On the Use of Emerging Design as a Basis for Knowledge Collaboration

Tiago Proenca, Nilmax Teones Moura, and André van der Hoek

University of California, Irvine, Department of Informatics, Irvine CA 92697, USA
{tproenca,nmoura,andre}@ics.uci.edu

Abstract. Abstractions in software engineering have been used for guidance and understanding of software systems. Design in particular is a key abstraction in this regard. However, design is often a static representation that does not evolve with the code and therefore cannot help developers in collaborating after it becomes out-of-date. Our research group is exploring the use of Emerging Design, a dynamic abstraction, as the basis for knowledge collaboration through its implementation in a coordination portal called Lighthouse. This paper presents the state of the art of Lighthouse and discusses three knowledge collaboration problems that we are currently addressing.

1 Introduction

Collaboration is related to mutual sharing of knowledge [1] and has become an essential part of software development and indeed an important research field in software engineering. Today, most knowledge sharing is either informal or decoupled from the actual artifacts to which it pertains. For instance, in the Knowledge Depot [2], an email-based group memory tool, knowledge is stored in a separate repository that must be queried to find a particular piece of information. This not only creates a hurdle to accessing knowledge, but also leads to the update problem, i.e., in the presence of changes, one has to update two places: the artifacts themselves and the Knowledge Depot.

Our research group is exploring a different kind of solution, one where the knowledge is essentially attached to an abstraction that we are creating as part of a collaboration infrastructure. This abstraction is called Emerging Design [3] and is defined as the design representation of source code as it changes over time. With each code change, the Emerging Design is updated accordingly. Emerging Design satisfies the traditional roles of abstraction (guidance and understanding [3]) and includes support for new roles such as coordination, project management, and detection of design decay.

While the original focus of our use of Emerging Design was on detecting conflicts in code changes [3], we believe it is a particularly promising abstraction to address a broader class of collaboration issues. In this paper, we talk about three such collaboration problems and how we believe Emerging Design serves as good basis for exploring them.

K. Nakakoji, Y. Murakami, and E. McCready (Eds.): JSAI-isAI, LNAI 6284, pp. 124–134, 2010.
© Springer-Verlag Berlin Heidelberg 2010

The remainder of this paper is organized as follows. In Section 2, we review Emerging Design and its implementation in Lighthouse. Section 3 presents three knowledge problems in collaboration and how we believe they can be addressed by building upon Emerging Design. In Section 4, we summarize our ideas and discuss some challenges and future directions to improve our work.

2 Emerging Design

Since a design document illustrates the interactions among modules, it can help developers to gain an understanding of the high-level structure of the system and its interactions [3,4]. However, design is often a static representation that does not evolve (automatically) with the code. Therefore, as it becomes out-of-date, it loses value for developers who need to collaborate.

Our research group is exploring the use of Emerging Design as the basis for collaboration. Emerging Design is defined as the design representation of source code *as it changes over time*. It is a live document that stays up-to-date with all changes made to the system. It is annotated with information about the changes made, helping developers to be aware about how the code structure evolves, and with whom they may need to coordinate their actions in order to reduce and prevent conflicts.

We implemented this approach as an Eclipse plug-in called Lighthouse [5]. Lighthouse presents the Emerging Design view as a UML-like class diagram which is built dynamically as developers implement or make changes in the code. One particular characteristic of Lighthouse is that it does not require check-in of the changes made. Instead, it tracks workspaces, since the goal is help people collaborate and coordinate before sending the changes to the source code repository, so merge conflicts are avoided.

Figure 1 shows the Emerging Design basic representation. It shows the primary elements found in UML class diagrams, such as classes, fields, and relationships, as annotated with additional information. In particular, Lighthouse shows information about the evolution of the code. The *plus* symbol represents an addition of a class/method/field, *minus* represents a removal, and *triangle* represents a change. For instance, in the `ATM` class, the field *value* was added by Max. Another example is the field *balance_inquiry*. We can see that Theo and Bob changed that field, and finally Anna removed it. Notice that this history of changes is presented in a top-down manner, time-ordered with the most recent changes at the bottom.

