ProveBit

A trusted timestamping application by:

Stephen Halm

Daniel Boehm

Noah Malmed

Matthew Knox

Qiuhua Ding

Shishir Kanodia

Table of Contents

1. Table of Contents
2. Project Description
3. Process Description
    a. General Process
    b. Issues Addressed
4. Requirements and Specifications
    a. Use Case Diagram
    b. User Stories and Task Breakdown
5. Architecture and Design
    a. Description of Architecture
    b. Key Systems
    c. UI Architecture
    d. Application/Daemon Communication
    e. Framework Implications
6. Future Plans
    a. Personal Reflections
    b. Future Plans
7. Bonus: Real World Use Case Example

Project Description

Our files have creation times and modification times, but they can be corrupted or even tampered with by a utility program. Thus, we should not place high trust in them as proof that a student completed their homework on time, or as evidence in court. ProveBit is an application that certifies file timestamps without relying on a costly third party service. Now users can have all their documents creation time certified automatically when ProveBit runs behind the scenes.

Process Description

General Process

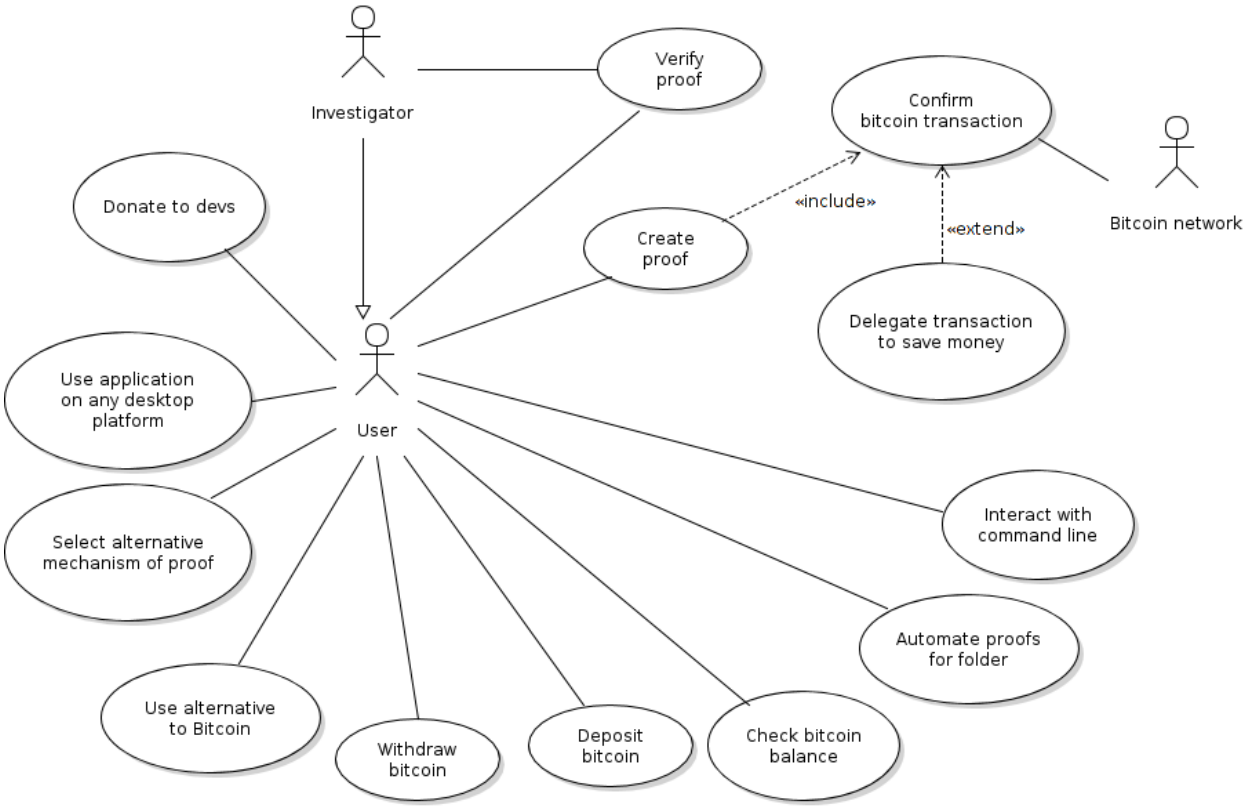
Our process goal was to follow XP as closely as possible. To this end we tried to implement and enforce all major components of XP. We had short, bi-weekly meetings where we discussed state-of-the-project issues and planned the next work steps for each deliverable. We enforced pair-programming, and mandated TDD (except in extreme cases). We set up continuous integration via Travis CI and used Maven for project management. We also set global coding standards and created a Q&A team to review code submissions for compliance with our policies. Finally we developed user stories and use cases, and kept track of our estimated and actual completion time for each user story. By employing all these core XP policies we managed to make steady progress each week.

