errors on the used logging system, so periodically checking log files can aid detecting bugs early in order to fix them.

Long-running applications like web servers usually store textual output into a simple text file for persistence. To check this file's content, one needs access to the server's file system where it's stored. This involves launching a suitable command line or visual tool, logging in, locating said file and displaying it. Once opened, the amount of aggregated content can make it hard to distinguish critical warnings and errors from less important output, since there is no differentiation in format as seen in Figure 4.4a. Furthermore, stack traces of deeply nested exceptions may either claim plenty of lines or be cut off at some point.

In contrast, DoodleDebug exposes its output on a web page, letting the user skip some steps to view it. Moreover, Throwables have a built-in special visualization, letting them clearly stand out from most other doodles as seen in Figure 4.4b. Based on DoodleDebug's semantic zooming feature, nested exceptions aren't fully printed to the main output screen. In fact, just the two outermost levels are initially visible, and clicking them reveals more deeply nested exceptions.

```
...
No printLogoImg property found under Node de
No printLogoImg property found under Node fr
No printLogoImg property found under Node en
Connecting to SMTP server...
Email sent to ananymousform@example.com
User logged in: Alan Turing
User changed password: Alan Turing
User logged out: Alan Turing
User logged in: Haskell Curry
User ordered a pizza: Haskell Curry
   with pepperoni
   with extra cheese
   with onions
   with olives
No printLogoImg property found under Node de
No printLogoImg property found under Node fr
No printLogoImg property found under Node en
java.lang.RuntimeException: java.lang.
    ↪ NumberFormatException: For input
    ↪ string: "2,3"
   at webform.FormParser.main(FormParser.
    ↪ java:36)
Caused by: java.lang.NumberFormatException:
    ↪ For input string: "2,3"
   at sun.misc.FloatingDecimal.
    ↪ readJavaFormatString(Unknown Source)
   at java.lang.Double.parseDouble(Unknown
    ↪ Source)
   at webform.FormParser.main(FormParser.
    ↪ java:34)
Connecting to SMTP server...
Email sent to pizza@example.com
User logged out: Haskell Curry
...
```

(a) Sample plain text output of a web server application. An exception can be hard to spot in between less important output if there is no strong visual differentiation.

(b) With DoodleDebug, exceptions are rendered in a way that visually highlights them, using flashy colors and a monospace typeface.

Figure 4.4: Comparison of an exception between other printed objects in plain text and DoodleDebug

5 DoodleDebug Internals

DoodleDebug includes a number of improvements over plain-text monitoring systems. While some of those advantages originate in the basic idea behind DoodleDebug and were there since DoodleDebug 1, others have been introduced in this iteration with the clustered, more modular setup.

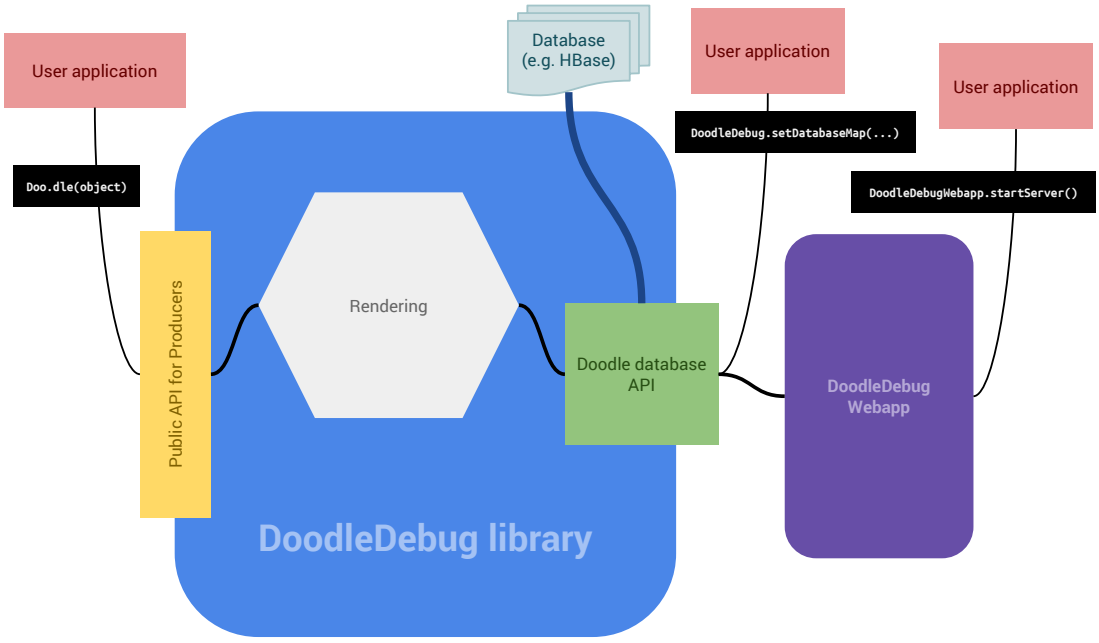


Figure 5.1: The DoodleDebug library with its public APIs and how it interacts with databases, the DoodleDebug webapp and user applications

As seen in Figure 5.1, the main DoodleDebug library provides two major public APIs: One for user

applications producing output and one for those reading it. The public API for producers has barely changed since DoodleDebug 1, it contains methods to write objects to the log and customize visualizations by providing plugins. However, a new API was introduced to manage the connection and handle accesses to used databases. Also, a standalone web application was created to eradicate dependencies to Eclipse and make DoodleDebug more widely applicable.

This section first describes general steps for doodling an object, a mechanism that hasn't changed in concept since the beginning. After that, newly introduced features and patterns in DoodleDebug 2 are described from a technical point of view.

5.1 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.

5.1.1 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 is found or all leaves are reached. The only type excluded from this search is the `Object` type, since it might be reached before some interface types.

5.1.2 Output

As output format, DoodleDebug uses HTML. This assures great compatibility among various platforms and facilitates integration into third-party applications. Previously, Eclipse's built-in browser component would be used to present the resulting page to users and handle communication needed for interactivity. As described before, all dependencies to Eclipse were removed as of DoodleDebug 2, among other things by embedding Jetty for a stand-alone HTTP server and switching client-server communication to WebSockets. Users now view DoodleDebug's output directly in a standalone web browser.

5.2 Doodle Database API

The newly introduced database API manages how and where doodles, plugins and other meta data are stored. It's used internally by DoodleDebug itself, but is also partially exposed to the public: User applications can perform actions like reading doodles or configuring custom database connections.