The use of Lighthouse in a large software product naturally introduces scalability issues with respect to the visualization. This can harm a user's ability to spot a particular events of interest. As a first step to make the Emerging Design more scalable, we developed a variety of filters to improve the user's capability. With these filters, the information shown can be reduced by focusing on particular packages, developers, modifications or some combination of them. As a result, crowded visualizations that clearly indicate a problem can be examined for what that problem exactly is.

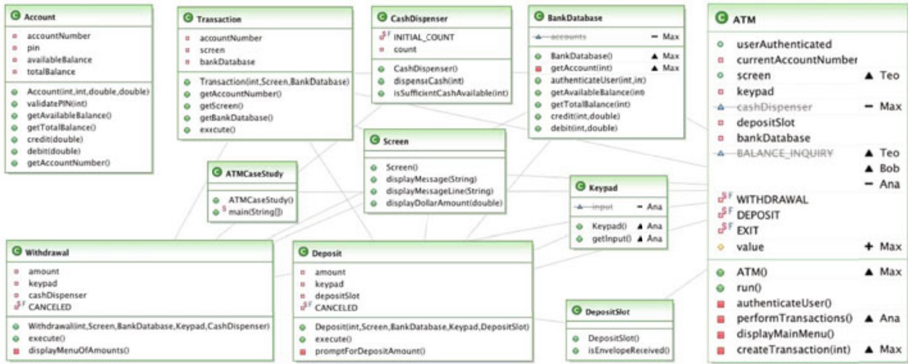

Fig. 1. Emerging Design basic representation

Figure 2 shows an Emerging Design representation with several classes, where each class has numerous events representing the activities of four developers (Max, Bob, Ana, Jim), who are all coding a particular part of the project. The following picture (Figure 3) illustrates how Lighthouse allows users to turn on the filter by developer. In this specific example, the user has chosen to show Bob's code changes. At this manner the user could be aware of any event that is happening in Bob's workspace.

The second filter uses the concept of Java packages. We believe that this feature is very pertinent, because most of the time, while developing, users interact

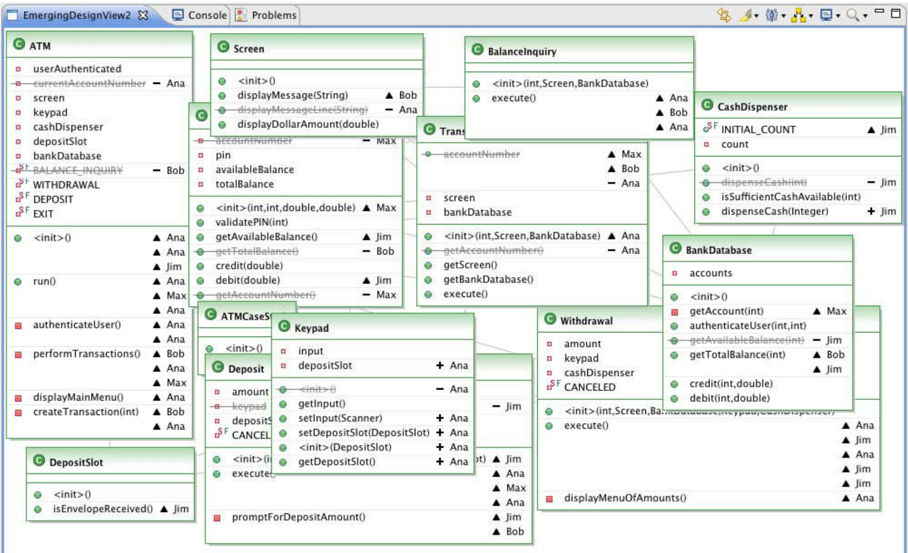

Fig. 2. Emerging Design basic representation with four developers (Max, Bob, Ana, Jim)

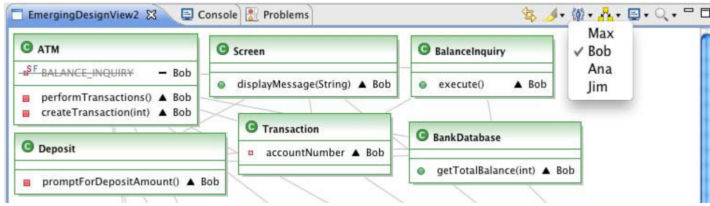

Fig. 3. Emerging Design basic representation with filtering. Just modifications in Bob's workspace are shown.

