SparkleShare

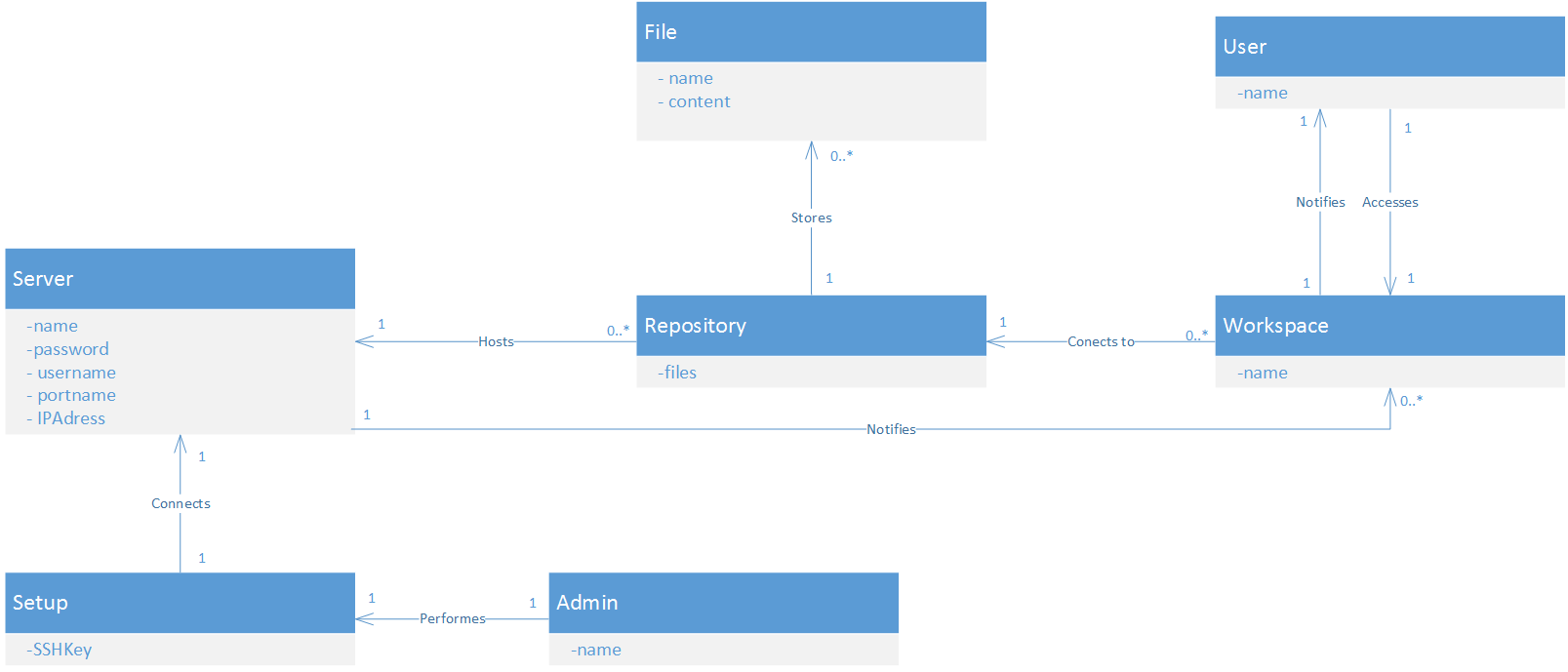
|  |  |
| --- | --- |
| **Alistair Mersereau**  **Scott Goldwater**  **Joshua Whatley**  **Genevieve Chafouleas**  **Jad Khoriaty** | **9021957**  **9338071**  **9643877**  **5647770**  **1959220** |

# Summary of Project

SparkleShare follows a similar idea as Dropbox in the way that it provides file synchronization between some client machine and some remote server machine. It differs from Dropbox in that it is self-hosted, it provides mechanisms for client-side encryption as well as version control, and it is free and open-source. Sparkleshare uses git as its versioning control system to manage its files, meaning it can only handle smaller frequently changing files. Sparkleshare is completely customizable. You can create and provide your own server or use outside sites like Github and Bitbucket to host your files.

