JTourBus: Simplifying Program Understanding by Documentation that Provides Tours Through the Source Code

Christopher Oezbek, Lutz Prechelt
Freie Universität Berlin
Institut für Informatik
Takustr. 9, 14195 Berlin, Germany
{oezbek, prechelt}@inf.fu-berlin.de

Abstract

Many small and medium-sized systems have little or no design documentation, which makes program understanding during maintenance enormously more difficult when performed by outsiders. Thus, if only minimal design documentation is available, which form should it take to maximize its usefulness? We suggest that it is helpful if the documentation describes a tour through the source code, leading the user directly to relevant details. This work reports an evaluation of this conceptual idea in the form of a controlled experiment with 59 student subjects working on a difficult program understanding task in the context of the 27 KLOC JHotDraw graphics framework. One group received a plain text documentation, the other received tour-structured documentation which they navigated by using an Eclipse plugin called JTourBus that we constructed for the experiment. The results indicate that program understanding can be achieved somewhat faster (albeit not more correctly) with JTourBus than with a plain text document.

Keywords: program understanding, design documentation, delocalized concern

1   Introduction

A substantial part of maintenance effort is spent on program comprehension [14], that is, understanding the basic ideas, sufficiently understanding the design of the affected parts of the system and locating the spots where changes should be applied. This is particularly pronounced if it is not the original author who faces this task. If neither prior knowledge of the system's design nor suitable design documentation are available, the programmer has to rely on reverse engineering, and program understanding becomes an extremely laborious process. Therefore, design documentation plays a crucial role during the maintenance phase.

Even large systems are notorious for having design documentation that is sufficiently outdated to be ignored by the maintenance programmers [10]. Presumably, a large fraction of small and medium size systems (10 to 200 KLOC) has little or no design documentation.

In the long run, CASE tools and model driven architecture promise to remedy this situation by making it easy to produce useful design documents as a by-product of developing an application. However, so far most projects still employ other kinds of solutions, typically in the form of natural language text, possibly complemented by a high-level architecture diagram and perhaps a few class diagrams.

1.1   Research question

We assume the following context: We consider small or mid-sized projects with a primary software designer, that have released at least one version of the product and that so far have not produced any durable design documentation. Now the designer decides to invest a small amount of time (less than one day) in order to produce at least a minimal introductory design documentation covering the most important aspects for an initial orientation in the source code.

Our baseline format is a plain text file written in whatever organization the designer considers most appropriate for the software. Now we would like to improve on this. If a graphical representation or diagram of the architecture exists in addition, all the better, but due to the assumed shortness of time, it is extremely unlikely that a diagram will be produced and text will have to provide the core of the documentation. We will hence not consider graphical documentation in this context at all.

As an example for this scenario consider Open Source Software projects (see [2] for an overview). A first version of a project might have been released with little if any design documentation but the author is interested in leveraging the collaborative maintenance skills of the Open

Source community to make the bugs of his/her project "shallow"[7]. How to efficiently make the project comprehensible to outsiders then becomes a key factor. An industrial scenario might be given by a project started by a small number of developers under time pressure without writing documentation. With changing teams, for instance prompted by moving from development to maintenance stage, the need for introductory documents arises.

Our research question is thus: What is the most useful format and organization that minimal textual design documentation should take?

2 Tours through source code

Our answer to this question originates in two observations. First, it seems superfluous, even counter-productive, to duplicate explanation already available in the source code instead of relying on the scaffolding provided by it. Second, many types of design documentation attempt to provide a high-level overview in order to actually give insight into the low-level code structure that often suffers from delocalization [11] and from technical and algorithmic complexities.

2.1 Concept

The basic idea is to organize design documentation as a set of *tours*, one for each important aspect of the design.[1] A tour consists of several *stops* in a particular order. A *tour browser* presents the tours and supports navigation from stop to stop. A stop is a pair of (1) a particular spot in the source code and (2) a fragment of documentation attached to this spot. By connecting stops into tours, delocalizations can be reconnected, and by putting stops into the source code, the source code provides a lot of detail and background information and thus is reused as documentation scaffolding.