Issues Addressed

Nearly all development issues we encountered were solved by our adherence to XP. Our semi-weekly meetings made collaboration easy and kept all group members updated as to the state of the project, and allowed review and improvement of in-progress code. Similarly, the meetings provided everyone an opportunity to see areas where refactoring may be necessary, either because the previous code needed to be improved or to prepare a system for future additions. In following XP, our testing philosophy was simple, TDD is required except for extreme cases, and in those extreme cases test cases still needed to be developed as soon as appropriate.

Requirements and Specifications

Use Case Diagram



Use Case Diagram

User Stories and Task Breakdown

Story Description	Task Breakdown
As a user, I can manage document certification from ProveBit's graphical interface	1. Image mockup of the GUI 2. Empty Model of the GUI implemented in Java 3. Controller implementation 4. Data Model implementation
As a user, I can deposit bitcoins into ProveBit's account balance	1. Implement application specific wallet 2. User can move funds from their wallet into the application wallet 3. User can move funds from the application wallet to their wallet

As a user, I can generate a creation time certificate (simple non final certificate)	1. Parse YAML from within Java 2. Write YAML files according to proof spec 3. Serialize path through Merkle tree relative to file 4. Retrieve the path of a transaction in a bitcoin block 5. Embed proof into Bitcoin transaction 6. Trigger proof generation on transaction confirmation 7. Serializing progress while transaction is being confirmed 8. Generate proof file from within UI
As a user, I can verify a creation time certificate	1. Retrieve blocks from BitCoin network 2. Parse and interpret proof file
As a user, I can configure the daemon from the GUI	1. Runnable without ProveBit application running 2. Run periodically 3. Interfacing with application 4. Signals for when files are altered, deleted 5. Track multiple directories and discrete files
Further interfacing between the Daemon Thread and ProveBit Application	1. Implement Server and Client inter-process communication 2. Design messaging protocol for Server and Client classes 3. Fully extract daemon from application (separate process)