# Class Diagram of Actual System (5 points)

## Conceptual Diagram



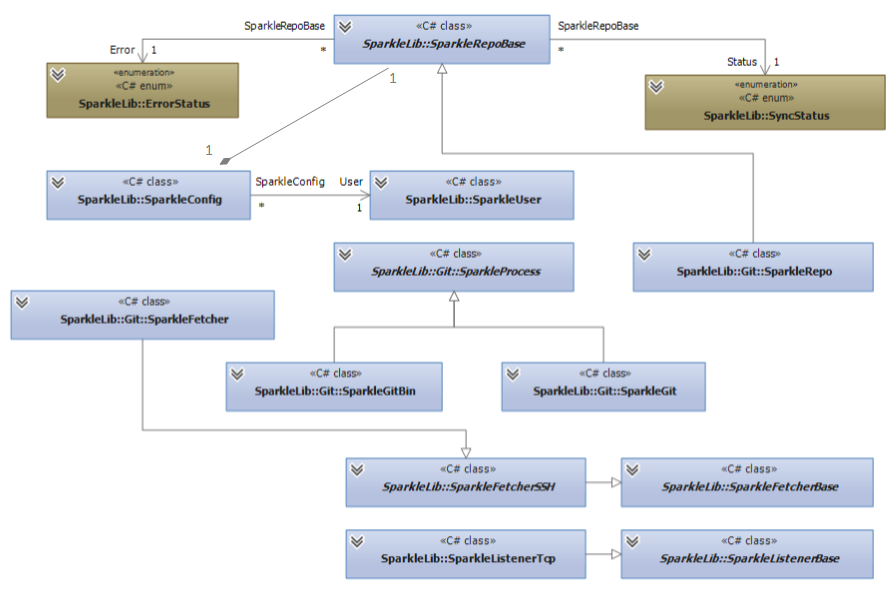
## Class Diagram

Our focus is on the SparkleLib library within the SparkleShare project. Since the project is heavily event-driven, the class diagram is quite divided. We take advantage of this to break down its presentation and corresponding explanations.

### Overview

This class diagram provides an overview of the interactions between the major classes in the SparkleShare environment. For detailed class diagrams for each individual subsection, refer to the three following sections.

