Towards an IDE to Support Programming as Problem-Solving

Nicholas Nelson
EECS
Oregon State University
nelsonni@oregonstate.edu

Anita Sarma
EECS
Oregon State University
anita.sarma@oregonstate.edu

André van der Hoek
Department of Informatics
University of California, Irvine
andre@ics.uci.edu

Abstract
Programming is inherently a problem-solving exercise: A programmer has to create an understanding of the situation, externalize and contextualize thoughts and ideas, develop strategies on how to proceed with the task, enact changes according to the most appropriate strategy, and reflect to learn from each problem. Therefore, programming is clearly more than just code input, testing, and maintenance. Current Integrated Development Environments (IDE), however, largely focus on the "writing code" parts of programming. In this position paper, we revisit which activities and actions constitute programming, and highlight six challenges to supporting these activities. We then briefly describe a new paradigm of interacting with the IDE on which we are working to more directly support each of the six activities.

1. Programming as Problem-Solving
Programming is more than dealing with language syntax and semantics: it is inherently an exercise in problem-solving that extends beyond the act of editing code in an Integrated Development Environment (IDE). We are not the first to observe this. For instance, programming has been characterized as an iterative process of refining mental representations of computational problems and solutions and expressing those representations as code (Loksa et al., 2016). Other studies have found evidence for the facts that programming requires gathering information from multiple sources (Sillito et al., 2008), includes creating mental models of program structures (Von Mayrhauser & Vans, 1995), and involves exploring and evaluating many alternatives (Hartmann et al., 2008).

We surveyed the literature from the perspective of programming as problem-solving, with Table 1 summarizing key activities that developers employ when programming. These activities can be partitioned into six categories (*Activities*), with specific actions that represent in more detail how the high-level activities manifest themselves in practice (*Actions*). Clearly, not every task involves all of these problem-solving actions, and there is no linearity to the order in which they are employed. Sometimes an action may not even be observable when it takes place solely in a programmer's head. At the same time, literature has documented that all of these actions do occur and play an important role in how programmers arrive at a solution to the programming problem at hand.

2. Challenges
We believe that it is necessary to fundamentally rethink IDEs, so that they seamlessly and intrinsically support programming as problem solving. Supporting the full set of actions comprising problem-solving in programming, however, involves numerous challenges that must be addressed. These challenges span all six categories of activities, and even those activities that have traditionally been supported by IDEs (*A4*) exhibit gaps when reviewed through the lens of problem-solving requirements.

1. ***How to support programmers' formulation of problems and reflection on potential solutions?*** Programmers do not just arrive at a solution out of nowhere. They need to first contextualize the computational problem in terms of what they know and how they can progress towards a possible solution. This involves exploration, articulation, and reflection on different alternatives, with these actions being interleaved, sometimes even happening at the same time (e.g., developers are known to reflect on a code solution while they articulate it) and typically encompassing several relatively quick iterations. Often there is no single correct solution, and the best solution requires mixing and matching elements from multiple alternative solutions.

Table 1 – Activities and Actions of Programming as Problem-Solving

Activities		Actions
A1	Understanding the situation	Identifying goals Recalling prior knowledge Constructing models Interpreting code artifacts Filling knowledge gaps
A2	Externalizing thoughts & ideas	Representing relevant information Contextualizing information Preserving contextual information
A3	Developing strategies	Generating alternatives Articulating and refining alternatives Understanding and assessing alternatives Recombining aspects of alternatives
A4	Enacting change	Translating strategies to actions Tracking progress Evaluating and assessing change
A5	Collaborate	Feedback solicitation Team work Group think Leverage group knowledge Synchronization
A6	Retrospect	Reflect on work Preserve work

2. ***How to provide programmers access to the relevant context in a problem space?*** Programming solutions must exist in the context of the rest of the codebase and its related artifacts. Programmers need to understand where a code snippet fits in a code base, what it calls out to, and what calls into it (de Souza & Redmiles, 2008), the desired behaviors of the existing codeand the code to be produced (e.g., computational speed, usability, features), organizational policies (e.g., licensing, process standards, code style), and historical development (e.g., has a solution previously been tried and rejected). Information that defines this context is not always readily available and, instead, must be cobbled together from multiple different types of and sources of artifacts. A developer needs to know where these individual pieces of information reside and how to obtain just those items that actually pertain to the problem at hand.

