CodeBasket: Making Developers' Mental Model Visible and Explorable

Benjamin Biegel, Sebastian Baltes, Ivan Scarpellini, and Stephan Diehl
Department of Computer Science
University of Trier
Trier, Germany
Email: {biegel,s.baltes,diehl}@uni-trier.de

***Abstract*—One of software developers' most important activities is exploring the broader context of a certain programming task, which strongly requires navigating source code and working out a mental model of the collected information. Without tool support, creating and maintaining this mental model leads to significant cognitive load because developers have to handle both relating relevant source code entities to their mental model as well as remembering already explored search paths. Furthermore, the opposite direction, that is, recalling relevant facts out of the mental model, and subsequently, seeking corresponding entities within the source code, demands similar cognitive efforts. In this paper, we introduce *CodeBasket*, an approach for making developers' mental model visible and explorable. As for that, *CodeBasket* helps developers keeping their mental model persistent by providing a two-dimensional canvas on which they can freely arrange visual representations of source code entities, named *eggs*. Since those eggs are linked to the underlying source code, eventually, they can be used for navigating directly to related source code entities. We implemented a first prototype as a touch-enabled web application that is connected to a conventional integrated development environment. In order to get early feedback on our approach, we used *CodeBasket* within a formative study.**

I. Introduction

Developers spent plenty of time comprehending software, which requires extensive searching and navigating within source code documents [1], [2]. Previously visited source code entities are revisited frequently [3], [4] and developers' navigation paths, from one entity to another, are mostly specified by lexical similarities or structural dependencies [5]. By navigating across source code, in respect to a given programming task, developers collect facts about relevant source code entities [6], relate them to previously considered entities, and finally bring them together in a common task context [7]. While in theory this procedure seems to be a simple and very systematic approach, the opposite is true in practice. Developers mostly keep task contexts in their head, and thus, creating, maintaining, and recalling such mental models simultaneously leads to significant cognitive load, is very time-consuming, and, especially after interruptions, parts of the model are getting lost [8], [9]. It is not surprising that developers only put effort in creating new task contexts or extending existing ones if it is absolutely necessary [10].

Being aware of the previously mentioned cognitive efforts, it is desirable to provide tools that are capable of externalizing developers' mental models, thus making them permanently available [8]. But in practice, such tools have a low acceptance and are rarely used [2], [10], [11]. An explanation could be that developers prefer to use tools that are seamlessly integrated in the working environment they are familiar with [12], [13]. Alternatively, in order to keep or to built up a mental model, developers take notes [11], create sketches [14], use the current dynamic state of the integrated development environment (IDE) [1], or simply rearrange source code in such a way that related source code fragments are close nearby, for example, by grouping dependent or similar methods together [15]. Nevertheless, those techniques still require manually searching for and navigating to corresponding source code entities.

Motivated by the previously mentioned observations and findings, we introduce *CodeBasket*, a novel approach that addresses both externalizing developers' mental models into a visual representation as well as using those representations as navigation maps for instantly seeking corresponding source code entities. This not only makes developers' mental models visible, but also explorable. In *CodeBasket*, source code entities are represented by colored and labeled ellipses, named *eggs*, which can be arranged spatially free on a two-dimensional canvas. Those eggs are permanently connected to their corresponding source code entities, and thus, can be used for searching and navigating to entities that are relevant for a given task context.

II. Concept and Prototype Implementation

In this section, we first summarize the iterative process of creating a mental model. Then, we introduce the general concept behind *CodeBasket*, and especially, discuss which role *CodeBasket* could take within the process of program understanding. Finally, we introduce our first prototype implementation and discuss several application scenarios.

A. Creating a Mental Model

In the following, we summarize the model of program understanding introduced by Ko et al. [1] as a representative for other models. Before solving a specific programming problem, developers might have questions or unconfirmed assumptions about parts of a program. At this point, developers are forced to start a program understanding cycle, and thus seek for relevant source code entities that could lead to the required answers and facts. Ko et al. describe three phases of

