Facilitating Information Management in Integrated Development Environments through Visual Interface Enhancements

Haipeng Cai
Department of Computer Science and Engineering
University of Notre Dame, Indiana, USA
Email: hcai@nd.edu

***Abstract*—In the past decades, integrated development environments (IDEs) have been largely advanced to facilitate software engineering tasks and improve developer productivity. Yet, with growing information needs driven by increasing complexity in developing modern software with demands for high quality and reliability, developers often need to switch among multiple user interfaces, even across different applications, in their development process, which breaks their mental workflow thus tends to adversely affect their work efficiency and productivity.**

This paper discusses challenges faced by the current IDE design mainly due to working context transitions imposed on developers during their search for multiple information sources for their development needs. It remarks the primary blockades behind and initially explores some high-level design considerations for overcoming such challenges in the next-generation IDEs. Specifically, a few design enhancements on top of modern IDEs are proposed, attempting to reduce the overheads of frequent context switching commonly seen in the multitasking practice of developers.

Keywords: Information need, integrated development environment, context switching, automatic recommendation, programming interface, software visualization

I. Introduction

One merit with visual programming [1]–[3] is that its integrated interface empowers smooth transitions among the workflow steps of developers—the interface provides all programming elements (of visual forms) so that the developers involved in the interface can easily maintain their mental workflow models by focusing on mostly just one type of interface (i.e., visual interfaces).

With most existing IDEs (e.g., ECLIPSE [4]), however, developers often face challenges from frequent transitions between coding (text interface) and visual aids (graphical interface), or even between disparate applications (and their different interfaces) [5]. Since traditional (textual) programming involves typically a demanding logic reasoning process, such transitions and context switches can cause large inefficiency [6] in the development process and thus even greater risks to the quality of the resulting software.

The reason underneath is that context switching tends to interrupt the overall workflow of developers [6]. More importantly, the problem can be exacerbated by the growing information needs for developing modern software of increasing scale and complexity. Unfortunately, on the other hand, modern IDEs tend to grow in the complexity of their interface in a way that, seemingly facilitating developers to meet their needs for multiple sources of information, actually compounds the problems with switching among increasingly more contexts.

As it stands, research on facilitating programming tasks through interactive graphical environments exists [7], mostly focusing on providing visual aids within IDEs. For instance, Dragon [8] shows visual windows for program dependence, debugging information, data structure state, memory layout, and similar other visual gadgets during common tasks of developers. However, this framework is limited to passively responding to user requests—it fails to automatically *push* information to developers to assist their program-analysis tasks as if it were an integral part of the whole task pipeline. In contrast, a more useful interactive programming environment needs to deliver informative visual aids not only on demand but in a proactive manner, so as to minimize context switches during the entire development workflow.

Another challenge to today's IDE interfaces lies at their falling short of meeting the growing amount and *variety* of information needs by developers. To finish a coding task, for example, developers usually have to consult many information sources that are diverse and distributed across disparate interfaces even applications. Although modern IDEs mostly have various supports for integrating diverse functionalities by means of plug-ins or GUI extensions, information from those extraneous modules often has to be passively retrieved under the requests of developers—the multiple sources of information are available, yet not well integrated *in synergy* [9] with other elements of the IDEs such that developers can concentrate on their ongoing tasks.

To address these specific issues, this paper preliminarily explores several novel IDE features that could effectively assist developers with handling multiple tasks while minimizing the

costs of context switching during their development workflow. Specifically, three features are proposed focusing on interface design: (1) extending the traditional coding view to include coworker views, (2) offering automatic recommendation based information for API usage and code examples, and (3) providing in-situ mechanism for most commonly used code-editing related operations driven by the current task context. And two major visualization features are proposed as well, including (1) multiple code visualization views and (2) interactive linked visualizations. By illustrating the needs and benefits of these instrumental features, this paper demonstrates how the next-generation IDEs could be designed to offer better aids to developers in ways that improve development efficiency and productivity.

In summary, this paper highlights the context-switching issue in the design of today's IDEs that hinders the effectiveness of using them, and illustrates such issue using example usage scenarios; it discusses three interface design features that potentially reduce developers' overall cost of switching among multiple contexts in search of various sources of information; it also envisions two interactive visualization features that enable holistic integration of multiple information in synergy so as to reduce developers' need of switching contexts when searching for various information.