3. ***How to support different information processing style and workflow of programmers?*** Programing is a creating activity; no two programmers arrive at the same solution in the same way. For example, female programmers tend to process information comprehensively, seeking a more complete understanding of the problem before starting (Grigoreanu et al., 2012), whereas male programmers are known to use a more heuristic (or selective) approach. As another example, visuospatial reasoning is critical for abstract knowledge and inference, and is a core component of how we view the world, but it has been observed that programmers bring their own personal visuospatial reasoning and information processing style to evaluating and organizing artifacts, solutions, and ideas (Tversky, 2005).

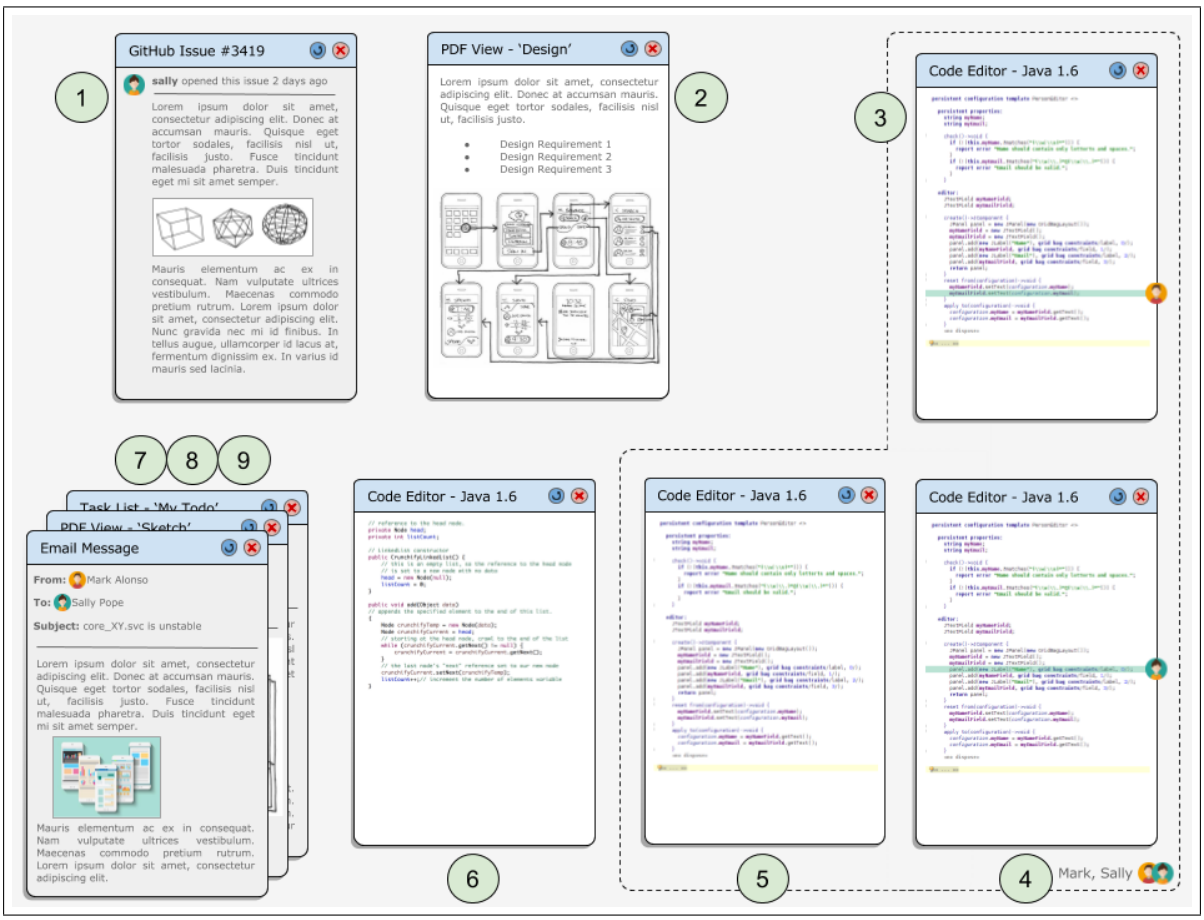
4. ***How to support programmers in relying on past experience?*** Problem-solving in programming is not a one-off activity performed in isolation. Instead, programmers rely upon past experiences with similar problems, knowledge gained from previous artifact interactions, and prior mental models. The question arises of how this prior experience can be brought to bear, easily, when a programmer faces a new situation. Apart from the programmer memorizing what they might have done in the past and looking this up, current IDEs provide no support in this regard. The challenge lies in how a new IDE can provide this kind of assistance.

