Lighthouse: A Practical Use for Emerging Design

Jeff Fulkerson
Prof. André van der Hoek
University of California, Irvine
December 2006
For as long as programmers have been collaborating, there has been the question of how best to coordinate the efforts of multiple developers working on the same project simultaneously. The sharing of files is something that must occur in this process, but it is also something that has been the source of numerous conflicts. Code bases have been managed in many ways, including via some sort of shared repository such as CVS (Concurrent Versions System) or SVN (Subversion). In other cases, programs such as Perforce, a client/server configuration management system, are used. While these alternatives offer a solution, they do not offer an ideal solution. They facilitate the sharing of files, yes, but they do not prevent figurative collisions between team members. It is not uncommon for one programmer to be tailoring her code to fit the interface of the graphics framework, while a fellow programmer is busy restructuring that same framework. When the time comes to submit their work, there will be disappointment because much of that hard work was in vain, as the code does not fit together. What is needed is a mechanism to reduce or eliminate situations such as this; there should be a way to keep track of team members’ activities in real time.

One way to accomplish this would be to have a coordination platform that dynamically updates a visual representation of the current state of the code base. This would give us an abstraction that represents the emerging design. Accordingly, this visual representation would facilitate greater coordination between team members, as well as prevent undesired changes by providing managers with an accurate depiction of the current state of the project. Lighthouse is a project that is designed to meet these
goals. Lighthouse is written in Java™ as a plug-in for Eclipse, a highly customizable open-source development environment. By working with Subclipse, a form of Subversion developed specifically for Eclipse, Lighthouse is able to keep track of changes to the code base. A diagram is created using information about the code, and it is updated when changes occur. In order to create a user-friendly diagram, the Lighthouse display implementation is based on GEF (Graph Editing Framework), which allows users to move, resize, and reshape portions of the diagram. Customization allows for greater understanding and ease of use, as each team member may tailor the display to fit personal preferences.

The Lighthouse architecture is composed of three main conceptual models: the Client module, the Event module, and the CM (Content Management) module. First of all, each Client module is composed of a Lighthouse client and an Eclipse client. Together these clients keep track of changes to the code and take steps to update the display accordingly. There may be any number of Client modules in the Lighthouse architecture. Secondly, the Event module stores events, and is responsible for sending events between clients. Unlike the Client module(s) however, there may only be one Event module. Lastly, there is the CM module, which is responsible for keeping track of the conceptual design as well as for storing the source code. A simple representation of this architecture may be seen below.
The modules will now be broken down into further detail. For a visual representation of the breakdown, please refer to the image at the end of this paragraph. To start, let's look at the Event Service. The Event Service keeps track of all events, and is responsible for sending out events to all Lighthouse clients. The events are archived in the Event Database, which simply maintains a history of all events for potential future use. The Eclipse Wrapper listens for and captures certain Eclipse events, such as addition/removal of variables, methods, etc. When these events occur the Wrapper sends them on to the Event Logic. The Event Logic then translates those Eclipse events into Lighthouse events, and sends them on to the Local Model and Event Service components, as appropriate. The Event Logic component may also receive events from the Event Service or Interactive View components. The Local Model element then takes the events and incorporates them into the local data model. For each workspace that Lighthouse is aware of, the Local Model would store an emerging design model for each client. Although, even though the model is kept local to each client, the Local Model component of each client should be the same. The Display Logic component decides how elements of the Local Model should be drawn. Additionally, whenever the Local Model changes, the Display Logic updates the current view accordingly. The Interactive View component is what actually provides the visual representation of the Local Model. Any interaction with the View will generate events that will either be sent to the Event Logic or Eclipse Wrapper components. Essentially, that is how Lighthouse is structured. However, this paragraph is merely a brief snapshot of what occurs beneath the surface.
In order to better understand the structure outlined in the previous paragraph, we will walk through a typical scenario. We will assume that a user has just deleted a source item. The first thing that occurs is that the Eclipse event listener will capture the event, which the Eclipse Wrapper will pass on to the Event Logic. The Event Logic will translate the event into a Lighthouse "Remove Element" event, and send it on to the Event Service component, which in turn will send it to other developers (as well as store the event in the Event Database). Each Lighthouse client will then update the Local Model of its own workspace based on the event. The Display Logic will then determine what changes need to be made to update the display, and the Interactive View will then display those changes. This is just one possible scenario. Similar sequences will transpire for each of the event types, and the reader may infer some of the differences
based on this example. By constantly updating the display in this manner, Lighthouse allows developers to constantly be up to date about the current state of the code base.