Architecture and Design

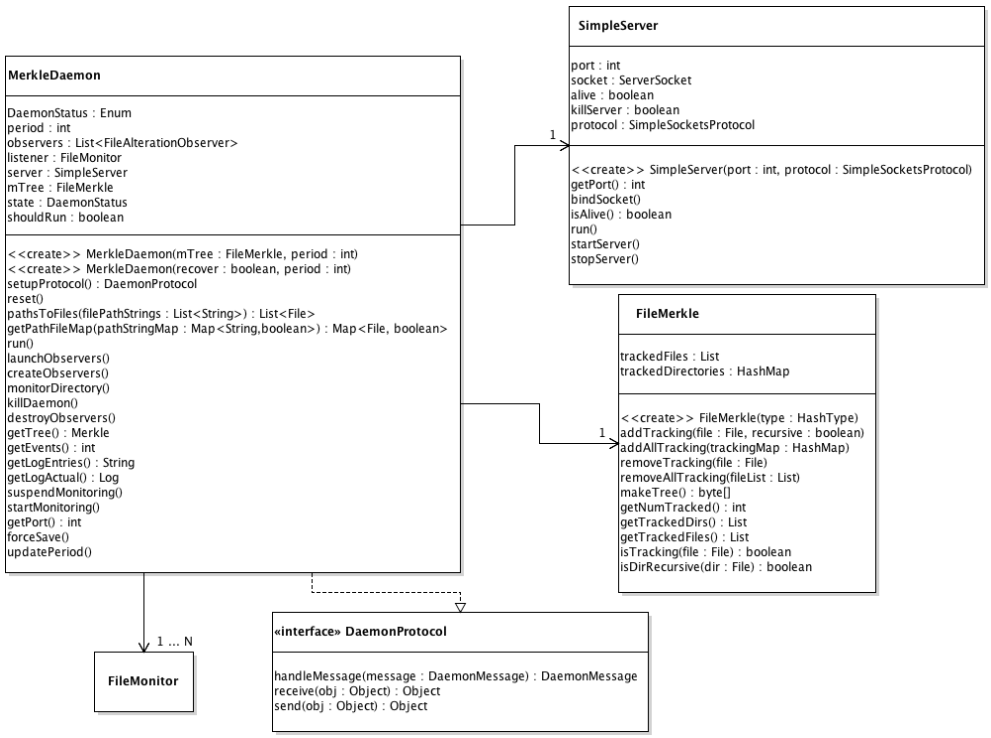
Description of Architecture

The ProveBit application has 4 key systems that provide the bulk of its functionality. The **Daemon** system monitors files so that the application knows when a file has been changed. The **Merkle** system builds merkle trees from files and from bitcoin blocks. The **Proof** system proves files by embedding them into Bitcoin transactions and also verifies those proofs. Finally, the **UI** system implements a GUI that interfaces with the user and the rest of the systems.

Key Systems

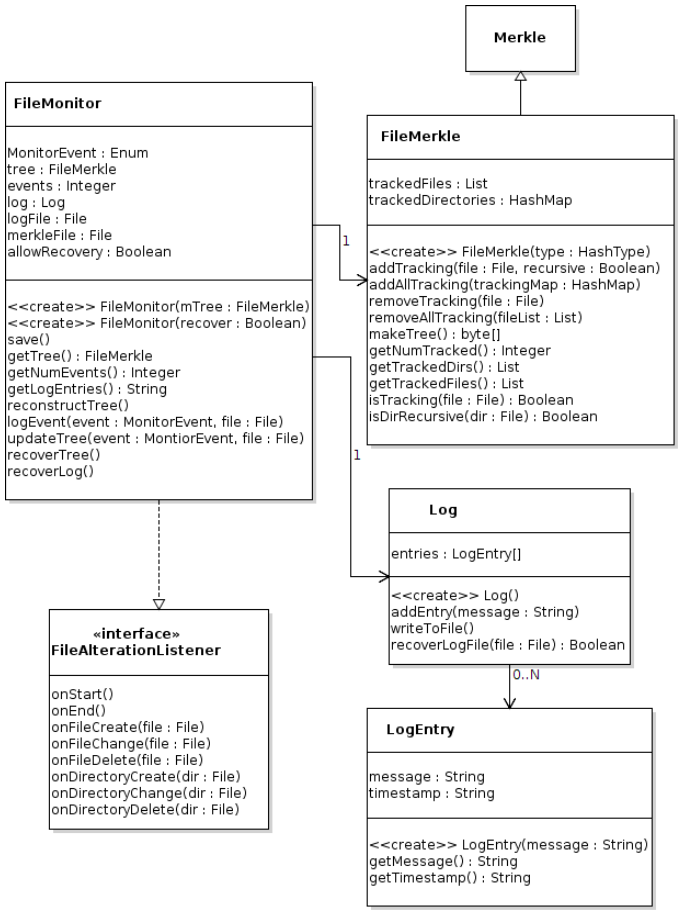
Daemon

The Daemon package provides the ProveBit application with methods for tracking files automatically through the FileMonitor class. The Daemon allows the user to designate a time period after which it will update the application GUI with any changes made to files currently being tracked. The Daemon is implemented as an extension of the Thread class and uses the DaemonProtocol class as well as the Client and Server model contained in the SimpleSockets package for communication between the ProveBit GUI and the Daemon. A UML diagram of the MerkleDaemon class structure follows.