5.2.1 Reading Doodles Programmatically

The doodle database API exposes public methods for loading stored doodles. To do so, a third-party application may simply create an instance of `DoodleDatabase`, which will automatically connect to the currently configured underlying database. Relevant methods for reading are `hasNewDoodles()`, checking whether doodles exist that haven't been read yet, and `loadNewDoodles()`, loading new doodles and returning a `List<Pair<String, String>>`. Each doodle is represented by two strings, containing HTML and CSS information, respectively.

However, those strings do not consist of plain HTML/CSS, but are wrapped into a JavaScript snippet that will insert them in the right place inside the output page once executed. For more modularity, it would be desirable to store and return HTML/CSS snippets unwrapped and let client applications manage their insertion into output. This issue is also listed in the future work section (7.2.1.4).

5.2.2 Custom Database Connection

When not specified explicitly, DoodleDebug defaults to a local HBase as database. To connect it to any other storage system, DoodleDebug provides the abstract class `DoodleDatabaseMap`, which is a partial implementation of the `Map<String, T>` interface. At run time, users may reconfigure the database connection by calling `DoodleDebug.setDatabaseMap(Class<? extends DoodleDatabaseMap>)` with a custom implementation class.

5.2.3 Clustered Data and Computation

One requirement was that DoodleDebug 2 should scale well for many instances of the same or different applications running parallel and logging to the same database. Availability and response time of the system should remain stable with any amount of doodling traffic, possibly requiring more hardware if necessary.

We decided to use HBase as default database for persistence of Doodles and other meta data. That way, we outsource part of the problem, since HBase includes a clustered mode and is widely used in industry.

When many application instances generate doodles at the same time, a lot of computation load needs to be handled for rendering objects to HTML. One option would have been to let client applications simply submit raw objects to be doodled to a central service, which would do all computation work. In case of high traffic of doodles being submitted by clients, a single server node running a single-threaded application would at some point fail to handle all requests in time. Computation would need to be distributed among a clustered network of processors or computers, which in turn required implementation of a sophisticated paradigm like MapReduce [5].

Instead, we decided on fat clients. The library used by client applications (Figure 5.1) includes the whole rendering mechanism. It generates HTML code from objects and submits it along with necessary meta data to a central database, which is by default an HBase, but can be configured otherwise. Since all rendering work is done on the client side, there is no central application carrying out heavy computation that might require clustering. The only resource shared among clients and possible bottleneck is the central database, which by default is an HBase and thus ready for big data and scaling.

The same library in Figure 5.1 provides another API to read doodles as explained in 5.2.1. It's necessary to have the whole rendering mechanism available when reading doodles. When doodles are inspected by a user, inner parts are dynamically generated as the user navigates. In summary, there is just one DoodleDebug library providing two APIs, one for writing and one for reading.

5.3 The Webapp

Though DoodleDebug's output was always based on HTML, it only worked as an eclipse plugin in DoodleDebug 1. As of DoodleDebug 2, we decided to eliminate all dependencies to Eclipse to achieve greater flexibility. We built a webapp that utilizes our doodle database API for reading doodles and displaying them on a web page.

5.3.1 Live Updates via WebSockets

DoodleDebug's dynamic nature with instant appearance of new doodles and users able to inspect them requires bidirectional communication between the front and the back end of the application. Previously, output was displayed in a browser component embedded into eclipse which could be controlled programmatically. As the front-end consisted of the embedded browser's page context, and the back-end was eclipse plugin code, communication was quite simple and straightforward. HTML could be directly passed to it and displayed, JavaScript snippets directly executed and listeners attached to events like location changes.

With the browser no longer being controlled from the outside, a new way of font-back-end communication was required. Now, a WebSocket[1] connection is maintained between the webapp and each client page, emulating communication in both directions.

The WebSocket protocol is one of the major innovations coming with HTML5, allowing applications to have a steady connection between two nodes, e.g. client and server of a web site. In contrast to communication via HTTP, no more repetitive client-side polling is needed and overhead is reduced since the HTTP header is omitted. As of 2014, WebSockets are widely supported by web browsers and thus reasonable to use as part of a developer tool.

5.3.1.1 Server to Client

In DoodleDebug 1, the server-side application controlled the browser and thus could simply execute a JavaScript snippet inside. Recurrent operations, like appending doodles to the output, were triggered that way. For instance, adding a doodle's HTML code worked by executing a snippet of the form `addCode('<p>test</p>')`, where `<p>test</p>` is that doodle's HTML representation. Such functions as `addCode` were all pre-implemented on the client side, i.e. in the page context.

Websockets are message-based, i.e. one node sends a message to the other, consisting of a key and a value. We simply replaced JS function calls on the server side by socket messages with a function name as key and argument as value. On the client side, a listener works as a proxy and invokes those functions with received arguments. Using our above example, a Websocket message with key `addCode` and value `<p>test</p>` would be sent to the client, which executes the corresponding function.

5.3.1.2 Client to Server

Communicating from client-side code to the server application had been tricky inside an embedded browser, since a website is basically running inside a sandbox. As a workaround, we had introduced pseudo-protocols in DoodleDebug 1: Client-side JavaScript code triggers a location change, as if the user would navigate to a new page. But instead of a hyperlink like `http://example.com/`, the target location looks like `key:value`, where `key` is one of several pre-defined pseudo-protocol ids and value the content of a message. For instance, a user might click on a doodle to inspect it, e.g. the one with id `23`. The client-side JavaScript code would now trigger a location change to `doodledebug:23`, virtually representing a request for inspecting doodle number 23. The server would listen to the embedded browser's location change events, decode the message and take appropriate actions. In our example, this would mean to render the clicked doodle's next level of semantic zoom and insert it to the output.

Replacing this mechanism with websockets was straightforward: We replaced the location change calls by messages to the remote socket, using `key` as message key and `value` as message content.

[1] Specification: `https://tools.ietf.org/html/rfc6455`

5.4 User Classes

When inspecting a doodle (see 3.2), new doodles popping up in a lightbox are generated on the fly. For instance, when an addressbook's doodle is visible in the output and the user clicks on a certain contact inside, DoodleDebug will foreground that contact by rendering it exclusively with more detail. As this rendering is done dynamically, objects like this contact need to be somehow stored and revived when needed. To deserialize an object from database, its own class file and dependencies need to be loaded into the current JVM. Furthermore, a custom rendering plugin for the object's type may exist, so this one needs to be loaded as well to always get the same rendering results. Since doodle inspection can happen anywhere and at any time, all classes required for an object's deserialization and rendering need to be stored in a database.

5.4.1 Deriving Dependencies

To resolve dependencies of objects to be doodled, DoodleDebug uses ASM[2], a Java bytecode analysis and manipulation framework focusing on performance. For manipulating and reading static program data, ASM uses the visitor design pattern [6]. Dependencies are mined by using a custom `Remapper` that doesn't change anything, but keeps track of all relevant types, like fields, while passing them.