The rest of this paper is organized as follows. First, Section II gives a development scenario regarding information foraging that motivates the following programming interface design. Then, Section III and Section IV summarize the concrete features in the new programming environments, on interface and visualization design, respectively. Section V discusses the key issues pertaining to the implementation of the proposed design features, where the strategy for empirical evaluation is outlined with technical limitations and practical challenges in applying the new IDEs also addressed. Finally, Section VI recaps the paper with concluding remarks.

II. Motivating Example

During software development, programmers gain most of the information they need from the source code they are working on [10], [11]. Yet, they also need information beyond that [12], such as those produced by program analysis tools, to obtain better understanding of the software [9]. Examples of such additional information include call graphs, dependence graphs, and type hierarchies. While most present IDEs do provide functionalities to help developers obtain these information, they force developers to proactively make requests for them. However, responding to user requests may not be sufficient in many situations. Rather, a more effective IDE should provide developers with a *guiding* interface instead of question responder, as developers may not have *prerequisite* information for them to initiate those requests or to do so in the most efficient way overall. In consequence, excessive context switches ensue when developers have to resort to other contexts or even applications for obtaining missing information.

In a typical usage scenario, a developer wants to know the overall design of the component-level architecture of a software for which he just finished the coding for one of its many packages. With a program analysis tool integrated in the IDE he is using, the developer needs to choose a button or menu item relevant to the functionality on the call graph of the entire program. Further, the developer proceeds by looking for all possible interfaces compatible for a function call of interest. Thus the developer has to traverse the call graph and hover mouse cursor over all relevant modules one by one.

However, without prior experiences with the very details of this software, it is infeasible for the developer to know how to make the preceding requests. The key issue, which is really the main obstacle here, is the requirement for the user to recognize which requests to make without auxiliary information from the program analysis tool. As such, the value of the visual aids apparently diminishes. In this case, there is a crucial need of the developer for an IDE that incorporates interactive program analysis tools where the transitions from graphical to textual settings, and also the other way around, are as seamless as possible. This work is motivated by such an observation and the consequent requirement in the design of interactive programming interfaces.

III. Interface Design

Developers spend most of their time on their code for adding new features, making changes, debugging, and understanding source code [10]. When doing these tasks, developers often need also external assistances, such as automatic code completion [13], integrated in their workflow environment which facilitate their development efficiency. To meet such needs, a tentative framework could incorporate three interface design features to help reduce context transitions of developers when they are working around their code.

Figure 1 gives an overview of these design features. Aside the traditional coding view, there are a few other coworker views that assist with communication and collaboration tasks typically seen in a team-development scenario; at the bottom, the context-driven API/example view attempts to provide code examples that are recommended based on current coding context to assist programmers with using APIs of which usages are not familiar to them; finally, the in-situ interface shown in the main code view illustrates the design of porting convenient shortcuts for code manipulations, which are mostly spread over varying places in existing IDEs, to the current focus of editing.

A. Context-driven API/example View

While coding, programmers often have questions about the usage of some third-party functionalities or features [12]. And while implementing a feature, they face hard questions concerning which functions or objects they should pick [11]. To some extent, these questions can be reduced to the needs

