2016

TFTP Framework

SOFTWARE DESIGN PATTERNS IN JAVA

NAZMUL ALAM

T00152975

Contents

Framework Implementation

- Refactor to patterns
- Remove code smells
- Follow design principles e.g. SOLID

Implemented

The design patterns listed below are the working patterns I have refactored so far. My aim is to achieve at least 8 design patterns.

1. Logging (Common)

```
Normal Singleton -> Threadsafe -> Enum
```

2. State Pattern (Client)

```
Converted authentication to state pattern

When `loginButton` is pressed, Authentication class wrapper delegates to its current state reference.

Eliminated if/else case from UiWindow actionPerformed
```

3. Proxy Pattern (Common, Server, Client)

```
Modified implementation to proxy pattern

DataSocket interface is the Subject

DataSocketImpl class is the Proxy on the server side
```

```
Server class is the client

ClientHelper is the proxy on the client side



N.B.

* All return types must be primitive or serializable

* A virtual proxy is a placeholder for "expensive to create" objects.

  The real object is only created when a client first requests/accesses the object.
```

4. Command Pattern (Server)

```
  Created command pattern to make the client requests simpler

  Code is easier to read now

  Code is still not completely following the OCP

  Need to remove the switch case smell by Strategy/State pattern
```

5. Strategy Pattern (Client)

```
 Cleaning up code from the UiWindow

 Single Responsibility Principle
```

6. Composite Pattern (Server)

```
 Using pattern to manage users
```

Possible Implementations

Here are the possible design patterns I could implement. This includes code smells and code principle violations.

1. Swing UI (Client)

```
MVC pattern (Observer + Composite)
```

2. Decorator (Common)

```
File input and output (IO)
```

3. Template Method (Client, Server)

```
The request processes
```

4. Observer Pattern (Client)

```
Display logArea (JTextArea) messages using observer pattern

This will make the classes as a SRP
```

Singleton Pattern

Intent

“Ensure a class has only one instance, and provide a global point of access to it (Shvets, n.d.).”

Before

Using Log4j Library, I created a constant variable in the classes I needed to log information e.g.

```
17  public class Server {
18
19    static final Logger LOGGER = Logger.getLogger(Server.class);
20
```

Figure 1 Server class before Singleton Implementation

However, after a thorough research, I found Singleton pattern is a good choice for logging, along with database connection or file manager. According to IBM, using Singleton patterns is a classic example for logging.

In this implementation, all the messages I log are saved to a single file. Each logged information includes the name of the class it has received the log information from. Although, it is sometimes useful to know where the log message has originated from, storing the log information in the file system could potentially mean there are more than one instance of the logger, and therefore, it is using more than one thread. This may result in a corrupted file.

Motivation

JDK uses Singleton for its Logger API.

After

```
22  public class LoggerSingleton {
23
24    private String LOG_EXTENSION = ".log";
25    private static Logger rootLogger;
26    private static String logFileName;
27    private PatternLayout patternLayout;
28    private static final Logger LOGGER = Logger.getLogger(LoggerSingleton.class);
29    private static LoggerSingleton loggerInstance;
30
31    private LoggerSingleton() {
32      rootLogger = Logger.getRootLogger();
33      rootLogger.setLevel(Level.DEBUG);
34      patternLayout = new PatternLayout("%d{yyyy-MM-dd HH:mm:ss} [%t] %-5p %c{1}:%L - Status: %m%n");
35      rootLogger.addAppender(new ConsoleAppender(patternLayout));
36    }
37
38    public static synchronized LoggerSingleton getLoggerInstance() {
39      if (loggerInstance == null) {
40        loggerInstance = new LoggerSingleton();
41      }
42      return loggerInstance;
43    }
```

Figure 2 LoggerSingleton thread safe implementation

[GitHub Commit]

In the above example, I created a thread-safe Logger service, and wrapping the Log4j's Logger API in the implementation. According to Joshua Bloch, we should enforce Singleton property with a private constructor or an Enumeration type. Making a class Singleton can make it difficult to test its clients, as it's impossible to substitute a mock implementation for a singleton unless it implements an interface that serves as its type. Joshua recommend to make an enum type with one element. Here is an implementation of Singleton by Enumeration: [GitHub Commit]

```
17  public enum LoggerSingleton {
18    INSTANCE;
19
20    private static final Logger LOGGER = Logger.getLogger(LoggerSingleton.class);
21
22    private LoggerSingleton() {
23    }
24
25    public static LoggerSingleton getInstance() {
26      return INSTANCE;
27    }
```