Third-party types are detected by their package name: By convention[3] [7], we assume that all types inside a package matching `ch.unibe.scg.doodle.*` are related to DoodleDebug and all others come from a third party.

5.4.2 Storage and Instantiation

In the used database, a separate table is created solely for third-party classes. Their file's content is stored in base64 encoding, indexed by canonical name. As a consequence, one class file will never be stored multiple times. There is no versioning mechanism yet; if a class file has changed since the last save, the old one will simply be overridden, which might lead to unexpected results in the output. However, fixing this problem is not trivial and part of our future work list (7.2.1.2).

When required for rendering an object, classes are read from the database, stored into the OS' temporary directory and instantiated through a `URLClassLoader`.

[2] `http://asm.ow2.org/index.html`
[3] `https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html`

6 Validation

As DoodleDebug was already validated in terms of usability in previous work [16], we'll focus more on technical aspects in this section, especially related to changes coming with DoodleDebug 2.

6.1 Performance

The fact that DoodleDebug features more sophisticated renderings than classical logging solutions suggests that its computation cost for visualizing a certain objects will be notably higher. This raises the question, how much of a performance penalty is to be expected, and where the bottlenecks in the doodling pipeline are. This section documents performance measurements we conducted and draws conclusions from acquired data.

6.1.1 Setup

We wrote a small application for repetitive logging of an object using one logging mechanism at a time. A total of 8 different logging setups were combined with two different objects to be logged. For a test run, the object in question would be logged 10^n times, with $0 \leq n \leq 6$. After each test run, the application was stopped in order to make sure potential initialization processes by logging systems needed to be executed every time for equal preconditions.

All HBase-related tests were executed on the system the HBase was running on. In the case of a clustered HBase, the running machine held the role of the HBase master node. Furthermore, log4j's logging file was configured to a location on the same drive as standalone HBase's persistence directory.

Execution time was measured by reading `System.nanoTime()` right before the first and right after the last logging call. To prevent random outliers, each test was run three times and the median result taken into account. Test runs that had not terminated after one hour of execution were aborted and left blank in the results.

Tests were carried out on a machine running Linux Mint 17.1 with an AMD FX-6100 (6 cores, 3.30GHz) CPU, 8 GB DDR3-1333 RAM as main storage and a Samsung SSD 840 Basic for mass storage.

6.1.1.1 Logged Objects

The first set of performance tests logged an object of type `NullPointerException`, directly instantiated by using the constructor `new NullPointerException()`. The resulting doodle can be seen in Figure 6.1.

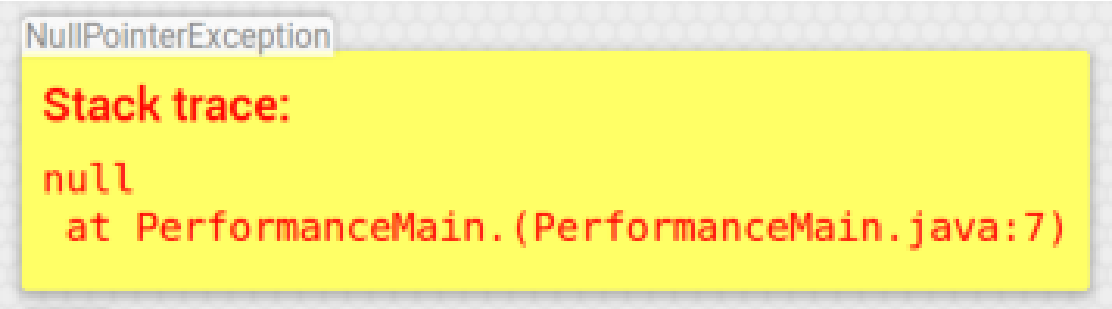


Figure 6.1: Doodle of the `NullPointerException` used for logging performance tests. Since it was instantiated directly and never thrown, the stack trace consists of one single item only.

The second object used was a `String` containing 121 characters. It can be seen in Figure 6.2 in its doodle representation.

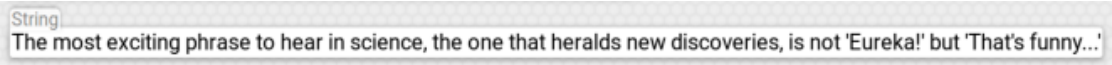


Figure 6.2: Doodle of the `String` used for logging performance tests.

6.1.1.2 Logging Configurations

As output channels, we used the following configurations:

- **DoodleDebug with clustered HBase**: Doodles and other meta data are persisted to a clustered HBase containing three nodes. The system running our test acts as master node.
- **DoodleDebug with standalone HBase**: Data is persisted to a local, standalone HBase. This means that no HDFS[1] is used, but data is persisted against the local file system.
- **DoodleDebug without HBase**: Instead of connecting to an HBase, data is persisted to RAM, which prevents expensive I/O operations. This is achieved by providing a custom `DoodleDatabaseMap` which maintains a static `Map` keeping all (serialized) data.
- **DoodleDebug without any database**: The same as before, except that the step of serializing objects for databases is skipped too and objects are directly stored to the Java map.
- **Log4j with stout and file**: Objects are logged using a log4j setup which stores output into a log file and prints it to stout (the console).
- **Log4j with stout**: Log4j only uses stout as appender (output channel).

[1] `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

- **Log4j bare**: Log4j is configured with no appenders at all. This should reveal the cost of log4j's core, similar to DoodleDebug without any database.
- **`System.out.println`**: Objects are directly printed to stout (the console).

  Additional measurement set-ups (DoodleDebug with Jackrabbit) are discussed in 6.2.1.