We hypothesize advantages of this approach for both the author and the user of a design document. As a benefit to the author, the documentation may become shorter: program identifiers and obvious code structures provide information that can now be omitted from the documentation text. This reduces the redundancy of documentation against the code and thus also makes it more resilient to code changes. A disadvantage is the distributed nature of the text, which makes it harder for the author to produce a coherent document.

There are several potential advantages for the user of the document. First, tours provide additional flexibility for fluent switching between different comprehension strategies

---

[1]Such aspects are usually domain and technology dependent. Some recurring examples include error handling, logging, synchronization, storage/persistence, change propagation, undo handling, configuration, start up and shut down.

by moving high-level and low-level information closer together. Second, a tour plus a tour browser provide design-aware browsing support. This is more efficient than the understanding approaches a programmer would normally use (namely deciding what needs to be investigated next and then using IDE features such as outline or class hierarchies for navigation). Third, tour stops allow for opportunistic contextualization of individual code parts: for instance, if the user is reading GUI controller code and finds a tour stop describing its data storage behavior, tours make it easy to switch to the larger context of data storage design in general. For a detailed discussion of tool support for program comprehension see [12].

The tour browser therefore (1) must provide support for creating and changing tours and (2) must allow all stops in the source code belonging to a certain tour to be enumerated and navigated easily and efficiently.

2.2 Implementation

For an implementation of the tour idea we chose the target programming language Java and Eclipse as an IDE. The resulting tool - JTourBus - was first based on the new Java 1.5 concept of annotations, but the syntax for annotating Java elements multiple times with the same kind of annotation is cumbersome, thus making it difficult for the same code location to be present in tours more than once. A second disadvantage is that the compiler needs to know about the annotation, thus adding an import dependency to the code. These reasons prompted us to embed the tour documentation syntax into JavaDoc instead. To attach a tour stop to a Java element like a field, method or class, documentation authors now add a tag `@JTourBusStop` into the JavaDoc documentation and provide the following three parameters: A number to indicate the position in a tour, the name of the associated tour and a description of the stop. An example stop attached to a method comment is shown in Figure 1.

Technically, we experimented with using the Eclipse Java Model, the Eclipse abstract syntax tree and a regular expression based on the Eclipse search engine and found the latter to be more efficient and compatible with the incremental compile semantics offered by Eclipse. As an additional side benefit of using regular expressions, tour stops may occur anywhere in the source code and are not bound to JavaDoc comments. To summarize the architecture, JTourBus is realized as a plugin for the Eclipse OSGi framework and will hook itself into the JavaCore element change mechanism. Any change performed to a Java source file will then prompt a background search for `@JTourBusStop` tags, which will update the internal model and the view presented in the bottom of Figure 1. We are confident that JTourBus can be easily ported to other IDEs and text editors, as it re-

quires only a Perl-compatible regular expression engine and a means for displaying the results.

2.3 Usability and benchmarking

Two iterations were performed to optimize the usability of JTourBus for both documentation authors and users. During these iterations we eliminated problems such as the following: Some pilot users preferred context menus instead of icon buttons and were puzzled when not finding them. For tour authors we added reordering support via drag and drop and the ability to rename stops by using the context menu. The last usability test before the experiment received no complaints from the pilot testers. The resulting interface and an example tour stop are shown in Figure 1.

With the incremental compilation support, JTourBus does not incur a noticeable delay except for the initial build of the tour roster. As a benchmark test, we added 18 stops into the source code of the Eclipse JFace API (which includes 39 KLOC code and 38 KLOC comments in 223 classes) and measured an average scanning time of less than one second for files already opened and under ten seconds when the whole set of files needed to be read from disk first (measured on a 1.7 GHz Pentium M).

To support larger projects we would like to add persistence support in a future version. The Eclipse platform already supports this feature transparently for plug-ins using the IMemento interface. We will also need to support hierarchical ordering of tours to accommodate larger numbers of tours.

2.4 Related work

2.4.1 Wiki Tours

JTourBus was inspired by the tour bus principle used on Wiki Wiki sites like Wikipedia or the C2 Wiki[2] to connect to other distant wikis in a handy fashion and provide a helpful "What's interesting?" view for newcomers.

2.4.2 Plain JavaDoc

Java's ubiquitous JavaDoc in-source documentation format bears some similarity to the characteristics of JTourBus, but also has important differences.

