

Lighthouse: an Extensible Collaboration Platform

Nilmax Teones Moura, Tiago Proença and André van der Hoek

Department of Informatics

University of California, Irvine

Irvine, CA 92697 USA

{nmoura, tproenca, andre}@ics.uci.edu

Abstract—In the last decade, there has been a tendency to integrate tools directly into the software development environment. This approach is particularly interesting because it reduces the effort spent switching over between tools, which makes the process of collaboration and development more seamless. For developers, this means that they can communicate and coordinate their work through the same tools they use to develop code. However, with the increasing availability of such tools, the problem of overloading the software environment arises—only some decorators, views, and notations can be inserted, seen or active at a time. This type of atomistic approach is clearly undesirable, distracting from work and detracting from the overall effectiveness of a more holistic development environment. To overcome this challenge, we introduce Lighthouse, a novel collaboration platform that unifies previously disparate collaboration functionalities together into a single location. This paper presents the state of the art of Lighthouse and shows four examples of extensions using the Lighthouse infrastructure.

Keywords—coordination, collaboration, design, platform

I. INTRODUCTION

Software development is a collaborative process, where developers need to work together in order to understand the problem, make decisions and coordinate their work. In their daily activities, in addition to face-to-face interaction, developers use collaborative tools that can exist as an external application (e.g. email, instant messaging) or as an embedded plug-in into the software development environment (e.g. version control and issue tracking systems) [1]. These collaboration tools are essential in supporting both communication and coordination of work, especially in situations where the development teams have become increasingly dispersed, such as in open-source and off-shore development [1].

In the last decade, there has been a tendency to integrate these collaborative tools directly into the software development environment. The idea is to reduce the effort spent switching over between tools, which makes the process of collaboration and development more seamless [2]. For developers, this means that they can communicate and coordinate their work through the same tools they use to develop code.

The Eclipse platform is an important starting point for the foundation of an integrated collaborative software development environment [1]. This platform has frameworks

and extensions points that simplify the process of integration between Eclipse and third-party tools, which make it very appealing for developers to contribute their own add-ons. For instance, Mylyn [3], Palantír [4], and Expertise Browser [5] are third-party tools that make the process of collaboration more attractive by providing information about the context of the software being built to the developing team. In addition, Eclipse Marketplace [?] has made thousands of tools for Eclipse available online, which makes Eclipse a good choice as a development environment for developers.

However, with more and more such tools, the problem of overloading the software development environment arises—only some decorators, views, and notations can be inserted, seen or active at a time. Moreover, it spreads out related information functionality through out the software development environment, thus not fostering a condition in which the various tools can work together to synthesize and visually group the different yet interrelated properties of the project. This type of atomistic approach is clearly undesirable, distracting from work and detracting from the overall effectiveness of a more holistic development environment.

This paper introduces Lighthouse, a novel collaboration platform that unifies previously disparate collaboration functionalities together into a single location [6]. This single location emerges as a side by side display where code and coordination activities appear as discrete entities presented simultaneously. This strategy, which presents a variety of collaborative information in conjunction with the task of coding, frees developers from the burden of switching between views; it also improves the ability of developers to multitask, which is a central component of any development environment. Rather than being forced to look in multiple places for different development needs, developers can utilize Lighthouse while coding to gain an uninterrupted understanding of the collaborative activities that are taking place in the project. The Lighthouse collaborative platform exhibits five distinct features:

- 1) *It builds on Emerging Design, so all the collaboration functionality is anchored to a visual representation of the code as it changes over time.* Facilitating awareness of high level system interactions and evolutions in the code's structure, Emerging Design provides the benefits of a design document without the onus of

incessant revision to maintenance relevance.

- 2) *It separates code from the collaboration functionalities through a side-by-side view.* Lighthouse utilizes a two monitor setup with the first monitor displaying the software development environment and the second acting as a collaboration portal. The advantage of a secondary monitor configuration is twofold; in addition to maintaining cognizance of real time project evolutions, the developer's primary task, coding, is augmented rather than interrupted through the simultaneous display of related information.
- 3) *It provides a fine-grain data model where all events associated with source code changes are tracked and made available to developers.* The fine-grain data-model is the heart of the platform as it charts the project's evolution and contains all the events generated in the Eclipse IDE. While Eclipse represents source code changes as abstract syntax trees (AST) possessing a short shelf-life, this approach stores and indexes code changes with the aim of promoting fast fetch of data.
- 4) *It relies upon sophisticated, modifiable filters attuned to a developer's current needs, tasks, and fellow collaborators.* Filters anticipate the possibility of information overload faced with a platform like Lighthouse by sifting for and isolating relevant information identified by a developer. Extremely detailed sorting is achieved through the option of employing multiple filters simultaneously and filters can be implemented or modified in response to changing needs.
- 5) *It utilizes an extensible infrastructure, so new functionality can be easily added on top of the platform.* Developers can take advantage of Lighthouse's data model (fine-grained information) and unified visualization (Emerging Design). Since these two capabilities provide the foundation for new collaborative tools, the benefit of building tools on top of the Lighthouse platform is the prospect of reusing previously created functionalities to save time and develop software products that provide rich user experiences.

The cumulative advantages of these four features is best exemplified through an evaluation of the fourth point enumerated above; how successful is the extensible infrastructure feature and what advantages does it offer to the developer? A central goal of this report is to anecdotally convey the benefits of this infrastructure in the belief that further evaluation will bear similar results. Interesting situations from our own use demonstrate the effectiveness of this platform and a number of design lessons learned by building four examples of coordination mechanisms using this infrastructure.