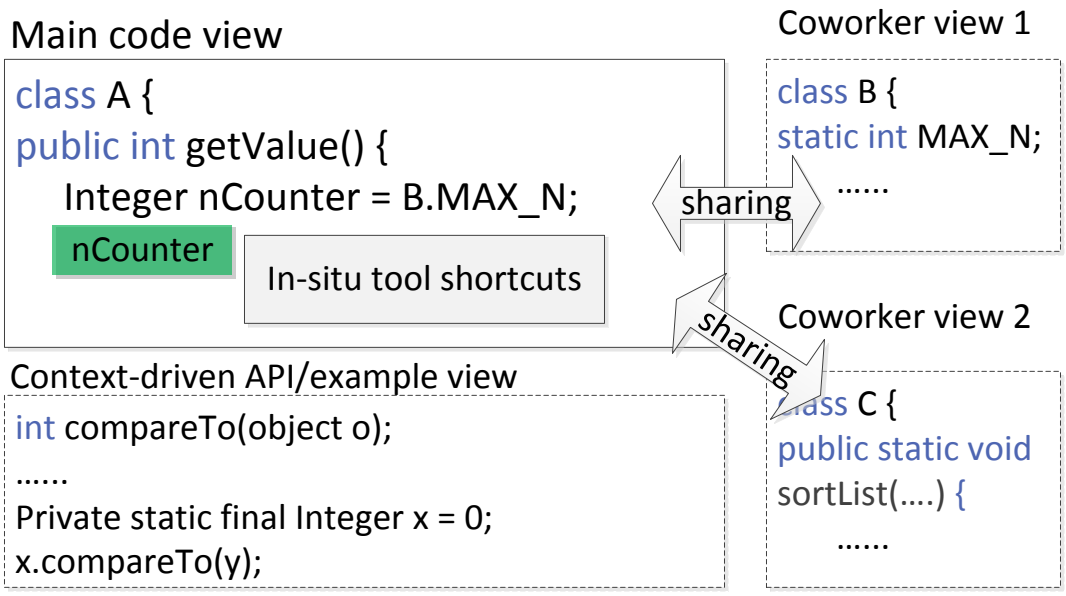

Fig. 1 A new interface design that helps reduce context switches of developers between coding and getting aids, and supports close collaborations among coworkers in software development teams.

for getting function usage information and, even further, illustrations of that usage with example code. Active code completion [13] via API menus already helps developers better than using separate API browsing views, yet it may not be sufficient as developers have to navigate through possibly long API lists (and hover on each one to see the function prototype or API documentation on a floating window as seen in ECLIPSE [4]), which could potentially break their mental model that is focused on the programming logic.

Alternatively, developers may put such assistance back into a separate view closely connected to the main code editing view (as shown at the bottom of Figure 1), where usage information of relevant APIs is displayed on demand based on the current context of object accesses or function calls. Importantly, all relevant APIs are ranked according to their frequency of being used recently as the default mode. Similar solutions have actually been explored more generally from a perspective of the information foraging theory and with respect to software engineering tasks such as programming and debugging [5].

A more important reason for providing the option of moving API usage information to a separate view is the need of combining code examples with the usage. While showing function prototype and/or API documentation is helpful to developers to fill in arguments, it is more beneficial to show them code examples thereof with the usage synopsis. In practice, programmers search code examples with respect to unfamiliar APIs very often (e.g., via Internet searches), even preferably over referring to API documentation.

In this regard, at least three sources of search for such code examples can be taken into account. The first one is the examples coming with APIs in their documentation. Another option is searching in the current code base for relevant examples using context similarity measurement (e.g., calling context and/or type of the object from which the API would be invoked). The code shown in the API/example view of Figure 1 illustrates the result obtained from this source: When the cursor lies immediately after the `Integer` object `nCounter`, the view shows candidate API lists applicable to objects of the `Integer` type, with ones most frequently used recently listed at the top (`compareTo` here) followed by the code example found in the current code base. Such examples give an instant and clear demonstration on how to use the relevant APIs. Finally, an automatic web search, using open search engine programming interface (e.g., Google API), can be initiated with queries for the function usage (e.g., "strtoul C++ example"). Then relevant content can be extracted and put back to the API/example view for developers' reference.

B. Coworker Views

p Another key interface design feature of the proposed framework is concerned with the information needs of multiple developers collaborating in a development team. Previous studies show that in collaborative development one of the primary information sources for developers is their coworkers [14]. In fact, it is very common that when developers have questions regarding how a function or feature is implemented, they tend to first resort to their teammate instead of software documentations [10]. To facilitate developers to take advantages of having coworkers to consult as their information needs arise, it is

potentially rewarding to incorporate a set of coworker views aside the main code editing view (shown on the right-hand side of Figure 1).

The rationale of introducing these additional views is two-fold. First, developers working in the same team can easily share their source code in real-time when necessary. One example case in which this sharing could be useful is when a senior developer coaches a team member in familiarizing him with the team project. Another example can be seen in agile development, where one developer could quickly prototype his function according to the ongoing implementation of a function being written or debugged by an another developer. As shown in Figure 1, the current developer is writing the method `getValue()` for class `A`, with a reference to the static variable `MAX_N` of class `B` that is being implemented by a coworker. Having the choice of checking the implementation of a component developed concurrently by a teammate, on which current coding task is dependent, will save a developer's time seeking for the information about that component in other more expensive ways.

Second, such views can enable close collaborations among physically distributed teammates. For instance, a developer who needs one of his teammates to demonstrate how to write or debug a piece of code would readily get the help from such views without moving to a different seat or office, or resorting to external instant-messaging tools. Furthermore, if screen space allows, such benefits can be even augmented with multiple coworker views open at the same time—enabling closer collaborations among a group of developers.