First, tour-like linking between documentation fragments can be realized in JavaDoc in two ways, via the `@see` tag or via the `@category` tag.[3] `@see` tags have the disadvantage of requiring fully qualified identifiers for classes

and interfaces not explicitly referred to by the class definition, which makes link writing cumbersome for the author. And indeed, `@see` tags are often automatically generated when overriding methods or to discuss other (mostly statical) relationships between elements. In contrast, JTourBus links tour stops implicitly by tour name. The `@category` tag allows grouping of Java elements but does not order them in any way.

Second, and more importantly, all JavaDoc documentation is usually read in its processed form as HTML pages and is then not connected to the actual source code. Worse, if the documentation is minimal, it is hardly useful without connection to the source code and will then hardly be used at all [10]. For instance, out of the 59 participants of the experiment described below, only 18 used the JavaDoc.

2.4.3 Concern tools

There are several other tools available which tackle the problems of delocalization of concerns. Most of them use one of two strategies: First, there are tools that attempt to physically undo the delocalization and unify concerns, most notably aspect-oriented programming [5]. Second, there are tools that leave program structure untouched but *present* concerns in a more accessible fashion, usually by automatic extraction and visualization [8].

JTourBus is similar to the second category, but its goals are related to documentation rather than code and its storage medium is the source code file itself rather than external files.

2.4.4 TagSEA

TagSEA provides support for collaborative documentation construction based on the idea of social tagging: Many developers independently mark source code locations with arbitrary keywords (tags) to "enhance navigation, coordination, and capture of knowledge relevant to a software development team" [13]. TagSEA has been recently extended to include tours (which are called *routes* and stored in separate files rather than embedded in the source code), but no validation of the usefulness of routes for users has been performed so far [13].

3 Empirical evaluation

In order to assess whether documentation based on JTourBus will indeed be more useful than plain text documentation (meaning either improved correctness of understanding or reduced time required, or both), we performed a controlled experiment comparing these two forms of presentation. We specifically tested for (1) improved correctness of understanding and (2) faster understanding.

---

[2]See the topics TourBus and TourBusStop in the Usemod.com Meatball wiki.

[3]`@category` is part of the Java Specification Request 260, which is in an early stage of discussion.
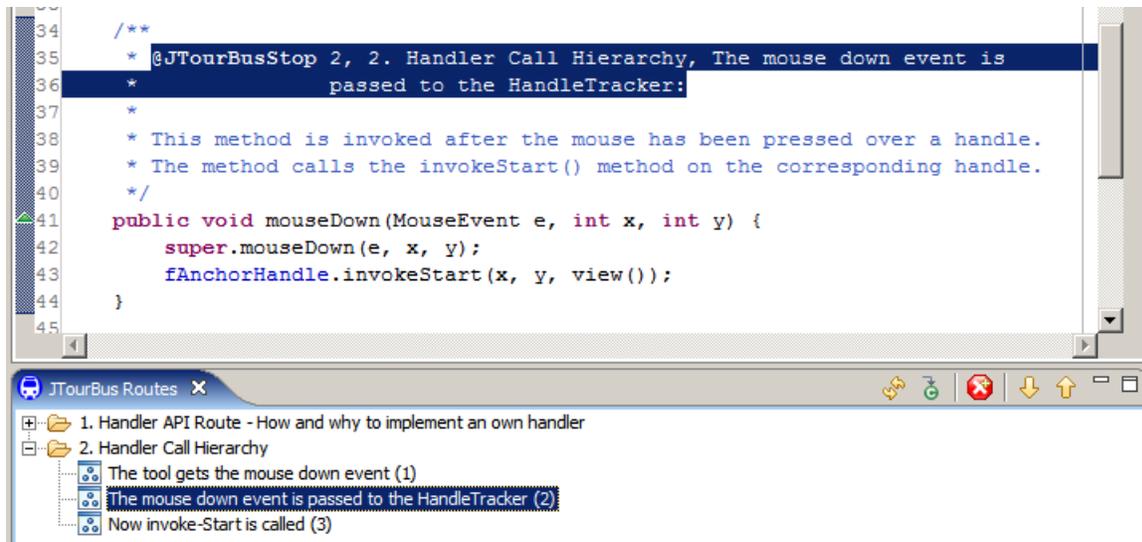
Figure 1. Screenshot of the JTourBusRoutes view in Eclipse after jumping to the second stop of the second tour. The example is taken from the source code used in the experiment.