RepositoryControl

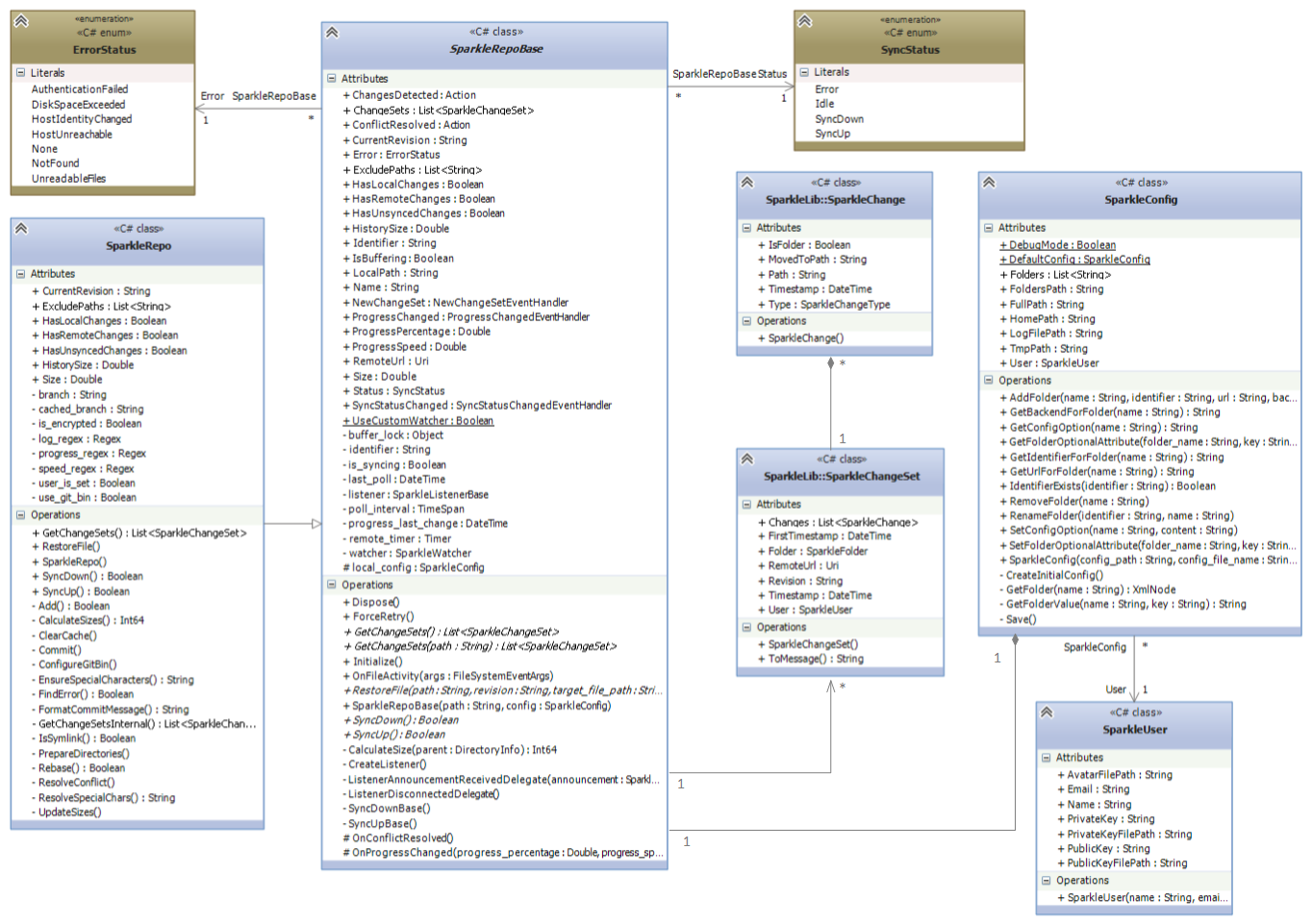


UpdateHandler

GitControl

The above diagram can be split into three separate groupings: Repository Control, GitControl, and Update Handler. Repository is mainly responsible for communicating with whichever server/method you have chosen to store the project. GitControl is the main class that deals with the git calls with regards to Sparkle Share. UpdateHandler is a group of classes that use a Broadcaster-Listener setup up to listen for updates in the file that is shared through SparkleShare.

