Bespoke Tools: Adapted to the Concepts Developers Know

Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman
Department of Computer Science
North Carolina State University, Raleigh, NC, USA
{bijohnso, rpandit}@ncsu.edu, {emerson, heckman}@csc.ncsu.edu

ABSTRACT

Even though different developers have varying levels of expertise, the tools in one developer's integrated development environment (IDE) behave the same as the tools in every other developers' IDE. In this paper, we propose the idea of automatically customizing development tools by modeling what a developer knows about software concepts. We then sketch three such "bespoke" tools and describe how development data can be used to infer what a developer knows about relevant concepts. Finally, we describe our ongoing efforts to make bespoke program analysis tools that customize their notifications to the developer using them.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments*

General Terms

Tools

Keywords

adaptive tools, IDE, concept models

1. THE PROBLEM

Today's integrated development environments (IDEs) are one-size-fits all; every developer gets the same IDE and set of tools. For example, developers who use Eclipse have thousands of tools cluttering their IDE's user interface, even though most developers use only a few tools [1]. As another example, while experts may find the git interface powerful, novices can find git's interface confusing [2].

One can customize their IDE and toolset, but the process is manual. For example, Eclipse offers about a dozen different distributions (Eclipse for "Java EE Developers", for "Eclipse Committers", and so on), each of which comes with a different set of tools. In the git example, a developer experienced with version control might use the default git client. However, a developer with less experience might choose a client designed for ease of use, such as SmartGit.[1]

[1]`http://www.syntevo.com/smartgit/`

Such manual customization is undesirable for several reasons. First, to choose among alternative tools, a developer must be aware that alternatives exist, yet lack of awareness is a pervasive problem in complex software like IDEs [3]. Second, even after being aware of alternatives, she must be able to intelligently choose which tool will be best for her. Third, if a developer's situation changes and she recognizes that the tool she is currently using is no longer the optimal one, she must endure the overhead of switching to another tool. Finally, customization takes time, time that is spent fiddling with tools rather than developing software.

2. WHAT IS THE NEW IDEA?

Our idea is *bespoke tools*: tools that automatically fit themselves to the developer using them. We envision bespoke tools that adapt their user interfaces based on models of what a developer knows about software *concepts*. By a concept, we mean an abstract notion that is applicable across software systems, such as polymorphism, erasure of Java generics, and the model-view-controller pattern. We argue that IDEs and tools that model developers' knowledge of concepts could help developers more effectively use software tools. In this paper, we specifically outline the benefits for new static analysis tool users, logging tool users, and developers that have difficulty with tool output. Bespoke tools can prioritize notifications and present information in ways that align the developer's experience with relevant concepts, improving each individual developer's experience.

3. WHY IS IT NEW?

Our own single most related paper describes degree-of-knowledge [4], which is similar to our idea of concept modeling in that both use developer data to model individual developers' knowledge. Others have modeled and used similar data. For instance, JADEITE models API usage to recommend API usage examples [5]. Likewise, WHOSEFAULT models code expertise to recommend an appropriate developer to fix a bug [6]. In contrast to such prior work, which models developers' knowledge of one system (for example, knowledge of method `foo`), our idea models conceptual knowledge (for example, knowledge of method overloading).

Our idea builds on two areas outside software engineering: intelligent tutoring systems and adaptive user interfaces.

Intelligent Tutoring Systems (ITS). ITS are designed to automatically adapt lesson plans to a student's individual needs [7]. Like our work, ITS also model concepts, albeit for students rather than developers. However, concept models in ITS are constructed by regularly posing questions to students, whereas asking professional software developers to answer programming questions may be prohibitively burdensome. Consequently, we aim to build developer concept models based on a developer's existing data, such as that from code, code reviews, and IDE interactions.

Adaptive User Interfaces (AUI). Benyon was one of the first to suggest how to adapt software's user interface, assess user differences, and identify factors to consider for adaptation [8]. Although there are many implementations of the AUI in various Human–Computer Interaction contexts, one is described in **the single most related paper by others**: Zou and colleagues' adaptive menus in Eclipse [9]. By modeling how often a developer uses menu items, this AUI removes menu items that are used infrequently. Such work aims at modeling software usage, rather than conceptual knowledge; our idea, in contrast to existing AUI approaches, aims to model developers' knowledge of programming concepts.

4. SKETCHES OF BESPOKE TOOLS

We next sketch three examples where we hypothesize that bespoke tools would improve developer productivity.