MerkleDaemon UML Diagram

The FileMonitor class is used by the MerkleDaemon to track changes to files and directories that are used in the construction of the merkle tree. It implements an Apache Commons interface that acts as the listener for the modifications and is the class that holds the reference to the FileMerkle object. The FileMonitor may be constructed with a boolean flag which will allow recovery of all previous state (Log and FileMerkle) recorded during the last operating cycle. The FileMonitor will detect and record creation, deletion, and modifications to files and directories, but since directory events do not directly influence the merkle tree construction they will not trigger a tree rebuild. The following figure represents the UML diagram for the FileMonitor class.



Use Case Diagram

Merkle

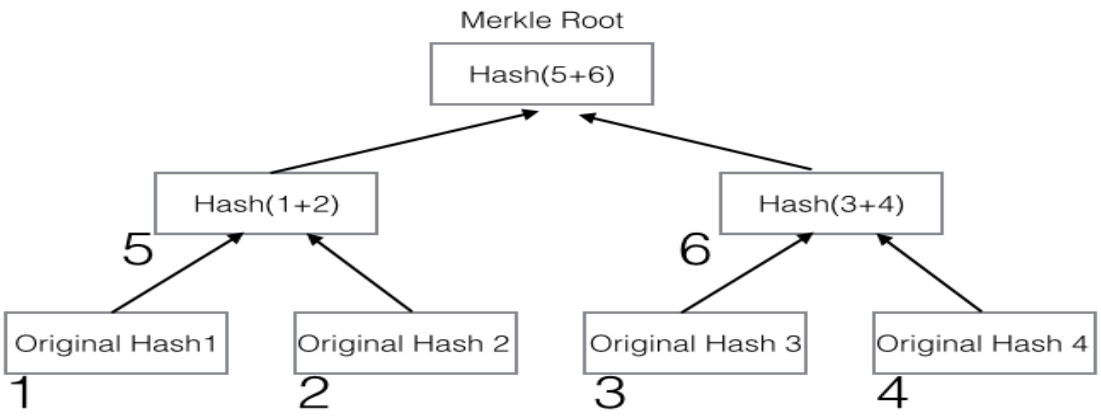
The Merkle package provides the application a way to hash multiple files together and a way to analyze where a transaction is with a bitcoin block. The main unit of this package is the Merkle class, which is our implementation of a merkle tree. A merkle tree is a binary tree where each leaf node represents a hash and every non-leaf node is the hash of the concatenation of its children. Once all of the tree has been formed, the top node (referred to as the merkle root) uniquely identifies the grouping of all of the leaf hashes.

The Merkle tree is stored as breadth first order tree array. In terms of indexing, this means that if a node has an index of I, then its children have indices 2i + 1(for the left child) and 2i+2 (for the right), while its parent (if any) is found at index (i-1)/2.

As mentioned before, we use the merkle class to hash together a group of files. This allows to prove multiple files at the same time by implanting the merkle root of a group of files into the block chain. In order to verify specific files, however, we need to save the “path” up the merkle tree. We define this path as a list of pairs of sides (0 for left, 1 for right) and sibling hashes. For example, if we wanted to get the path for hash 2 in the tree below it would be as follows:

