DoodleDebug

November 24, 2012


Doodle
Debug

Abstract

Introduction

Idea

For programmers, it's an everyday problem: Some feature of a program is not working correctly, but it's not clear where the problem originating. In many cases, it's not even clear what is wrong in the runtime data. Those problems, commonly called bugs, are approached by many various techniques who always have pros and cons, mostly it's a trade off between (manpower) time cost and quality of output. The simplest approach, just staring at code and build some mental model to play the program in one's head, not only demands a lot of concentration and imagination, but also consumes a huge amount of time if the affected code sample is rather big. In most cases, it's much more comfortable to automatically built a visual representation of runtime data, which can is provided by debuggers for instance. They allow users to stop at a particular moment in time, specified by a previously defined line of code that is run over. The whole program state is inspectable this way and commonly used data types have some improved textual representation. One big problem here is, that a program is only inspectable at one snapshot in time, but two states cannot directly be compared. Another widely used solution, printing on a console, eliminates this issue and has other advantages: Because such printing instructions are directly timed into code to debug, it's much quicker to use instead of setting a break point, opening a debugger and navigating to whatever one is searching for. Also, printing to a console allows to save space by just printing exactly what is needed and therefore display a bigger amount of useful information using the same space on screen. But still it lack some features, on one hand, because the only supported output format is text, on the other hand, a line that has been printed cannot be edited any more, therefore advanced formatting becomes very complicated.

Conscious of all those problems, we tried to model a new way of debugging, joining all advantages of classical tools and eliminate all discomfort. We considered console printing as a good starting point because it already brings a lot of desirable properties, so our tool should be able to be called directly from code and render its output into some console-like device. To get away from text-only format and also to support standards as well as portability, we decided to use html as output format.

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.

Smart Behaviour versus Configuration Hell

When designing user interfaces, most people tend to be unsure about best behaviour their UI could provide, or at least think that users will know better what they want. As a result of this, especially open source products come bundled with a lot of configuration possibilities, which is a good thing if some power users really know what they are doing and have lots of experience with this particular software. But most user will barely ever touch them, mostly because it's too expensive to find that specific check box they are searching for. Instead, it's mostly more convenient to have a smart default behaviour fitting the vast majority of people. We made using this mindset as one of the goals for DoodleDebug's user interface, which is why there is no properties dialogue or file.

Smart Scrolling

One Example is the behaviour of the scroll position in the console when new objects are rendered into it. As default (with no user interaction), the window keeps scrolling along with rendered objects, always jumping to the newest one. As soon as the user scrolls away from the bottom, this mechanism is stopped and the scroll position remains equal while new objects are appended on the bottom of the output silently. However, if the user decides to go to the bottom of the output again, the auto-scrolling mechanism is obtained again.

Focus When Likely Desired

Similar thoughts were made on focus handling of DoodleDebug's view in the space of Eclipse's whole UI. Eclipse's built-in console receives focus on any output event per default, but provides a button to deactivate this. For a programmer, it's likely desirable to generally be notified if something happened, but it can be annoying for many outputs over a long time, e.g. if they're working inside another view of Eclipse. In this case, they would probably want to be notified of sporadically occurring outputs. Based on these thoughts, we defined an algorithm to check time since the last output event and only gain focus if the previous event is longer than 4 seconds ago. For example some loop of a game which prints out it's calculation time for the last frame every time will only gain focus on the DoodleDebug view once, but a quiet application like a web server that only generates update when a user visits a site will usually regain focus on such an event. TODO: refer to design book here

Qualitative Study on Beta Version

One of DoodleDebug's big purposes is to compete with classical debugging mechanisms, so we decided to do a study with a handful of Java developers in order to proof this statement and to find bugs as well as usability problems.

Study Session Setup

A fully functional version of DoodleDebug was used, most probably the equivalent to a "release candidate". It was run inside Eclipse 4.2 (Classic edition), using a ThinkPad T410 with Windows 7 (x64) and an additional mouse (2 buttons + wheel, standard size). The screen was captured during the whole session and one instructor sitting beside the test subject for problem explanation and protocol.
Before the actual testing, the user had 15 - 30 minutes to work through a tutorial and play around with DoodleDebug inside a sandbox. At this time, the instructor was allowed to answer questions and support the subject.
For the actual session, there were 3 different small programs containing some manually inserted bug, which they had to find and eliminate. For one or two of them, they were allowed to use DoodleDebug and for the other one or two respectively, they had to fall back to classical tools. The permission to use DoodleDebug on a particular problem changed with every study session, i.e. if subject 1 was allowed to use DoodleDebug on problem A, then subject 2 would not be allowed, but subject 3 would.

References

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