At the first glance, the above interface designs seemingly conflict the goal of reducing context switches by developers, because those extra views potentially end up with more context switches. However, the overall cost of context switching will be mostly reduced indeed as the total time developers would spend on getting the information from these views can be much greater without these integrated views and information. To illustrate, consider finding the code example for an API again. Without the automatic code reference shown within the IDE, a developer would have to search online or consult to other sources that are available usually in different interfaces from the whole IDE (e.g., a different application such as internet web browser).

C. In-situ Interface Elements

Almost all IDEs today contain a main menu (usually placed at the top) of the entire interface, followed by one or several rows of tool shortcuts shown as buttons or icons. Although usually those menus or shortcuts can be situated differently, few of them is tightly incorporated into the working area of developers where the underlying functionalities of those tools will be applied to. For example, there has always been a considerable "visual distance" from the code being focused on by developers and the shortcuts to functionalities developers need to utilize on that code. While the context switches in such situations are not as large as those seen in cases where developers seek coworker resources without coworker views, such distances could be much reduced. Accordingly, two possible interface improvements to reduce the unnecessary distance can be investigated.

First, in-situ tool shortcuts can be added to the main code editing view. The presence of such gadgets is contingent on user actions of marking focus on (e.g., selecting) code elements to which the shortcuts are applicable; and the composition of the gadgets is determined by the characteristics of the code elements being focused on by developers. As an example, Figure 1 shows, in the main coding view, an "in-situ tool shortcuts" bar appears aside the object `nCounter` when it is selected through double-click and the mouse cursor hovers nearby—the gadget disappears once the selection is revoked or the cursor moves away the focused object. This is akin to the in-situ formatting toolbar in Microsoft Office, triggered by double-clicking on a word.

The more important part of this design is the demand-driven composition of the gadget. To effectively reduce perceptual transitions within the IDE, the in-situ tool gadget should contain most, if not all, shortcuts to functionalities that developers would possibly use for the focused object. This decision can be made in reference to developers' common information needs with respect to that object, based on such criteria as the object's type. For instance, for a function identifier in its invocation statement, example shortcuts would be "caller list", "rename", "declaration" and so on.

Second, the presence and layout of visual components should be demand-driven. As developers usually work on multiple tasks during their development workflow [10], they tend to switch among multiple sources of information. Yet, they can mostly focus on one task or information at a time only. The IDE thus needs to optimize the size and composition of the particular visual space that a developer has to concentrate on for completing a specific task, while diminishing the presence or even phasing out all visual components irrelevant to the current task.

For example, when a developer is right in the process of typing code, visual components, such as the top menu and main tool bar, side panels, and bottom debugging views, become irrelevant and thus should automatically disappear so that the main coding view gets its maximal visual space. Some IDEs, such as Microsoft Visual Studio, have already incorporated similar features (e.g., dockable gadgets), yet often just passively rely on user settings to apply those features. Also, the overall design there does not support automatic adaptation of interface layout and composition to user action and workflow contexts. The dockable gadgets, for instance, can be set to hide when mouse cursor moves away from them, but the layout and elements of the gadgets do not automatically accommodate developers' information needs varying in the development workflow. In addition, all visual components that