The remainder of this paper is organized as follows. Section 2 describes the design rationale, particularly intro-

ducing the five key design decisions behind Lighthouse. Section 3 briefly describes how Lighthouse works. Section 4 presents some considerations and design choices we took when developing this platform. Section 5 shows four examples of extensions using the Lighthouse infrastructure. Section 6 addresses related works and concluding remarks are presented in Section 7.

II. DESIGN RATIONALE

In order to reconcile very different collaborative functionalities, it is necessary to take into consideration the vast number of possible design decisions as they are going to shape the kind of collaboration tools that can be supported or not. In this section we introduce the four key design decisions behind Lighthouse platform and explain the benefits associated with each one.

A. Emerging Design

A design document is a valuable media to share with fellow collaborators the design decisions of the project being developed. It assists developers gain an understanding of the high-level structure of the system and its interactions [6]. The problem with traditional design documents, however, is that the design is a static representation that does not evolve (automatically) with the code [7]. As the project evolves, the design development becomes out-of-date and ultimately loses its intrinsic value for developers.

This unfortunate situation counteracts one of main benefits of the design document: it is singularly one of the most important sources of information at a developer's disposal for understanding the project at hand. In order to mitigate this problem, our research group is exploring the use of Emerging Design [6] as the basis for a coordination portal. Emerging Design is defined as the design representation of source code as it changes over time. It is represented as a live UML-like document that stays up-to-date with both gradual and sudden alterations made to the system. As seen in Figure 1, modifications associated with classes, methods and fields are attributed to authors with the history of events represented in a top down manner. Emerging Design is annotated with information about the changes made, helping developers to be cognizant of the code's evolving structure, and with whom they may need to coordinate their actions in order to reduce and prevent conflicts [7].

With Emerging Design, developers can enjoy the benefits of a design document without the burden of creating a document that requires constant maintenance to remain current. If collaboration is typically concerned with work that is happening now or work that took place in the past, it becomes essential to have a way of displaying both these components. Attending to the evolution of the project over time, the notion of Emerging Design works—as Figure 1 shows—to anchor information in the context of collaboration

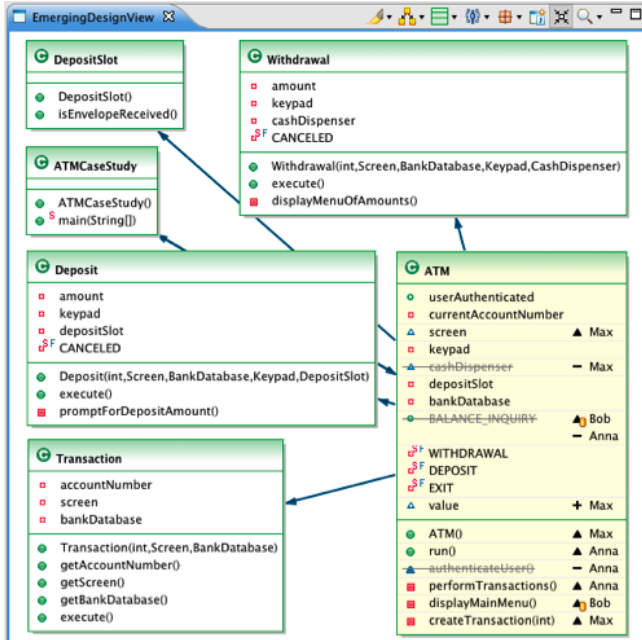


Figure 1. Emerging Design basic representation.

activities and, in doing so, aids developer's in their constant task of identifying relevant problems and possibilities.

B. Side-by-Side Presentation

It is essential to keep the Emerging Design continuously visible to gather all the benefits and capabilities it provides. However, this task can often be problematic, particularly considering the need to display large amounts of data in contemporary projects. The question then centers around the execution of this model. How can something that seeks to accommodate a developer's needs be accomplished without intruding upon or disturbing the developer?

The concept of peripheral visual awareness addresses this issue. Peripheral visual awareness is the ability to be cognizant of, but not distracted by, a significant amount of space/time volume relative to the particular setting and task at hand [8]; it is involved with detecting and understanding the big picture, the context and changes in our environment [8].

Peripheral visual awareness can be achieved through the use of a secondary display allowing for a more seamless integration of Emerging design and coding activities. This is particularly interesting, since a secondary display would keep the Emerging Design conspicuous, hence augmenting the context of what is occurring in the project. We envision the Emerging Design being displayed in a dual-monitor setup, side-by-side with the code, as depicted in Figure 2. With the first monitor assigned to the regular integrated development environment, the second monitor would function to generate peripheral collaboration awareness. This type of configuration avoids explicit context switching, which is

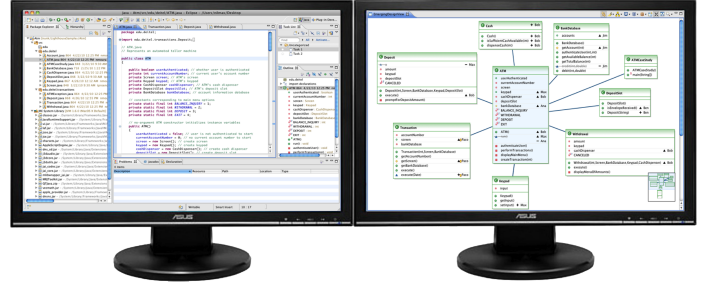


Figure 2. Side-by-side presentation.

known to be detrimental to maintaining effective levels of awareness in an environment [9], [10], and instead leads to an interactive visualization of planning and execution stages.

