Design Document: HCS Group

*Six Design Patterns*

**MVC**- The Model, View, and Controller pattern is the three main components to software development. Model: classes that maintain data or state of our program; do not contain information about how that state is shown to the user. View: classes that present the data in the model to the user, and/or allow her to interact with it (often this is the program’s GUI). Controller: classes for coordination of interaction with the user. Typically this includes event handlers for input and also perhaps the GUI components that gather that input. In our model package in our program, it contains two classes, which are the Revision Documents and the User. The revisions made are connected to a given User and are saved in a way that one is able to view what they have written previously using this function in this package. Our “View” is considered to be our MainGUI in the MainGUI package. This package contains a class called MainGUI as well as a Time Class that will keep track as things are saved at time. The controller is basically the MainGUI as well seeing that the text pane is inside of there and is where we edit as a user.

**Command**- The command pattern is known as a behavioral pattern, as it's used to manage algorithms, relationships and responsibilities between objects. Command is basically the interaction between server and client. In our program, we have a package called Server that has three different servers. These servers represent the Editor Server, Chat Server and Paint Server. These servers interact individually with the clients in the packages where these characteristics exists and update using the client. For example the PaintPanel talks to the PaintClient directly to have it update the server so that interactions happen between the windows.

**Template**- The Template Method pattern is used when two or more implementations of a similar algorithm exist. This pattern exists when an abstract class is created or exists. It can connect to a concrete class as we have in our program. This can be an example of how our GUI is made with functionality spread to different classes. Template pattern also may contain final variables that cannot be changed which we used throughout our program.

**Composite**- Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects. This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects. We are demonstrating use of composite pattern via following example in which we will show employees hierarchy of an organization. We used the composite pattern by separating functions of different things we have in our program. We separated the chat from the editor from the paint. These are all in different packages and they are grouped as different functions called by a MainGUI.

**Builder**- Defines an interface or abstract class for creating parts of a

Product. Lets one class (Director) deal with building the object while allowing the other class (ConcreteBuilder) to deal with a valid object.

Can be used to substitute the constructor call in a class. We used this design pattern by separating out editor from the MainGUI. All the features of the text edit is inside of the MainGUI, but it is called because all the features are in a different package in separate packages.

**Flyweight Factory Pattern**: Creates and manages flyweight objects and ensures that flyweights are shared properly, when a flyweight is requested, FlyweightFactory supplies an existing instance or creates one, if none exists. One of the main examples of flyweight is the idea of a word processor such as what we created. These can be things such as shared documents and permissions.

Design Doc (cont.): Project Requirements

**Server**

1. **Facilitates the creation of users:** We implemented this function inside our login window and we can create a new user and store it in a HashMap.
2. **Allows users to create documents:** The user can create a new document as soon as they login. This is just used as a new document created by user initially.
3. **Allows multiple users to connect simultaneously over a network connection using:** We are able to connect and work on a document together simultaneously. We can also paint over the network together on a Paint client that we had made.

**Server Socket objects**

1. **Maintains a list of users:** We are able to maintain a list of users through the HashMap created.
2. **Allows users to grant editing permissions or ownership to other users:** We are able to grant permission to edit a document by reading ones user name. Networking issues do arise with this though.
3. **Has a unique user name (String) and unique numerical identifier (id):** We are able to create a username using characters as well as numbers.
4. **Has a password that is stored only in hashed form:** The password is stored in a HashMap as required.
5. **Has the ability to change a password:** We are able to override someone password who has an account created.
6. **Has the ability to reset a forgotten password by providing their user name (*for the sake of simplicity, we will assume this is all they need to do*):** Yes, we are able to change a password that is forgotten by getting the forgotten and using the username and changing the password.
7. **Has a list of documents owned by the user:** We have the list popping up, but it won’t save to the list that we created. We created the text area to store, but it won’t maintain the documents.
8. **Has a list of documents editable by the user:** We have the list popping up, but it won’t save to the list that we created to be editable by a user. We created the text area to store, but it won’t maintain the documents.

**CSc 335 Final Project Spring 2015**

**Document**

1. **Contains rich text information which can be stored in any format: html, rtf, plain text with separate metadata – the storage format is up to you:** This function works with all the different things required such as bold, italics, etc. These were all implemented using HTML and all work well.
2. **Contains a revision history, where each revision contains the text modified, the time of revision, and the revising user. Sample revisions include:** The revision history on the panel does work and it does it to the time on mark.
   1. **user 'a' adding one line of text at 09:12;00.00**
   2. **user 'b' adding a line of text at 09:12;05.01 (note that this revision depends on the revision of user 'a' – by itself, it will not produce the document that they have produced together)**
   3. **user 'a' formatting some text at a given time**
   4. **user 'a' deleting some text**
3. **The document should, at every moment, have revisions that represent the final product. This means that if a document were to be created from a revision list, it should match exactly the document with that revision list.** We have not fully gotten the saved document that as maintained.
4. **The document must support rich text formatting, and must include these formats:** We support all of these using HTML tags.
   1. **bold text**
   2. **italicized text**
   3. **indented text**
   4. **colored text**
   5. **text in different fonts**
   6. **text in different sizes**
   7. **lists**
5. **The documents must also include more formatting elements as specified by you:** We have an align center, left and right.
6. **Formatting is not mutually exclusive; multiple formats can be combined. For example:** We can combine all of these features with multiple HTML tags. We used a JToggleButton for it to be a press when we are using it.
   1. **bold, indented text**
   2. **a list of text, where each list element has a different color and font size**
   3. **a sentence where each word has a different font and font size GUI**
7. **Prompts users to log in:** This does launch as soon as the program runs.
8. **Allows users to perform functionality listed under *User* (change password, reset password, etc.):** This does exist under the login screen.
9. **Displays a list of documents owned or editable by the logged-in user:** This does not exist, but the panel do exist.
10. **Allows for creation of a new documents:** This is created as soon as the user prompt launches.
11. **Allows for changes to the permissions or ownership of a document:** This is the feature of adding a user to a given document to give permission.
12. **Editor** 
    1. **Has buttons for each of the supported formats.** 
       1. **If a button is pressed while text is highlighted, apply that formatting to the text**
       2. **If a button is pressed while no text is highlighted, apply that formatting to all text entered until the button is pressed again. Make it obvious to the user when this is the case.**
    2. **Has a large text editing area (such as a JTextPane)**
    3. **Displays the revision history of the document. Clicking on a revision will display the document as if that revision was the most recent and no other revisions had been made. The editors can opt to “rollback” the document to a specified revision.**
    4. **Displays a list of currently connected editors, as well as all editors, and provides a chat panel for them to chat to each other without modifying the document.**

All of these functions work as stated above and the button is created as a JToggleButton and it is pressed and can work with multiple features at a time. We also have a display of the user and it displays the list of connected editors and it even shows who is online on the chat.