**Lighthouse Design Document**

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. This document describes Lighthouse architecture and highlights some of the design considerations that account for our implementation choices.

1. **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.

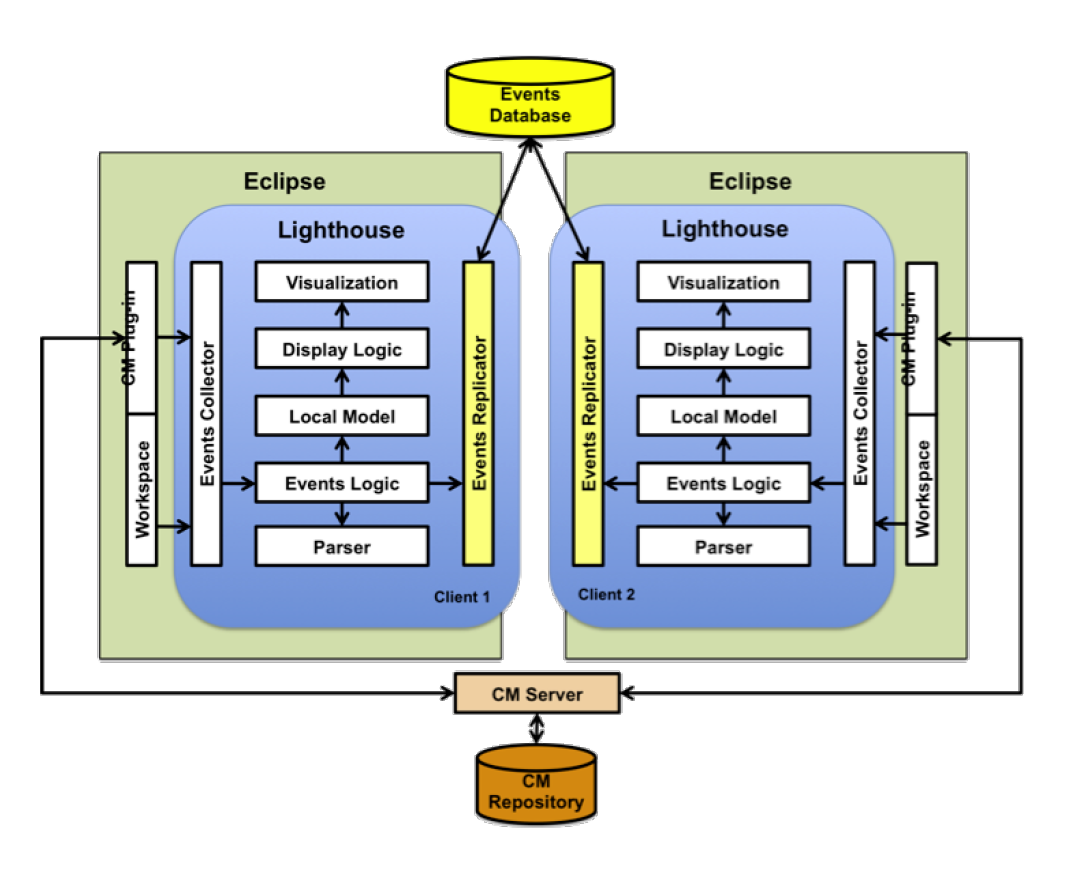
**Persistent outgoing buffer:** In order to address issues (2) and (4), we chose to use a persistent buffer for the outgoing messages sent to the server. If a network failure happens, all events generated by the client are queued locally in this buffer and, when a connection is detected, the buffer starts 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.

1. **Lighthouse Architecture**



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

* 1. ***Architecture Overview***

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.