C. Fine-grain Information

The experience of building a visualization of the Emerging Design raised many insights but also many questions: how could we show the evolution of the project as well as dependencies between artifacts and people, and how could we group artifacts to represent common concerns of interest? Instead of providing an absolute answer to all these questions, we decided to take a step back and leave them open; the idea here was to provide a fine-grain data model which developers can use to access all the information they need to create new applications on top of Emerging Design and, consequently, answer all these questions for themselves during this process.

This fine-grain data model is the heart of the developing framework. It not only contains SCM events (e.g. commits, checkouts, and updates), but also all the real-time events generated in the Eclipse IDE workspace (e.g. the creation of classes, methods, fields, and relationships). As Eclipse, we design this data-model to treat source code changes as first-class entities. However, Eclipse represents source code changes as abstract syntax trees (AST) while we have our own representation of source code change, which is indexed in the model for fast fetching of data.

D. Filters

One of the most difficult challenges a developer might face using a system like Lighthouse is the visualization of huge amounts of data generated by enormous, multi-faceted projects; the developer is invariably at risk of information overload. Side-effects of this state often include distraction, feeling overwhelmed, loss of focus/specificity, and loss of purpose/meaning. There is no easy solution to this condition as it is vital that the developer maintain awareness of macro-level imperatives such as collaborative needs, large-scale evolutions in the project, and final end-goals.

In consideration of these issues, Lighthouse relies heavily upon sophisticated filters to selectively reduce the volume of information displayed. Lighthouse provides different filters

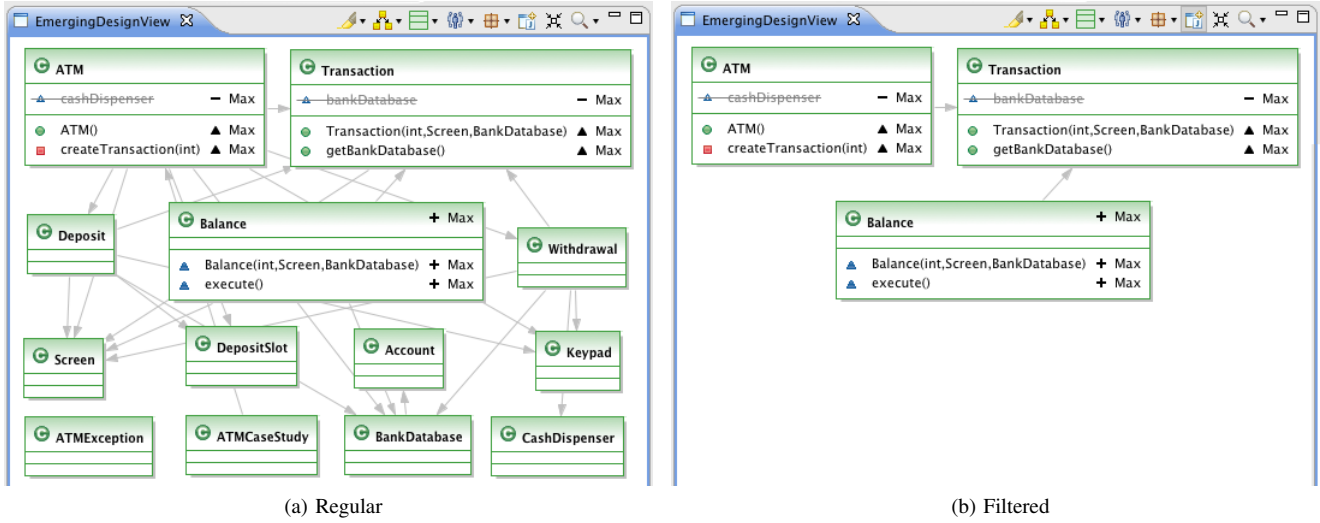


Figure 3. Example of the modification filter being applied to the Emerging Design view.

in a toolbar adjacent to the Emerging Design (Figure 3), which emphasizes ease of access and use. The filters display pertinent information focused on particular or multiple packages, developer(s), modification(s) or some combination of them, identifying information in accordance with a developer’s current needs, tasks, and fellow collaborators.

Figure 3 shows an example of the modification filter built in Lighthouse applied on the Emerging Design view. The purpose of this filter is to hide the classes that do not contain any modifying activities. Figure 3a shows the Emerging Design without any filter, while Figure 3b shows the effectiveness of this particular filter at mitigating the information overload issue. Note that this particular filter can not entirely solve this issue, but when used in conjunction with others a more fine grained level of filtering is readily available. Finally, in addition to built-in filters, Lighthouse provides mechanisms for developers to create new filters as pre-existing needs evolve and new needs develop. Unconstrained by permanent and/or unalterable filter configurations, the possibility of adaptive implementation is a hallmark of the Lighthouse design.

E. Extension-points and wizards

Developers can take advantage of Lighthouse’s data model (fine-grained information) and unified visualization (Emerging Design) for creating their own tools. Since these two capabilities provide the foundation for new collaborative tools, the benefit of building tools on top of the Lighthouse platform is the prospect of reusing previously created functionalities to save time and develop software products that provide rich user experiences.

