Lighthouse: Coordination through Emerging Design

Isabella A. da Silva*, Ping H. Chen**, Christopher Van der Westhuizen**, Roger M. Ripley**, and André van der Hoek**

*Federal University of Rio de Janeiro
COPPE - System Eng. and Computer Science
Rio de Janeiro, RJ 21945-970 Brazil
+55 21 2562 8675
isabella@cos.ufrj.br

**Department of Informatics
University of California, Irvine
Irvine, California 92697-3440, U.S.A.
+1 949 824 6326
{pchen, cvanderw, rripley, andre}@ics.uci.edu

ABSTRACT

Despite the fact that software development is an inherently collaborative activity, a great deal of software development is spent with developers in isolation, working on their own parts of the system. In these situations developers are unaware of parallel changes being made by others, often resulting in conflicts. One common approach to deal with this issue is called *conflict resolution,* which means that changes have already been checked-in and developers must use merge tools to resolve conflicts and then retest the code to ensure its correctness. Unfortunately, this process becomes more difficult the longer the conflicts go undetected. In order to address these issues, have been proposed conflict avoidance approaches that detect conflicts as soon as they occur. In this paper, we present *Lighthouse*, an Eclipse plug-in that takes the conflict avoidance approach to coordinate developers. Lighthouse distinguishes itself by utilizing a concept called *emerging design*, an up to date design representation of the code, to alert developers of potentially conflicting implementation changes as they occur, indicating where the changes have been made and by whom.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques - *computer-aided software engineering, user interfaces;* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement - *restructuring, reverse engineering, and reengineering*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organizational Interfaces - *collaborative computing, computer-supported cooperative work.*

General Terms

Design, Coordination.

Keywords

Design, reverse-engineering, coordination, awareness, abstraction, emerging design.

1. INTRODUCTION

Software development is a collaborative activity. However, a great deal of development time is spent with developers in isolation, working on their own specific parts of the system. In such setups, it is not easy to be aware of the other developers' tasks at all times, and conflicts regularly happen. Many of the existing tools for software development use Software Configuration Management (SCM) repositories to handle the shared code [1] [10]. Unfortunately, this means that the developers will only be aware of the other's parallel changes and potential conflicts once the changes are checked in. While the SCM tools usually have merging tools to deal with different versions of the same files, merging code and retesting it is a time consuming activity. Furthermore, the longer the conflicting changes go undetected, the greater will be the difficulty of correctly merging in one's changes [5] [7].We call this is a *conflict resolution approach.*

To alleviate these concerns, some tools [4] [6] have started using a *conflict avoidance approach*. In this approach, conflicts can be detected as they are emerging, allowing the developers to proactively coordinate to discuss and address a conflict before it reaches the repository or grows out of hand. Thus, the conflicts can be addressed sooner, avoiding the greater impact of late consideration. In order to enable such an approach, these tools have to deal with some specific requirements and desirable features:

- Instead of having only the repository monitored, each developer workspace has to be monitored for changes to the software artifacts, so that the conflicts are detected as soon as possible;
- The tool has to present relevant information so that the developer does not need to spend a lot of time finding and understanding what he needs to know about other's work. Since there can be a lot of information to show, the representation choice determines the tool's effectiveness;
- The information representation should be integrated with the tools used by the developers. Otherwise they would need to interrupt their development tasks to glean the necessary information, incurring a context switch that is unnecessary and disruptive [2][8];

- The information should be provided to the developers in a proactive manner, so that the developers do not need to take actions to receive the latest information updates [11].

Most existing conflict avoidance tools are file-based, informing the developers about new versions of the file by e-mail, by adding notes to the file, or by annotating the changed lines inside editors. However, we feel that, in order to best help coordinate developers, we need to present finer-grained information. Instead of simply informing the user which files have changed, we want to inform them of how each file has changed in order to provide more context. At the same time, we do not want to overwhelm the developers with implementation details. Thus, we chose to raise the abstraction level and show only information about changes that impact the software design.

In order to do so, we developed the concept of *Emerging Design* [12], an up-to-date abstraction of the code as it exists in the developers' workspaces. Emerging Design is built as the developers implement each part of the code and is automatically updated when the code evolves. In support of coordination, the design is annotated with information about the changes made, so that the developers can be aware of how the design has evolved throughout the implementation phase, who is responsible for each design change, and whether the changes have been checked into the repository and already checked out by other developers.

