Software Development and Tool Usability

Brian Dillon
Naval Surface Warfare Center, Dahlgren Division
Virginia Tech
Blacksburg, VA, USA
briand81@vt.edu

Richard Thompson
Naval Surface Warfare Center, Dahlgren Division
Dahlgren, VA USA
richard.p.thompson@navy.mil

Abstract—Tools are used at every stage of the software life cycle with particular recent emphasis on the maintenance period. Evidence shows that maintenance tools are underused, even by the developers who create them. Integrated development environments were created to empower developers, but they have remained virtually unchanged since the late 1990s. This paper examines the challenges of creating development tools, analyzes the usability of two frequently used tools, and suggests that poor tool usability may be inhibiting more efficient software development.

Keywords—human factors, usability, software development, tool

I. INTRODUCTION

Tools are used in every level of computer programming from requirements to design, implementation, test, and maintenance. Many of these are assimilated into the integrated development environment (IDE) to afford access during development. It is estimated that as much as 80 percent of the lifetime software costs are attributed to maintenance activities. Any automation that can replace some of those man-hours with machine-hours has the potential to increase the profitability of software. Because of this largely unfilled niche, much effort is currently being expended to create new tools by such notables as Google Inc. and the Massachusetts Institute of Technology, not to mention a number of companies in the international market.

A study of refactoring tools as used by professional developers [5] by Murphy-Hill, et al. showed three evidences of tool disuse. First, even among the toolsmiths, (a term for those who develop tools meant to be used by other developers) almost 90 percent of all refactorings were manually performed. Second, the average programmer was even less likely to use these tools. Third, when tools were used, the default settings remained unchanged in 90 percent of the cases. A second study [4] found that relatively few low-level refactoring tools (e.g., Rename, Move, Extract, Inline) accounted for 86 percent of all automatic refactorings. While Murphy-Hill found slightly different use patterns between average developers and toolsmiths, their findings showed a similarly high percentage of all tool use was attributed to the same refactoring tools.

Developers’ tools are designed from the bottom up, a term Pizka and Bauer [8] used to indicate that technology drives the requirements. The approach might be phrased as “We can do X, therefore we will give the user the option to do X.” Conversely, a top-down approach is built from and adapts to meet user requirements. Pizka and Bauer also noted that a user is more likely to adapt to the system if it is constructed as bottom-up. Manual editing by developers, even when a tool exists, may be a form of adaptation.

Pizka and Bauer go on to describe the shift from bottom-up to top-down design experienced by the gaming industry in the 1990s. This refocus on an improved user interface led to the explosive high-end gaming market we have today, but it was not without pain. They noted that otherwise excellent software failed because the interface did not meet the expectations of increasingly demanding users. They also noted that some successful games required significant interface improvement to keep pace with the state of the art.

Fig. 1. Comparable views of Visual Studio 97 and Visual Studio 2012 (used with permission from Microsoft Corporation).

Although we have seen vast improvements in other software domains, developer tools have not significantly changed. Consider the screen captures from Visual Studio 97 and 2012 in Figure 1. The 1997 release was a huge improvement over the previous technologies that were basically text editors. The 2012 version, however, is not much different even after 15 years. Moreover, independent sources, based on Google trends [3] and developer surveys [2], find that relatively rudimentary text editors like Vim and Notepad++ are still in competition with the full-blown IDEs like Eclipse and Visual Studio that were designed to replace them. If the IDE
has evolved, it has not yet come too far from its humble beginnings.

Finally, toolsmiths face a unique challenge. In any other field, end users know nothing about software development. As a result, a well-built interface will intentionally hide the details of the program from users in accordance with the Nielsen’s aesthetic and minimalist design principle [6] (i.e., that a task should be simplified for a user by hiding implementation details). With a software developer in the role of a user, however, there is no clear delineation between domain knowledge and implementation. Theoretically, a full description of implementation details will be understood by users. There is an apparent conflict among the match between system and the real world and aesthetic and minimalist design principles. Violation of the latter only complicates the task and retards the user’s efforts.

This prior work suggests that usability factors may be the heart of the problem. The tool usage by developers (or lack thereof) would be explained by a perception on their part that tools lack power and add little value, a criticism supported by some researchers [1] [9]. The bottom-up approach described by Pizka [8] has resulted in tools that fail to meet the needs of end users—end users for whom it is uniquely difficult to design a proper interface. Therefore, the authors contend that common developer tools are failing to meet a usability threshold that would make them acceptable to developers.

II. DEVELOPER TOOL USABILITY

To consider this hypothesis, we will compare the usability of tools commonly found in two popular IDEs: Eclipse and Visual Studio [3]. Several automated tools could have been selected, but Rename is common to both IDEs and is apparently the most successful at meeting user needs [4] [5]. Several refactoring and IDE tools were eligible counterexamples, as many reasonably good tools still have blemishes. The authors, however, selected Add New Method because, like Rename, it was supported by both IDEs and is a commonly performed action. To frame this discussion we will consider the sample code shown in Figure 2 and Nielsen’s usability principles [6].