The extension capability of the Lighthouse collaborative platform is implemented using the mechanism of extension and the extension points of Eclipse. This mechanism can be

more easily explicated by making an analogy to electrical outlets: the outlet is the extension point and the plug that connects to it, the extension.

An extension point is created when a particular plug-in wishes to enable plug-ins to extend or adapt parts of its functionality. The extension point defines a contract that extensions must follow to enable integration between them. The pivotal aspect of this relationship is that the only knowledge the plug-in being extended has of the extension plug-ins is that it conforms to this extension point contract. In other words, the host plug-in is not aware of anything beyond the contract—not how many plug-ins are being connected to it, nor the nature of the functionality being implemented by them. As a result of this contingent relationship, plug-ins designed and constructed by different parties are able to cooperate and connect despite a dearth of specific information on the part of both the plug-in being extended and the plug-in that is extending.

There are multiple inconveniences associated with extension points; taken individually, they can be described as minor annoyances, but when encountered in succession they tend to become exacerbating and time consuming. One must go through countless menus and dialogues to implement an extension point. For example, a developer must include dependencies to Lighthouse components, manually locate which Lighthouse extension point they are planning to extend, and identify which class/interface they have to extend/implement. Compounding these initial steps, developers must start from scratch as they do not benefit from having any basic code to start with.

In order to simplify the complications associated with the creation of extensions for Lighthouse, we created wizards that generate skeleton implementations of a given extension point functionality. Figure 4 shows a wizard that creates a

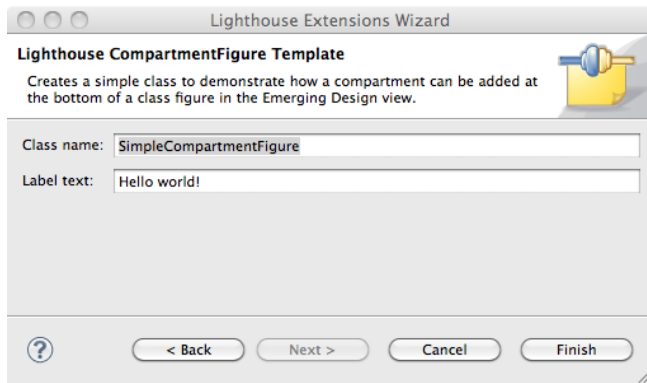


Figure 4. Lighthouse wizard dialog.

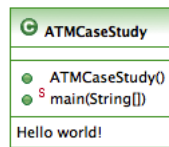


Figure 5. Example of new compartment created by Lighthouse wizard feature.

start-point project for the Lighthouse compartment extension point. In the first text field, the developer inputs the name of the desired compartment figure class and, in the second field, a string of his choice. After concluding this task, the developer can run his extension plug-in through the Eclipse platform, which will create a new compartment in the Emerging Design visualization, as depict in Figure 5. Ultimately, in addition to streamlining the process, upon clicking on finish the developer is greeted by a project that already possesses all the essential pieces needed to implement that extension point functionality.

III. LIGHTHOUSE: A BRIEF OVERVIEW

Consider the following scenario¹ where two developers, Max and Nicole, are charged with implementing new features in a project, which involve changing a set of related files. Before starting her task, Nicole decides to refactor a particular class behind an abstract interface; this separates the concretization of the functionality from its definition. However, this change will require the other pieces of the code to refer to the interface instead of the class. In the meanwhile, Max, unaware of Nicole's recent changes, is implementing the new feature by adding a piece of code directly to the class. Although Nicole refactors all related code to properly access the new interface, Max checks in his code to the repository before Nicole does and, consequently, commits a piece of code that uses a deprecated API.

As an alternative to the scenario described above, Figure 6 depicts the same series of events using the Emerging

Design basic representation. It shows the primary elements found in UML class diagrams, such as classes, fields, methods and relationships, as annotated entities with additional information. The plus symbol represents an addition of class, method or field; minus represents a removal; triangle represents a change. In particular, these annotations are seen as the realization of the evolution of the code. In this depiction, Nicole moved the methods `getAccountNumber` and `getScreen` from the `Deposit` class to the new `Transaction` abstract interface. In a similar fashion, Max modified the `execute` method of the `Deposit` class to include logging functionality. It is important to highlight that the history of changes is presented in the class in a top-down manner, time-ordered with the most recent changes at the bottom.

One quickly realizes that a crucial aspect of building and facilitating coordination amongst developers is awareness: in other words, maintaining cognizance of what other developers are adding to the project and what modifications they are contributing (or have contributed) to the code. With this insight in mind, it becomes imperative that awareness of project activities, which best aids developers in avoiding or lessening the impact of conflicts and cross-purposes, be made available to developers in real-time. As conveyed in the above example, if Max had immediate access to the changes Nicole was creating, he would have been able to adjust his actions in response to hers and, in doing so, avoid ensuing inconsistencies and design erosion of the code.