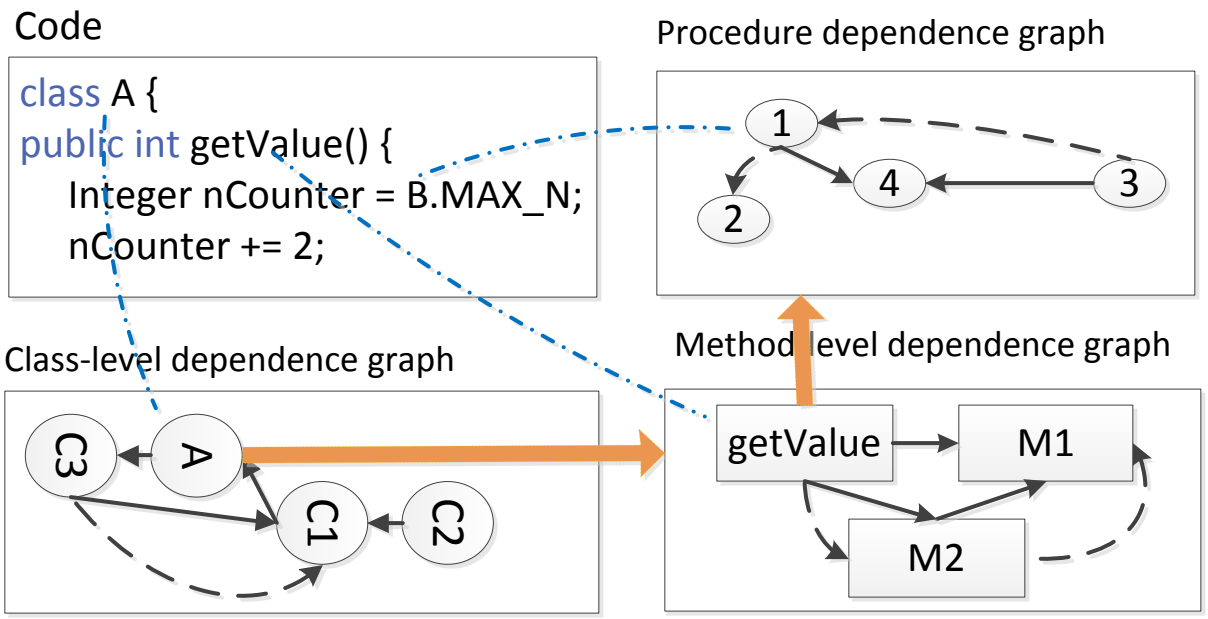

Fig. 2 Multiple linked visualizations of source code to integrate multiple code information in synergy.

are not applicable to the current developer action can phase out from the interface and come back when they become applicable again. In contrast, most IDEs choose to disable those components while still leaving them in the interface hence wasting the limited visual space of the IDE.

IV. Visualization Design

When offering means to help developers obtain various information, most IDEs also implicitly force developers to switch among different visualizations of those information, potentially leading to expensive workflow interruptions. An additional issue is that usually these visualizations are separated from each other, without explicit links among them, forcing developers to maintain an extra mental model linking those information in mind. In this context, it is reasonable to leverage multiple linked visualizations of source code, along with the source code itself, to facilitate the code understanding and navigation for developers.

A. Multiple Visualizations of Source Code

Not only can information visualization greatly aid data understanding, but multiple forms of information of the same data set could be also, or even more, helpful (e.g., [15]). During their mental workflow for code understanding as the primary task [10], developers can greatly benefit from multiple visualizations of source code information besides the textual form of the code itself.

A relevant proposal would be to utilize multiple visualizations of program source code, which are interconnected underneath the source code, to enable more effective program understanding. Figure 2 illustrates the visualization design feature for the next-generation IDEs using dependence graphs as the example information representation of source code (there are other forms of such information of code, such as call graphs and type hierarchies as mentioned above). Surrounding the traditional code editing view (upper left) that provides the textual representation of source code only, three other satellite views show the dependence graphs of the source code at three levels of detail, namely the (statement-level) procedure dependence graph (PDG) [16] (upper right), method-level dependence graph (MDG) (bottom right) [17], and class-level dependence graph (CDG) (bottom left). Different styles of arrow lines in the graphs illustrate different types of dependencies (e.g., solid lines are for data dependencies and dashed lines for control dependencies). The three dash-dotted lines across the views illustrate the links between the source code and each of these graphs. Given that navigating dependencies is one of the core steps in software development tasks (e.g., maintenance and evolution), visualizing these dependencies at multiple levels in the IDE can greatly reduce developers' efforts in seeking such information for comparison purposes [18].

Optionally, these visualizations can be selectively or all added to the IDE. A more synergetic design is to *seamless synthesize* the four views *in one*. In the latter design, instead of showing more than one or all views at the same time, only one view is visible at a time. The motivation is that, again, while developers can be greatly benefited from multiple visualizations, they could utilize one of them at one time only. The idea is to switch among these visualizations by *zooming in/out* operations (shown by two wide arrows in the figure).

As with Google Maps, when developers zoom out from a statement, they will first switch to the PDG visualization with central points automatically set to that statement; then they can navigate on the PDG and zoom in from any node thereof back to the source code view. If developers continue to zoom out when they are in the PDG visualization, they switch to the MDG view where they can also navigate, at the method level, and zoom in back to the lower levels of details. Switching between MDG and CDG is similar. Alternatively, zooming in from a method declaration while in the source code view can directly lead the developer to the MDG visualization and, similarly, zooming in from a class declaration in the source code leads developers to the CDG view immediately.

B. Interactions across Linked Visualizations

The multiple visualizations of source code are also mutually linked, since they all represent the same data (i.e., the source code). An additional merit of the multiple linked visualizations lies in more effective interactions. One simple example is that selecting all lines of code of a method can be more easily done by just selecting that method on the MDG visualization view; selecting or deleting a whole class will see greater benefits in similar ways. Moving code around through interactions on the dependence graph visualizations would be even more beneficial. For example, a developer can quickly start writing a method by cloning an existing one by copying the corresponding node on the MDG view. When multiple visualizations are shown simultaneously, more interactions can be enabled, such as moving or copying a method from one class to another. Of course, feasible interactions on the graphical visualizations are constrained by feasible automatic source code level operations.