To implement the Emerging Design concept, we are building an Eclipse [5] plug-in called *Lighthouse*. This plug-in uses Eclipse listeners to keep track of all changes being made to the code in the developers' workspaces and also listens to Subversion events such as check ins and check outs. Lighthouse then uses the collected data to build and update the Emerging Design view automatically.

The rest of this paper is organized as follows. In Section 2, a small motivational example for this work is presented. Section 3 introduces the proposed approach of Emerging Design and its use for coordination purposes. Section 4 shows how the approach was implemented as an Eclipse plug-in called Lighthouse. Section 5 details our Lighthouse integration with Eclipse. Finally, in Section 6, we present our conclusions and future work.

2. MOTIVATION

Imagine a situation in which two developers, John and Susan, part of a larger team, have been assigned to implement related features that involves changing some set of overlapping files. Susan is working to encapsulate a particular class behind an abstract interface, so that other pieces of code should refer to the interface instead of the class. In the mean time, John is adding a new piece of code that uses that class directly, without knowing about Susan's recent changes. So Susan refactors all the code to properly access the new interface, but John checks in his code that directly refers the class to the repository.

From the above scenario, one immediately notices that a critical requirement for supporting coordination is awareness. Developers should be aware of which other developers are contributing to the project and what changes they are making (or have made) to the code. In order to best help developers avoid (or lessen the severity and impact of) conflicts and inconsistencies, awareness information should be available to the developers in real-time; this allows developers to see what changes are being made without requiring that the changes be checked in. In the above example, if John was able to see the changes Susan was making in real-time, he would have been able to adapt his work appropriately and, in doing so, avoid the consequent inconsistency and code decay.

3. APPROACH

In this work, we apply the concept of Emerging Design, an up-to-date representation of the design as it exists in the developers' code. The Emerging Design diagram is built dynamically as the developers implement each part of the code, without the need to save or check in the changes made. The diagram is automatically updated with each code change, enabling the developers to always have an accurate representation of the design as it is currently exists in the developers' workspaces. For instance, if a developer adds a new class, the Emerging Design is updated accordingly, showing the new class representation on the diagram. The view is updated not only in this developer workspace, but in all developers' workspaces. Hence, all the developers have the same exact view of the current design, even if they have not yet checked in or checked out the latest changes.

The Emerging Design is annotated to present additional information about the ongoing changes. Through these, the developers can be aware of how the design has evolved throughout the implementation phase, who is responsible for each design change, and whether the changes have been checked into the repository or, already checked out by other developers.

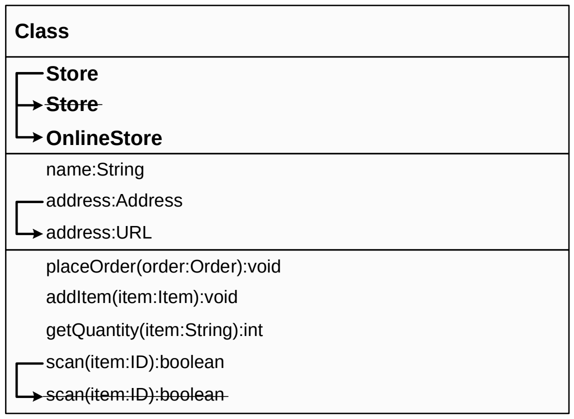


Figure 1. Emerging Design Basic Representation.

3.1 Emerging Design Representation

Currently, the Emerging Design is being represented as a UML-like class diagram, as can be seen in Figure 1. It presents the main elements found in these diagrams (classes, fields, methods, and relationships) with additional evolution information. Arrows are used to link the changes made in the same element. For instance, the class was originally named "Store". However, as indicated by the arrows, there were some changes made by the developers. First, someone deleted the class, as indicated by the next line, "~~Store~~". Also, another arrow points to the third line, indicating that the class was renamed to "OnlineStore". Notice that both arrows come from the same line. This represents a potential conflict, since it indicates that one developer removed the class in their workspace while another renamed it. We can also see that the “address” field had its type changed from “Address” to “URL”. Towards the bottom we can also observe that the method “scan” was deleted.