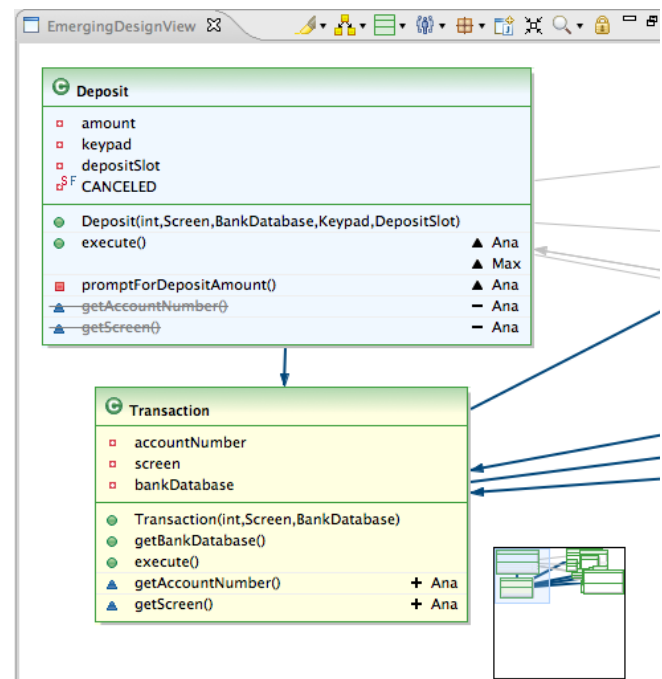


Figure 6. Developers activities illustrated using the Emerging Design representation.

¹Scenario adapted from [9]

IV. IMPLEMENTATION

As stated previously, we built our prototype implementation of Lighthouse on top of the Eclipse framework. Lighthouse uses Eclipse and Subversion extension points to fully integrate Lighthouse with the software development environment. In this section, we describe Lighthouse architecture and highlight some of the design considerations that account for our implementation choices.

A. Design Considerations

Lighthouse is collaborative platform where information about modifications are constantly exchanged among developers' workspaces. Of utmost importance in the design of an application like Lighthouse is the consideration of problems that arise when developing a real-time distributed system. We were interested in answering some significant design questions related to this issue:

- 1) *Event-handling*: How will a client be aware of other clients' events and how is this communication going to be performed?
- 2) *Fault-tolerance*: How can Lighthouse recover from a situation of failure;
- 3) *Time-synchronization*: How will Lighthouse maintain data consistency considering the heterogeneity of clients, time-zones and possible divergences in individual computers clocks?
- 4) *Data integrity*: How to keep the integrity of data without, for instance, losing events or performing mistaken updates?

In order to answer the design questions above, we conceived of Lighthouse with the following design decisions in mind:

- *Client-server architecture*: In order to address issue (1), we designed Lighthouse using a client-server architecture style where every event generated by the client is sent to a central server. In turn, the server stores the events in a central database, thereby making them available to everyone.
- *Persistent outgoing buffer*: In order to address issues (2) and (4), we chose to use a persistent buffer for the outgoing messages sent to the server. If a network failure happens, all events generated by the client are queued locally in this buffer and, when a connection is detected, the buffer starts to send them all to the server. Note that this decision also allows developers to work even when offline.
- *Server-centric time*: In order to address issues (3) and (4), we decided to time-stamp the events only in the server. As a result, there is no need to bother with time synchronization among clients in order to guarantee the real sequence of events.
- *Pulling data*: In order to address issue (4), we chose to use pulling to fetch new events instead of pushing.

Push works well for delivering new events, but if a client misses a message, the client has to implement different mechanisms to recover the missing data (e.g. bootstrap approach). By performing automatic and periodic pulling, in the event of disruptions not only is this aforementioned complexity avoided, but data integrity is secured and mistaken updates are eliminated. Note this automatic and periodic pulling is perspectival, appearing to the user under the guise of pushing.

B. Lighthouse Architecture

The architecture of Lighthouse has seven main components (Figure 7): *Events Collector*, *Events Logic*, *Events Replicator*, *Parser*, *Local Model*, *Display Logic*, and *Visualization*.

Lighthouse is implemented as an Eclipse plug-in and relies on a configuration management system (CM). The *Events Collector* intercepts all relevant events triggered by the CM system and from the Eclipse workspace and passes those events ahead to the *Events Logic* component. The *Events Collector* component is implemented as a set of Adapters, what is desirable since it provides a common and unique interface to the other components and allows Lighthouse to be easily adaptable if the underlying CM system changes.

The *Events Logic* translates events from *Events Collector* into Lighthouse events, and propagates them to both *Events Replicator* and *Local Model* components. Since we model source code changes as first-class entities, depending of the nature of the event, the *Events Logic* can invoke the *Parser* component, which allow us to generate a delta that represents the extent of a change. Note that all user actions are represent in the system as Lighthouse events.

Once, the *Local Model* receives the Lighthouse events from *Events Logic*, it merges these events with the current

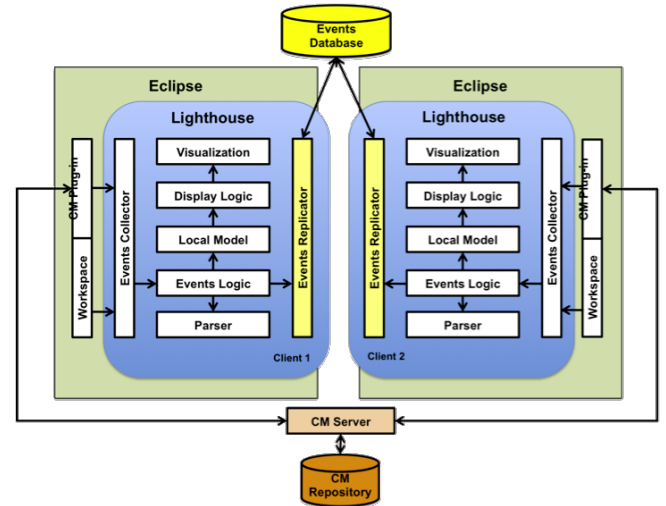


Figure 7. Lighthouse architecture.

state of the data model. The *Local Model* represents the Emerging Design and keeps track of all history of changes. Every time the *Local Model* is changed, a message is sent to the *Display Logic* that proper takes the actions necessary to refresh the *Visualization*.