![Code sample](image)

**A. Rename Operation**

Rename can be initiated from the definition or use of any identifier. In this example, lines 5, 13, and 22 are candidate initiation points. Once initiated, the tool locates instances of the identifier and comments matching the specific string (i.e., as in line 12). Rename allows developers to review the identified elements and exclude any false positives. Rename requires positive confirmation from developers before executing the proposed changes. During execution, all approved changes are made simultaneously. Rename also has an undo function that completely reverses the process.

Rename is available from the mouse menu or from predefined hotkeys. Execution with the mouse requires very few actions. Using hotkeys, it is possible to complete the entire transaction without interrupting contact with the keyboard. No training is necessary to use the tool; however, a novice may be unaware of its existence and effect. Visual Studio overcomes this barrier by identifying manual name changes to identifiers and prompting users to complete the action with the tool (see Figure 3).

![Potential Rename prompt (Visual Studio)](image)

**B. Rename Usability Analysis**

In this section, we will identify specific usability principles from Nielsen [6] and analyze the degree to which Rename tools in Eclipse and Visual Studio meet those characteristics.

*Visibility of system status:* Throughout the Rename operation, prompts guide users from one step to the next and ensure users know what the tool is doing and intends to do. This design also minimizes training in tool use as users are guided step by step. Providing ongoing dialog and complete transparency increases users’ trust in and promotes future use of the tool. Another key feature of system status visibility is WYSIWYG. Simply put, users are given direct insight into what will occur prior to an action. Rename does this well by allowing the user to directly review all findings. That leaves users with more confidence in the tool and confirmation that the tool is acting as expected.

*Error prevention, Help users recognize, diagnose, and recover from errors,* and *User control and freedom:* Rename has some key features that prevent errors or assist users to recover from errors. Specifically, users are given a full list of all changes that will occur. Users are also required to actively agree to those changes before they are enacted. Equally important is the ability to undo all changes. This affords users complete control over the process and reduces the likelihood and impact of an error.

*Flexibility and efficiency of use:* Both tools are designed with the novice and expert users in mind. In keeping with this principle, novice users can follow simple mouse-based menus to find and use the tool, but experts are able to complete the same action with a few keystrokes and only limited disruption of their workflow. The prompt in Visual Studio that was discussed above is another example of this principle. Furthermore, novice users can complete the action using default settings after only a few mouse clicks. More expert users may review and filter the affected identifiers, altering settings to match their preferences.

*Recognition rather than recall:* The Visual Studio version is superior to Eclipse in this respect. The ability to recognize a manual rename and to prompt a user to finish the action with the tool is an aid to novice and experienced users alike. The
prompt increases their awareness of the tool and only requires passive recognition rather than the more active recollection. Eclipse may have the same tool, but it requires knowledge and action to access the capability.

C. New Method Operations

In this example, we will consider the creation of a new method doSomething at lines 7, 14, and 26. The operation and outcome of the New Method tool is different for each initiation location and IDE. Each is enumerated as a distinct use cases or corollary.

CASE 1: The method signature void doSomething(); is inserted in ITool at line 14. When this occurs, both IDEs create a distant compilation error in Tool but, as shown in Figure 4, do not assist or even prompt the developer to further action. Once the developer investigates, the error message in Tool indicates that doSomething is not implemented.

COROLLARY 1.1: Eclipse gives users two options capable of resolving the error: add unimplemented methods; or Make type ‘Tool’ abstract. Eclipse does not, however, give any indication whether the latter choice will invalidate other code that instantiates the Tool object. That choice may create new compilation errors rather than resolving the matter and is probably incorrect given the previous user actions.

COROLLARY 1.2: The Visual Studio tool only suggests the implementation of the new method. This is required in C#, even from abstract types. Unfortunately, Visual Studio has a completely different problem. The error generated in Case 1 appears under the Tool type name. The automatic implementation, however, is only accessible from the ITool inheritance. This disconnect is shown in Figure 5.

COROLLARY 1.3: Initially, the automatically generated method at line 26 of Tool must be empty; at least until the developer can supply the appropriate implementation. Each IDE uses a language-specific strategy to persuade developers to do so. Visual Studio throws a NotImplementedException that ensures the developer will be notified of the missing implementation when the code is executed. This design choice can also cause the program to crash unexpectedly at run time. Eclipse generates a TODO tag that can be tracked in the Tasks view. When the program executes, the empty method will still fail but without even a whimper. Neither solution—catastrophic application failure nor silent untraceable failure—is ideal.

CASE 2: A call to tool.doSomething is inserted in User.use above line 6. In this case, both IDEs prompt the developer to resolve the deficiency in ITool, as shown in Figure 6, and support the automatic insertion in ITool at line 14. That step, of course, requires the manual execution of Case 1 as a separate process. There is no direct tool prompt to assist.