It is important to notice that the diagram only keeps track of changes that impact the software design. Internal implementation changes in methods are not considered in our current representation, which is a conscious choice relying on the nature and use of interfaces.

3.2 Coordination

With the Emerging Design, developers can follow the design evolution by simply observing the chain of changes for the diagram elements. However, it is also important for the developers to have additional information, such as who is responsible for a particular change, for them to be able to better coordinate their tasks. We therefore chose to annotate the basic Emerging Design with extra columns in the class representation, as shown in Figure 2. The first extra column holds location indicators for each change. A small circle is placed in the respective column, if the change is in the developer’s workspace, if it has been checked in to the repository, and if it is in other workspaces. Additionally, a line connects all circles if the change is present in all workspaces, meaning that all developers have checked out that change. Progressing from left to right, then, means that *my* changes are being adopted by others. Progression from right to left means that *I* am adopting other’s changes. This way, any developer can be aware not only of all design changes, but also know when these changes are checked in to the repository or checked out by the other developers.

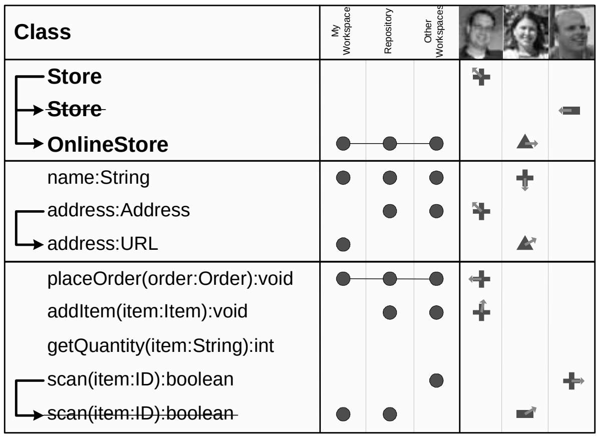


Figure 2. Coordination Annotations.

The method “AddItem” was checked into the repository and some developers have already checked it out. The new “Place-Order” method exists in the repository and has been adopted by all developers, as denoted by the line connecting all three dots.

The last columns indicate which authors made which changes. Three different symbols indicate the types of changes made. A plus symbol indicates an element addition, a minus symbol indicates a removal, and a triangle symbol a modification. Also, a small arrow is placed on the top of each symbol to indicate the recentness of the change. The arrow rotates in a clockwise fashion over time, so for the most recent changes the arrow appears pointing to the top.

3.3 Side-by-Side Presentation

It is important to keep the Emerging Design view always visible, so that the developers can keep constant peripheral awareness of design changes being made by colleagues. However, in order to deal with the amount of information displayed, Lighthouse requires a dual-monitor setup, as depicted in Figure 3. With this setup, the developers can keep a side-by-side view of the code and the Emerging Design view at all times. We do not consider this a problem, but rather a great opportunity. Awareness information has always had to be restricted to fit into an existing environment. We view Lighthouse as a first pilot in which one can design the interface as on desires. The two monitor setup particularly avoids explicit context switching, which is known to be detrimental to effective insertion of awareness in an environment [8].

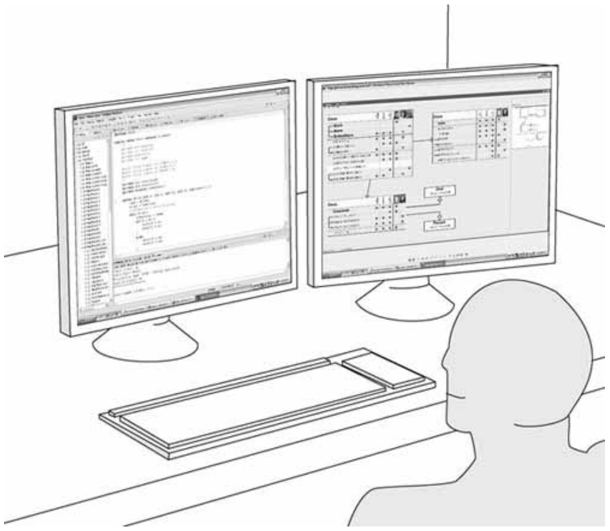


Figure 3. Side-by-Side View of Code and Emerging Design.