The concept of emerging design, which Lighthouse has embraced, endows developers with numerous benefits. As such, Lighthouse enables developers to easily grasp a high-level view of the system at any given time. Rather than perusing profuse pages of a design document, a simple diagram may save time and accomplish the same goal. The emerging design also allows developers to visualize dependencies within the code. This is something that is not always easy to do by looking at the source code, and it may ultimately help produce cleaner code. Additionally, the emerging design indicates who has authored and/or modified code modules. This goes along with the visualization of dependencies, because if a dependency is noticed, it is helpful to know who to talk to when working with a specific section of code. Keeping track of changes is another benefit, since it helps developers track down bugs as they arise. Generally speaking, emerging design gives developers something to compare with the original conceptual design, enabling them to evaluate the overall progress of the project. The benefits enumerated here are not merely theoretical; in fact Lighthouse makes them quite practical.

Assume that a group of programmers are creating a computer game. Each programmer is assigned to work on a different portion of the game, such as User Interface, Graphics, Performance Optimization, Gameplay, et cetera. These are all separate arenas, but they all must interact with each other in order to create a fun and
playable game. For example, the optimization programmer may decide that certain
player commands are cumbersome and redundant, and can safely be removed from the
code base. At the same time, the gameplay programmer may be designing a combat
system that uses some of those same commands. The optimizations are submitted, and
then later when the combat system is submitted, it no longer works with the current
code because certain commands have been phased out. To avoid this, it is important
for each programmer to be able to see what the others are working on. If the gameplay
programmer had known that the optimization programmer was working on the
commands, the two programmers could discuss the changes to ensure that
development continues smoothly. Lighthouse offers an ideal way to solve dilemmas
such as this one.

Diagrams have historically been used in education and business to promote
understanding of a concept or a current state of affairs. By applying this model to
software development, it is easy to extrapolate the benefits that will incur. In a given
scenario, a diagram provides a simplified version of a concept or idea, which makes it
easier to visualize. This promotes clarity and unification. For instance, the Lighthouse
diagram (see sample below) provides programmers with a general idea of how the
project is progressing, in addition to providing details about who is working on certain
areas of the code base.
Each class will have its own such diagram, including the methods and fields that it contains. The arrows on the left connect items that have been renamed. The circles in the center column indicate where each portion of the code is being worked on (i.e., in which workspace). The icons on the right show what type of change has been made, and when. A plus denotes an addition to the code, a minus denotes a removal, and a triangle represents a modification. The small arrow represents how recently the change was made, with North being the most recent. The arrow turns up to 360° as time progresses, until a certain period of time has elapsed. As can be seen above, the icons on the right are aligned with the programmer who made the change. Ultimately, each
user will be able to select which projects and classes to view in their own workspace. This provides a customizable way to promote cohesion among team members.

In an ideal scenario, Lighthouse is envisioned as being used in a dual-monitor workspace environment. The image below depicts what a developer will see when using Lighthouse while working on a project. As seen in the image, one monitor is devoted to Lighthouse, while the other is reserved for the developer’s coding environment. It is best that Lighthouse remain visible, because changes may occur at any time, and the developer needs to be informed of such changes. When any change occurs, the views of all developers are updated to reflect the change. If one developer adds a new class, all displays are updated. If another developer removes a method invocation, again all displays are refreshed. Any modifications to the code base that alter the design structure will be represented in the Lighthouse display. It is important to note that the display is updated in real time, rather than waiting until individual developers check in their changes to the repository. With the emerging design constantly available on a second monitor, potential conflicts or problems may be identified as they surface. Rather than losing time finding issues and solving problems later on, developers are able to address issues immediately and prevent them from becoming major problems.
My main contribution to the project was to create a listener class, initialized by the UI (User Interface) plug-in, that serves as an interface between the data model and the UI model. The first thing it does upon initialization is to add all children of the Lighthouse data model into the UI model so that they can be added to the display. Once the initialization is complete, the listener waits for an event to occur. The listener then looks at the type of event, and determines what to do with the information that was generated by the event. Events occur when code is added, removed, modified, or committed to the repository. To help facilitate the passing of information, my listener
takes the information generated by the event and puts it into a class structure that can be easily understood by the UI logic. This was a design decision that was agreed upon at an earlier stage, when another undergraduate researcher, Charles Tubbs, was working on the project with us. Regrettably, he quit the project before we had a working prototype. Upon his departure, I had to look at his code to determine how to get it to a fully functional point of operation. Charles had attempted to display the Lighthouse diagrams in an editor, rather than a simple view. These were options made available by GEF. A view would be a simple static display, whereas an editor would allow user interactions such as moving and resizing of diagram objects. I was able to make some small changes that brought the UI logic closer to being functional, but alas, the school year ended before I was able to fully understand Charles’ code and make it work as desired. Upon the expiration of my research term, our data model was working fairly well, and our UI was not far from completion.

The Lighthouse team has recently been able to complete a working prototype, and I have been informed that they have not yet had any problems with the code that I contributed. With a working prototype, the Lighthouse project may move forward with promoting the idea of emerging design, and potentially promoting itself as a standard for its implementation. In a more general sense, the concepts and ideas that have taken form during research and development should prove beneficial to present and future generations of software developers and researchers alike.