6.1.2 Results and Conclusions

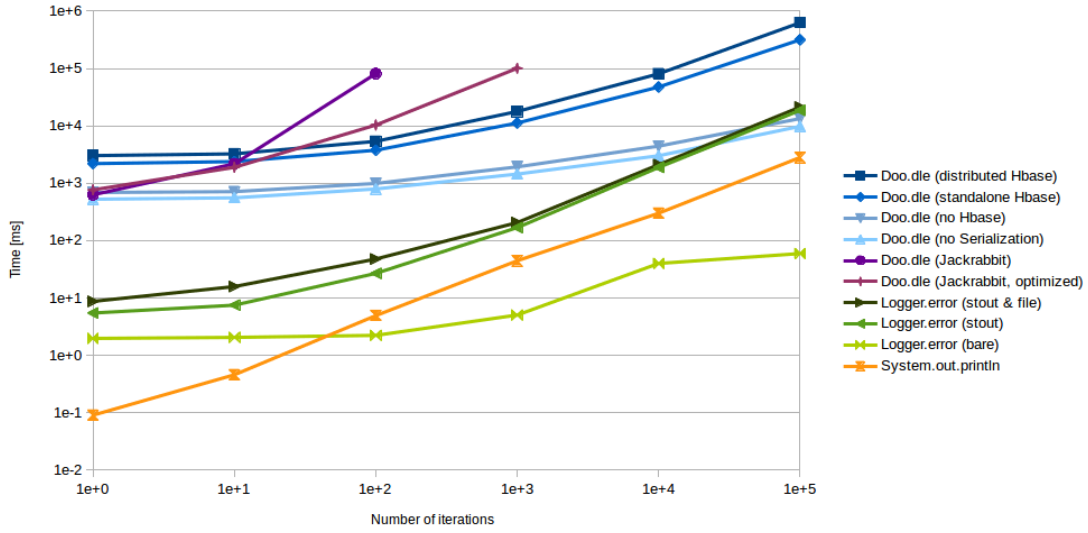


Figure 6.3: Performance while logging a simple exception object. Both axes use a logarithmic scaling. Raw data to this diagram can be found in the appendix (Table 8.1).

As expected, DoodleDebug cannot keep up with text-based logging system in terms of performance while attached to a local, standalone HBase. However, two noteworthy observations can be made.

6.1.2.1 The Bottleneck

When not attached to HBase, performance increases dramatically, even while serialization is still active. As seen in Figure 6.3 and Figure 6.4, serialization does not substantially elevate computation time. Therefore, the bottleneck is obviously communication with HBase.

Theoretically, this bottleneck could be reduced by switching to a database with a higher throughput, considering measurements without a database as a lower bound of computation time. However, this would require a database with significantly better write performance than HBase, which most popular ones cannot provide [4][18]. Another approach to reducing database-induced delays would be to execute operations asynchronously, as discussed in future work (7.2.2.6).

The slightly higher computation times using a clustered HBase implies that HDFS either doesn't provide a performance advantage over standalone file system storage, or that those advantages are drowned out by other additional computation costs due to the clustered setup. However, this difference appears rather small and bearable, considering that we have a distributed, randomly accessible database in the latter case.

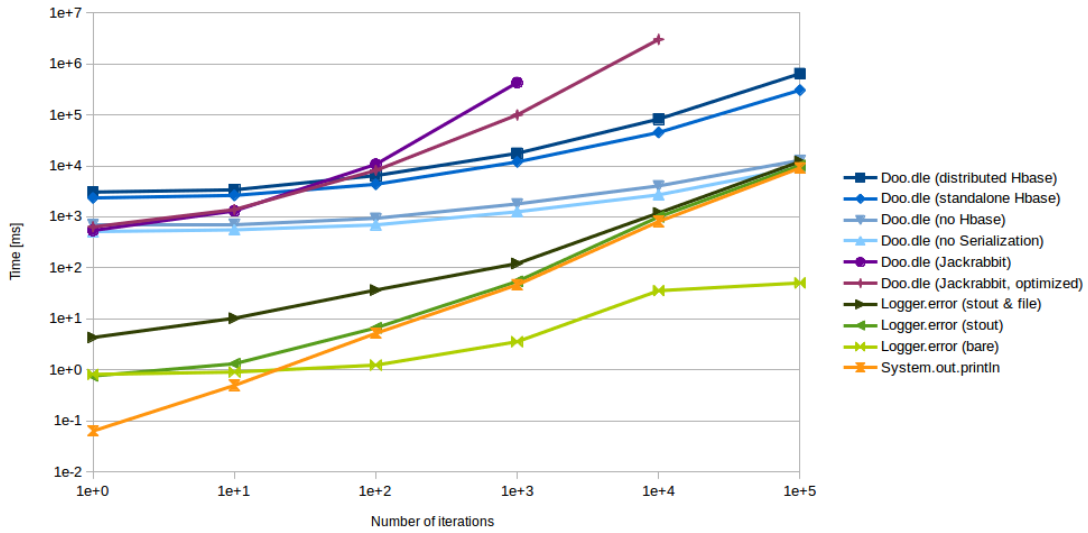


Figure 6.4: Performance while logging a string object. Both axes use a logarithmic scaling. Raw data to this diagram can be found in the appendix (Table 8.2).

6.1.2.2 DoodleDebug Scales Well

Both, Figure 6.3 and Figure 6.4, show that DoodleDebug's computation cost is far from linear. The reason is a relatively high cost for initialization, happening with the very first doodle being produced. For instance, when doodling a String object, DoodleDebug checks what renderings are available for this type (locally and in the database), then memorizes the most prioritized one henceforth uses that for all string renderings.

In Figure 6.3, we can observe that DoodleDebug without HBase required slightly less time to render 100'000 objects than log4j, even when including serialization. However, this might be misleading, as it doesn't actually indicate that DoodleDebug performs better than log4j. While log4j was configured to save its output to log files and thus did expensive IO operations, DoodleDebug's HBase connection was replaced by simply saving data to RAM.

6.2 Integration into Existing Software - Magnolia CMS

Since DoodleDebug 1 was an Eclipse plugin, it would only work while the application to debug was in development stage and running inside an Eclipse instance. As we moved away from that dependency and DoodleDebug is available as an independent library, embedding it into any application became possible.

Magnolia[2] is a complex, mostly enterprise oriented open source Java CMS, aiming on modularity and performance. The cost of these properties is a vast amount of classes present in the JVM, making it demand a large amount of memory, even when idling. Magnolia comes in the form of Tomcat webapps and is divided into modules, each providing a jar for that webapp. We created a Magnolia module to integrate the mechanism of logging with DoodleDebug, such that developers could use this one instead of log4j, which is integrated by default.

Besides allowing users to access the DoodleDebug API, we implemented a handful of Magnolia-specific features that plug into DoodleDebug as third-party elements and exhibit its modular nature.

[2] `http://magnolia-cms.com/`

6.2.1 JCR as Custom Database

Magnolia utilizes the Java Content Repository specification[3] as a database abstraction layer. All data is stored and managed based on that, including graphical interfaces for end-users to view and interact with them. As default implementing database, Magnolia ships with Apache Jackrabbit[4].

JCR uses trees of nodes as data structure. Starting with one root node, each node may have multiple child nodes, recursively. For storing data, nodes may contain properties, which consist of a key and a value, like a string, number or binary content.

To avoid being dependent on an HBase, we wrote a class to make DoodleDebug connect to JCR instead. This class simply extends `DoodleDatabaseMap<T>`, which means it needs to implement some methods of the `Map<String, T>` interface in a way that map contents persist, i.e. outlive the object itself.