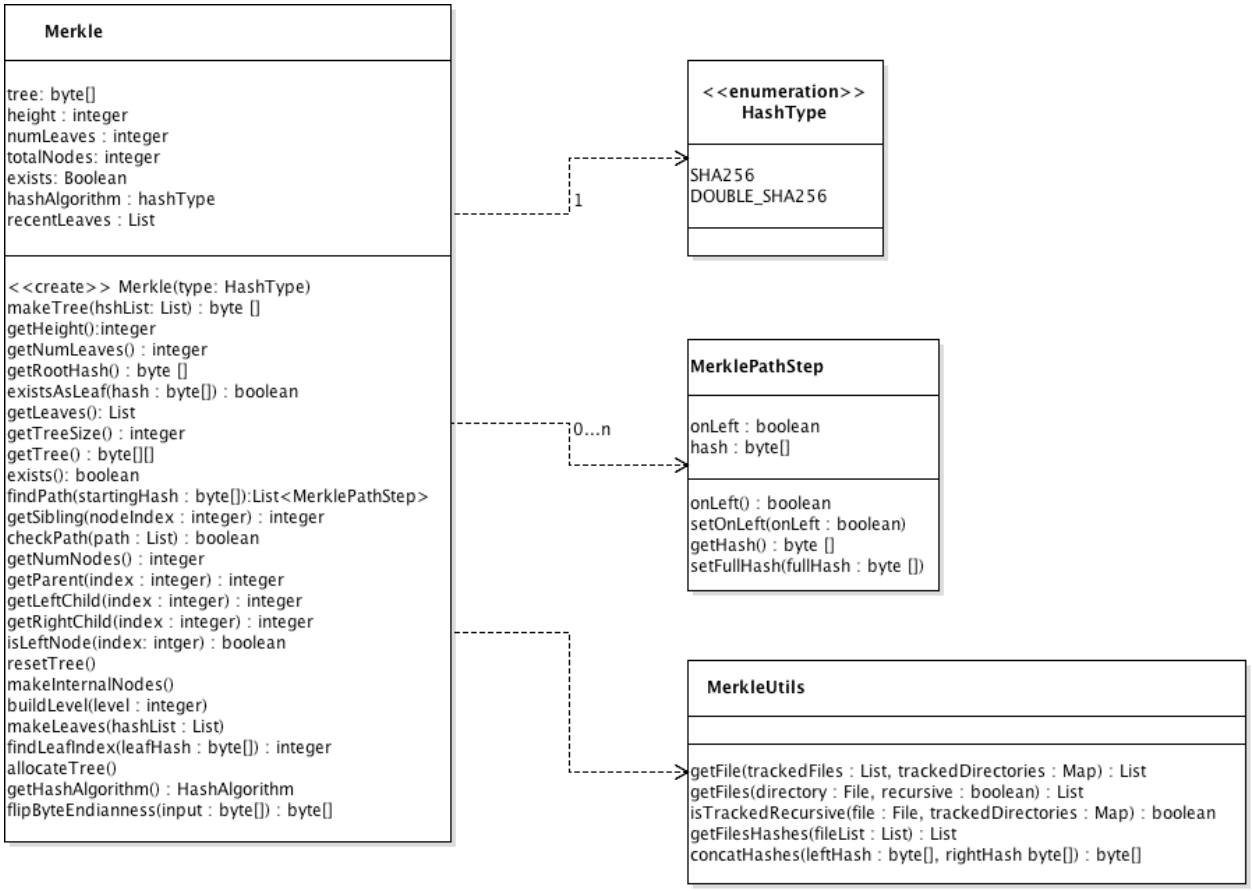
```
{(1, [hash1]), (0, [hash6])}
```

With this list, given a hash and a merkle root to check, you should be able to verify that that hash in the merkle tree or not.