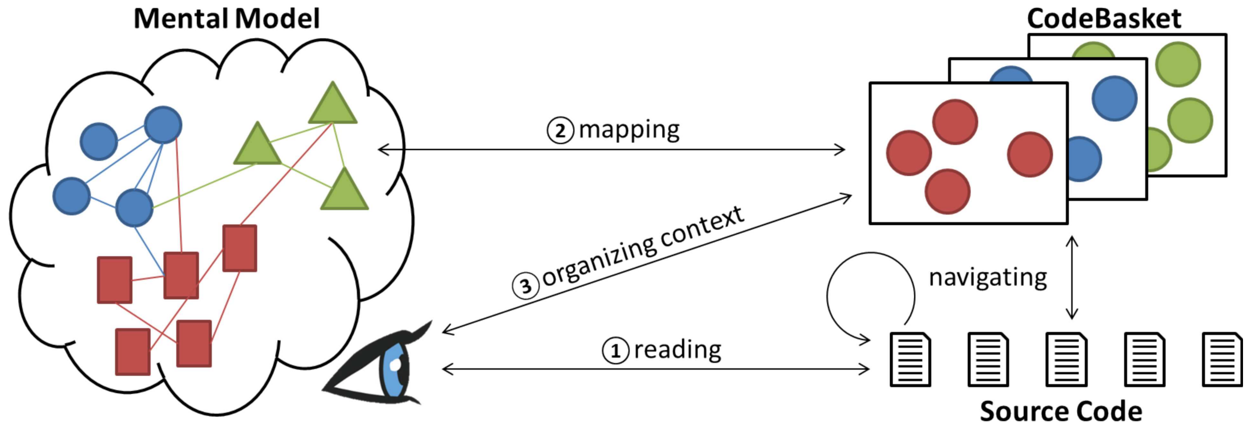


Fig. 1. Externalizing the mental model: The role of *CodeBasket* within the iterative process of creating task context.

this understanding process that are intrinsically tied to each other:

1) *Search:* Developers search for a source code entity that is relevant for a specific task context. Based on information given by the IDE, developers try to decide if the considered entity could be relevant or not. Thus, the *search* phase preselects candidates that could be of relevance.
2) *Relate:* After preselecting a candidate, developers try to understand its context, and thus, relate it to other dependent entities. In general, this step includes lexical and structural navigation and requires extensive explorations of multiple search paths. Relationships to the current entity give developers valuable information if the candidate is relevant and further reveal dependent entities that could also be important for the task context.
3) *Collect:* During both previous mentioned phases, developers permanently try to keep source code entities in mind that could be important for completing the current task.

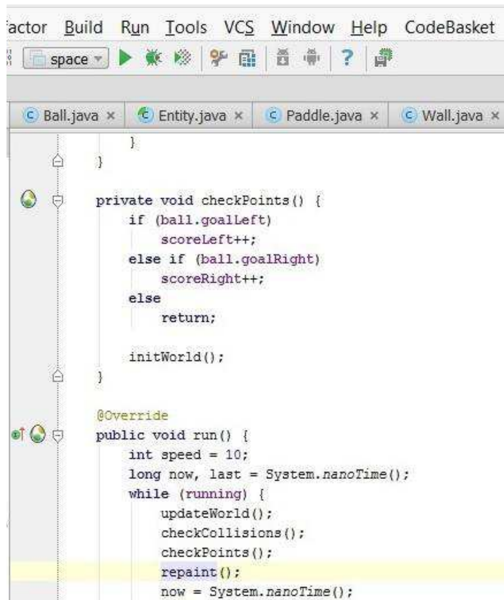
It is possible that further questions or assumptions arise during an understanding cycle, and developers are forced to handle multiple understanding cycles at the same time. During each cycle of program understanding, developers built up small, abstract, and highly connected models [5]. Instantly, at the moment developers think they have collected sufficient information for solving a specific problem, they stop this process of program understanding.

B. Concept

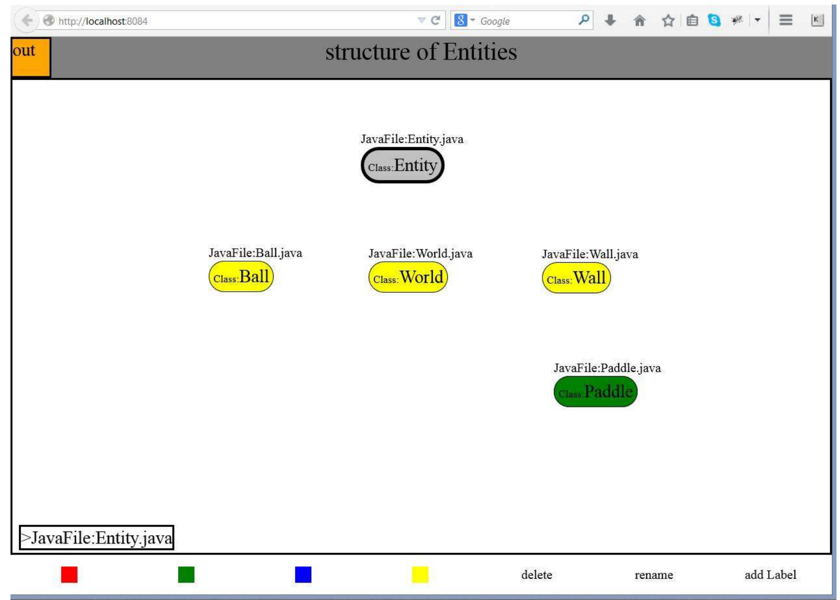
First, we expose which role *CodeBasket* could take within such an understanding process. Since humans are experts in processing and remembering visual information efficiently [16], we believe that externalizing developers' mental model in a visual representation could be a valuable improvement. Figure 1 illustrates the integration of *CodeBasket* into the process of creating task context. Mapping the mental model to a visual, thus visible, representation could enable developers to loop their collected and related information back to the current understanding cycles, especially for further refinement, clarifying, keeping an overview, and handling and maintaining multiple understanding cycles at the same time. A spatial visual representation could help organizing relationships of relevant entities more easily without losing the broader context. Furthermore, an external model can be made interactive and explorable, and thus, it can be used for searching and navigating the underlying source code.