### Repository Control Class Diagram



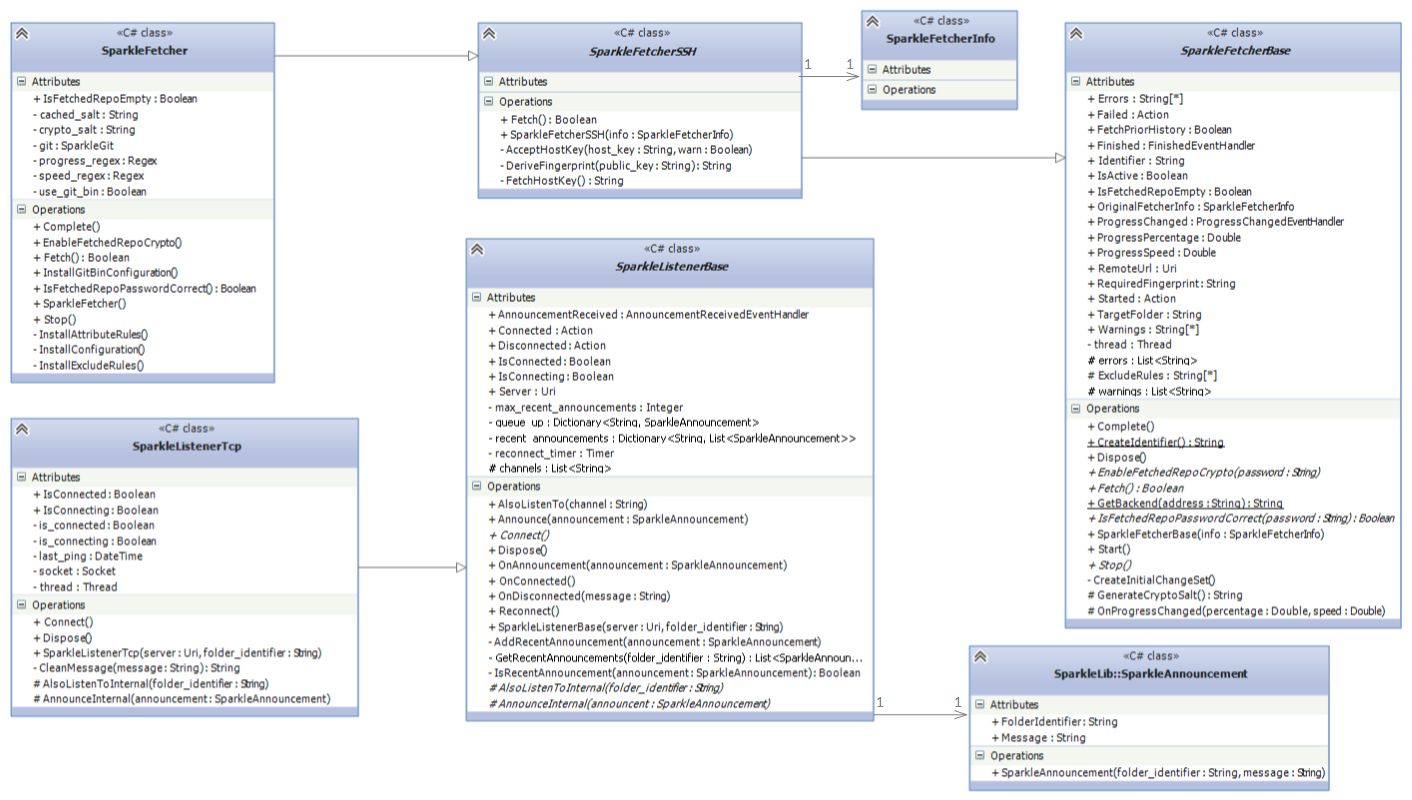
### The RepoControl grouping of classes is responsible for everything related to the files and folders that are used in the repository. This grouping provides the same functionality as the “Repository, File, User, and Workspace” classes in the conceptual diagram. Both SparkleRepository and SparkleBase map to the Repository and Workspace class in the concept diagram, SparkleUser maps to the User class. In their actual class diagram there is this discrepancy between the Repository and Workspace classes, because SparkleShare doesn’t have a concept of local vs remote repository. This does not have any serious repercussions in regards to the design of the system.

### Git Control Class Diagram



The GitControl Grouping are classes that provide git functionality to the Repository. If compared to the Conceptual diagram this isn’t a feature that we have included. This isn’t included in our conceptual diagram because in concept it is not assumed that the user will be using a specific Source Control Software.

### Update Handler Class Diagram



The UpdateHandler grouping are classes that provide the functionality of updating the repository for the users that have access. This grouping can be compared to our conceptual diagram, and it basically performs the same role as the Server class, which leaves the admin class which is unused in the real class diagram. This class diagram separates all of the server functions into a few smaller more manageable classes, rather than managing all server functionality in the same class. This leaves a discrepancy with the admin class which the class diagram does not use because SparkleShare considers a user and an admin to be the same user.

## Reverse Engineering Tools

We used the Visual Studio 2013 architecture feature to reverse engineer the SparkleLib library. This feature enables the construction of a variety of UML diagrams based on the associations present in the code.

## Relationship between SparkleConfig & SparkleUser

The SparkleUser class has 5 references in the SparkleConfig1 class all located within the getter/setter of the user in the config. The five references are marked by yellow underlining. Further references to the user class are through the instance stored in the “User” attribute in SparkleConfig. A code map is provided which outlines further connections after SparkleConfig.