V. Discussion

To demonstrate the practical benefits of the new design features proposed in this work, a tool implementation and an empirical study on effectiveness of those features are both required. This section outlines major considerations for such next steps and presents an even broader perspective on enhancing IDE design for other software engineering tasks than cooperative development and program comprehension as discussed above.

A. Evaluation

While ideally a brand-new IDE would be implemented to fully realize the design principles for the next-generation programming interface that maximizes development productivity, it is more realistic at this stage to start with incorporating experimental features into existing IDEs that are being commonly adopted in practice. For instance, the three interface design enhancements can be implemented as plug-ins for Eclipse using the JDT Text and UI components [19]. In particular, the context-driven API usage and code examples can be retrieved from the local code base (i.e., the developer's project repositories) and results of online code-search service (e.g., [20]) through existing web-search extensions of Eclipse (e.g., [21]).

In contrast, implementation of the two visualization features would be facilitated by using some external information-visualization libraries, such as VTK (with Java wrapping if necessary) [22]. On the other hand, the underlying data for such visualizations will come from the results of relevant program analysis such as the computation of program dependencies at different levels of granularity (e.g., statement, method, and class levels). The program analysis can be either realized through existing relevant plug-ins if readily accessible, or implemented on top of a third-party program-analysis frameworks (e.g., Soot [23] for Java programs).

Using these implementations, the proposed features can be evaluated through user studies where professional developers are to be asked to use both the enhanced IDE and an existing one to perform a same set of common development tasks. Two key elements of the user-study design will be the list of tasks that developers would regularly perform during software development, and the metrics that measure the usefulness of the added features. Among others, example tasks for such a study may include (1) developing a program according to given specifications, which necessitates using a set of APIs unfamiliar to the developers, (2) describing the functionalities of a software component, which requires reading the source code first, and (3) finding faults in a given program, which needs debugging within the new IDE.

To gauge the effectiveness of the new features, main metrics may include, respectively corresponding to the above example tasks, (1) the degree of conformance of the resulting program to the specifications, (2) the accuracy of the functional description, and (3) the outcome of being either success or failure in locating the bug. In addition, the time costs of finishing these tasks will be another key metrics as well. At least two groups of developments, one group using the baseline IDE (without the new features incorporated) and the other using the new IDE, are needed to compare the effectiveness between them so as to indirectly measure the benefits of the proposed design. And a hypothesis testing can be used to examine the statistical significance of the benefits, through that of the differences in the metrics between the two groups.

B. Extensions

As suggested in the foregoing design of empirical evaluation, developers need not only the information collected from external sources (e.g., API usage examples) but also that extracted from the program itself. And the latter usually comes from the outputs of certain program-analysis techniques. In fact, for many development tasks, such as regression testing and debugging, the results from program analysis are needed more often than those immediately available in existing artifacts

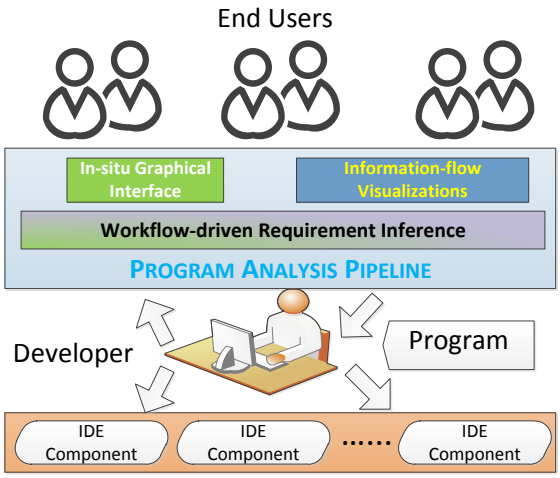

Fig. 3 Illustration of a generalized IPAE framework (top) working in synergy, through interactions of the developer (middle), with an IDE that incorporates the proposed design features and beyond (bottom).

such as source code and documentation. For instance, other than dependence representations (graphs) as mentioned before, such results can also include a selected subset or prioritized ordering of a test suite to be executed from a regression test selection or prioritization analysis, program entities potentially impacted by an ongoing change to be inspected from an impact analysis, and candidate code region possibly containing bugs from a fault-localization analysis.