3.1 Experiment overview

The experiment population consisted of 114 computer science majors who had just finished a senior-year software engineering class (see Section 3.5 for details) and were asked to solve a difficult program understanding task regarding the 27 KLOC JHotDraw drawing editor framework (Section 3.2), which is written in Java. Since participation was voluntary a total of 59 valid data points could be collected.

The task was to describe where source code changes would need to be applied in order to implement a particular, single requirements change (Section 3.4). Only a small number of the subjects had some prior acquaintance with JHotDraw, most, however, had never worked with it and its source code before.

The subjects were randomly assigned into two groups, one solving the task with a minimal design documentation consisting of two JTourBus tours (group T), the other using an alternative documentation consisting of a single plaintext file with two paragraphs (group P).

We carefully ensured that in terms of direct information content both documentations were equivalent (Section 3.3). Also, the documents were prepared by neutral outsiders.

We collected data on each subject's background, time taken for solving the task, quality of the solution (mostly just correct/incorrect) and also coarse-grained information about activities of the subjects during the experiment (Section 3.6). The latter is not discussed in this article, though.

3.2 Software to be understood: JHotDraw

The task to be solved in the experiment involves understanding some aspect of JHotDraw, which is a graphical editor framework originally developed for teaching purposes by Erich Gamma and Thomas Eggenschwiler. JHotDraw is an advanced classroom example of clean and heavy use of object-oriented design patterns [3].

JHotDraw as used in the experiment consists of 26,895 lines of Java code (plus 33,791 lines of comments plus 8348 empty lines using the Eclipse built-in source format) in 448 classes. The code is documented using JavaDoc comments.

JHotDraw contains several applications that instantiate the framework. For the experiment task we chose `JavaDrawApp`, which is a vector-based graphical drawing editor with undo support, multiple views and about a dozen different figure primitives. We slightly simplified the source code by removing the references to the other applications and by refactoring all test classes into a separate source folder.

JHotDraw has found some use in the academic world, most notably with research on aspect mining [1], aspect refactoring [4] and in teaching.

The two subject groups had to receive different versions of JHotDraw in our experiment. The plain text group P received the original, unmodified JHotDraw, while the tour group J received an extended version as described below. The full source code of both versions can be found on the web [6].

3.3 The documents: Tours and Plaintext

With respect to the nature of the documentation provided to the subjects, we took great pains to reach the following goals:

- We attempted to produce documents which resembled those a software designer might have written whose goal it was to provide the best possible information for conceivable maintenance tasks — and to do so within a very limited time frame.

- In particular, the J and P version of the documentation should mention the same facts, i.e., contain equivalent information, and should have been created using roughly the same amount of time.

- and the selection of this information should not be biased towards favoring JTourBus.

In order to avoid bias, we did not produce the documents ourselves but rather employed three top students of a previous software engineering class. The first and second author wrote the J and P documentation together, to the best of their capabilities, but without knowing the task the subjects would be given. They were given a verbal introduction into the design of JHotDraw and received all the design documentation that was part of JHotDraw: a PowerPoint presentation containing an architecture overview diagram and a few notes.

A third author then reworked both of the resulting documents. He was told to modify both documents in such a way that they contained equivalent information and none would favor its users over the other. Using this collaborative process helped to ensure that the quality of the documents would be sufficient to be used in the experiment, yet small enough to fit the requirement for minimal design documentation that can be efficiently produced. The documents then were corrected regarding spelling and grammar and reformated for the experiment.

The result of this process was a text file of 99 lines for the P group and 107 lines of additional comments (forming 2 tours with a total of 15 stops) scattered across the JHotDraw source code for the J group. The difference between the groups results from syntactic overhead in the stop headers (see Figure 1) and reformating.

3.4 The experiment task

When drawing a polyline figure, `JavaDrawApp` does not provide handle points for resizing this figure. The task was to find out which methods in which classes needed to be modified in order to provide resizing support for polyline figures.

This means the task is a corrective maintenance activity and tests the first two stages of the corrective activity [14], namely understanding the existing system and generation of a solution hypothesis. The later stages of correcting the problem, i.e. evaluation of the hypothesis, repairing the code and testing the changes, were explicitly not part of the task. Rather, the subjects were asked to simply name the methods in need of change.