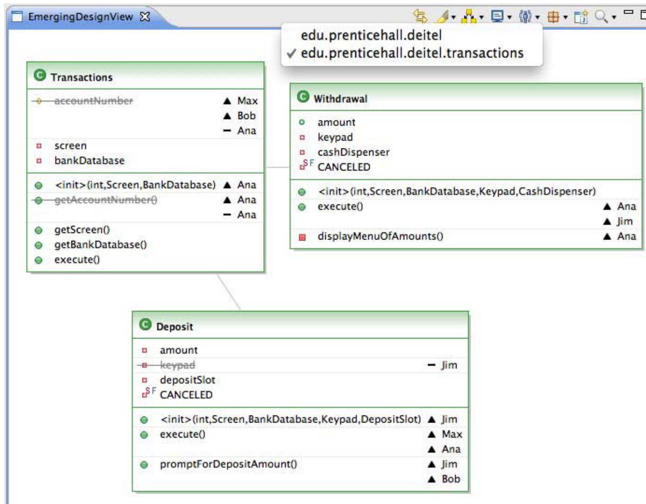

Fig. 4. Emerging Design basic representation with filtering. Just modifications in the selected package are shown.

with only a few package of the whole project. By using this filter, users could pay more attention to specific packages that are related with the task in hand, as can be seen in Figure 4.

The third and last filter only shows the classes that have any modification. So, instead of showing all the classes as in Figure 1, the tool decrease the numbers of nodes from the visualization by hiding the classes that are not being modified for any member, as shown in Figure 5.

To date, Lighthouse is a collaboration portal focused on detecting conflicts. It uses the Emerging Design to show who is making the changes where, and by looking at that, enabling developers to find where their changes may be conflicting with somebody else's. In this paper, we take this work a step further. We outline how we believe the concept of Emerging Design is not only useful for detecting conflicts, but also as a basis for knowledge collaboration. In the next section we talk about three particular knowledge collaboration problems and how Emerging Design can be used as a basis for exploring them.

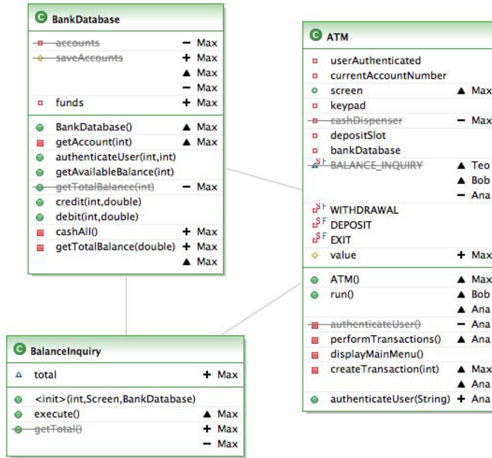

Fig. 5. Emerging Design basic representation with filtering. Just classes with modifications are shown.

3 Three Knowledge Problems

Knowledge collaboration manifests itself in many different forms and may revolve around many issues. In this section, we discuss the following three problems: (1) How to support developers in determining where the implementation is deviating from the original design; (2) How to support finding the right expert related to a given design; and (3) How to support identification of those parts of the program with less than ideal quality.

3.1 Design Decay

It is well known that software changes, and that such changes involve modifications to the original design that may lead to design decay [6]. Prior to the implementation phase, some conceptual design diagrams are usually constructed to guide developers and help them understand the project's high-level picture. The reasons *why* a particular design decays generally are not available, and therefore could be said to represent a knowledge collaboration problem: at some future point in time, other developers must understand why a certain piece of code is like it is, and much rationale resides behind the code changes from the original design to the current state.