Based on those considerations, *CodeBasket* provides a two-dimensional canvas, named *basket*, on which developers can freely arrange their mental model. Corresponding source code entities are represented by ellipses, named *eggs*. A Source code entity can be a class, a method, or any other code fragment within a code base. Beside the spatial arrangement of eggs, users can also color and label them. By using the same color for different eggs, users are able to visually express contextual relations. Instead of using specialized visualizations, we believe that simple visual representations give developers the freedom to create context and relationships that are not necessarily represented in the source code by lexical or structural dependencies, for example, methods that could be responsible for a bug, several parts of a program that have performance issues, or just a collection of code snippets they like.

For each understanding cycle, a new basket can be created and at any time, developers can switch back to previously created baskets. Baskets are persistent, permanently available, and thus, can be used among multiple coding sessions and can be shared with colleagues. The linkage of *CodeBasket* and the source code displayed in an IDE enables bidirectional searching and navigating: On the one hand, developers can click or tap eggs for instantly navigating to the corresponding source code entities. On the other hand, within the IDE, linked source code entities are marked by egg icons and developers are able to highlight corresponding eggs within the basket by clicking on those icons. While navigating or selecting code, a temporary egg is displayed in the current basket. By simple clicking or tapping the temporary egg, developers can make it persistent and keep it in the current basket.



(a) Linked source code is marked by egg icons on the left.



(b) Pong's class structure represented in the *CodeBasket* webview.

Fig. 2. Screenshots from our *CodeBasket* prototype.

In order to get a better acceptance among developers, we propose *CodeBasket* to be displayed in a separate view, possibly on a tablet besides the desktop, as motivated by Parnin et al. [17]. This enables developers to use *CodeBasket* as an extension of their familiar working environment and is less invasive than other approaches.

Finally, we discuss to which extent *CodeBasket* could be beneficial for each of the previously mentioned phases of the understanding process:

1) *Search:* It might be possible that relevant source code entities in previously created task contexts could also be relevant for following tasks. Thus, in some cases, *CodeBasket* could provide possible candidates that might be relevant for developers' current problem.
2) *Relate:* Since *CodeBasket* makes the mental model visible, we believe that relating considered source code entities to the current task context is much easier. Furthermore, developers have the opportunity to compare different baskets, and thus, relate different models to each other.
3) *Collect:* By making mental models permanently available, *CodeBasket* reduces their inherent volatility and ephemeral nature. When developers are interrupted, the collected source code entities are not getting lost and baskets can serve as reminder to help developers finding quickly back into the task context.

C. Prototype Implementation

Our first prototype supports parts of the concept and was developed as a touch-enabled web application that has a permanent WebSocket connection to an IDE. Figure 2 presents screenshots depicting the integration into the IntelliJ[1] IDE (2(b)) and the web application itself (2(a)). The IntelliJ plugin provides the whole infrastructure for communicating with the webview, especially creating links and enabling bidirectional navigation. Egg icons beside the source code indicate links to corresponding eggs in the basket. In the webview, those eggs are represented as colored ellipses. By default, they are labeled with the name and the location of the related source code entity, but that can be changed by developers at any time. Creating visual connections between eggs is not possible yet. At top, the name of the current open basket is shown. In the top left corner, a menu for loading other baskets is placed. In the bottom, there is a toolbar that provides a predefined color set for coloring selected eggs, and functions for deleting and renaming eggs as well as creating labels. These labels can also be freely arranged on the canvas. In order to avoid using the keyboard on a tablet for text entry, it is possible to use the keyboard connected to the computer running the IDE.