In a much broader sense, the proposed design features will be part of a more general framework of interactive program analysis environment (IPAE) that works in synergy with the next-generation IDEs, as schematized in Figure 3. Overall, the IPAE infrastructure is characteristic of a *human-centered* design: The developer is at the center of the entire work flow of software development, and acts both as the user of various IDE components (at the bottom) and the IPAE, as the two arrows on the left indicate, and as the producer of the software being developed for which the end user (at the top) requests, as symbolized by the two arrows on the right. The software requirements originate from the end user, while the ultimate output of the infrastructure feeds the developer.

Highlighted in the diagram is the three-layer hierarchy of the IPAE infrastructure, underlaid by the *program analysis pipeline* which analyzes the information flow between the developer and the environments (i.e., IPAE and IDE) so as to automatically trigger program analysis relevant to current developer actions and recommend the resulting data to developers. This automatic recommendation of information that developers may need in their specific task scenarios differentiates the proposed approach from existing alternatives like the Dragon [8] and Dimple frameworks [7], which entirely rely on user inputs to passively produce additional information. Note that when developers need to deal with a great amount of information in a complex development environment, automatically recommending the most relevant information they are most likely to access for their current tasks can also avoid excessive context switches that would ensue for searching those information.

Specifically, to generate these automatic recommendations, the bottom layer of the program-analysis pipeline hosts a *workflow-driven requirement inference* component. This component *monitors* developer operations (e.g., running unit tests, editing code, and debugging, etc.) and *infers* potential next information needs. Utilizing history data on developer operations and decisions, the inference can be realized through automatic reasoning and statistical learning algorithms. Then, the prediction results are sent back to the program-analysis pipeline, which will invoke analysis routines required for generating the information needed next and transfer the results to the *information-flow visualization* and/or the *in-situ graphical interface* modules above. Finally, the visualization module renders the recommended information (if appropriate to visualize) and push the visual representation to developers through specific IDE components (as containers of the visual representation); in the meanwhile, the interface module responds by creating new in-situ gadgets that offer visual aids for developers to proceed with the current development task, and by adjusting the IDE layout as well when necessary.

It should be noted that these two modules, while part of the IPAE infrastructure, are both linked to the IDE through particular components (e.g., panels, menus, and dialogs, etc.) and they share the spirit of design with the visual interface enhancements as laid out for improving the IDE itself in previous sections. Also, in terms of implementation, the IPAE infrastructure and the IDE can share common utilities.

C. Limitations and Challenges

The development of large-scale software systems is usually collaborative, relying on cross-team communication and coordination. While the design improvements proposed are able to assist developers with such collaborative activities, they might not solve all issues as possible consequences of promoting this development paradigm. First, the cowork-view features can help with the collaboration among a group of developments but for groups with limited sizes only. For example, when ten or more developers debug a large component consisting of multiple smaller submodules in a collaborative manner, it may not be feasible for the IDE to host ten or more coworker views each dedicated to the operations of an individual developer simultaneously although ideally it could do so: The impediments due to the resulting visual clutters would overweigh the benefit of having these coworker views, even more so if the developers want to track some debugging information (in additional views or gadgets) only relevant to individual interests at the same time. In addition, when also

incorporated in the IDE, the in-situ interface needs to take extra visual regions in the workspace too, which may further aggravate the challenge, potentially forcing the developers to dismiss either the in-situ feature or (at least some of) the coworker views.

An another challenge lies in possibly negative effects of computational costs underneath the visualizations on the work efficiency of developers using the proposed visualization features. Take the multiple visualizations of source code for example, computing the dependencies at various levels of granularity naturally incurs extra time which may not only slow down developer tasks but also interrupt the mental workflow of developers. The problem can be even worse when the program under analysis is very large or highly complex such that the corresponding dependence graphs are large and dense, since visually manipulating such graphs is a challenging task itself.

One possible solution is to create and visualize the dependencies for code segments that the developers are currently working on instead of for the entire program. Yet, a drawback of this solution is that the developers would need to manually maintain the dependencies across multiple such code fragments (and their dependence graphs). On the other hand, to reduce the time overheads for computing the dependencies, the graphs may be all built beforehand and then be retrieved on demand (from the offline computations). However, such a precomputation approach would impose limits on how much and often the developers can change the source code with respect to reusing the dependencies: Once the code is modified, developers may need to check updated dependencies in accordance with the updated program.