The *Display Logic* determines how the Emerging Design should be displayed. It is compounded by a set of filters that can be customized and combined. Whenever the data model is updated, this component is notified and then updates the visualization accordingly.

All user actions are translated into Lighthouse events and placed in a queue in the *Events Replicator* component. The *Events Replicator* periodically pushes events from this queue to the *Events Database*, while pulling new remote events left by other clients to the data model. This queue acts like a buffer, which keeps Lighthouse working even when network connection is not available. The *Events Database* keeps the history and the detailed evolution of multi-developer projects. Note that, while it can exist any number of Lighthouse clients, each deployed configuration will only have one *Events Database*.

V. EXAMPLE OF LIGHTHOUSE EXTENSIONS

In this section we describe interesting situations from our own use of Lighthouse. The goal is to show the effectiveness of this platform for developing new collaboration tools. We begin our discussion our presenting our simplest extension: the tagging plug-in.

A. Tagging

A tag generally refers to a keyword assigned to a piece of information providing both description and support for search. Tagging is the collaborative activity of marking shared content with tags as a way to organize content for future navigating, filtering, or searching. In an individual perspective, developers use tagging to make annotations so that information is easier to recall at a later date. From a collaborative viewpoint, tagging also provides a means to visualize and be made aware of the tags used by other developers. Using the Lighthouse platform, developers could use taggings like bugs (failures and faults), tasks (todo and fixme), and concerns (security, log, database etc) overlaid on the Emerging Design. As a result, an association between keywords and design artifacts would emerge that could then be shared with the rest of the team.

We implemented the Tagging extension using Lighthouse's compartment extension point. This extension point enables developers to create an extra compartment at the bottom of a class figure. The idea here is to provide a mechanism to associate tags with classes, which makes this particular extension point very useful. In order to do so, we had to extend the Lighthouse data model to accommodate the tag information. Figure 8 shows the resulting Tagging extension overlaid on the Emerging Design view. The tag

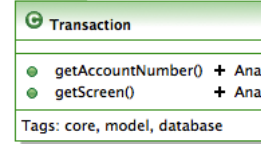


Figure 8. Tagging Plug-in.

is done by the developer clicking in a particular Tag compartment. An input box will pop up to add/remove/modify the tags with these actions being reflected in all developers' workspaces.




Although for now a simplistic approach, this example demonstrates the possibility of creating a customized extension where developers can enter and share tags. While this idea currently has limitations because too many tags can result in widespread tag devaluation, in the future we are planning to expand this extension to provide an interface where developers to identify and agree on tags that are meaningful for the entire team.




B. 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 [11]. 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 are clear collaboration and expertise needs when one must understand how some class/method works, why it is as it is, and how it may need to evolve.

Most current approaches to determining developers' expertise rely on the heuristic that a person committing changes to a file possesses expertise in that file [12]. Although this is often the case, we can identify two major limitations. First, it ignores all the real-time events that occurred between committing changes. Depending on the size of the task at hand, developers can spend weeks without committing code. This approach does not account for individuals who, possessing the freshest information but left out of the collaborative equation, could actually be of great assistance. Second, they operate in pull mode, producing recommendations only after a developer's explicit request. Aside from simply lacking a readily available cataloguing of experts from which to choose, the demands of this manual request and retrieval may lead a developer to rely upon their memory or network of colleagues as opposed to someone barely known but with a high level of expertise.

We implemented the Expertise finder extension using Lighthouse's compartment extension point. Figure 9 shows this extension overlaid on the Emerging Design view. Note how experts are always shown inside each class in the diagram to clearly indicate relevant expertise. The developer no longer needs to perform a manual pull operation to gather this information. In other words, our extension operates

ATM		
○ keypad	▲ Max	
	▲ Alex	
■ cashDispenser	▲ Bob	
	▲ Alex	
◇ bankDatabase	▲ Ana	
	▲ Alex	
ATM()		
	▲ Bob	
	▲ Alex	
▲ run()	▲ Max	
	— Ana	
	— Alex	
		

Transaction		
○ screen	▲ Paco	
◇ getScreen()	▲ Paco	
▲ execute(Date)	+ Paco	
		




BankDatabase		
○ accounts	▲ Jim	
◇ getAccount(int)	▲ Jim	
▲ credit(int,double)	— Jim	
		

Figure 9. Expertise Plug-in.

in push mode, reflecting real-time changes and delivering results continuously.

We took advantage of the Lighthouse fine-grained data model to provide expertise recommendations via three different dimensions: (1) the one who owns the most lines of code; (2) the one who is responsible for the most real-time events; (3) the one who interacts most frequently with an artifact. The point here is not to suggest this is the best method for locating experts. Rather, by exploring how such a system can be easily implemented on top of the Lighthouse platform, we wish to convey how an extension like the Expertise finder can address limits and identify new possibilities for something like expertise collaboration. One example of many possibilities, in the future this initial Expertise finder extension can be expanded to annotate the Emerging Design with more detailed expertise information (e.g. at the method level).