Figure 3 Enumerated singleton

Design

- Class is responsible for lifecycle
- Static in nature
- Needs to be thread-safe
- Has private instance
- Has private constructor

Pros

- Singleton pattern prevents other objects from instantiating their own copies, thus, ensures a single instance
- As it controls the instantiation process, it has the flexibility to change the instantiation process

Cons

- Singleton pattern is considered as an anti-pattern and are often overused.
- Making everything Singleton may slow down your application.
- They are difficult to unit test.
- Violates SRP by controlling their own creation and lifecycle
- Almost impossible to subclass a Singleton
- Creates tightly coupled code that is difficult to test

State Pattern

Intent

“Allow an object to alter its behavior when its internal state changes. The object will appear to change its class (Shvets, n.d.).”

Before

UiWindow.java [GitHub Commit]

```
278     @Override
279     public void actionPerformed(ActionEvent event) {
280
281       if (event.getSource() == loginButton) {
282         if ("Connect".equals(loginButton.getText())) {
283           LOGGER.info(ProtocolCode.LOGIN + " Login request sent");
284           login();
285
286         } else if ("Disconnect".equals(loginButton.getText()) && loggedin) {
287           LOGGER.info(ProtocolCode.LOGOUT + " Logout request sent");
288           logOut();
289         }
```

Figure 4 UiWindow switch case code smell

In the above implementation, authentication is processed with a single button i.e. loginButton press from the GUI. The text of the button changes to “Disconnect” on successful login, thus, effectively allowed me to add another if block to process logging out.

Sequence of if or switch statements are considered to be a code smell. Often code for a single switch can be implemented in multiple places in the application. When a new condition is added, I would have to find all the implementation and modify it. This breaks the Open closed principle i.e. open to implementation but closed for modification.

Motivation

Logging in and Logging out are two separate states. Localizing the state behavior, will allow me to store the state in an object, rather than in mixed of variables i.e. logggedin. This effectively makes me follow the Open Closed Principle.

After

UiWindow.java [GitHub Commit]

```
274     @Override
275     public void actionPerformed(ActionEvent event) {
276
277       if (event.getSource() == loginButton) {
278         auth.authenticate();
```

Figure 5 State pattern integrated

As you can, this simply calls the authenticate method from the Authentication context class and wraps the states within an Object polymorphically. Although, implementing a state pattern may have been an overkill for the login/logoff, it still makes the code more simple and readable.

In the first implementation, I was still calling the login() / logOut() method from UiWindow class in the LoggedOutState and LoggedInState respectively. However, I fixed it in the later implementation. There was a trade-off to bring the authentication processing implementation within the state objects. I had to pass the input from the UiWindow to the State objects and set the values to the Authentication context class. [GitHub Commit]

```
@Override
public void processAuthentication(JTextField serverInput,
                                  JTextField portInput,
                                  JTextField userInput,
                                  JPasswordField passwordInput,
                                  JTextArea logArea,
                                  ClientHelper helper) {
```

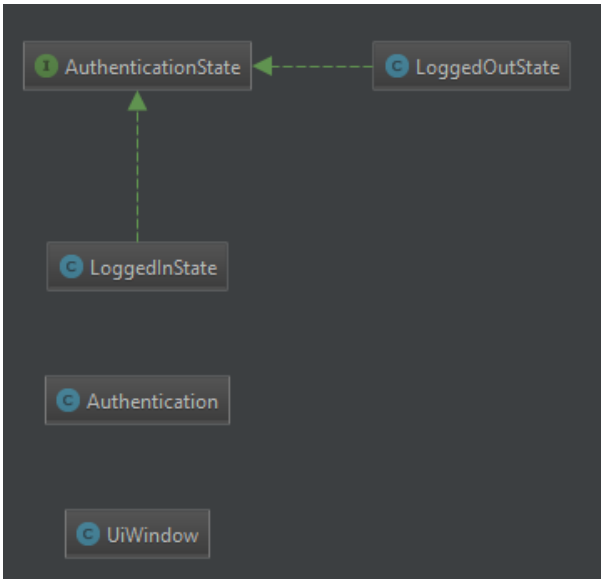
Figure 6 New method implemented required in the State

```
private void setAuthenticationContext(Authentication auth) {

  auth.setServerInput(serverInput);
  auth.setPortInput(portInput);
  auth.setUserInput(userInput);
  auth.setPasswordInput(passwordInput);
  auth.setLogArea(logArea);
  auth.setHelper(helper);
}
```