|  |  |
| --- | --- |
| public class SparkleUser {  public readonly string Name;  public readonly string Email;  public string AvatarFilePath;  public string PrivateKey;  public string PrivateKeyFilePath;  public string PublicKey;  public string PublicKeyFilePath;  public SparkleUser (string name, string email)  {  Name = name;  Email = email;  }  } | public class SparkleConfig : XmlDocument {  ...  public SparkleUser User {  get {  XmlNode name\_node = SelectSingleNode ("/sparkleshare/user/name/text()");  XmlNode email\_node = SelectSingleNode ("/sparkleshare/user/email/text()");  string user\_name = name\_node.Value;  string user\_email = email\_node.Value;  SparkleUser user = new SparkleUser (user\_name, user\_email);  string [] private\_key\_file\_paths = Directory.GetFiles (Path.GetDirectoryName (FullPath), "\*.key");    if (private\_key\_file\_paths.Length > 0) {  user.PrivateKey = File.ReadAllText (private\_key\_file\_paths [0]);  user.PrivateKeyFilePath = private\_key\_file\_paths [0];  user.PublicKey = File.ReadAllText (private\_key\_file\_paths [0] + ".pub");  user.PublicKeyFilePath = private\_key\_file\_paths [0] + ".pub";  }  return user;  }  set {  SparkleUser user = (SparkleUser) value;  XmlNode name\_node = SelectSingleNode ("/sparkleshare/user/name/text()");  XmlNode email\_node = SelectSingleNode ("/sparkleshare/user/email/text()");  name\_node.InnerText = user.Name;  email\_node.InnerText = user.Email;  Save ();  }  }  ...  } |
| **Code Map:**    *Grey represents references in SparkleConfig*  *Green represents references in SparkleRepo*  *SyncDownBase is located in SparkleRepoBase* |

# Refactoring

## Suggested Refactorings

### Large Class & Long Method

The general solution to the bad code smell “Large Class” is to extract class or extract subclass. By doing so, we will end up with several smaller classes that are coupled to less classes and have a smaller Cyclomatic Complexity. To fix the “Long Method” bad code smell, we have to shorten the method by extracting methods from the main method.

The SparkleRepoGit class can be divided into two classes, one that handles exceptions and one that takes care of syncing up and syncing down. To accomplish that, every single method has to be examined and identified as handling exceptions or taking care of syncing the files. If there are methods or data that are needed by both tasks, we should create a new class that will be used as parent class to both the other classes. That will make this method and data accessible by both classes.

Methods to be placed in the class that handles exceptions (non-exhaustive):

* FindError(string) : bool
* ResolveConflict() : void
* ResolveSpecialChars(string) : string
* RestoreFile(string, string, string) : void

The rest of the methods can be kept in the class that is responsible for syncing.

In addition to this, some methods have to be broken down into several smaller pieces. For example, the GetChangeSets method has to be extracted. Its Cyclomatic Complexity is of 36, it is coupled with 22 other classes in 138 lines of code all to itself. The methods that are going to be extracted will be called by the initial method.

### Put all data in constant file and resource file:

Currently in SparkleShare we have identified a code smell of Hidden Assumption and potential duplicated code. There is a lot of strings and data that are just loosely written in the code and are not associated with any meaning or variable name. This makes it very hard to identify why these specific numbers or strings are used. We plan on moving all loose numbers to a constant file and all loose strings to a resource file using the replace magic number with symbolic constant refactoring. This will make maintainability easier in the future.

This will be done the following way:

For the strings first identify a loosely used string. Create a new resource string with the string value. Find every location with the same string and if it matches the same usage of resource string. If matches replace string with resource string name. Repeat for every common string. Compile code and test functionality.

For the data first identify a loosely used data. Create a new constant with the data value. Find every location with the same data and check if matches the same usage of the new constant. If matches replace data with constant. Repeat for every common data. Compile code and test functionality.

### No use of Interfaces

For the core refactoring within SparkleShare we will be changing the code to an Adapter pattern. As of now the code git is hardcoded into SparkleShare, which makes supporting additional types of repositories difficult. If the code moved to an adapter pattern all the developer would need to do to create support for a new repository protocol would be to implement a generalized repository class. The code presently uses abstract classes in order to combine repositories to the program’s processes however this doesn’t allow the developer to have absolute flexibility within the code. For instance, within SparkleShare creates an abstract class named “SparkleProcess” which defines methods that a user has to overload or disregard when extending the class. If SparkleProcess was turned into ISparkleProcess (an interface for all process) the user can simply implement their own methods without having to disregard code.

## Refactoring Analysis (No use of Interfaces)

The following class diagram demonstrates what the system design will look like after performing this refactoring.