D. General Application Scenarios

While designing *CodeBasket*, we considered several application scenarios and asked ourselves which kinds of relationships or contexts could be represented. A first idea was to use the spatial arrangement for expressing structural dependencies. *CodeBasket* could be used, for example, to quickly outline class hierarchies, similar to classical tree structure representations, beginning at the top with the super class and putting its subclasses side by side in the next lower row (see Figure 2(a)). In contrast, by making call graphs visible, eggs could be arranged from left to right. Moreover, semantically dependent eggs could be grouped spatially together. Furthermore,

[1] https://www.jetbrains.com/idea/

colors can be used to express close semantic relationships between eggs. Besides understanding dependencies between source code entities, developers could also be interested in relationships that are not reflected in the source code, for example, collecting all entities that belong to specific bug reports, task tickets, or unit tests.

III. Formative Study

To get early feedback on our prototype implementation, we conducted a formative study with four PhD students, all involved in software development. After a short introduction to *CodeBasket*, one of the authors presented the approach by explaining the source code of a Pong game developed in the context of a lecture. The IDE and the *CodeBasket* webview were run on two different monitors, the latter being a touch screen. During this presentation, two different aspects of the game were explained: first, the inheritance hierarchy of the involved classes (see Figure 2(b)) and second, the game loop. After the presentation, we interviewed the PhD students in a focus group. The interview lasted one and a half hours. In the following, we describe certain aspects of their responses.

A. Use Cases

One concrete use case for *CodeBasket* that was mentioned during the interviews was keeping an overview of important classes and methods during programming tasks. One participant described the baskets as a "collection of bookmarks", which he would use while debugging. Another proposed use case was using *CodeBasket* in computer science education for explaining source code. The two-dimensional spatial arrangement of the eggs on a canvas could provide a global overview for the students or it could show the order in which the source code artifacts are presented.

B. Feature Requests

One participant posed the question which parts of the *CodeBasket* approach could be automated. He was particularly interested in assistance for filling baskets. For instance, he mentioned the possibility to add all called methods inside a method to the basket. Another participant proposed to generally alter the approach in such a way that the basket is filled automatically and can be filtered afterwards. Moreover, it was proposed to automatically generate baskets from source code navigation paths inside the IDE, adding methods in which the developer spend a certain amount of time. It was also noted that eggs could be automatically arranged in the basket according to the inheritance hierarchy of the involved classes. Furthermore, it was requested to be able to not only use the automatically generated links between eggs and source code, but also to be able to link baskets with each other.

Regarding the appearance of the eggs, several participants asked for a better visual distinction of different kinds of eggs (e.g. an egg representing a method or an egg representing a class). To this end, one could use either different icons or differ the shape of the eggs. However, to was noted that too many icons or too many different shapes could lead to visual clutter on the canvas.

C. Device Setup

At the end of the focus group interview, a discussion about the preferred device setup emerged. The participants compared having an additional monitor with or without touch features, or having a tablet as a second view to the IDE. A disadvantage of using a normal monitor and a touch screen with the same computer would be that the mouse cursor would switch to the touch monitor every time the user taps on it. For a more or less decoupled second view, a tablet was the preferred choice. However, for most use cases, the participants preferred a normal second screen without touch features. They also mentioned that they would need to test the different setups themselves, as it is difficult to judge after just seeing a presentation. One participant said that the setup must fit the specific workflow.

IV. Discussion and Related Work

During the focus group interview, participants mentioned that manually creating eggs could sometimes be cumbersome, and thus, demanded mechanisms for automatically filling baskets with eggs and then manually filtering them by relevance. As for that, different application scenarios were discussed, for example, creating eggs that represent developers' navigation paths, complete call graphs belonging to a specific method, or other lexical or structural dependencies. Obviously, these ideas require to also provide an automatic layout mechanism. We plan to implement some of these automatic filling and layouting mechanisms in future work. The Mylyn tool, introduced by Kersten and Murphy [26], provides such an automatic creation of working sets with links to relevant source code entities that belong to a certain task context, based on a degree-of interest model. With their tool REACHER, LaToza and Meyers [25] introduced a novel approach for visualizing and navigating call graphs, which is represented in a separate view besides source code.

In our focus group interview, participants preferred using a classical two-screen monitor setup. At first glance, they saw no immediate advantages in using a touch device compared to a mouse. Thus, when introducing device setups to extend developers' workspace, similar to the proposed setup by Parnin et al. [17], one has to convince developers of the usefulness of such setups. In future studies, it has to be investigated in which application scenarios additional touch-enabled devices could be beneficial. In previous work, we used an additional view on a tablet to link parts of a sketch with source code entities [23]. We experienced that mobility and touch interactions, provided by tablets, fits well in the scenario of collaborative sketching, for example, within developer meetings.