C. Soft-lock

A developer may not be working on a particular component of a project yet may have an interest in how other developers continue to interact with it. It could be because the developer previously designed the component, alterations could affect the developer's current task or it is a core component that determines aspects of the larger project's success. We developed the Soft-lock extension for Lighthouse to provide developers with a way to indicate particular classes they are interested in and be notified when changes occur.

Where the previous two examples were implemented using Lighthouse's compartment extension point, the Soft-lock extension took advantage of two different extension points: the Click extension point and the Filter extension point. The first forwards click-events in the Emerging Design view to

other plug-ins/implementations and the second provides a framework for creation of new filters.

This particular extension involves three pieces: 1) adding classes of interest to a soft-lock list 2) a table viewer that displays ongoing modifications happening on the soft-locked classes; and 3) a new filter that will only display the classes the developer soft-locked.

To add classes to a soft-lock list, the developer simply right clicks on the class of interest and selects the "add to soft-lock list" popup menu item (Figure 10a). This information is kept local and, from this point, any event that is received is checked against this list. If a match is found, an entry is created in the table viewer that alerts the developer about which class is being touched and by who (Figure 10b). A double click in this entry highlights the class in the diagram, making clear to the developer the extent of the changes that have happened or are happening. From the point of design, we found the table viewer to be less intrusive than other options. If a developer does not want to be distracted by continually occurring changes, the table can be hidden and easily reaccessed at a later time if desired. The soft-lock filter (Figure 10b) is useful to keep track and manage which classes the developer soft-locked.

While a developer's interest in a particular component may arise for an array of reasons, developer accountability for errors and mistakes is one example of how the Soft-lock extension point becomes very useful. With a record of modifications associated with the person responsible for these changes, the Soft-click extension point is one example of using the Lighthouse platform to monitor various components of a larger software development environment in order to prevent conflicts, address individual mistakes and maintain cognizance of the overall project.

D. Impact Analysis

Often in software development developers are working in parallel in order to maximize speed in the production. However, this parallel nature of software development can lead to conflicts, which are typically detected after they have been introduced in the source code. Since, Lighthouse captures real-time events from developers' workspaces, Lighthouse creates an opportunity for creation of new tools aimed to detect conflicts in an early stage.

CASI[13] is a Eclipse plug-in that uses the Lighthouse platform infrastructure to be able to detect indirect conflicts taking place in a project. The rationale behind CASI is that providing tool support to developers might allow them to detect and prevent potential indirect conflicts.

To be continued...

VI. RELATED WORK

There exist a number of software tools that are aimed to improve the way people collaborate. The War Room Command Console [11] shows in a public display the

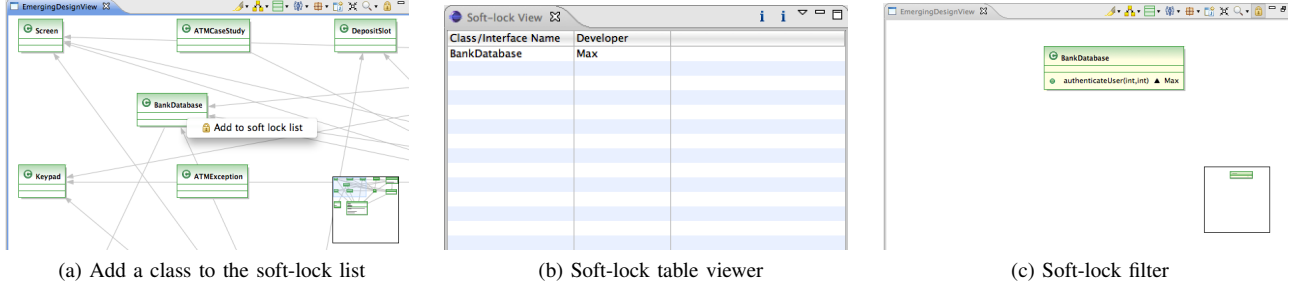


Figure 10. Soft-lock plugin.

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 [4] 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 [14] and CollabVS [15] 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 an historical view of changes made.

ProjectWatcher [16] is an Eclipse plug-in that focuses on mining local interaction histories among developers and artifacts. The local changes are automatically committed to a shallow CVS repository and then analyzed in order to extract useful information relevant to the development team. The information is displayed through two different visualizations: (1) an activity awareness view that shows past and current activities performed on the project artifacts, and (2) a proximity awareness view that provides a sense of distance among developers in terms of system structure and dependencies of artifacts. ProjectWatcher's idea is similar to Lighthouse since it believes that the local interactions between developers and artifacts is a valuable source of

information and hence should be considered in workspace awareness tools.

Syde [17] is an Eclipse plug-in that provides team-awareness by sharing information about changes and conflicts across developer workspaces. Syde provides many visualizations that help developers be aware about different properties of the project at hand. As Lighthouse, Syde continuously tracks information about changes from developers' workspaces, storing them in a fine-grained representation. Unlike Lighthouse, Syde overloads the software development environment with several views, decorations and annotations. This can be viewed as undesirable, distracting from work and detracting from the overall effectiveness of a more holistic development environment. Lighthouse provides a unique visualization that synthesizes and visually groups the different yet interrelated properties of the project.

The Jazz platform [1] is built on top of Eclipse, and is closer related with our work, since it also provides an extensible platform upon which other collaboration tools can be built. However the main focus of Jazz is to integrate development activities, artifacts, and teams throughout the software lifecycle. Lighthouse on the other hand focus in provide a shared understanding of all the real-time events that happens on a software project, in which can be used as a basis for new software collaboration tools.