4. IMPLEMENTATION

This section details the implementation of *Lighthouse*, the plug-in that brings the approach presented in the previous section to Eclipse. Lighthouse dynamically builds the Emerging Design view from Java code being developed in Eclipse workspaces. It also adds the proposed coordination annotations to the design, using information from both source code changes and Subversion [10] event listeners. First, Lighthouse architecture is specified. Then, the current implementation status is presented.

4.1 Architecture

The architecture of Lighthouse is shown in Figure 4. As can be seen, Lighthouse relies on the Eclipse development environment and a standard configuration management (CM) system (the current Lighthouse implementation utilizes the Subclipse Eclipse plug-in [9]). Components in dark-gray make up the architecture for the Lighthouse client: ECLIPSE WRAPPER, EVENT LOGIC, EVENT REPLICATOR, LOCAL MODEL, DISPLAY LOGIC, and VISUALIZATION.

Through the use of the ECLIPSE WRAPPER, Lighthouse intercepts all relevant events from the CM system and from Eclipse and passes those events onto the EVENT LOGIC component. It is then the responsibility of the EVENT LOGIC to translate the events into Lighthouse events that can be understood by the rest of the Lighthouse client. This way, the Lighthouse system is shielded from the details of the original events and can continue to work even if the underlying CM system and event changes; one simply adjusts the ECLIPSE WRAPPER. Following this translation, the EVENT LOGIC propagates events to the rest of the Lighthouse client as well as to the LOCAL EVENT DATABASE. Upon receiving the events, the EVENT LOGIC updates the LOCAL MODEL appropriately and then broadcasts a data model event notifying other components that the data model has changed. The LOCAL MODEL stores the data model that describes the emerging design. In addition to storing the emerging design, the LOCAL MODEL is also responsible for keeping track of each element's change history. The DISPLAY LOGIC component determines how the elements in the LOCAL MODEL should be displayed. Whenever the LOCAL MODEL is updated, this component is notified (via data model events) and then updates the VISUALIZATION accordingly.

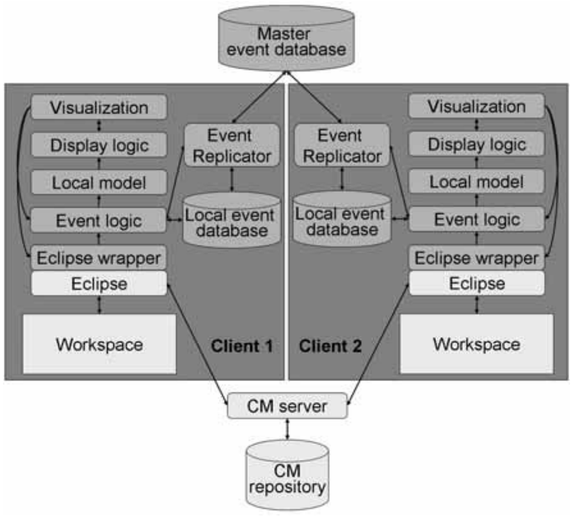


Figure 4. Lighthouse Architecture.

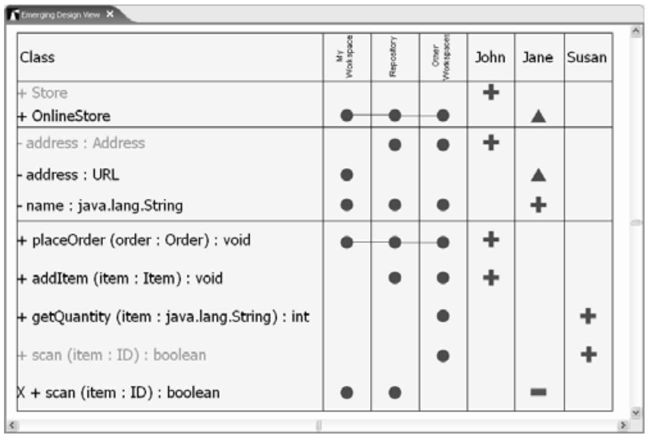
Finally, the EVENT REPLICATOR periodically pushes local events that are stored in the LOCAL EVENT DATABASE to the MASTER EVENT DATABASE, while pulling new remote events left by other Lighthouse clients into the LOCAL EVENT DATABASE. This allows for disconnected operation without any loss of functionality. The MASTER EVENT DATABASE keeps a history of all events, supporting bootstrapping of information when new developers join a project. It should be noted that, while there can be any number of Lighthouse clients, each deployed configuration will only have one MASTER EVENT DATABASE.