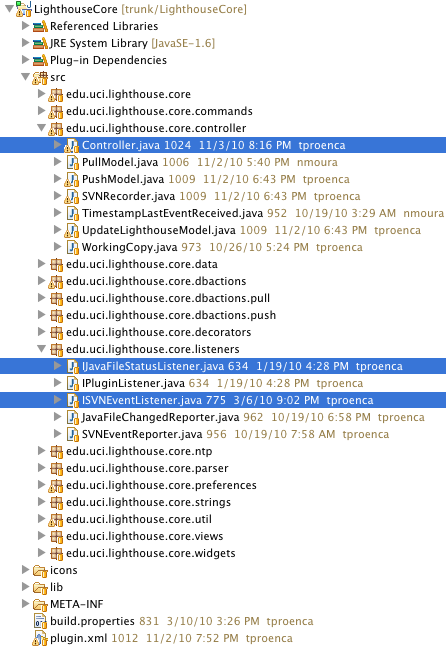
The **Events Logic** translates events from **Events Collector** into Lighthouse actions and events, and propagates them to both **Events Replicator** and **Local Model** components respectively. Since we model source code changes as first-class entities, depending of the nature of the event, the **Events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

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

* 1. **Events Collector**



The **Events Collector** intercepts all relevant ***events*** ***triggered*** by:

* CM system

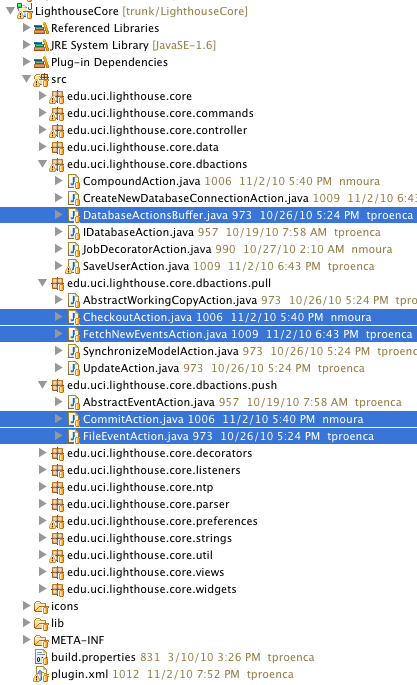
(*ISVNEventListener.java*)

* + Checkout
  + Update
  + Commit
  + Conflict
* Eclipse workspace

(*IJavaFileStatusListener.java*)

* + Open File
  + Close File
  + Add File
  + Change File
  + Remove File

In order to catch all the triggered events the class *Controller.java* implements *ISVNEventListener and IJavaFileStatusListener* interfaces.

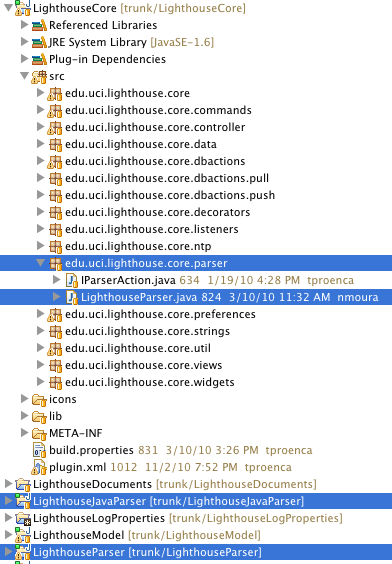


* 1. **Events Logic**

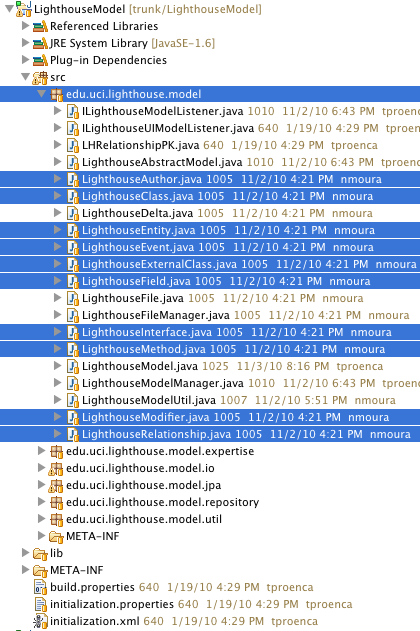
The **Events Logic** translates events from **Events Collector** into Lighthouse actions. The actions are propagated through **Events Replicator**, such as *CheckoutAction.java, FetchNewEventsAction.java, CommitAction.java, FileEventAction.java etc.*

