0.1 Study Sessions

For every candidate, a screen capture video was taken as well as hand notes. The main setup and action plots of each subject is documented here. Add rest of documentation

0.1.1 Alpha

The main purpose of our very first session was to test our study setup and its synthetic programming problems. At this point, we only had the first two mentioned problems, namely "Sorting" and "Serialization", the third one, "Decimal Alignment", was added afterwards. Aware of those unequal conditions, we include those results nevertheless, because this study only has a qualitative background and intended to deliver statistically relevant results.

Problem	DoodleDebug	Classical
Sorting	✓	
Serialization		✓

Table 1: Problems for Alpha to be solved using DoodleDebug

Allowed Tools

General Behaviour. Alpha is a typical System.out.println() user: Too "lazy" (according to themselves) to open a debugger and start thinking, better just print out whatever could be affected.

Successes. After creating a dummy project while learning from the Tutorial, they immediately were able to understand how to use the `Doo.dle(object)` method, even used the auto-completion template `dd` and never typed the whole method name again.

Errors. When working through the tutorial, Alpha had to face the situation where they had to use DoodleDebug's `Doodleable` interface. This interface contains two methods for generating a visual representation supporting semantic zoom, one for its normal representation and one for reduced size. In that version of DoodleDebug, they were called `drawOn(Canvas c)` and `drawSmallOn(Canvas c)`, and the Canvas object received as parameter had two methods to draw objects onto it, also supporting semantic zoom, even though `Canvas.drawSmall(Object o)` is intended to be used only in special cases. Instead of adapting the idea of semantic zoom and reducing an objects representation in `drawSmallOn(c)` method to as few as possible parts, Alpha basically drew the same fields as in `drawOn(c)`, but used `c.drawSmall(o)`. Conceptually, this is a bad idea, because repeating this behaviour in nested objects can withdraw any size reduction when it's always delegated to inner objects. At best, this would cause all objects of the normal representation still to be rendered, but maybe smaller (depending of the innermost objects) and thus be reduced to something similar to geometrical zoom.

Discussion. They argued that it's probably too much work to implement an Interface like `Doodleable`, even if there are two methods to implement. They argued that one would suffice in most cases. From our point of view, one method is enough when a problem context only contains few information, just like in our study problems. As software grows bigger, structure will become nested deeper and deeper, which forces either very careful selection of what one want to see on their screen (which must be done before writing print statements into their code) or some technique like semantic zoom.

Consequences. After this session, a couple of API methods were renamed in order to enhance intuitiveness and semantic distinctiveness of them.

0.1.2 Bravo

Problem	DoodleDebug	Classical
Sorting		✓
Serialization	✓	
Decimal Alignment		✓

Table 2: Problems for Bravo to be solved using DoodleDebug

Allowed Tools

General Behaviour. In contrast to any other subject, Bravo used Enums during their tutorial walk-through. We had not considered any rendering for Enums, because only very few programmers actually use them with Java, so there was no meaningful output for them and we had to find and supply some visual representation for them afterwards.
For the sorting problem, Bravo started using Eclipse' debugger, but gave it up again quickly due to problems handling it technically, so they switched to System.out.println() usage.

Successes

Errors

Discussion

Consequences

0.1.3 Subject

Allowed Tools

Problem	DoodleDebug	Classical
Sorting		
Serialization		
Decimal Alignment		

Table 3: Problems for ??? to be solved using DoodleDebug

General Behaviour

Successes

Errors

Discussion

Consequences

0.2 Lack of Documentation

In order to provide API functionality such as `Doo.dle()` or the `Doodleable` interface to users, we had to find some way of providing Java code automatically from our plugin to the current eclipse workspace. Don't talk about "had to find some way." In its way, this section has the least priority. It should probably be called "Implementation." Because it isn't very important, try and be brief: what were the implemtation challenges, what were the solutions? You're kind of saying that, just say it briefer. You haven't come around to this, have you? Please do. One may say that everything is well-documented in the Eclipse documentation Too prosaic. Also: doesn't really matter. This is science., Still true. Please address all comments. Potentially by answering them, rather than following them. I'm skipping this chapter, since it seems to have been addressed already. but the challenge was to find the right part in the documentation. Eclipse developer forums neither could help, but finally a hint was given by some Stackoverflow user: In order to provide code to workspace Java projects, a plugin needs to use the extension point `org.eclipse.jdt.ui.classpathContainerPage`. From then on, another big help was the source code of JUnit, which uses the same technique, otherwise, it would probably have taken several more weeks until everything was working finely. Rewrite, away from what you did, towards a problem-solution focus.

For programmers, it's an everyday problem: Some feature of a program is not working correctly, but it's not clear where the problem originating from. The simplest approach, just staring at code and build some mental model to play the program in one's head, is usually time and concentration intensive. For more comfort, tools help to automatically build a visual representation of runtime data, which is provided by debuggers for instance. They allow users to stop at a particular moment in time, the whole program state is inspectable and commonly used data types have some useful 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. Excellent. Really good sentence. Maybe move this sentence further up. It's kind of key to the whole thing. But we need it here... Another widely used solution, printing on a console, eliminates this issue and has other advantages: As our study results show, programmers like it for its simplicity and assumed quickness, because such printing instructions are directly written into code snippets to be debugged. However, we could not observe any proof for this technique to be actually quicker than using a debugger. Also, printing to a console allows developers to precisely represent an object with needed data only using its `toString()` method (in Java). This paper introduces a solution which aims to eliminate problems of the above mentioned tools Should I refer to the related work section here? and includes a small qualitative study for its verification.

0.3 Planning

Instead of immediately starting to implement, we followed several well-proven planning steps from literature to avoid costly redesigning or refactoring later on. Following the principle of "not mixing up the design and engineering processes" seemed to be a good way to avoid losing the initial path[**?**].