5. ***How to enable collaboration between programmers across all artifacts involved in problem solving?*** As already highlighted, programmers contextualize their work with all sorts of different ar-

tifacts. Yet, current IDEs generally only allow sharing of the code being worked on. Successful collaboration, however, requires sharing of all different artifacts so all participants have access to the full context. Moreover, it requires ongoing programmers to know how their changes may affect others, and who may be making changes that affect their own work (de Souza & Redmiles, 2008). Finally, they also need to understand the provenance of design decisions, and how and why these decisions were made.

6. ***How to utilize different pieces of information and context to support the act of coding?*** Solutions to computation problems must eventually be represented in code. Converting a conceptual solution into actual lines of code is a non-linear activity (coding sessions start and stop), occurs concurrently with other problem-solving activities, and is loosely organized (e.g., solutions are partially implemented, abandoned, and recovered). Creating a simple, elegant software solution hinges on complex, exploratory coding sessions, and the IDE must support this process fluidly.

Figure 1 – Cards-based User Interface of a Problem-Solving IDE



3. Toward A New IDE

We are in the initial stages of rethinking what an IDE might look like when it is designed from the ground-up to support programming as problem solving. We particularly are exploring new metaphors for how users would interact with the IDE (compared to the bento-box paradigm underneath today's IDEs), as rooted in the specific actions that we know programmers engage in and that must thus be supported (conform Table 1). At present, the leading metaphor for our design is that of cards (as inspired by code bubbles (Bragdon et al., 2010)): the IDE presents all information and enables the developer to make changes through individual cards (see Figure 1). We believe cards, and the affordances that they have, allows the envisioned IDE to better match what programmers actually do. We briefly highlight five key elements in this regard together with which challenges (*C#*) we identified they would address.

- Cards serve as containers for different entities created during problem-solving sessions. They span different artifacts (e.g., an open issue that is being worked on (Card 1), sketches that capture designs (Card 2), or code snippets for reviewing and editing (Cards 3–6)). This diversity of cards is crucial, as it unifies how programmers interact with different types of information that they need in formulating problems and reflecting on them (*C#1*).

- Cards can be spatially arranged or grouped by programmers in the open canvas. Since cards can be grouped in different ways, this supports different information processing styles and workflows (*C#3*). For example, cards can be placed side-by-side (Cards 4–6) allowing for the exploration, reflection, and mixing and matching of alternative solutions (*C#1*). Such comparative reasoning has been shown to be critical for creating good solutions (Hartmann et al., 2008).

- Cards can be organized in *groups* or *stacks*, which allows programmers to organize the different entities to create the context that is needed in the problem solving exercise (*C#2*). For example, cards can be organized spatially (Cards 3–5) such that all relevant cards for a given context are visible at the same time, or they can be stacked (Cards 7–9) to consolidate cards that are relevant, but are not immediately needed.

- Card groupings (in *groups* or *stacks*) are type-agnostic, that is, cards of different types can be combined together (Cards 7–9). Cards can also reflect programming elements at different points in time (Cards 4, 5), allowing comparisons between different versions of the same artifact to provide historical context to the current change (*C#4*).

- Cards are created within an individual programmer's workspace, but are easily shared between different programmers (e.g., Cards 3–5 are shared with two teammates: Mark and Sally). Further, each card is dynamic, automatically updating itself to reflect the latest changes, which provides an awareness of what other programmers are currently working on (*C#5*), thereby allowing developers to create some form of workspace awareness (Da Silva et al., 2006).

To date, we have merely engaged with the 'theoretical' exploration of these ideas. Our next step is to construct an actual prototype implementation to take them forward toward supporting programming as problem solving.

4. References

Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., et al. (2010). Code Bubbles: A working set-based interface for code understanding and maintenance. In *CHI* (pp. 2503–2512).

Da Silva, I. A., Chen, P. H., Van der Westhuizen, C., Ripley, R. M., & Van Der Hoek, A. (2006). Lighthouse: coordination through emerging design. In *OOPSLA-ETX* (pp. 11–15).

de Souza, C. R., & Redmiles, D. F. (2008). An empirical study of software developers' management of dependencies and changes. In *ICSE* (pp. 241–250).

Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., & Kwan, I. (2012). End-user debugging strategies: A sensemaking perspective. *TOCHI*, *19*(1), 5.

Hartmann, B., Yu, L., Allison, A., Yang, Y., & Klemmer, S. R. (2008). Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *UIST* (pp. 91–100).

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016). Programming, problem solving, and self-awareness: Effects of explicit guidance. In *CHI* (pp. 1449–1461).

Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and answering questions during a programming change task. *TSE*, *34*(4), 434–451.

Tversky, B. (2005). Visuospatial reasoning. *The Cambridge Handbook of Thinking and Reasoning*, 209–240.

Von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, *28*(8), 44–55.