In order to illustrate this problem, consider the following scenario: Ana is a developer in a large team and has been assigned to a task that involves making changes in a part of the system with which she is unfamiliar. The previous person that developed that specific piece of code is on vacation and is not available for questions. However, Ana remembers that the project has some documentation, including a detailed UML design diagram that was made before the system's

implementation. Ana finds out that this document includes some notes on rationale for some structural decisions and uses it to find the information she needs to complete the task. Ready to work, Ana realizes that the design does not match with the source code. Some elements have changed, others elements are missing, and no rationale was provided to understand why this has happened and whether or not the changes happened accidentally or intentionally. Ana is left to study the source code in detail to try and understand how to accomplish her task, an unfortunate situation [3].

We can address this issue by marking the Emerging Design, so it shows deviation, and providing facilities for developers to provide contextual information pertaining to the changes they make. Imagine a developer restructuring a certain piece of code in a certain way that is counter-intuitive. By leaving a note, directly visible on the diagram (Figure 6) they now can motivate their change. Other developers can respond either in the affirmative or by expressing concerns and such. A discussion can ensue, for which it is crucial to note that the discussion takes place directly in the context of the artifacts and as the changes are happening. Design decay can be avoided this way, and design evolution becomes under joint ownership of the developers.

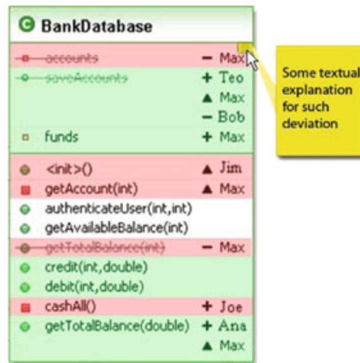

Fig. 6. Design Decay Representation

In Figure 6, the green overlays are used for elements that are present in both the conceptual and emerging design, i.e., the ones that were implemented according to the original design. Red overlays are used for items that are in the emerging but not in the conceptual design, meaning that the implementation diverges from the original design. Elements left in white are the ones that are in the conceptual but not in the emerging design. These elements have not been implemented yet.

The Emerging Design provides a natural basis for addressing design decay because it already tracks design evolution. By now using this basis with simple but powerful extensions, the Emerging Design provides instant knowledge collaboration, both implicitly because it makes visible the design as it evolves

and explicitly because its evolution can be gauged, questioned, discussed, and resolved as needed.

We also note that this can take place both among individual developers at the level of individual or small sets of changes, and by team leaders and architects based upon views of the code as a whole.

3.2 Expertise

The time taken to find an expert is one of the major reasons that co-located work tends to take less time than similar development work split across sites [7]. Quickly finding the right expert related to a given design and/or implementation issue is critical to the success of any software development project. There is a clear knowledge collaboration problem when one needs to understand how some class/method works, why it is as it is, and how it may need to evolve. For instance, in the previous section, because of the absence of the expert, Ana found herself in a situation where she had to study the source code in detail, which implies more time in order to accomplish her task. Often, an expert can provide useful assistance in this regard.

We again explore how the basis of Emerging Design can be leveraged to address this problem. Particularly, we envision exploring the use of a visualization to allow users to browse through the Lighthouse diagram in order to find the proper expert. Since Lighthouse already provides the basis for who made which