* 1. **Events Replicator**

The **Events Replicator** periodically pushes events from this action queue (*DatabaseActionsBuffer*.java) to the Events Database

* 1. **Parser**

The **Events Logic** also translates events from **Events Collector** into Lighthouse Events that is propagated through **Local Model** see method *Controller.generateDeltaAndSaveIntoModel(…)*. In order to generate Lighthouse Events we need to invoke the parser *Controller.parseIFile(…)* that allow us to generate a delta that represents the extent of a change. The java parser that we used is called by **Sourcerer** (more info at http://sourcerer.ics.uci.edu/)

* 1. **Local Model**

The **Local Model** represents the Emerging Design and keeps track of all history of changes

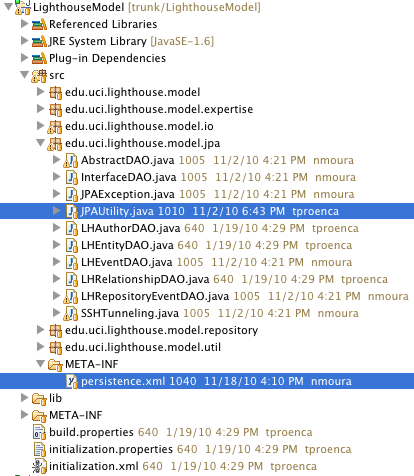
* Model
  + **Entities***: class, Interface, field, method…*
  + **Relationship**: *extend, implements, inside, call…*
  + **Events***: type, timestamp, author…*

***LighthouseModel.java:*** Basically this class has a list of entities, list of relationship, and a list of events. It also have a bunch of methods that perform actions on those lists such as *addRelationship()* and *addEvent().* This class also fires any modification that happen to the visualization component through the listener *ILighthouseModelListener*.java.

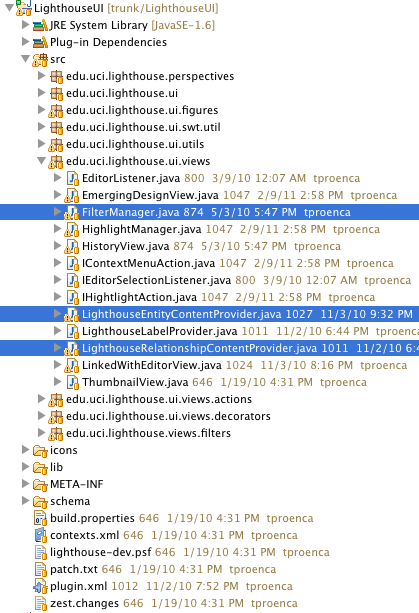
***LighthouseModelManager.java:*** , it is responsible for building and update lighthouse model objects.

***LighthouseFile.java*:** represents a "Small Lighthouse Model" i.e. it represents all the entities and relationship of ONE Java class**.**

***LighthouseFileManager .java*:** It is responsible to manipulate the *LighthouseFile.java*

* + 1. **Model Persistence**

We decided to use JPA framework in order to facilitate the lighthouse model persistence. The objects that are stored in the database are authors, entities, relationships and events (see *persistence.xml* file). Here the most important class is called *JPAUtility.java*that is responsible to manage all the database connections and transactions.

* 1. **Display Logic**.

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**. In the end of the day the **Display Logic** will determine how the Emerging Design should be displayed.

* 1. **Visualization**

We decided to use Zest Framework (more info at <http://www.eclipse.org/gef/zest/>), the projects that implement the UI are *LighthouseUI*, *org.eclipse.zest.core* and *org.eclipse.zest.layouts*.

1. **Open Bugs**

All the documentation about the remaining bugs can be found at https://calico.ics.uci.edu/bugs/