4.2 Current Status

We have developed a Lighthouse prototype with the specified architecture. All the features described in the approach section are already supported by the architecture, but some still need to be implemented by the Visualization component. The current state of the plug-in Visualization is depicted on Figure 5. Comparing it with Figure 2, one can notice that the main proposed features are already available, but some details, such as the arrow links and the recentness indicators are not yet implemented.

5. ECLIPSE INTEGRATION

The Lighthouse integration with Eclipse is made through the ECLIPSE WRAPPER component. This component is broken up into two main parts, the Eclipse event listener that monitors source code changes and the Subclipse listener that monitors CM activity.



Class	My Workspace	Repository	Other Workspaces	John	Jane	Susan
+ Store				+		
+ OnlineStore	●	●	●		▲	
- address : Address		●	●	+		
- address : URL	●				▲	
- name : java.lang.String	●	●	●		+	
+ placeOrder (order : Order) : void	●	●	●	+		
+ addItem (item : Item) : void		●	●	+		
+ getQuantity (item : java.lang.String) : int			●			+
+ scan (item : ID) : boolean			●			+
X + scan (item : ID) : boolean	●	●			−	

Figure 5. Lighthouse on Eclipse: Emerging Design View.

5.1 Eclipse Listener

The Eclipse event listener is implemented as an Eclipse JDT's IELEMENTCHANGEDLISTENER and monitors changes to source code elements. The IELEMENTCHANGEDLISTENER is notified of changes to resources (projects, directories, and files) and fine-grained changes to Java elements such as classes or methods. Each change event contains a set of deltas describing the Java element or resource that was changed, annotating the element with either added, removed, or changed. Furthermore, there are a number of flags that provide additional information such as if the scope modifiers or the type hierarchy has changed. The wealth of information provided in a change event helps simplify the logic required to translate Eclipse events into Lighthouse events.

The listener itself is only responsible for parsing each change event to determine the types of changes and the elements that were changed and pass this information on to the EVENT LOGIC component. Before the logic can broadcast the event to other Lighthouse clients, it must first translate the Eclipse event to a corresponding Lighthouse event. The logic combines pairs of related events based on some simple heuristics. For example, the logic will combine the deletion and then the addition of a similar element with a different name into a rename event. As part of the translation process, the logic must also determine whether the change event was the result of a CM action such as checkout or update. This information can be obtained from the Subclipse listener as described in the following section.

5.2 Subclipse Listener

The Subclipse listener is implemented as an IConsoleListener that intercepts all the interaction with the Subversion repository. Currently, Lighthouse only supports the most basic types of CM operations such as check-in, check-out, update, add, and remove. While the IConsoleListener itself is not meant to be an event listener, it originally was meant to be a logger, it provides a great deal of flexibility in the types of information we can gather and allows us to easily customize the CM operations of interest. The listener operates in three basic phases: (1) begin operation, (2) log the file(s) that are involved in the operation, and (3) complete the operation or log any errors that occur. Our listener implementation remembers the state it is currently in and the operation/files, and also passes the information on to the original Subclipse console listener. Once the operation completes successfully, the listener stores the information in a centralized table for the Event Logic to access.

5.3 Lessons Learned

While both the Eclipse and Subclipse listeners are able to capture all the necessary information, the integration with Eclipse was not smooth. One of the most difficult aspects of the integration is filtering out extraneous Eclipse events. For example, it was difficult to determine if our listener always needs to process changes made to "working copies" of Java elements. In certain cases, we needed to ignore events dealing with working copies since they are also used in order to test potential problems with a refactoring. We currently filter these events with heuristics, but a better solution would be to hook into the refactoring system.