Figure 7 Setting values to Authentication context in UiWindow class

UML Diagram



Design

- Abstract class / Interface i.e AuthenticationState
- Concrete State classes i.e. LoggedInState and LoggedOutState
- Context i.e. Authentication
- Client i.e. UiWindow

Pros

- Simplifies cyclomatic complexity
- Removes switch case smell

Cons

- Requires you to know your states before implementation
- Adds extra classes within your application
- Keeps logic out of context
- Identifying what triggers the state can be tricky

Proxy Pattern

Intent

“Provide a surrogate or placeholder for another object to control access to it (Shvets, n.d.).”

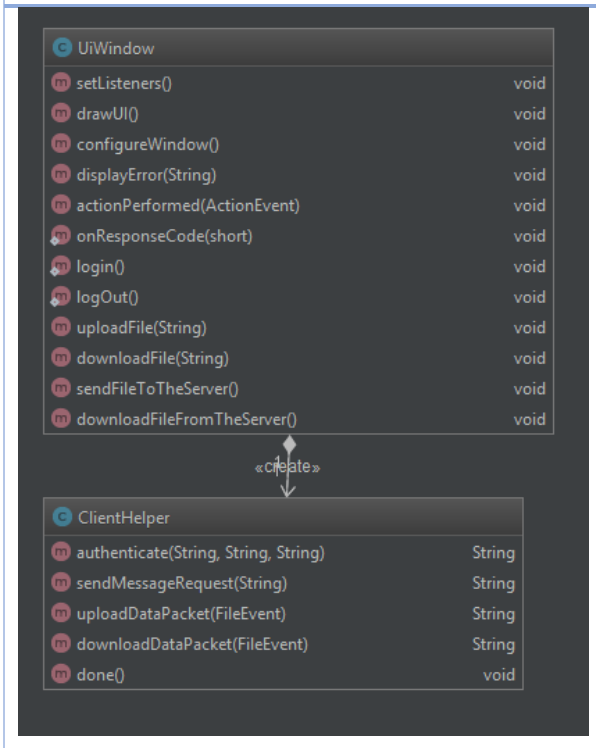
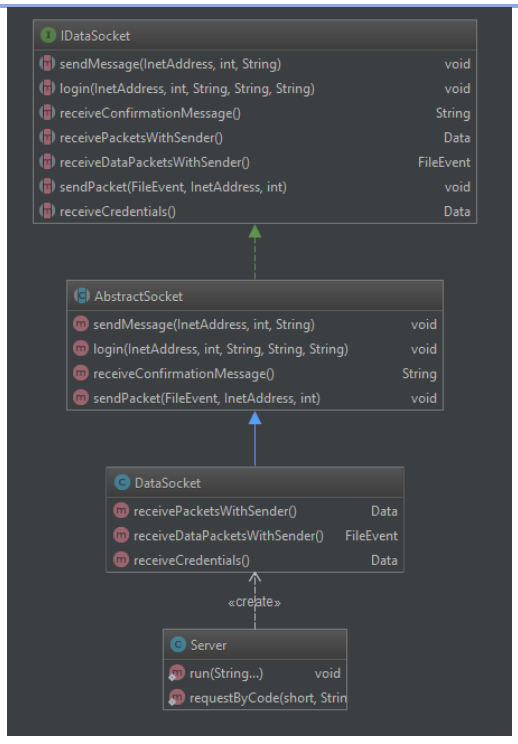
Before

In this exercise, I am refactoring a client-server based FTP application using Java’s Socket API. I have implemented this application as part of my Distributed Computing module.

Most of the implementation in the application was already using the proxy pattern. However, few things needed to be refactored.