Finding a solution involved several steps: The participants first had to gain a basic understanding of the relationship of tools, figures and handles. Since the task description only told the participants that the `ScribbleTool` creates a scribble but lacked the class name `PolyLineFigure`, the participants had to discover this on their own. The task description as a next step provided the hint that the triangle figure sports resize handles. We were confident that participants would discover the interface method `handles()` common to all figures and port the single crucial line from the `TriangleFigure` method to the `PolyLineFigure`. The second part of the answer was the location where a figure is actually resized (`basicDisplayBox`), which is more difficult to find.

3.5 Subjects

The 59 subjects of the experiment were 45 graduate students and 11 undergraduate students (3 unknown), most of them in their fifth to seventh semester. All participants were enrolled in the software engineering class at the time of the experiment and had a background in using Java as the principal programming language in teaching at Freie Universität, Berlin.

In a questionnaire administered at the beginning of the experiment, the subjects self-reported experience with programming in general, Java, Eclipse, dealing with large software projects, and JHotDraw. It was also possible to relate the experiment results to the results of the following final exam in the course Software Engineering for 54 of the 59 students. Of all the participants, 21 had worked on an exercise sheet handed out the week before which contained a task similar in style to the one given in the experiment, but using JUnit rather than JHotDraw. Additional documentation was not provided for this exercise, so the participants used JTourBus for the first time during the experiment. Around half of the participants (27) had watched several videos about advanced code browsing techniques in Eclipse.

Participants were randomly assigned into the groups J (JTourBus condition, 30 participants) and P (Plaintext condition, 29 participants). The participants did not know in advance which group they belonged to; in fact they did not even know there were several groups.

3.6 Experiment setup and conduct

The experiment was carried out in February 2006. It was announced as a voluntary practical exercise in a third-year software engineering course at the end of the semester. The topic of the exercise explicitly was not part of the exam the week afterwards. Participants were informed that their data on this exercise would be anonymously collected and that their participation was voluntary and could be terminated at any time. Participants then had the choice to agree by providing their consensus or abort the experiment. We did not tell participants until debriefing that there were two groups.

Each participant worked in one session. All participants of a session started at the same time and were allowed to leave when they had finished. Sessions were officially set for 90 minutes, but participants were free to stay as long as they wanted. An instructor/supervisor was present in each room for moral support, but did not play an active or important role.

After login to a specially prepared account, each participant found a ready-to-go Eclipse installation with JHotDraw already loaded. The participants were able to execute and test JHotDraw if they wanted to. They were guided through the experiment by a web-based instruction system, which also collected the subject background survey data, consensus, and answers. Depending on their group, participants additionally received information on how to access the corresponding documentation. For the J group this included a guide on how to use JTourBus.

Time and performance data was collected fully automatically and non-disruptively through additional instrumentation software running in each account and invisible to the participants [9].

3.7 Results

Here we consider the difference between the subject groups J and P with respect to the correctness of their answers and to the time required for working on the experiment task. We also search for other variables that can explain some of the variance observed. The statistical evaluation was performed with R 2.2.

3.7.1 Correctness of answer

When reviewing the answers provided by the participants, we found no useful way of categorizing beyond the discrimination of *incorrect* and *correct*. Although many (in fact most) of the correct answers mentioned only one of the two places where changes needed to be applied, we counted them as completely correct, because the task description called for answering with the "first hypothesis that you think is correct".

In total numbers, in the J group (P group, respectively) there were 8 (8) participants who answered correctly, 13 (16) who did so incorrectly and 9 (5) who did not answer at all or explicitly gave up after trying.

For the incorrect answers, a closer inspection revealed that 11 (J 5, P 6) participants did not understand the relationship between Tools, Figures and Handles sufficiently to answer correctly and 4 participants (2 J, 2 P) gave answers relating to the Zoom functionality which is not related to Resizing.

The modest fraction of correct answers indicates the high difficulty of the task. Yet more than 75% of the participants were able to understand the software sufficiently well to formulate a hypothesis and declare that they would now move on to verifying this hypothesis.