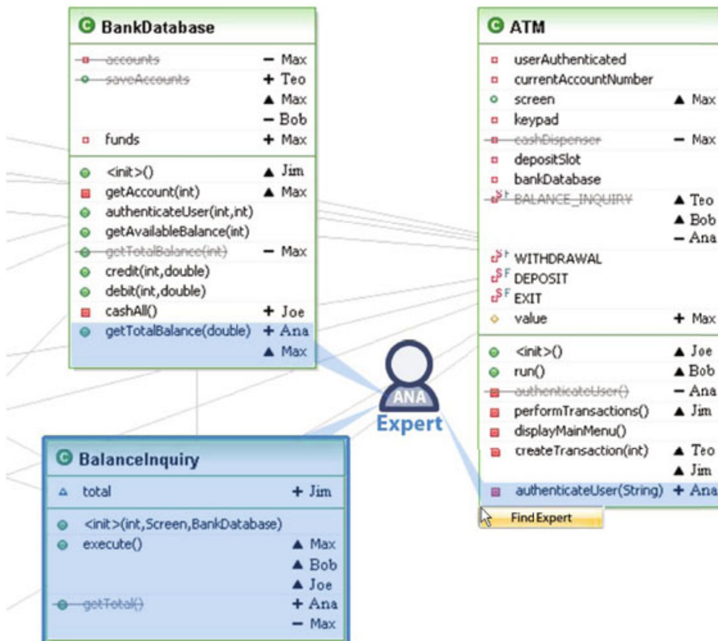

Fig. 7. Expertise Representation

changes, now we can actually build various overlays that make it possible, for instance, to click on one of the authors of a particular method and have the other pieces that they changed highlighted.

In another form, we note that it is often difficult to find someone with broader knowledge pertaining to multiple artifacts and methods. We plan to develop a feature that allows the user to select a group of methods and classes in order to find the expert related to that set of artifacts, as shown in Figure 7.

The advantage here is that, while most expertise systems are limited to work at the level of artifacts, our approach can provide more fine-grained as well as a broader range of answers.

3.3 Code Quality

Software quality metrics can drive software process improvement [8]. Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs [9]. Some information that could be useful in this regard is the overall quality of each class, which if available would enable the identification of the most problematic or complex parts of a project. This kind of information is not usually accessible, representing the third knowledge collaboration problem that we address in this paper.

The use of Emerging Design in this situation would help developers and managers to quickly spot code that is growing without proper quality. We envision a software quality visualization that will show individual factors, such as *number of developers*, *number of recent bugs*, *how well the class/method was tested*, and *number of changes/code volatility* at the bottom of each class. We also take these

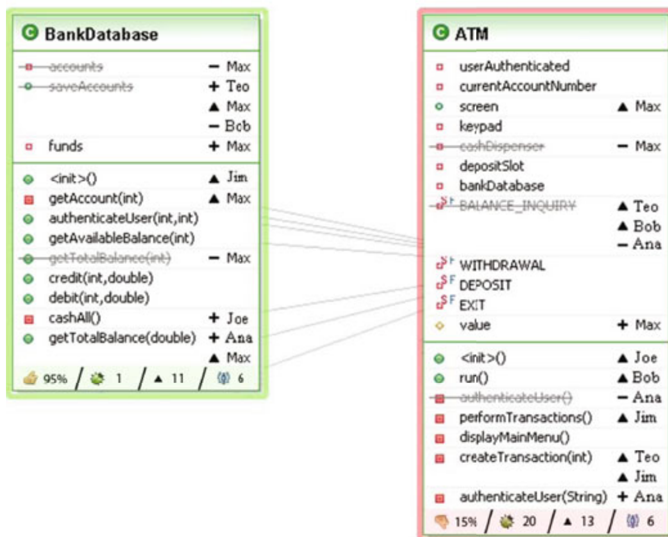

Fig. 8. Code Quality Representation

individual factors in consideration to provide an overall quality measure, and we represent this high level awareness information by using colored border, in which green means good quality and red means bad quality (Figure 8).

We extend the capability of Emerging Design to deal with software quality issues. This approach can help understanding which classes/methods are producing higher quality code. In this way, managers would be able to identify areas that need attention, and also tell what parts of the project are in need of more tests and what parts have enough coverage already.

4 Related Work

Several tools have been created to help people collaborate and to enhance individuals' awareness. The War Room Command Console [10] shows in a public display the current state of a system across workspaces in real-time. The visualization shows the ongoing changes made by developers in thumbprints, a graphical representation of the source code, displayed in a topographic layout. This work, like Lighthouse, uses a program-centered approach to show how changes made by developers are related with the artifacts and how the system is evolving. Its display, however, is in a central location and not on a per-developer basis. Furthermore, the information that it shown is compacted, and does not allow easy access to details.

Palantír [11] provides real-time awareness of changes made by developers and estimates the impact of how severe these changes are. Palantír, like Lighthouse, does not require developers to check-in the changes made and presents a view with information of all developers' workspaces. However, Palantír differs from Lighthouse since it uses a low-level abstraction that focuses on files, while Lighthouse uses the concept of Emerging Design.