Furthermore, sometimes the same action may result in different change events depending on the circumstances. One such example is updating files from the CM repository. Eclipse actually broadcasts different events based on whether or not the file is open in the Eclipse editor. In the end, our listener had to include an increasing number of specific tests to ensure that we capture exactly the right events. Unfortunately, this makes the listener very specific and difficult to maintain or extend in order to accommodate new types of events.

In order to capture CM interaction, we initially attempted to implement the Subclipse listener as an IResourceStateChange-Listener, which is notified when a resource is synced up with the repository or is modified to be different from the repository. Unfortunately, it was impossible to tell how the file was changed once it has already fallen out of sync with the repository. That is, once the file has been modified, it was impossible to differentiate between whether the file has been changed through an update from the repository or through additional user changes. In order to differentiate between the modifications from a CM update and user changes, the Subclipse listener is therefore implemented as an IConsoleListener that interprets the UI console events. Unfortunately, this forces us to work at a low level, in some cases requiring the listener to parse log strings in order to determine the files involved and CM operations. This creates an unnecessary dependency between the listener and the format of the logs as well as the data flow of the Subclipse UI console framework. This though seems to be the best solution to date.

6. CONCLUSION AND FUTURE WORK

In this paper we presented Lighthouse, an Eclipse plug-in that implements the Emerging Design concept, an up-to-date design representing the code as it exists on the developers' workspaces. The design is annotated with additional information to help developers be more aware of other's changes and, thus, to better coordinate their work.

Besides coordination, we still want to explore the use of Emerging Design in two more contexts. The first would be to detect design decay, i.e., to determine where the implementation is diverging from the original design. In order to do that, we plan to overlay the Emerging Design on top of a conceptual design, made prior to the implementation. This will allow us to highlight design deviations, an important source of development problems. The second context we want to explore is the application of Emerging Design to project management. The main idea is to try to answer some common management questions with the help from the Emerging Design representation.

7. ACKNOWLEDGMENTS

This research has been funded by a 2005 IBM Eclipse Technology Exchange grant, a 2006 IBM Technology Fellowship, and by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724.

8. REFERENCES

[1] Allen, L., Fernandez, G., Kane, K., Leblang, D. B., Minard, D., and Posner, J. 1995. ClearCase MultiSite: Supporting Geographically-Distributed Software Development. In *Selected Papers From the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*, LKNCS 1005. Springer-Verlag, 194-214.

[2] McCrickard, D. S., Chewar, C. M., Somervell, J. P., and Ndiwalana, A. 2003. A model for notification systems evaluation—assessing user goals for multitasking activity. *ACM Trans. Comput.-Hum. Interact.* 10(4), 312-338.

[3] Eclipse, Eclipse, http://www.eclipse.org

[4] Hupfer, S., Cheng, L., Ross, S., and Patterson, J. 2004. Introducing collaboration into an application development environment. *In Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, 2004, 21-24.

[5] Perry, D. E., Siy, H. P., and Votta, L. G. 2001. Parallel changes in large-scale software development: an observational case study. *ACM TOSEM.* 10(3), 308-337.

[6] Sarma, A., Noroozi, Z., and van der Hoek, A. 2003. Palantír: raising awareness among configuration management workspaces. *Twenty-fifth international Conference on Software Engineering*, 444-454.

[7] Sarma, A., van der Hoek, A. 2004. A conflict detected earlier is a conflict resolved easier. Fourth Workshop on Open Source Software Engineering".

[8] Speier, C., Valacich, J. S., and Vessey, I. 1997. The effects of task interruption and information presentation on individual decision making. *Eighteenth International Conference on Information Systems*, 21-36.

[9] Subclipse, Subclipse, http://subclipse.tigris.org/

[10] Subversion, Subversion, http://subversion.tigris.org/

[11] Van der Hoek, A., Redmiles, D., Dourish P., Sarma, A., Silva Filho R., and De Souza, C. 2004. Continuous coordination: a new paradigm for collaborative software engineering tools. Workshop on Directions in Software Engineering Environments

[12] Van der Westhuizen, C., Chen, P. H., and van der Hoek, A. 2006. Emerging design: new roles and uses for abstraction. *2006 International Workshop on Role of Abstraction in Software Engineering*, 23-2