Finally, as explored in [24], it would be desirable to have customized interface features in IDEs that are particularly useful for specific language features. As such, some of the design enhancements proposed here would best fit certain languages yet may not well suit others. Apparently, designing an IDE that equally benefits all of them would be even more difficult than improving an IDE for one particular language. When considering the differences between languages of disparate programming paradigms (e.g., a domain-specific or dynamic language versus a procedural one), the challenge potentially augments. To help address such challenges, finding a common ground of different IDE design features for different languages could be an initial step: prioritizing IDE features that are less dependent on specific language syntax or constructs (e.g., the visualizations of source code) over those that are more so (e.g., the in-situ graphical interface).

VI. Conclusion

Today's developers usually deal with multiple tasks simultaneously during their software development process, seeking variously sources of information for interleaving tasks such as coding, documenting, testing, and debugging. Accordingly, modern IDEs try to incorporate an increasing number of interface elements to provide sources meeting those multiple information needs, yet mostly tend to reduce developers' productivity by imposing on them an implicit need for frequently switching among many different contexts.

This paper thus explores in that regard and envisions several interface and interactive visualization design features for enhancing today's IDEs in a way that helps developers meet multiple information needs more efficiently. It demonstrates the needs and benefits of incorporating those features in next-generation IDEs and also discusses limitations and challenges of doing so in practice along with possible solutions. Although these benefits need supports of empirical evidences, the discussions here serve as an important preliminary step in advancing programming interfaces and environments.

As two immediate next steps, implementation and evaluation of the proposed design are outlined. In addition, an interactive program analysis environment providing automatic recommendations and generalizing the proposed design has also been sketched up. Beyond what has been presented, there are potentially more IDE design features and principles to be probed in the future.

Acknowledgment

This work was partially supported by ONR Award N000141410037 to the University of Notre Dame. The author would also like to thank the anonymous reviewers for their valuable comments and suggestions that helped improve this paper over its original version.

References

[1] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a cognitive dimensions framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[2] R. Metoyer, B. Lee, N. Henry Riche, and M. Czerwinski, "Understanding the verbal language and structure of end-user descriptions of data visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 1659–1662.

[3] H. Cai, J. Chen, A. P. Auchus, and D. H. Laidlaw, "Composing dti visualizations with end-user programming," *CoRR*, vol. abs/1310.2923, 2013.

[4] "Eclipse luna," https://eclipse.org/.

[5] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[6] M. Czerwinski, E. Horvitz, and S. Wilhite, "A diary study of task switching and interruptions," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004, pp. 175–182.

[7] W. C. Benton and C. N. Fischer, "Interactive, scalable, declarative program analysis: From prototype to implementation," in *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2007, pp. 13–24.

[8] B. Chapman, O. Hernandez, L. Huang, T.-h. Weng, Z. Liu, L. Adhianto, and Y. Wen, "Dragon: An open64-based interactive program analysis tool for large applications," in *International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2003, pp. 792–796.

[9] A. Zeller, "The future of programming environments: Integration, synergy, and assistance," in *Future of Software Engineering*. IEEE Computer Society, 2007, pp. 316–325.

[10] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 492–501.

[11] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*, 2010, p. 8.

[12] ——, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 185–194.

[13] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 859–869.

[14] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th international conference on Software Engineering*, 2007, pp. 344–353.

[15] B. Hanciles, V. Shankararaman, and J. Munoz, "Multiple representation for understanding data structures," *Computers & Education*, vol. 29, no. 1, pp. 1–11, 1997.

[16] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[17] H. Cai and R. Santelices, "Abstracting Program Dependencies using the Method Dependence Graph," in *Proceedings of International Conference on Software Quality, Reliability, and Security (QRS)*, 2015.

[18] A. J. Ko, H. H. Aung, B. Myers *et al.*, "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks," in *Proceedings of 27th International Conference on Software Engineering*, 2005, pp. 126–135.

[19] "Eclipse java development tools," https://eclipse.org/jdt/.

[20] "Source code search engine," https://searchcode.com/.

[21] "Eclipse web search," http://marketplace.eclipse.org/content/web-search.

[22] "Eclipse web search," http://www.vtk.org/Wiki/VTK/Java_Wrapping.

[23] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a java bytecode optimization framework," in *Cetus Users and Compiler Infrastructure Workshop*, Oct. 2011.

[24] P. Charles, R. M. Fuhrer, S. M. Sutton, Jr., E. Duesterwald, and J. Vinju, "Accelerating the creation of customized, language-specific ides in eclipse," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009, pp. 191–206.