FastDash [12] and CollabVS [13] both use a collaboration-centered approach to display the artifacts' interaction among developers. Unlike Lighthouse, this approach uses real-time awareness of developers' activities instead of focusing on program artifacts. The visualization shows people and the activities they currently undertake, e.g., who has which file open or who is editing which file. This approach has the drawback of not providing a spatial awareness of artifacts and it does not provide a historical view of changes made.

5 Summary

In this paper we recapped Emerging Design and presented our vision of the potential role it can play in knowledge collaboration. We described Lighthouse briefly and addressed three knowledge collaboration problems: *design decay* by providing developers with the rationale resides behind the code changes from the original design to the current state; *expertise* by finding the proper expertise for a particular group of methods and/or classes; and *code quality* by providing developers the identification of parts of the program with less than ideal quality.

The benefit we can see is that the knowledge is directly anchored to the artifacts to which it pertains and is thereby easily accessible and intuitive since it fits with the task that a developer is currently working on. Presently, we are engaged in providing this support and we will perform various explorations and evaluations as we build our extensions to Lighthouse. A particular question is whether Emerging Design is useful to support other knowledge collaboration problems as well. Another question is how it can support multiple problems in parallel, as some of our solutions use similar techniques and thus cannot be used at the same time.

Acknowledgments

Effort partially funded by the National Science Foundation under grant number 0920777.

References

1. Rus, I., Lindvall, M.: Knowledge management in software engineering. IEEE Software 19(3), 26–38 (2002)
2. Kantor, M., Zimmermann, B., Redmiles, D.: From group memory to project awareness through use of the knowledge depot. In: CSS 1997: California Software Symposium (1997)
3. Van der Westhuizen, C., Chen, P.H., van der Hoek, A.: Emerging design: New roles and uses for abstraction. In: ROA 2006: Proceedings of the 2006 International Workshop on Role of Abstraction in Software Engineering, pp. 23–28. ACM, New York (2006)
4. Parnas, D.L., Clements, P.C.: A rational design process: How and why to fake it. IEEE Transaction on Software Engineering 12(2), 251–257 (1986)
5. da Silva, I.A., Chen, P.H., Van der Westhuizen, C., Ripley, R.M., van der Hoek, A.: Lighthouse: Coordination through emerging design. In: Eclipse 2006: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange, pp. 11–15. ACM, New York (2006)
6. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Transactions on Software Engineering 27(1), 1–12 (2001)
7. Herbsleb, J.D., Mockus, A., Finholt, T.A., Grinter, R.E.: An empirical study of global software development: distance and speed. In: ICSE 2001: Proceedings of the 23rd International Conference on Software Engineering, Washington, DC, USA, pp. 81–90. IEEE Computer Society, Los Alamitos (2001)
8. Livingston, J., Prosise, K., Altizer, R.: Process improvement matrix: A tool for measuring progress toward better quality. In: Proceedings of 5th International Conference on Software Quality (1995)
9. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. In: ICSE 1976: Proceedings of the 2nd International Conference on Software Engineering, pp. 592–605. IEEE Computer Society Press, Los Alamitos (1976)
10. O'Reilly, C., Bustard, D., Morrow, P.: The war room command console: shared visualizations for inclusive team coordination. In: SoftVis 2005: Proceedings of the 2005 ACM symposium on Software visualization, pp. 57–65. ACM, New York (2005)

11. Sarma, A., Noroozi, Z., van der Hoek, A.: Palantír: raising awareness among configuration management workspaces. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, pp. 444–454. IEEE Computer Society, Los Alamitos (2003)
12. Biehl, J.T., Czerwinski, M., Smith, G., Robertson, G.G.: Fastdash: a visual dashboard for fostering awareness in software teams. In: CHI 2007: Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 1313–1322. ACM, New York (2007)
13. Hegde, R., Dewan, P.: Connecting programming environments to support ad-hoc collaboration. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008, pp. 178–187. ACM Press, New York (2008)