VII. CONCLUSION

In this paper, we presented Lighthouse, a novel collaboration platform that unifies previously disparate collaboration functionalities together into a single location. Building upon the Eclipse software development environment while identifying the limits of its atomistic approach to collaboration, Lighthouse utilizes a dual monitor display where code and information functionality appear side by side. Lighthouse thus deconstructs the false but seemingly enduring division between collaboration and coding; intimately connected, these two activities must function in tandem to achieve optimal, time-effective results.

Through the use of Emerging Design, Lighthouse's collaboration functionality is wedded to a visualization of the code as it evolves. This design feature allows for the display

of current information and provides all the benefits of a design document with none of the drawbacks of obsolescence. While liberating developers from the need to continually switch between views, Lighthouse fosters an uninterrupted awareness of collaboration needs melded with the principal task of coding. The use of the fine grain model means events associated with modifications to the source code are tracked and supplied to developers to facilitate fast awareness of the changing development environment. Filters anticipate the possibility of developer exhaustion brought on by information overload by sorting and isolating relevant information identified by a developer as being of use. The extensible infrastructure makes Lighthouse a dynamic collaboration platform that can be adapted and customized to the various needs of developers and satisfy the ever-increasing demand for rich user experiences.

From tags to expertise to soft-locks and CASI, examples of the benefits of the extensible infrastructure for the creation of coordination mechanisms highlight the intrinsic, multidimensional functionality of the Lighthouse platform. Lighthouse can be put to use to address countless coordination needs, facilitating a development environment where developers who imagine new extensions evince Lighthouse's broader ability to bring individual contributions and team collaboration a few steps closer together.

ACKNOWLEDGMENT

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

REFERENCES

- [1] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson, "Jazzing up eclipse with collaborative tools," in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, ser. eclipse '03. New York, NY, USA: ACM, 2003, pp. 45–49. [Online]. Available: <http://doi.acm.org/10.1145/965660.965670>
- [2] *Collaborative Development Enviroments*, vol. 59. Academic Press, August 2003.
- [3] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 1–11. [Online]. Available: <http://doi.acm.org/10.1145/1181775.1181777>
- [4] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: raising awareness among configuration management workspaces," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 444–454. [Online]. Available: <http://portal.acm.org/citation.cfm?id=776816.776870>
- [5] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 503–512. [Online]. Available: <http://doi.acm.org/10.1145/581339.581401>
- [6] C. Van der Westhuizen, P. H. Chen, and A. van der Hoek, "Emerging design: new roles and uses for abstraction," in *Proceedings of the 2006 international workshop on Role of abstraction in software engineering*, ser. ROA '06. New York, NY, USA: ACM, 2006, pp. 23–28. [Online]. Available: <http://doi.acm.org/10.1145/1137620.1137626>
- [7] T. Proenca, N. T. Moura, and A. Van Der Hoek, "On the use of emerging design as a basis for knowledge collaboration," in *Proceedings of the 2009 international conference on New frontiers in artificial intelligence*, ser. JSAI-isAI'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 124–134. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1881958.1881972>
- [8] *Peripheral Visual Awareness: The Central Issue*, vol. 7, 1996.
- [9] I. A. da Silva, P. H. Chen, C. Van der Westhuizen, R. M. Ripley, and A. van der Hoek, "Lighthouse: coordination through emerging design," in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, ser. eclipse '06. New York, NY, USA: ACM, 2006, pp. 11–15. [Online]. Available: <http://doi.acm.org/10.1145/1188835.1188838>
- [10] C. Speier, J. S. Valacich, and I. Vessey, "The effects of task interruption and information presentation on individual decision making," in *Proceedings of the eighteenth international conference on Information systems*, ser. ICIS '97. Atlanta, GA, USA: Association for Information Systems, 1997, pp. 21–36. [Online]. Available: <http://portal.acm.org/citation.cfm?id=353071.353080>
- [11] C. O'Reilly, D. Bustard, and P. Morrow, "The war room command console: shared visualizations for inclusive team coordination," in *Proceedings of the 2005 ACM symposium on Software visualization*, ser. SoftVis '05. New York, NY, USA: ACM, 2005, pp. 57–65. [Online]. Available: <http://doi.acm.org/10.1145/1056018.1056026>
- [12] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 385–394. [Online]. Available: <http://doi.acm.org/10.1145/1806799.1806856>
- [13] F. Servant, J. A. Jones, and A. van der Hoek, "Casi: preventing indirect conflicts through a live visualization," in *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '10. New York, NY, USA: ACM, 2010, pp. 39–46. [Online]. Available: <http://doi.acm.org/10.1145/1833310.1833317>
- [14] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "Fastdash: a visual dashboard for fostering awareness in software teams," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ser. CHI '07.

New York, NY, USA: ACM, 2007, pp. 1313–1322. [Online]. Available: <http://doi.acm.org/10.1145/1240624.1240823>

- [15] R. Hegde and P. Dewan, “Connecting programming environments to support ad-hoc collaboration,” in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 178–187. [Online]. Available: <http://dx.doi.org/10.1109/ASE.2008.28>
- [16] K. Schneider, C. Gutwin, R. Penner, and D. Paquette, “Mining a Software Developer’s Local Interaction History.” Scotland: 1st International Workshop on Mining Software Repositories, 2004.
- [17] L. Hattori and M. Lanza, “Syde: a tool for collaborative software development,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 235–238. [Online]. Available: <http://doi.acm.org/10.1145/1810295.1810339>