6.2.1.1 Implementation and Performance

The most obvious reduction of map data to JCR is straightforward: Each map corresponds to a JCR node, and each key/value pair is represented as a property of said node. This leads to a tree with a limited number of nodes, which receive more properties with new objects being doodled, as shown in Figure 6.5a. This structure was used for the first implementation and performance measurements carried out. The results, shown in Figure 6.3 and Figure 6.4, suggest that Jackrabbit performs well for few doodles, but suffers from an exponential slowdown. Apparently, creating new properties on nodes becomes slower with more properties already being present.

In order to keep property numbers per node low, we implemented a second version which adds an additional separate child node for each property, as described in Figure 6.5b. Figure 6.3 and Figure 6.4 show a notable increase in performance, though there is still an exponential growth in computation time to be observed. The Jackrabbit wiki[5] explains that big sets of child nodes per node negatively affect write performance. As a solution, deeper hierarchical nesting should be considered. In the specific case of DoodleDebug maps, each digit of a key could be represented as a jcr node, automatically leading to a deep tree. For instance, a tuple with key `4321` from a map named `clickables` would be stored in the node `/doodledebug/clickables/4/3/2/1`.

6.2.2 Impact of Full DoodleDebug Integration On Performance

While the above performance measurements provide numbers on computation time with DoodleDebug attached to Jackrabbit, they don't give insight into how a full switch from log4j to DoodleDebug would impact overall Magnolia performance under realistic conditions. As a repeatable procedure with a fair amount of logging traffic, the start up phase was chosen as reference to acquire a relevant comparison.

Since Magnolia consists of several modules which all may contribute to the log, replacing all logging statements with `Doo.dle` ones would have been an ineffective approach. Implementing a custom log4j appender in order to redirect logging requests to DoodleDebug might have introduced additional overhead skewing results and brought along other conceptual issues, as discussed in 6.2.6. As an alternative, an unmodified Magnolia was started up and the number of items in the log counted. Based on previous performance measurements, the additional delay caused by DoodleDebug was estimated. Time savings due to removal of log4j events were neglected since they are much smaller than additional time consumption by DoodleDebug.

A full startup under default configuration in Magnolia 5.4.1 took 67,175 ms (median of three runs) and generated 559 log4j events. Using a linear interpolation between previous measurements of 100 and 1000

[3] `https://www.jcp.org/en/jsr/detail?id=170`
[4] `https://jackrabbit.apache.org/jcr/index.html`
[5] `http://wiki.apache.org/jackrabbit/Performance`

```
* doodledebug
  * clickables
    - 0 : [...]
    - 1 : [...]
  * doodles_html
    - 0 : [...]
    - 1 : [...]
...
```

(a) First, most obvious implementation - each node under `doodledebug` represents a map, each property is one tuple in that map. With more objects being doodled, no new nodes are generated, just new properties added to existing ones, leading to a very flat tree in the long term.

```
* doodledebug
  * clickables
    * 0
      - 0 : [...]
    * 1
      - 1 : [...]
  * doodles_html
    * 0
      - 0 : [...]
    * 1
      - 1 : [...]
...
```

(b) This second implementation adds a new node for each map tuple, and stores data into a property of it. As a result, there are no more single nodes with huge amounts of data, which slightly increases performance of Jackrabbit.

Figure 6.5: Schematic excerpts of resulting JCR trees with the two different implementations of `DoodleDatatbaseMap`. * marks a node, − marks a property.

(see 8.1.2) doodles, an additional delay of 224,129 ms for the naive implementation and 54,812 ms for the optimized one would be expected. However, since the measured times appear to grow exponentially, a linear interpolation would exceed the actual expected time and should only be seen as an upper bound. On the other hand, a proportional continuation of measured time for 100 doodles could be referred to as a lower bound, which would would be 60,031 ms for the naive and 45,487 ms for the optimized implementation. In any case, the impact on start-up time would be heavy, making it a questionable solution for real-world users. Further optimization through smarter organization of data in JCR as discussed in 6.2.1.1 might solve this problem.

Assuming the same scenario as before, but with a standalone HBase connected to DoodleDebug, would result in an estimated additional time consumption of 8,228 ms. In contrast to Jackrabbit connections, execution time with HBase didn't grow exponentially, but slower than linearly. As a consequence, linear interpolated values should be considered lower bounds, and proportional continuations from lower data points as upper bounds. In this case, based on the measurement of 100 doodles, an additional computation time of 24,321 ms could be calculated as an upper bound. Thus, the actual delay caused by DoodleDebug would make up notably less than 50% of the original start-up time in any case, which could be regarded as a reasonable trade-off for the gained benefits.

6.2.3 Reading the Log

As mentioned before, DoodleDebug 2 consists of two parts, one that receives data form the user's application to save it to a database in a standardized way, and another one which reads from there to present it to users. While embedding the former into an application is fairly straightforward by including appropriate libraries and setting up a database, integration of the latter is more complex. It consists of a full-fledged, standalone web server and features highly dynamic pages through excessive communication with clients via WebSockets.

Having such a log reader directly integrated into an application's user interface appears desirable, as users don't need to open a different website in order to read the log. Magnolia's user interface consists of

apps, which of each serves one main purpose, similar to smartphone operating systems. Apps are grouped by topics and can be hidden for certain users, allowing us to only let developers and system administrators access the DoodleDebug log.

Even though the primary use case for our DoodleDebug web app is to use it as a standalone application and start it via command line, there's a programmatic hook, `DoodleDebugWebapp.startServer([port])`, for launching it from another application. We're using a custom magnolia module which listens for a startup event and launches the DoodleDebug webapp on a separate port. The magnolia app itself is configured to simply display this page embedded into the UI. Figure 6.6 shows the resulting opened app with some sample doodles.