4.1 Initial Experience with Static Analysis

Static analysis tools, which analyze source code to find potential bugs, are cheap to run, and can find substantial problems in software [10]. Still, one problem with such tools is that they can produce thousands of notifications, and developers can have difficulty prioritizing these notifications [11]. Thus, initial user experience with static analysis tools could be improved if they helped developers prioritize which notifications should be addressed first.

Tools like Coverity[2] prioritize bugs by *severity* (how bad the bug is) and *confidence* (how sure the tool is that the bug is really a bug). While these are important aspects of prioritization, neither is sufficient to guarantee a positive initial user experience. We propose two additional criteria to prioritize notifications in a bespoke static analysis tool:

- **Prioritize notifications that the developer will understand.** For example, Coverity may suggest a developer "Prevent multiple instantiations of singleton objects" by pointing out that an instance of a non-volatile, static singleton method is not thread safe. To fully understand this notification, the receiving developer should know something about the concepts of lazy initialization, volatile, static members, object initialization, and multithreading.
- **Prioritize notifications that are easy to fix.** For the notification above, the tool's documentation suggests making the singleton field volatile. To assess the cost of fixing this notification, the developer should understand the concept of volatile fields, including the performance penalty incurred by volatile fields.

We argue that integrating these two prioritization criteria with severity and confidence could lead to a better initial user experience of static analysis tools. The notification about lazy initialization would be prioritized if the developer is knowledgeable about the concepts relevant to the notification and its fix. But how can a bespoke tool predict what a developer knows about those concepts?

We propose that IDEs for bespoke tools model a developer's knowledge (Figure 1). Each concept would have a different model, and each individual developer would have a different instantiation of each model. For example, a bespoke static analysis tool would have a model for what a developer knows about Java's volatile keyword, a model for lazy initialization, and a model for multithreading performance. It might further create models that aggregate knowledge from submodels, such as a model of general multithreading knowledge that combines relevant models. The input to a model is a variety of developer data; the output is a number representing a developer's knowledge of one concept.

Suppose we wish to model what a developer knows about lazy initialization. One source is the source code she has authored; for instance, the more times she has implemented a lazy factory pattern, the more she may know about lazy initialization. Another source is a developer's code review data; if she has spent time interacting with code that implements this pattern, she may know more. Yet another source would be the tools a developer uses; if she has used the "Introduce Factory" refactoring tool, she may know more. Each of these sources of data are weighted differently in the model. For instance, the developer's code may be weighted more heavily than her tool usage data.

As a whole, we imagine an IDE that continually monitors a developer's behavior (Figure 1, top) to build and refine models of a wide variety of concepts (Figure 1, middle). Bespoke tools then synthesize these models to adapt their user interface accordingly (Figure 1, bottom).

4.2 Improving Code with Logging Tools

Logs support developers when diagnosing run-time problems, and logging tools can help them to effectively implement logging. A recent tool called LogAdvisor recommends where a developer should add a log statement based on contextual features that commonly surround log statements [12, 13]. For example, LogAdvisor might infer that logging is often performed within `catch` statements when a `FileNotFoundException` exception is thrown; then, LogAdvisor makes a recommendation to insert logging where those features appear, but no logging exists.

We argue that bespoke logging tools could further assist developers with logging decisions by using conceptual models to determine when and how to make a suggestion. Although useful to developers new to logging, other developers may have made the informed decision to not include logging statements in some places. Therefore, developers knowledgable about logging may not find the notifications useful, or even annoying at times. We propose two additional ways to provide developers with logging advice:

1. **Provide advice when in unfamiliar contexts.** To know how to log in a particular context, it helps if the developer has experience with the concepts relevant to that context. For a developer to know to log when a `FileNotFoundException` is thrown, they would have to have done so at some point or have written code that catches that exception. If the developer understands logging, exception handling, and `FileNotFoundExceptions`, LogAdvisor could skip providing advice.
2. **Provide advice based on individual developer knowledge.** Logging tools could filter advice and suggestions by using models that can predict an individual developer's knowledge of logging as a concept. For