The results thus indicate that JTourBus has neither increased the fraction of correct answers nor seriously disrupted the understanding process. We note though, that more people did not answer in the J group than in the P group.[4] This may be a real effect of the added complexity from tool usage, and it may be advisable to take these people out of consideration for further analysis steps; see below. If correctness is identical for J and P, the benefit of tours, if present, should be visible in the time taken for solving the task (whether correctly or not and either including or excluding the no-answer givers-up). In Section 3.7.2 we will see that tours do indeed save time.

But what other explanation can we find for the differences in correctness, if JTourBus does not make one overall? We attempted to find explanatory variables to predict whether a solution would be correct by constructing decision trees via recursive partitioning. When applying sensible fit-versus-generalization tradeoff criteria, the best such tree uses only one predictor variable, namely the number of points achieved in the course exam at the end of the semester (after the experiment had been conducted). See Figure 2 for the resulting tree: Of the 16 correct answers, 14 (or 88%) were given by students from the top two fifths of the class (who had 41 or more out of 90 possible points in the exam). Thus, this simple criterion predicts correct solutions with 88% accuracy and incorrect solutions with 79% accuracy. All other plausible predictors such as a subject's experience in programming or knowledge about Java, Eclipse or even JHotDraw were not significantly useful.

3.7.2 Working time

Overall, the working time used for the task was 8.8 minutes less for the J group than for the P group (Welch-corrected normal-theory 90% confidence interval: 1.5 to 16.2 minutes), which is 23% of the average time; see also Figure 3.

---

[4] but not significantly more: Fisher exact $p = 0.36$
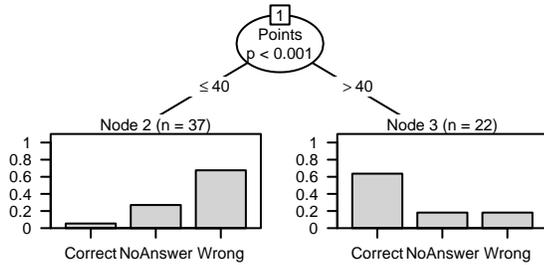
Figure 2. Conditional tree for the correctness of answers. The only admissible split found is "more than 40 points in the exam", which constitutes a 22:32 split of the participants and predicts correct solutions with 88% accuracy and incorrect and no solutions with 79% accuracy.
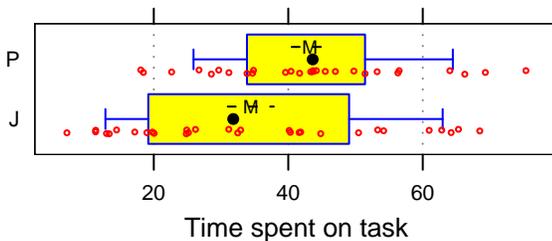

Figure 3. Time spent on task for groups J and P. The fat dot marks the median, the box and wiskers indicate the 10%, 25%, 75% and 90% quantiles, the M and its dashed line give the mean plus or minus one standard error. On average, the JTourBus group finished the task 8.8 minutes faster.
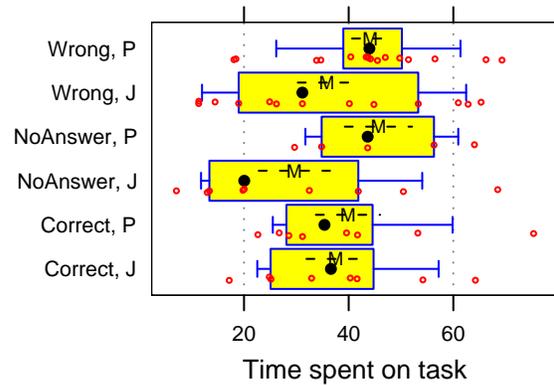

Figure 4. Time spent on task for groups J and P in each of the three correctness categories. The speed advantage of the J group is pronounced only for incorrect/wrong answers and for the givers-up, but not for correct answers.

However, the difference depends a lot on the correctness of the answer, as is shown in Figure 4. The participants with correct answers are hardly faster at all, while there is a big difference not only for incorrect hypotheses, but also for the participants who gave up (no answer).