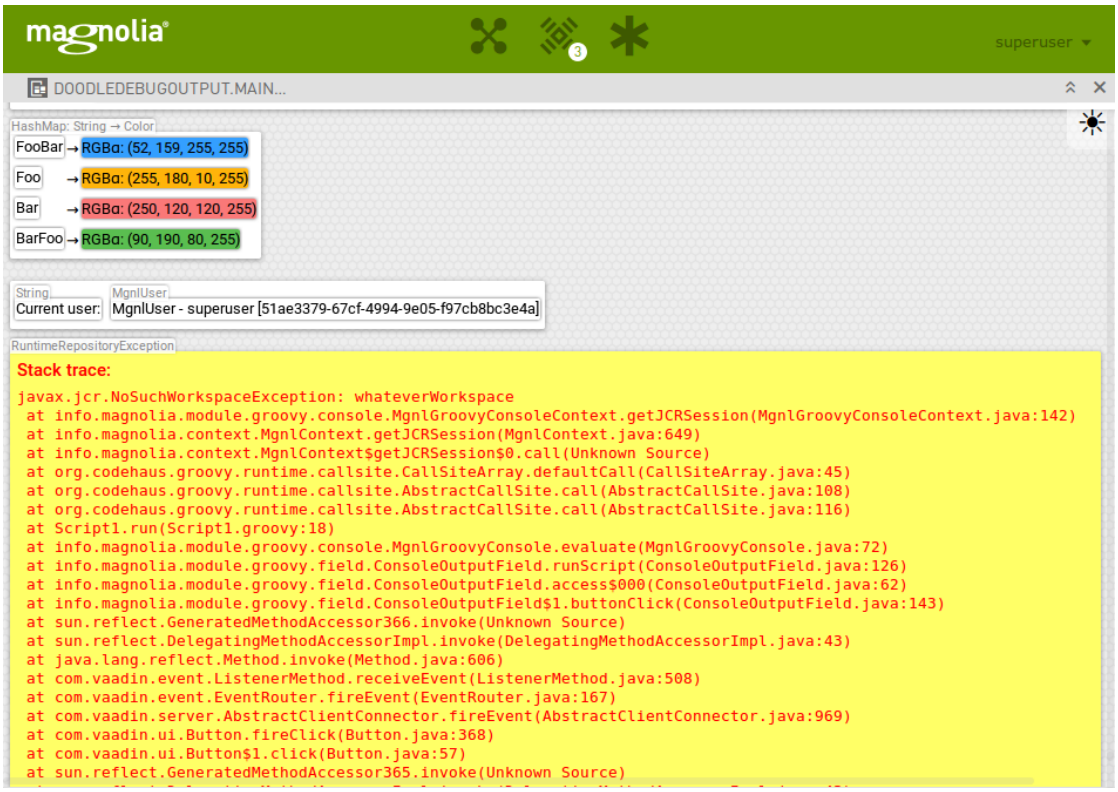


Figure 6.6: The DoodleDebug output embedded into a Magnolia app.

6.2.4 Multiple Instances

Magnolia is able to run on multiple instances, which are all separate Tomcat webapps. By default, there is one so-called author instance, where editors manage and review content that's not publicly accessible yet. Once it's ready for publication, data is sent to one or more so-called public instances which serve web sites to internet users.

Those instances may run in the same Tomcat, and thus write their output into the same log file, or run in different Tomcats and write to different logs. However, DoodleDebug allows developers to actively decide which way to go. Doodle logs of multiple instances can easily be merged by connecting them to one and the same database, e.g. a clustered HBase. On the other hand, two different instances may declare unique log names, virtually separating their doodle logs for readers (3.5.3).

6.2.5 Limitations

When trying to doodle an object that's heavily entangled into the system, e.g. an instance of `MgnlContext`[6], an `OutOfMemoryError` may occur. The reason for this issue is that DoodleDebug tries to serialize an object and all of its dependencies for later reconstruction when inspecting its doodle in the log, which naturally consumes more memory with bigger object graphs.

In many cases, such an error may cause the program to crash completely, since recovering from it is difficult as handling it might lead to its re-occurrence [2].

As discussed in 7.2.1.1, overcoming this issue programmatically is not trivial. As a workaround, users may simply increase the amount of memory dedicated to the JVM they're running.

6.2.6 Migrating Logging Code to DoodleDebug

By default, Magnolia uses log4j for managing application output, usually through the slf4j API, and thus it's obvious to do the same for third party modules. For migrating logging statements in existing code to DoodleDebug, there are mainly two possible ways to go: Write a custom log4j appender or replace all code snippets in question.

Writing a custom log4j appender which doodles each item to be logged seems like an elegant solution, since existing code would remain untouched, and the logging to DoodleDebug could be arbitrarily switched on and off. However, there are two drawbacks to this approach.

On the one hand, DoodleDebug internally uses log4, which might lead to recursive invocation of another log task, ending in an infinite loop. This would require a detection of circular calls, which exclude DoodleDebug-induced log requests from being propagated.

On the other hand, slf4j works with string based messages. But one of DoodleDebug's strengths is its ability to visualize objects based on their type and storing their run-time state for later inspection. Just doodling textual representations would boil DoodleDebug down to the equivalent of a simple textual log file.

Replacing all log4j related code snippets by appropriate `Doo.dle` statements would obviously incur higher costs, but give developers the opportunity to make use of all features. Simple tools like search and replace could be utilized for quick, naive replacements over large amounts of code. However, since log4j and DoodleDebug don't exclude each other and can be used simultaneously in the same application, a continuous migration should work without difficulty.

6.2.7 Required Effort

With the goal of more modularity, such integrations should be possible with a reasonable amount of effort and only little changes or additions in code. The efforts required to achieve above features contained two parts: Java coding and configuration of Magnolia.

Coding We work with a total of two custom Java classes. One is `JcrDatabase`, which extends `DoodleDatabaseMap` and implements methods for persisting a map to JCR. The other one is `DoodleDebugModule`, an class that's automatically generated by the maven archetype used to generate the module. It manages the module life cycle and contains `start` and `stop` methods executed on application start up or shut down, respectively. We added code for registering our JCR database in DoodleDebug as described in 6.2.1, and for starting/stopping the webapp for reading doodles.

[6] `https://nexus.magnolia-cms.com/content/sites/magnolia.public.sites/ref/5.3.6/apidocs/info/magnolia/context/MgnlContext.html`

Configuration The only configuration required was the definition of a Magnolia app for displaying doodles, which simply embeds the output page of our webapp, as described in 6.2.3.

All those efforts are straightforward and worked well with intended paradigms of both, Magnolia and DoodleDebug. Everything was achieved by adding code or configuration, i.e. no changes in the existing applications or other hacks were required. A developer familiar with both worlds could probably complete everything in a few hours.

7 Conclusion and Future Work

In this chapter, we look back at the second iteration of the DoodleDebug project, what has been achieved, where we are standing now, and what could be improved in future iterations.

7.1 Conclusion

In previous work, the concept of DoodleDebug had been introduced. The main idea was to create a tool for developers to understand program states by combining the most useful features of existing solutions in a smart way. The implementation at that time focused on the development process - it was based on the Eclipse IDE and data would be volatile with no external database attached.