[2]`http://www.coverity.com/products/code-advisor/`

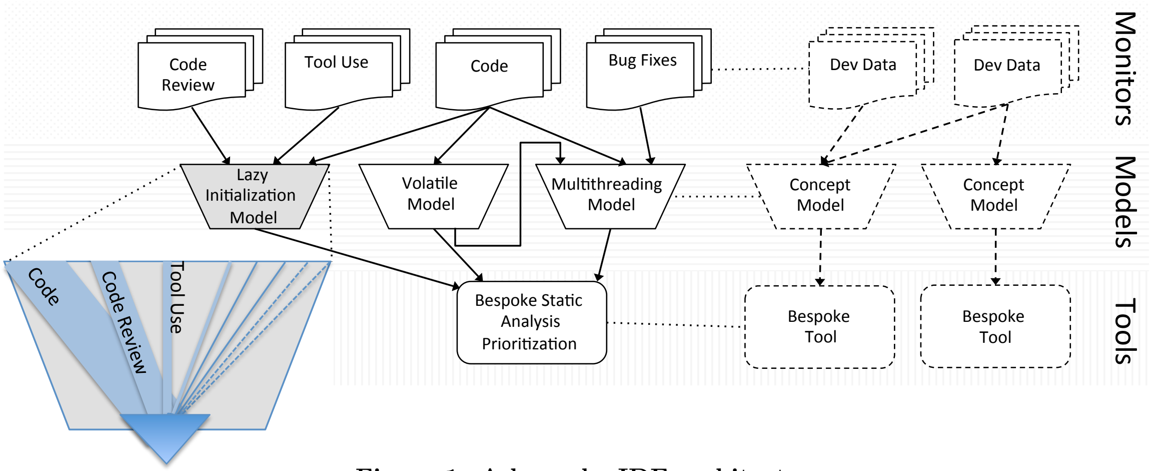

Figure 1: A bespoke IDE architecture

example, a bespoke tool could predict how well she understands logging based on how often she uses it correctly (without major subsequent revisions) in her code. As her experience with logging increases, the tool provides advice and suggestions less often while keeping the suggestions available if the developer is interested.

A similar set of models proposed for static analysis tools could be used for a bespoke version of LogAdvisor. One would be a model of exception handling knowledge. To model exception handling, one source of data would be code; the more she has written involving exceptions, the more she may know. Nonetheless, more does not necessarily imply greater knowledge, as some developers might add exceptions routinely without much consideration. We could also determine how often the developer moves, removes, or changes relevant code. Presumably if she removes, moves, or changes just the exception or code involving the exception, she has an informed reason for doing so.

4.3 Understandable Program Analysis Output

While prioritization of static analysis notifications may help with initial user experience, developers still may encounter notifications that they do not understand [11]. Other types of program analysis tools, such as code coverage tools and model checkers, output complex notifications that developers may likewise have trouble understanding.

Bespoke program analysis tools could improve the ability for tools to communicate in ways that align with the experience and knowledge of the developer using the tool. For example, FindBugs provides the following notification:

> There is a statement or branch that if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions).

A developer who is less familiar with the concepts of `null` values and pointers might find this notification difficult to understand. In contrast, a developer who is very familiar with both concepts might find this notification too verbose.

A bespoke tool that models what a developer knows about programming concepts could adapt the notification to the developer looking at it. To continue the above example, if a developer knows little about the concept of `null` values, the message would be more verbose, perhaps providing links to relevant online material. If a developer knows all the relevant concepts well, the notification could instead simply say “potential null dereference,” and point her to the code where `null` value may be dereferenced and the relevant prior statement or branch.

Like the other examples, the code that the developer has written is a useful source of data for hints about a developer's knowledge of a concept. For the `null`-pointer notification, this includes checks for `null` values and catch blocks for `NullPointerExceptions` that have actions. Another source would be notifications she has already resolved; if the developer has resolved notifications of the same kind or notifications that involve the same concepts, the developer likely has some knowledge of the underlying concepts.

5. PROGRESS SO FAR

We have started to explore the feasibility of our idea by collecting data to answer the research question, “Can we predict conceptual knowledge using existing developer data?” As a starting point, we chose to begin with the concept of dereferencing `null` values because it is a common concept for many program analysis tools. We have started to analyze publicly available code as a source of developer data.

We recruited developers from GitHub and students from our university to complete a concept inventory we created to assess each developer's knowledge of `null` value dereferencing. The items on the inventory serve as an oracle for what each developer knows about the concept. Then, for each developer, we analyzed their GitHub repositories and counted the `null` checks that they added and removed. Next, we built a linear regression model to correlate a developer's concept inventory score with her `null` check counts. We expect to see a correlation between the two as a proof of feasibility of our idea. As a baseline for developer experience, we used lines of code added by the developer; research suggests that one major indicator of developer experience is the code they have written [4, 14]. Therefore, we also built a linear regression model to correlate a developer's concept inventory score with lines of code she has written for comparison.