In academic literature, further approaches for supporting understanding and navigating software projects were introduced. Tools like TagSEA [21], Pollicino [22], and FEAT [24] enhance code bookmarks by providing structural organization with tree-like representations. By using a separate view besides source code, those approaches influence developers' working environment little. Some approaches also offer to share their

bookmarks with colleagues, and thus, enable working collaboratively on a common task [21], [22].

Other approaches are much more invasive and introduce completely novel development environment paradigms. Code Canvas [18], Code Bubbles [19], and Jasper [27] made use of spatially arranging source code fragments on a canvas. Since these approaches avoid using additional views, developers are forced to give up their familiar working environment, which could lower their acceptance. Nevertheless, Debugger Canvas [20] has proven that paradigm shifts can be applied successfully in practice.

V. Conclusion

In this paper, we introduced *CodeBasket*, a novel approach for externalizing developers' mental model, and thus, making it visible and explorable. We have discussed how *CodeBasket* could be integrated in the program understanding process, and further, to which extent it could be beneficial for this task, especially, when handling multiple understanding cycles at once. In order to gain first insights into the general idea of *CodeBasket*, we used a prototype implementation in a formative study. In an interview, the participants gave us valuable feedback and ideas, that will help us improving *CodeBasket*. In future work, we would like to enhance our concept and implementation, and evaluate *CodeBasket* in different application scenarios.

References

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Software Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[2] J. Singer, T. C. Lethbridge, N. G. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada*, 1997, p. 21.

[3] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. M. Burnett, and R. K. E. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011*, 2011, pp. 109–116.

[4] K. Kevic and T. Fritz, "Towards developer- and task-tailored navigation models," in *1st International Workshop on Context in Software Development*, ser. CSD, Hong Kong,China, November 2014, p. Epub ahead of print.

[5] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 7–18.

[6] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, 2007, pp. 361–370.

[7] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Cubranic, "The emergent structure of development tasks," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, 2005, pp. 33–48.

[8] 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*. ACM, 2006, pp. 492–501.

[9] C. Parnin, "A cognitive neuroscience perspective on memory for programming tasks," *Programming Interest Group*, p. 27, 2010.

[10] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, p. 31, 2014.

[11] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 255–265.

[12] M. Petre, "UML in practice," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 722–731.

[13] B. Biegel, J. Hoffmann, A. Lipinski, and S. Diehl, "U can touch this: touchifying an IDE," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2014, Hyderabad, India, June 2-3, 2014*, 2014, pp. 8–15.

[14] S. Baltes and S. Diehl, "Sketches and diagrams in practice," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 530–541.

[15] B. Biegel, F. Beck, W. Hornig, and S. Diehl, "The order of things: How developers sort fields and methods," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, 2012, pp. 88–97.

[16] S. M. Kosslyn, "Graphics and human information processing," *Journal of the American Statistical Association*, vol. 80, no. 391, pp. 499–512, 1985.

[17] C. Parnin, C. Görg, and S. Rugaber, "Codepad: interactive spaces for maintaining concentration in programming environments," in *Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010*, 2010, pp. 15–24.

[18] R. DeLine and K. Rowan, "Code canvas: zooming towards better development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 207–210.

[19] A. Bragdon, S. P. Reiss, R. C. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 455–464.

[20] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger canvas: Industrial experience with the code bubbles paradigm," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 1064–1073.

[21] M. D. Storey, L. Cheng, J. Singer, M. J. Muller, D. Myers, and J. Ryall, "How programmers can turn comments into waypoints for code navigation," in *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, 2007, pp. 265–274.

[22] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen, "Collective code bookmarks for program comprehension," in *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, 2011, pp. 101–110.

[23] S. Baltes, P. Schmitz, and S. Diehl, "Linking sketches and diagrams to source code artifacts," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 743–746.

[24] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, 2002, pp. 406–416.

[25] T. D. LaToza and B. A. Myers, "Visualizing call graphs," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011*, 2011, pp. 117–124.

[26] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, Illinois, USA, March 14-18, 2005*, 2005, pp. 159–168.

[27] M. J. Coblenz, A. J. Ko, and B. A. Myers, "JASPER: an eclipse plug-in to facilitate software maintenance tasks," in *Proceedings of the 2006 OOPSLA workshop on Eclipse Technology eXchange, ETX 2006, Portland, Oregon, USA, October 22-23, 2006*, 2006, pp. 65–69.