During this second iteration, DoodleDebug was detached from its dependencies on Eclipse and enhanced to a more adaptable framework. It has been split up into a part producing doodles and a second one being able to read them. Both of these can be connected to any kind of database as specified by users through the framework. On the one hand, developers may still use it as a debugging tool while coding on an application, for instance by persisting data to the local file system. On the other hand, DoodleDebug can be utilized for large-scale software running on multiple nodes. Several instances of an application may be configured to send doodles to one and the same database, for instance through the built-in HBase connection. Maintainers of the system may configure the DoodleDebug webapp to read from said HBase and get insights about current or historical program states.

In terms of a logging solution, DoodleDebug silhouettes itself against text-based tools especially through its ability to conserve run time states of an object, and its following support for post-mortem inspection. As a common use case, additional technical context of an observed error can be acquired without having to reproduce it locally, which often would be difficult and expensive.

However, persisting snapshots of object states causes fairly high resource costs. Computation time increases significantly with many objects being doodled, especially compared to other logging frameworks. While serialization is already expensive, another critical factor is the database being used for persistence. Big amounts of data need to be stored in real time and should be randomly accessible afterwards. DoodleDebug currently runs completely in the thread where a visualization is triggered, in order to assure correct order of doodles in the output. As a consequence, the main program is being blocked during that time.

While the principles and features coming with DoodleDebug constitute noteworthy innovations in terms of logging solutions, it still suffers from several shortcomings to be resolved before using it in real world production systems. Those issues and possible solutions are listed in the future work section (7.2).

7.2 Future Work

During conceptual design, implementation and validation of DoodleDebug, several issues and possible features were noted. They've been grouped into problems in the current version of DoodleDebug and potential new features.

7.2.1 Solving Current Problems

This section lists issues with DoodleDebug that have been observed during this work, and suggests possible solution approaches.

7.2.1.1 Smarter Serialization of Objects

Currently, deep copies of doodled objects are stored into the database for later reconstruction, using XStream for serialization. Creating deep copies with XStream leads to problems when operating on objects which are strongly entangled into big software. Due to growing memory needs with more dependencies, an `OutOfMemoryError` may occur.

While solving this issue is relatively easy for users by just increasing the JVM's memory, logging software like DoodleDebug should under no circumstances break the main application.

One option would be to find a serialization solution that features a lower memory footprint. Another approach would be to introduce smarter serialization, e.g. only serialize up to a certain number of hops on the object's dependency tree. However, this could lead to new problems, like missing data when trying to inspect a doodle.

7.2.1.2 Versioning of Classes

A DoodleDebug log can be arbitrarily long-living, which leads to the problem that user-provided visualizations might change over time. Technically speaking, classes containing rendering information may change internally while keeping the same canonical name, resulting in a collision inside the class map kept by DoodleDebug. Neither keeping the old one nor replacing it with the newer version will yield fully satisfying results. Instead, old doodles created before a certain change in visualization code should use the old version and vice versa to assure consistency.

Technical Difficulties One of the biggest problems is that a class can only exist once in a Java application with the same canonical name. Attempting to change a class' name would lead to probably insurmountable problems, like broken dependencies. Instead, classes could be replaced by a different version for the time of a specific rendering.

7.2.1.3 Old CSS

A related problem is that each doodle has a snippet of CSS code associated, which gets inserted into the output page along with its corresponding HTML code. On the one hand, this is redundant when doodling the same type with the same rendering multiple times. On the other hand, a new version of a type's CSS interferes with the old one.

A possible simple solution would map each CSS snippet to its doodle, based on a unique identifier, e.g. using an HTML class attribute. However, a more efficient way would be to detect changes in rendering, using above discussed class versioning system, and only load new CSS when a new version is available. Doodles would then get an HTML class attribute referring to a rendering version:

```
<div class="doodle string version-0">
   <p>Hello World</p>
</div>
<div class="doodle string version-1">
   <p>Hello World</p>
</div>
```

and CSS snippets would address one version each:

```
.string.version-0 p {
   font-family: monospace;
}
.string.version-1 p {
   font-family: sans-serif;
   color: blue;
}
```

7.2.1.4 Storing Doodles Unwrapped

Currently, a doodle is stored in the database as a snippet of HTML and another one of CSS, both wrapped into JavaScript. The historical reason for this is discussed in 5.3.1.1.

This wrapping limits flexibility for third-party applications trying to integrate DoodleDebug into contexts different from the standard webapp. Pure HTML and CSS snippets would be more flexible to handle and thus introduce more modularity.

7.2.1.5 Security

At present, DoodleDebug doesn't use any kind of security mechanism to prevent third parties to read data. There are two parts to be protected from unauthorized access: The database containing all doodles and meta data, and the webapp exposing rendered output.

Database As of version 0.92, HBase supports SASL authentication of clients accessing it[1]. Thus, given a central HBase configured like that, DoodleDebug would only need to provide an API for users to deposit login data when reading and writing doodles.

Webapp Currently, the DoodleDebug webapp neither requires a login nor sends data in an encrypted way. In order to prevent random users knowing the webapp's URL from accessing output pages, jetty's built-in authentication and authorization features[2] could be utilized. On top of that, user account and access right information could be configured via an additional web interface and persisted in the same database as doodles and other meta data.

However, a simple login mechanism does not solve the problem of eavesdropping attacks, where a third party logs all traffic between server and client, and can therefore reconstruct data or even passwords, as long as they're sent as plain text. Also, malicious third parties may perform a man-in-the-middle attack, which means they mock the server towards the client and vice versa, and thus can manipulate all

[1] http://hbase.apache.org/book.html#hbase.secure.configuration

[2] http://www.eclipse.org/jetty/documentation/current/configuring-security-authentication.html

communication. To counteract this problem, a secure communication protocol like HTTPS is required. On the one hand, this prevents man-in-the-middle attacks by safely identifying a server from a client's point of view, using certificates from a trusted third party instance. The drawback here is, that such a certificate needs to be registered first at a so-called certificate authority. On the other hand, HTTPS encrypts all data for their transport between client and server in both directions.

7.2.1.6 Concurrency

Since doodling is executed synchronously, we did not consider concurrency so far. When accessing DoodleDebug from multiple threads at the same time, problems may occur, especially related to its database connection. The simplest way to prevent such problems would be to run everything synchronized using one global monitor. However, when introducing parallelism for performance reasons, a more sophisticated concept needs to be formed.

7.2.2 Desirable Features

DoodleDebug's flexible nature clears the way for new features not feasible with classical logging or debugging systems. This section outlines possible, but non-trivial feature ideas collected during development. It discusses difficulties coming with them and possible approaches.

7.2.2.1 Meta Information About Doodles

There are situations where a developer sees some piece of content in the output, e.g. an error message, and would like to locate the line of code which printed this error in order to have a good point to start analyzing and debugging. In DoodleDebug, this could be realized by tracking back the stack trace to the `Doo.dle` call, then integrate class name and line number information in each doodle's visual representation.