COROLLARY 1.3: Neither tool refocuses on the newly created method, and only Eclipse refocuses on the method signature created in Case 1. These are all signs of disconnected although clearly related capabilities with poor status updates.

The system also repeatedly fails to correctly update users on the tool’s status. Consider the lack of error message described in Case 1 and the lack of transparency in Corollary 1.1. Users would also expect the auto-generated method to create some visible warning as described in Corollary 1.3. Users expect the tool to give solid feedback on the impact of a proposed change. Not matching users’ expectations in this respect violates this principle and creates user distrust.

D. New Method Usability Analysis

As in Subsection B, this subsection will discuss how well specific usability principles were implemented in the New Method tool.

Visibility of system status: It is difficult to evaluate the tool on this principle simply because there is very little continuity between individual steps. For example, executing Case 2 followed by Case 1 results in an incomplete end state. Theoretically, the tool should then prompt users to continue, but instead the process is immediately terminated. In Corollary 1.3, neither tool refocuses on the newly created method, and only Eclipse refocuses on the method signature created in Case 2. These are all signs of disconnected although clearly related capabilities with poor status updates.

Error prevention, Help users recognize, diagnose, and recover from errors, and User control and freedom: As discussed previously, Case 1 actually creates a compilation error. In Corollary 1.3, both languages create future program errors. Without even a warning at compile time, users can create and distribute an unstable product with a built-in failure mechanism. Even in Case 3, where Eclipse supports automatic generation of the method signature in ITool, it does so only in response to an error message created by the @Override annotation. Unlike Rename, the New Method tool seems to create errors and force developers to resolve them manually as a standard mode of operation.

Flexibility and efficiency of use: Novice users will identify a number of drawbacks in the New Method tool. Case 1
generates an error that can be multiplied by naïvely following the suggestions described in Corollary 1.1. Novice users will not even know how to automatically correct the error described in Corollary 1.2 and in Case 3, novice user will likely fail to properly annotate the method in Eclipse. Neither of these tools was designed with novice users in mind.

Are the tools any better for experts? An expert developer might be able to navigate or at least understand the sequence of error messages created in these use cases. Accessing the tools, however, would require repeated alternate use of the mouse and keyboard. One might even expect an expert developer to intentionally avoid all such errors by executing the cases in a specific sequence (i.e., case 3, 1, and 2), but doing so would be no faster than manual development. On average, expert users may not even make use of available tools because they offer no advantage.

Recognition rather than recall: As noted above, an expert user might make use of any or all of these cases if a user is aware of the available tools; however, cases 1, 2, and 3 all require users to overcome errors instead of suggesting solutions. In each case, the IDE recognizes the error, but provides a variety of options consistent with the error and not previous user actions. A better implementation of Case 1, for example, might identify all implementing classes and suggest implementation. This theoretical tool might also suggest a default implementation for non-abstract classes, resolving corollaries 1.1, 1.2, and 1.3. The tool becomes more valuable by offering suggestions and default choices rather than forcing users to synthesize a new solution.

III. ANALYSIS

Through long association with Marines and others used to hardship, the authors have come to recognize that users do not generally know what they are missing until a better alternative is given to them. Landline phones, folding maps, and paper books all seem quite useful until one has held all three at once in a single smart phone. Or in the words of Nielsen [7], “Definitely don’t believe what people predict they may do in the future.”

Users often believe they are content with the status quo and accept inefficiencies of design that place an added burden on users. These design deficiencies are excused or overlooked because the conditions have existed for so long or, as advocated by Nielsen, users believe they know what they want but their actions show a very different preference.

In a very real sense, software developers may be considered a disadvantaged user class. They are forced to communicate with a device that uses its own language and rules. They are forced to manually correlate thousands of individual tasks, spread across multiple files, without clear insight into dependencies between those files and tasks. The tools they have are a significant improvement over their predecessors, but are far behind the current state of the art in other software domains. They were built, as described by Pizka and Bauer [8], bottom-up and where the tools have failed, developers have adapted by not using the tools.

Repeatedly in our analysis of the New Method tool, the authors identified situations where a user would be forced to take manual actions. It was even noted that by failing to meet user expectations, give useful feedback, and provide task continuity, expert users might learn to avoid the available tools altogether. This is consistent with the 90 percent manual refactoring rate identified by other researchers [5]. When expert developers consistently avoid tools supposedly designed for their use, even built by them, the reason cannot be a lack of training or unfamiliarity. We might profitably use Nielsen’s rule of thumb [7] “[w]atch what people actually do.” What they are doing is not using these tools.

IV. CONCLUSIONS

In this paper, we have given clear evidence that developer tool usability is sometimes very low. By examining commonly used automatic development tools available in popular IDEs, we have shown that a large disparity exists between extreme usability cases. Consistent with the findings of various other researchers, we have shown that poor usability can force developers to use manual development techniques, avoid the use of tools and, in the case of novices, actually make code worse. This analysis can be repeated for all varieties of development tools in an effort to improve their usability by applying a top-down construction paradigm solidly based on user requirements.

REFERENCES