To compare the time it took for sufficient comprehension to occur so that a hypothesis could be stated, we then removed all participants who did not answer, i.e. gave up. The remaining J group is still 6.2 minutes faster on average (Welch-corrected normal-theory 90% confidence interval: 2.2 to 14.6 minutes), which is 16% of the average time.

Although ideally JTourBus would obviously speed up correct answers as well, the overall effect is still useful, as wrong initial hypotheses occur frequently in practice and the quicker they are made, the faster they are challenged (and hopefully finally discarded), so that the overall process can progress.

Note also that the 'wrong' category of Figure 4 has a much wider box (higher variance within the middle half of the values) for the J group compared to the P group. This probably stems from the unfamiliarity of the JTourBus tool and the resulting working style. We expect that after an initial learning stage the right end of the box would move left, which would increase the JTourBus advantage.

In an attempt to explain some more of the observed variance in the groups, we built various linear models of working time, trying out many combinations of plausible predictor variables. The influence of most variables we investigated was either minor or very noisy. The best model we found is shown in Table 1 and has a multiple $r^2 = 0.156$,

that is, it explains 16% of the overall observed variance.

	Estimate	Std. Error	p
(Intercept)	23.29	8.30	0.00
Group == P	11.72	4.47	0.01
JHotDraw	-10.30	5.10	0.05
ReadingTime	1.00	0.60	0.10

Table 1. Linear model of time on task as a dependent variable of membership in group J or P, whether JHotDraw was known before the experiment and the time spent reading the experiment description and JHotDraw introduction before moving on to the task.

	Estimate	Std. Error	p
(Intercept)	27.60	10.11	0.01
Group == P	8.97	5.42	0.11
JHotDraw	-6.93	6.09	0.26
ReadingTime	0.77	0.74	0.30

Table 2. As above, but excluding the data from all those participants who provided no answer.

This model suggests working time to be additively composed of the following factors: First, a base time of 23 minutes. Second, for the P group a group penalty of 12 minutes (0 for the J group). Third, for people with previous knowledge of JHotDraw a time reduction of 10 minutes. This factor guarantees that the obvious advantage of prior JHotDraw knowledge has not distorted the experiment results. Fourth, for each minute that somebody took for reading the experiment introduction and instructions (which does not count into the work time) a corresponding minute of additional work time on the task proper. This factor explains some of the variability that stems from people who either work very thoroughly or are generally slow.

Compare these results to the 8.8 minutes plain group difference mentioned above (which is 23% of the average time and which, when interpreted as a model, explains 6.6% of the variance). The group difference of 11.7 minutes as postulated by the extended model here is 30% of the average time, and the model explains 16% of the variance. We conclude that this model strengthens the claim that tours and JTourBus do indeed save a significant amount of the time spent for program understanding.

However, let us make sure that this result, too, is robust against the exclusion of the no-answer subgroups of participants. The resulting model is shown in Table 2. We see that qualitatively the model remains intact, but the coefficients

are smaller and the noise bigger. However, the group coefficient is still as large as the bare group difference of 8.8 minutes for the full list of participants.

In summary, we find convincing evidence that a programmer yet unfamiliar with the software to be understood can save program understanding time on the order of 20% by using JTourBus. This holds at least for the task and programmer population given in this experiment.

3.8 Threats to validity

This section discusses which effects could make the results of the experiment as reported here incorrect (threats to internal validity) or inapplicable to other situations (threats to external validity) and how likely we consider these effects to be.

3.8.1 Internal validity

Internal validity is the degree to which the observed performance differences between the groups arise only from the experiment variable (form of documentation), rather than other sources. As this experiment involves true randomization and fairly large groups, it has the potential for perfect internal validity. Nevertheless, several threats to internal validity come to mind:

- Information leaks into future sessions. Students were asked to not discuss the nature of the experiment before all subjects had completed the experiment. A question was included in the survey asking the students whether they had learned anything about the experiment in advance. We are confident that essentially no leaks occurred that could have distorted the results.

- Subject motivation. It is possible that JTourBus may have unfairly motivated (or demotivated) the subjects just by being new and more technological than plain text. At all times we have carefully avoided characterizing JTourBus as particularly new, clever, cool or any other positive quality. We believe motivation effects to be negligible in our experiment.

- Biased documents. It is possible that despite all our attempts at producing a fair pair of documents (as described in Section 3.3), the documentation is biased in favor or against one condition. We believe that document bias, if present at all in the experiment, is too small to be relevant.

