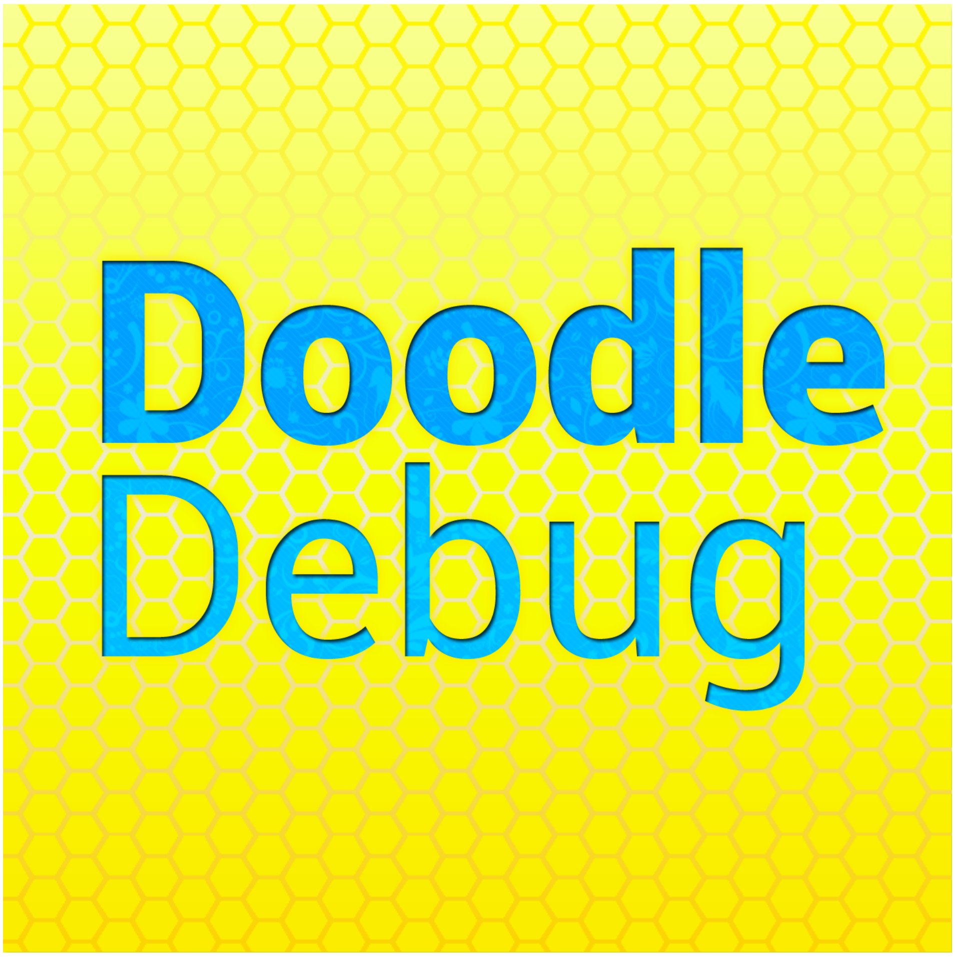
DoodleDebug

November 3, 2012


Doodle
Debug

Abstract

Introduction

Development

Planning

Programming

Providing Code from Eclipse Plugin to Workspace

Lack of Documentation

In order to provide API functionality such as `Doo.dle()` or the `Doodleable` interface to users, we had to find some way of providing Java code automatically from our plugin to the current eclipse workspace. Even though it should intuitively be a quite simple operation, it took weeks in the dark to figure it out. One may say that everything is well-documented in the Eclipse documentation, but the challenge was to find the right part in the documentation. Eclipse developer forums neither could help, but finally a hint was given by some Stackoverflow user: In order to provide code to workspace Java projects, a plugin needs to use the extension point `org.eclipse.jdt.ui.classpathContainerPage`. From then on, another big help was the source code of JUnit, which uses the same technique, otherwise, it would probably have taken several more weeks until everything was working finely.

What Is Provided to the User Workspace?

On one hand, all API methods must be visible from user projects. On the other hand, users should only see the very minimal amount of DoodleDebug's code in order to prevent them from using it in an unintended way, which could cause unexpected behaviour and is much less future-proof, i.e. when DoodleDebug is internally changing. The compromise made between those two requirements was to mainly provide simple, well-documented interfaces (such as `Doodleable` and only show a proxy of fully implemented classes (e.g. `Doo`).

Communication between Java Virtual Machines

Because a user program is running in a different Virtual Machine (VM) than Eclipse itself, we had to find a way to somehow communicate between those two in order to display the DoodleDebug rendering in an Eclipse tab. As a first attempt, we tried to use Java's built-in Remote Method Invocation (RMI) what resulted in frustration as some Java Security Manager always put obstacles in our way. Looking for alternatives, we found SIMON (Simple Invocation of Methods Over Network)[1], a more comfortable alternative, which allows to create a registry on a specified port of localhost and add a Server object to it. The Server class implements an Interface and the client, knowing the server's Interface, can then search for this server name and send messages to it, including simple data types such as Strings. In Order to transport any kind of Java objects, a serialization util is used.

Serialization for the Transport between VMs

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

TODO: Maybe simple benchmark of XML vs. JSON here

Communication between Java and JavaScript

From Java to JavaScript

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

From JavaScript to Java

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

User Interface

Output

Output format and rendering engine

In favor of portability and simplicity, DoodleDebug uses html as output format. For its handling, Eclipse provides the package `org.eclipse.swt.browser`, which among others includes a `Browser` class to render and integrate into the Eclipse UI. However, this `Browser` class is not completely platform independent, it uses the default browser rendering engine of its current host operating system (and not the standard browser). When running on Windows for instance, Internet Explorer's rendering engine is used, even though Firefox is set as the System's standard browser. This fact forced us to be even more careful with the usage of html5 features, because a lot of Eclipse-using Java developers are running Windows.

Semantic Zoom

In order to save space and keep information available to users, DoodleDebug uses the concept of "Semantic Zoom"[link/reference]. Object visualizations are divided in levels of nesting, where level 0 represents outermost objects, referenced in `Doo.dle(object)`, level 1 objects are (semantically) nested ones inside level 0 etcetera. Saving space is achieved by only completely rendering level 0 and 1 objects, and use a smaller representation for a objects of level 2. Level 1 objects are clickable, which will cause them to be repainted as new virtual level 0 objects, so previous level 2 objects will move to level 1. This pattern allows to arbitrarily explore an objects nesting tree, similar to a debugger.

References

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