These findings suggest that, though there is currently a large margin of error with our small data set (17 developers), `null` checks added and removed may predict an increase in knowledge better than lines of code added. Although both models predict knowledge within 1 point of the developer's total inventory score (out of 9) 47% of the time, there is a positive correlation in the `null` checks model while there was a negative correlation between the lines of code added and the inventory score. One way to interpret these results is that the more code a developer writes, the less often they have to think about the `null` checks they write in their code. Though they may generally understand the concept, as no participant did poorly on the inventory, they may eventually move detailed knowledge of `null` objects and dereferencing out of active memory. This suggests time is another variable to consider when measuring knowledge. Another interpretation, which could coexist with the first, is that an increase in `null` dereference checks would be more likely to predict an increase in knowledge, which is what we intend to monitor, than using lines of code written alone.

6. CHALLENGES

The idea of bespoke development tools presented in this paper appears promising, but let us consider a few of the challenges in implementing bespoke development tools. One major challenge is gathering the requisite developer data to build concept models. The easiest data to collect for model building may be source code, since version control systems are widely used, but other data such as IDE usage data are less common. Even if a developer decides to start collecting usage data to take advantage of bespoke tools, past uses of the tool might not have been recorded at all, and thus some concept models could initially be inaccurate. This could lead to false positive adaptations, such as treating a developer with more experience like one with less experience due to lack of data. It would beneficial to consider allowing developers access to augmenting their model with data of their own. Without the ability to easily do so, developers may not want to use the tools at all [11].

Another challenge is that changing tools' user interfaces may make it difficult for developers to transition between user interfaces. The advantage of the unchanging tool interfaces we see today is that the developer can expect to see the same interface the next time they return to it. When bespoke tools change their user interfaces as developers become more experienced, such a change could be disorienting and confusing. Such bespoke tools should therefore ensure that the user interface changes gradually. To decrease the number of needless adaptations, bespoke tools could allow developers to self-report their experience to improve the predictions. It should also be possible for the developer to turn off the adaptations.

A third challenge is the effort involved in scaling up the idea. The space of potential concepts that could be modeled is enormous; for instance, the concepts for a single programming language are described in the language's specification, and many developers use multiple programming languages and tools. There are numerous relevant considerations, such as how often the developer writes relevant code and time in between uses. From a toolsmith's perspective, it is difficult enough to create one developer-friendly tool, let alone to create a tool that changes over time. These challenges highlight the need for reusable patterns and frameworks that can make the job of implementing bespoke tools easier.

7. CONCLUSION

In this paper we advocated for the idea of bespoke tools. While today's IDEs and the tools within them require the developer to change her behavior to fit with the way tools work, we believe that bespoke tools can enable the opposite; tools that change to fit the ways that developers work. Although the evidence for feasibility that we have presented here is modest and the challenges to overcome are substantial, we believe bespoke tools could one day improve developers' ability to use tools effectively.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant numbers 1217700 and DGE–0946818.

8. REFERENCES

[1] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, "Improving software developers' fluency by recommending development environment commands," in *FSE*, 2012, pp. 1–11.

[2] S. Perez De Rosso and D. Jackson, "What's wrong with git?: a conceptual design analysis," in *Onward!*, 2013, pp. 37–52.

[3] T. Grossman, G. Fitzmaurice, and R. Attar, "A survey of software learnability: metrics, methodologies and guidelines," in *CHI*, 2009, pp. 649–658.

[4] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *ICSE*, 2010, pp. 385–394.

[5] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *VL/HCC*, 2009, pp. 119–126.

[6] F. Servant and J. A. Jones, "Whosefault: Automatic developer-to-fault assignment through fault localization," in *ICSE*, 2012, pp. 36–46.

[7] T. Murray, "Authoring intelligent tutoring systems: An analysis of the state of the art," *IJAIED*, vol. 10, pp. 98–129, 1999.

[8] D. Benyon, "Accommodating individual differences through an adaptive user interface," *Human Factors in Information Technology*, vol. 10, pp. 149–149, 1993.

[9] Y. Zou, M. Lerner, A. Leung, S. Morisson, and M. Wringe, "Adapting the user interface of integrated development environments (IDEs) for novice users," *JOT*, vol. 7, no. 7, pp. 55–74, 2008.

[10] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *PASTE*, 2007, pp. 1–8.

[11] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE*, 2013, pp. 672–681.

[12] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *ICSE*, 2014, pp. 24–33.

[13] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," Supp. to ICSE, pp. 452–461, 2015.

[14] J. J. Cañas, M. T. Bajo, and P. Gonzalvo, "Mental models and computer programming," *IJHCS*, vol. 40, no. 5, pp. 795–811, 1994.