- Usability problems. Rather than measuring the tour idea, our experiment obviously measures our specific implementation in the form of JTourBus. If that implementation has usability problems, the real difference between the P and J conditions is larger than we have

seen in the experiment. We expect this problem to be present, but modest.

- Learning effects. Program understanding based on tours implies a somewhat different working style than program understanding based on a plain text design description. Furthermore, the manner of IDE usage is different with JTourBus than without. Both differences disfavor the J group, because this was the first time they worked with tours and JTourBus, and lack of training will have hampered their actions. This problem is undeniably present in the experiment and we cannot quantify its size.

3.8.2 External validity

External validity refers to the degree to which similar differences between the J and P conditions can be expected in different settings of program understanding with no prior knowledge of the system and only minimal design documentation present. Such settings might involve different subjects, target software systems, program understanding tasks, workplace conditions, time pressure, etc.

We are fairly optimistic in this respect. There are certainly many factors that hugely change the program understanding performance in different settings. These involve the inexperience of our subjects in both program understanding in general and the domain of graphical editor frameworks in particular, their lack of human help, the potentially much larger size of other target software systems, the choice of the task given, the quality of the documents, and many more.

However, we cannot see why these factors should greatly influence the size of the relative *difference* in performance between the two conditions J and P. The given task and system were far from trivial and we expect whatever particular phenomena have produced the J versus P difference in our experiment to recur in most realistic settings.

4 Discussion, Conclusion, and Further Work

So is the tour idea a suitable answer to the question of how to write minimal textual design documentation? After all, it is only a 20 percent improvement over plain text rather than a factor 10 improvement, and the correctness of answers has not improved.

We think that, yes, tours (via JTourBus, TagSEA, or some other way) are worth introducing when constructing minimal design documentation, because tours have all the same features that made Javadoc such a huge success: they are technologically simple, easy to learn, have tight integration with the only documents of guaranteed relevance (namely, the source code) and can provide good technical support for navigation, which provides significant added value.

Nevertheless, many things remain to be done:

- Compare tours with other forms of documentation, most notably graphical representations such as UML diagrams.

- Understand the suitability of the tour idea for types of software other than frameworks and its scaling behavior to smaller or larger software systems.

- Study the implication for authors of delocalized tours when updating and writing them in contrast to other types of documentation especially with regards to collaborative settings.

- Understand the user behavior underlying the 20 percent time savings. Is program understanding with tours done in a substantially different manner than with plain text documentation? In which situations did participants save time?

- Experiment with different perspectives on tours such as integration with HTML Javadoc or as a virtual text document composed of the delocalized stops.

- Fine-tune the usability of JTourBus as an implementation of the tour idea, provide support for large projects and allow for connecting non-textual documents into a tour.

Acknowledgments

References

[1] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[2] J. Feller and B. Fitzgerald. A framework analysis of the open source software development paradigm. In *Proceedings of the 21st Annual International Conference on Information Systems (ICIS 2000)*, pages 58–69, Brisbane, Australia, 2000.

[3] E. Gamma, T. Eggenschwiler, and W. Kaiser. JHotDraw homepage. http://www.jhotdraw.org/. Visited 2006-04-03.

[4] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005. ACM Press.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[6] C. Oezbek and L. Prechelt. JTourBus homepage. http://www.inf.fu-berlin.de/inst/ag-se/jtourbus/.

[7] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[8] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, Orlando, Florida, USA, May 2002. IEEE Computer Society.

[9] F. Schlesinger and S. Jekutsch. ElectroCodeoGram: An environment for studying programming. In *Workshop on Ethnographies of Code*, Infolab21, Lancaster University, UK, March 2006.

[10] J. Singer. Practices of software maintenance. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, pages 139–145, Washington, DC, USA, 1998. IEEE Computer Society.

[11] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[12] M.-A. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, Washington, DC, USA, 2005. IEEE Computer Society.

[13] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 195–198, New York, NY, USA, 2006. ACM Press.

[14] A. M. Vans, A. von Mayhauser, and G. Somlo. Program understanding behavior during corrective maintenance of large-scale software. *Int. J. Human-Computer Studies*, 51(1):31–70, 1999.