Other meta data of interest for developers might be the date of a doodle or name/IP of the machine generating it (in a clustered setup).

7.2.2.2 Categorization of Doodles

Log4j uses built-in categories for users to categorize messages by their level of importance. Based on this, logging output can be filtered in order to adjust it to specific use cases, like debugging, where a verbose output is desirable, or just simple monitoring, only showing warnings and errors. In DoodleDebug, we could introduce a similar system, or even a more powerful one with features like multiple categories (tags) per doodle or arbitrary importance levels, based on a floating point number. One huge benefit of HTML-based output is that categories could be differentiated more powerfully, for instance by mapping each one to a color and framing them with it.

7.2.2.3 Versioning of Doodles

As stated in previous work, one major use case of `System.out.println` and DoodleDebug is comparison between objects or an object's different versions by printing them one after another [16]. To enhance this type of debugging, DoodleDebug could compare objects and highlight differences in the output. Comparison would most probably happen between consecutive prints of the same type and could either be based on raw objects or generated HTML code of doodles.

7.2.2.4 Command Line for Inspection

We previously argued that a big advantage of debuggers over `System.out.println` is the ability to inspect any object to arbitrary detail at a certain point of execution, or even evaluate custom code expressions [16]. Since a snapshot of the object behind each doodle is stored in the database, inspecting that appears feasible to a certain point. Instead of working with plain java expressions, a DSL like groovy[3] could be considered for more convenience.

7.2.2.5 Parallelism

Currently, DoodleDebug runs completely synchronous in the thread where the `Doo.dle` call is made. A drawback of this method is the performance penalty caused by relatively expensive rendering and serialization steps.

Letting the DoodleDebug run asynchronously and let the original thread continue might result in a strong boost, but introduce new problems:

Mutations Objects may mutate while rendering is not complete yet, leading to unexpected results. To overcome this issue, each object could be synchronously cloned, then its clone be doodled. Yet, this would only make sense if the cost of cloning objects doesn't exceed the benefits of parallel execution.

Order Chronological order of doodles in the output may not be preserved anymore, since different objects take different times to render. As a solution, each new object to be doodles could be assigned a timestamp, allowing applications to insert them in the right place inside the output.

7.2.2.6 Asynchronous HBase Communication

As explicated in 6.1.2, performance is heavily affected by HBase communication, but only slightly by serialization. As soon as an object to be stored has been serialized, the main thread could safely continue without the danger of causing inconsistencies, while a different thread would asynchronously execute HBase operations based on the serialized object.

While this would work flawlessly for write-only operations, there are cases where asynchronous database operations could lead to problems. For instance, doodles are indexed with consecutive numbers, new ones based on the last number in the database. To avoid inconsistencies caused by uncoordinated read and write access to HBase, a cache object could be introduced. This would proxy the actual HBase and cache all communication, i.e. periodically synchronize its data with HBase.

[3] `http://groovy.codehaus.org/`

Bibliography

[1] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[2] John Boyland. Position paper: Handling "out of memory" errors. In *ECOOP Workshop*, page 150, 2005.

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

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[8] Jim Gray. Why do computers stop and what can be done about it?, 1985.

[9] Ceki Gülcü and Scott Stark. *The complete log4j manual*. QOS. ch, 2003.

[10] 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.

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

[12] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[13] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126, nov 2008.

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

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

[16] Cedric Reichenbach. Doodledebug – a shot-gun marriage between system.out.println and object inspectors. Technical report, University of Bern, 2013.

[17] 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.

[18] B.G. Tudorica and C. Bucur. A comparison between several nosql databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5, June 2011.

[19] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael Jordan. Large-scale system problems detection by mining console logs. Technical Report UCB/EECS-2009-103, EECS Department, University of California, Berkeley, jul 2009.

[20] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. *SIGARCH Comput. Archit. News*, 38(1):143–154, March 2010.

[21] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[22] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *SIGPLAN Not.*, 46(3):3–14, March 2011.

[23] H. Zawawy, Kostas Kontogiannis, and J. Mylopoulos. Log filtering and interpretation for root cause analysis. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–5, sep 2010.

8 Appendix

8.1 Logging Performance Measures Logging

8.1.1 Exception Object

	1	10	100	1000	10000	100000
Doo.dle (distributed Hbase)	3035,362457	3256,101743	5378,092369	17838,51116	80836,11213	628695,5176
Doo.dle (standalone Hbase)	2202,30823	2388,061641	3765,550716	11223,29829	47622,71727	316763,8301
Doo.dle (no Hbase)	690,625234	715,625209	993,961481	1929,919923	4437,770438	13412,86478
Doo.dle (no Serialization)	527,621299	557,623999	796,754017	1447,900566	3018,674109	9825,87062
Doo.dle (Jackrabbit)	629.661565	2184.778766	81544.423888			
Doo.dle (Jackrabbit, optimized)	767.59644	1897.966263	10363.266205	101041.076279		
Logger.error (stout & file)	8,735859	15,821908	47,905965	206,986058	2107,475018	21281,48053
Logger.error (stout)	5,473467	7,504814	26,864249	169,059736	1887,433613	18654,00048
Logger.error (bare)	1,975852	2,060413	2,235754	5,048011	40,110254	59,639311
System.out.println	0,090877	0,459299	4,935718	44,692925	300,86698	2815,498811

Table 8.1: Time [ms] taken to log an exception object up to 100000 times

8.1.2 String Object

	1	10	100	1000	10000	100000
Doo.dle (distributed Hbase)	3060,228806	3389,720309	6449,460724	17631,99923	82559,87353	643915,0131
Doo.dle (standalone Hbase)	2348,98924	2637,000372	4350,842893	11952,64217	45157,55335	305115,0385
Doo.dle (no Hbase)	696,911506	717,205528	937,908382	1793,138938	4052,352456	12806,28842
Doo.dle (no Serialization)	515,00131	556,582099	696,858178	1258,19551	2704,244234	9302,363002
Doo.dle (Jackrabbit)	542.014391	1307.886133	10739.061657	429151.24842		
Doo.dle (Jackrabbit, optimized)	636.937298	1390.13083	8137.219618	99656.403972	3004560.727409	
Logger.error (stout & file)	4,3182	10,29832	36,714118	122,023731	1193,747782	12131,40346
Logger.error (stout)	0,760116	1,315547	6,658909	54,29045	997,787684	10197,36325
Logger.error (bare)	0,815181	0,905926	1,238802	3,567737	35,704638	50,53007
System.out.println	0,063425	0,495399	5,205929	46,444227	809,674804	9098,191995

Table 8.2: Time [ms] taken to log a string object up to 100000 times