Merkle Tree

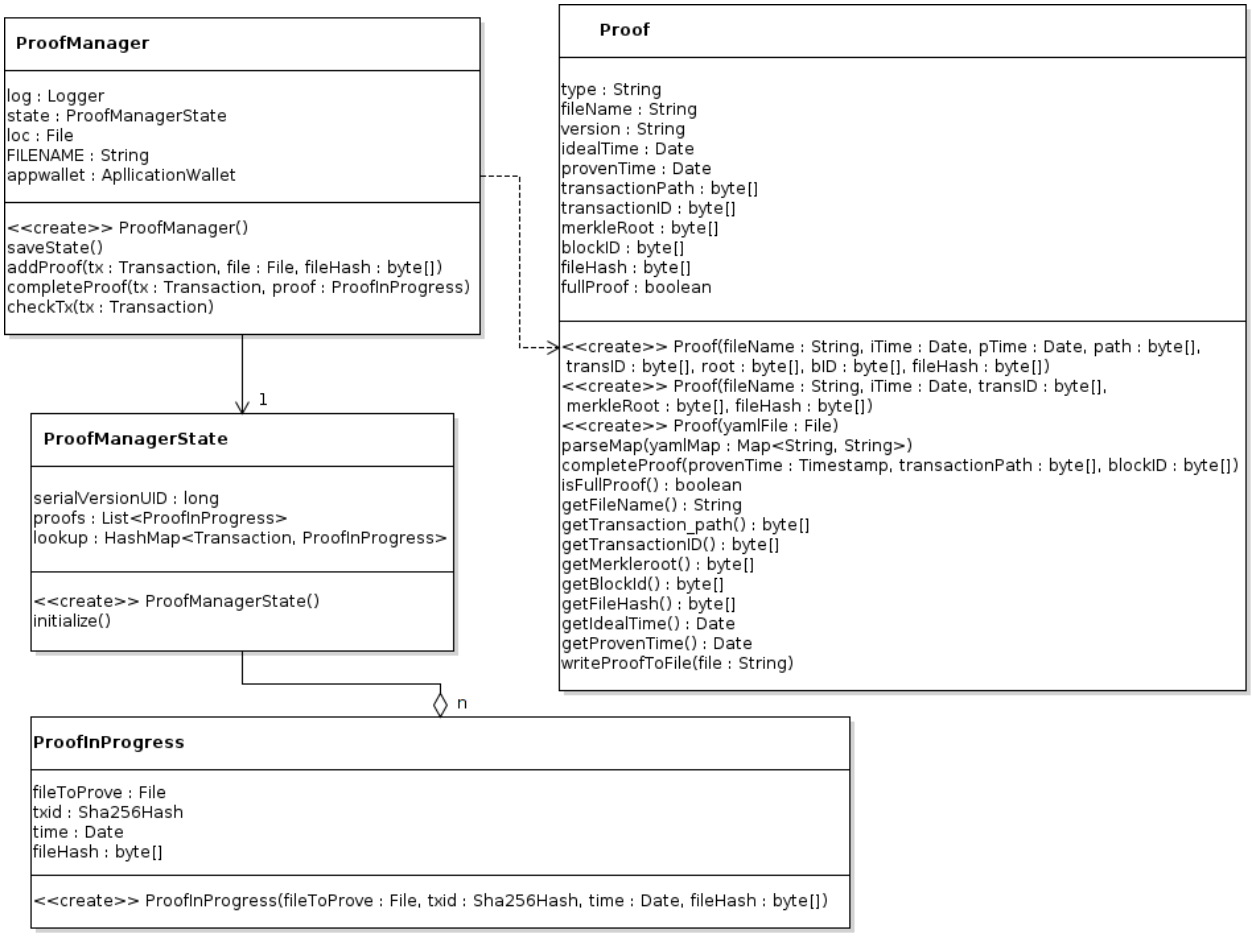
As for analyzing Bitcoin data. Bitcoin blocks store all of their transactions in a merkle tree. We use the merkle class to construct our own merkle tree with the bitcoin transaction data. This allows us to verify whether a transaction is in a block by constructing and verifying a path using the same strategy detailed above. The following figure represents the UML diagram for the Merkle class.



Merkle Class UML

Proof

The proof package provides the ProveBit application abilities to monitor and create proofs. ProofManager is a singleton class which tracks the progress of proofs within the application. The addProof(Transaction tx, File file, byte[] fileHash) method takes the hash of the file and transaction to create a new ProofInProgress object and add it to state monitor. We keep track of the transaction, and we get an event that our confidence of it being accepted by the Bitcoin network changed, we check if at that point the transaction is completed. If so, we write the completed proof file. The saveState() method will save the current proof state when the application closes.



Proof Class UML

UI Architecture

To implement our GUI we followed the MVC design pattern. In particular, we have multiple MVC components which represent each major section (tab) of our application. Making each tab its own unique MVC had two major advantages: work could progress in parallel without any significant conflict between application components, and we were better able to logically separate the application components (no reason for Bitcoin wallet implementation to be next to the Daemon implementation).

The package structure of our UI packages under this design pattern are as follows

```
+ org.provebit.ui
   RunGUI.java (Main launcher)
   AdvancedTab.java
   DaemonTab.java
   GeneralTab.java
   WalletsTab.java
   +   org.provebit.ui.main (Outer frame, tabbed pane in here)
       MainController.java
       MainModel.java
       MainView.java
   +   org.provebit.ui.advanced (Holds MVC components for Advanced tab)
       AdvancedController.java
       AdvancedModel.java
       AdvancedView.java
   +   org.provebit.ui.daemon (Holds MVC components for Daemon tab)
       DaemonController.java
       DaemonModel.java
       DaemonView.java
   +   org.provebit.ui.<other tabs>...
```

When the application is launched, Main MVC gets instantiated. When the MainView class is created it creates a JTabbedPane and adds each component.

```
...
private JTabbedPane tabbedPane;
...
public MainView(MainModel model) {
        this.model = model;
        menuItems = new ArrayList<JMenuItem>();

        setLayout(new MigLayout());

        setSize(800, 600);
        setMinimumSize(new Dimension(500, 375));
        setTitle("ProveBit");

        // set up menu and tabs
        addMenuBar();
        addStatus();
        addTabs();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
}
...
private void addTabs() {
        tabbedPane = new JTabbedPane(JTabbedPane.TOP);
        tabbedPane.setName("Main");
        tabbedPane.addTab("General", new GeneralTab().getPanel());
        tabbedPane.addTab("Wallet", new WalletsTab().getPanel());
        tabbedPane.addTab("Daemon", new DaemonTab().getPanel());
        tabbedPane.addTab("Advanced", new AdvancedTab().getPanel());
        add(tabbedPane, "grow, push, span");
}
```

Application/Daemon Communication

In order to fully separate the daemon from the application, we needed to develop some method of IPC that worked across JVMs. Among the candidates were traditional RMI, CORBA, and sockets. While RMI and CORBA would've provided a more robust interface, the overhead involved in setting up either such IPC mechanism would've been overkill for the simple types of messages that we needed to communicate with the daemon, thus we chose to implement a socket based IPC scheme.

To facilitate communication we developed a reusable (in Java) server/client socket suite called SimpleSockets, which abstracts the details of dealing with sockets and presents the user with a simple interface for designing their own protocol which is used by the server to handle sending and receiving messages where the data is a serializable java object.

```
public interface SimpleSocketsProtocol {
    /**
     * Receive callback, called when data is received on a socket
     *
     * @param data - Serialized object received
     * @return reply object, or null if no reply necessary
     */
    Object receive(Object data);

    /**
     * Sending callback, called when data is to be sent on socket
     *
     * @param data - Application data to be turned into serializable object for
transmission
     * @return Serializable object to send on socket as reply
     */
    Object send(Object data);
}
```

We extended this interface when developing the daemon protocol to handle DaemonMessage objects which encapsulate the type of message and the payload (if any) for that message. This enabled us to cleanly communicate with the daemon, and allowed us to very simply implement new message types as we found a need for them.

Framework Implications

The BitcoinJ library is of fairly high quality and thus provided quality interfaces for our project. For Bitcoin network events that we need to track, we implemented event listeners, which fit our MVC style architecture. The UISpec4J testing framework we used allowed us to do GUI testing effectively by describing actions on the UI in a concise way, so we did not need to adapt the UI code in any specific way.

Future Plans

Personal Reflections

Daniel

My most valuable experience working on ProveBit was working on a large scale project that was built from nothing but an idea. Such a complete view of the software engineering process is unique and richly rewarding. I also learned an enormous amount about Bitcoin and the issue of trusted timestamping. As for the Java language this was my first time programming any sort of threading or synchronization code.

Steve

ProveBit was based on an idea I have wanted to implement for a while, and it is satisfying to see the progress we have made. I have been interested in doing Bitcoin development, and this project was a great opportunity to work with both Bitcoin itself and the BitcoinJ library. I value the gained experience of working on a major project from scratch and using real world software engineering tools: Git, Maven, and Travis CI, which I had not used much beforehand.

Matt

Being able to see a project be build up just a few hundred words on a page was probably the most rewarding experience of this project. Working on ProveBit gave me some insight into what producing and maintaining a large code base is like, as well as managing all the moving parts required for the final product. With the integration of Travis CI, it was rewarding to see freshly committed code versions passing test cases and motivating otherwise.

Noah

Coming into this project I didn't really know anything about Bitcoin or cryptography at all. Working on ProveBit really introduced me to both of these concepts in a "trial by fire" manor. I had to learn certain concepts quickly, however having teammates to discuss these concepts with was really useful.

Quihua

I was interested in Bitcoin, but didn't have any professional knowledge about it or anything about cryptography. This project led me into that world. This is one of the largest project I worked on. I was able to get some experience about working on a large code base and using tools like CI and Maven. Also, working with a library I had never seen was a valuable experience.

Shishir

I had an amazing learning experience while working on ProveBit. Working on this project from scratch, using the Bitcoin network that already intrigued me, and working with helpful and great fellow computer scientists was the most enriching experience of my college life. We used challenging technologies and almost all of us had to overcome quite a few hurdles while

working on the project. I'm glad I chose this project and that I got the opportunity to work with a wonderful team to implement a much needed and useful idea behind ProveBit.

Future Plans

ProveBit is an open source project that we released under the Apache License 2.0. Overall, the project is of prototype quality and would greatly benefit from continued work. We thus plan to continue working on the project after the semester ends. We still have a good amount of work to do on the remaining user stories. A major task among them is implementing the new scripting language, DVScript 0.2, and a new proof file type that uses the scripting language. Since we are on GitHub, we will review and accept pull requests from any new contributors who wish to join.

Bonus: Real World Use Case Example

We have used ProveBit to certify the state of our repository before submission (excluding this documentation).

The code is tagged:
https://github.com/thereal1024/ProveBit/releases/tag/prove-repository-state-test

And the following .tar.gz is what we proved:
https://github.com/thereal1024/ProveBit/archive/prove-repository-state-test.tar.gz

Here is the prove-repository-state-test.tar.gz.dproof contents:


```
{
  Transaction Data:
0100000001c00f6407b7552513162585d6cdb93e5846e2d244cc81d3b64b38bc260
986f404010000006a47304402200b4d415612103c36686cd86ed6397f44b3e40b0b
f48d2e19cf8b9c33240e5b4402205ff18babd79d6982fbd579247c6be0e59b5e780
e6245cb9acf8eed513a860106012102cbd58869420ba0720bc34d8ec6aa489d8916
c45f654b93e665d8414caa1e0405ffffffff02b03001000000000001976a91481cfa
ab9c9fa02d497fa19add4e0c68d5770ddd788ac0000000000000002a6a2850726f
7665426974deca796d99b379752b221d63acc9ce7557fbe131f247f7dead1379b7e
6965c6900000000,
  Block ID:
0000000000000000a0c582f32fc17cf397882c26616ef331246927d0863a945,
  Type: Document Proof,
  Proven Time: 'Sun May 03 23:05:56 CDT 2015',
  File Hash:
deca796d99b379752b221d63acc9ce7557fbe131f247f7dead1379b7e6965c69,
  Transaction Path:
013d49a0e06b219d5e102641468b6249b7fe056cb62765271c1c7fa2a58670a6090
1ecbf49932981c9b8f5b03251f79d9e83c50f4070ee4f96a61989e3a8e254fbda01
89a58c315867d4d685fe39653457db5e7aebc0f8623d4e6137a0144dbd6d97b2005
```

```
33c47c63ee622b1651d39dc08bcf53ce3c2abc4dece0972dd690c030950e4990156
30558947461db7f3aea0ddcbf274f3f4b8ac1333feecf33a5b0563a21f90f501865
3ac34978cfefc800f3297290adbd102237fe60d7615935c59d7b0319669dc017502
938eb71b8093603cb70c2febbc57f2c47479bb88eaa2359cdeebced4f9a700fbcb3
5895dbd23e431f730d210eacb829570a82aeac448d478d944a212236b62002c3b86
d38c3eebfe50859195f5f48cf1a2115d192952018321edbec9a0dfbf880162d9adf
2153c393d996ba736682ebcdebd63b8ff45d33e40d5392b4bcc163c15,
  Version: '0.1',
  Merkle Root:
deca796d99b379752b221d63acc9ce7557fbe131f247f7dead1379b7e6965c69,
  File Name: ProveBit-prove-repository-state-test.tar.gz,
  Ideal Time: 'Sun May 03 22:53:03 CDT 2015'
}
```

You may verify the transaction has indeed occurred before the due date (Sun May 03 23:05:56 CDT 2015) on a blockchain explorer: https://insight.bitpay.com/tx/d39323b7a17e68aba85d61adf1f83854797f570860bcd15e1dbaf258fdbd4aa5

Thus, we have PROVED our on time submission!