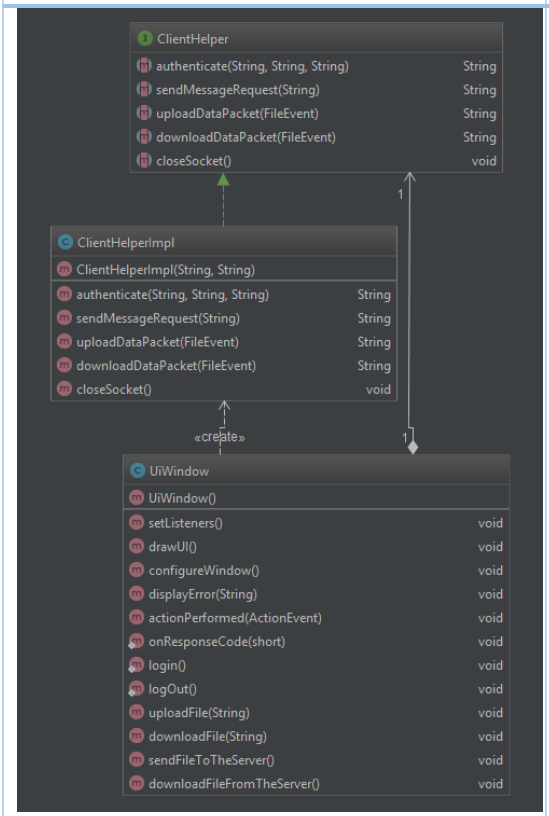
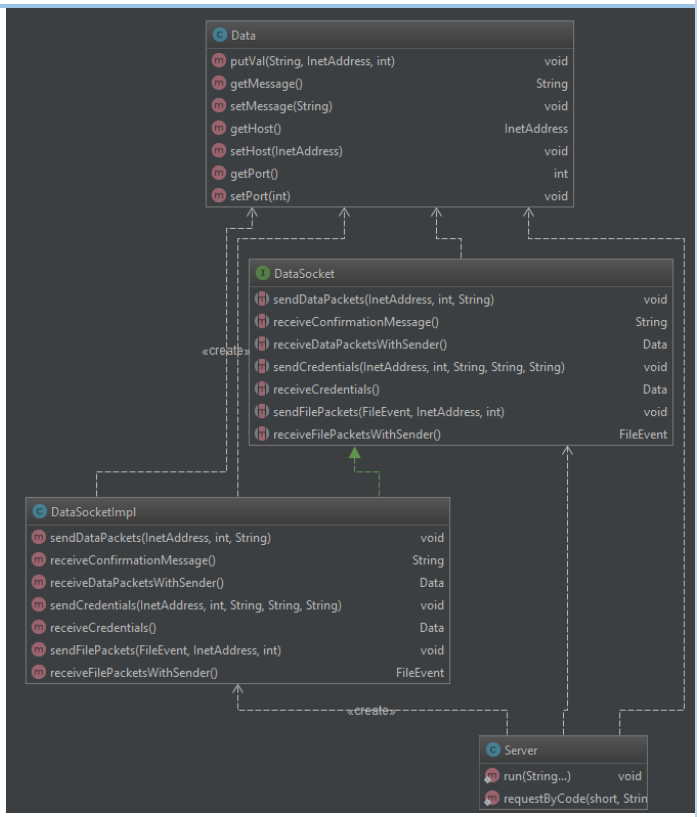
UML Class Diagrams (Before)

Table 1 Client-Server proxy pattern [GitHub Commit]

Client	**Server**
	

UML Class Diagrams (After)

Table 2 Client-Server proxy pattern [GitHub Commit]

Client	Server
	

The proxy is used to solve remote calls from client and server layers. The server listens to localhost on port 3000 and based on the request sent from the client, the server processes the request accordingly.

```
while (true) {
  Data request = socket.receivePacketsWithSender();
  String message = request.getMessage();
```

Figure 8 Server.java waiting for a request

```
365         helper = new ClientHelperImpl(host, port);
366         logArea.append("Status: Logging into " + host + "\n");
367         responseCode = helper.authenticate(Constants.LOGIN, username, password);
```

Figure 9 UiWindow.java Client sends login request [GitHub Commit]

Motivation

Proxy pattern is widely applicable in almost every distributed system.

Design

- Interface based
- Interface and Implementation Class where proxy resides

Pros

- Adds functionality at compile time
- Great utilities built into Java API
- Used by Dependency Injection or Inversion of Control Frameworks
- Decoupling clients from the location of remote server components
- Separation of housekeeping code from functionality

Cons

- Only one proxy instance is allowed
- Adds another abstraction layer
- Less efficiency due to indirection
- Very similar to decorator, or other patterns, making it very confusing.

Command Pattern

Intent

“Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations (Shvets, n.d.).”

Before

```
if (loggedInUser != null && loggedInUser.isAuthenticated()) {
  loginPacket = new LoginPacket(opcode, message, request, socket);
  writePacket = new WRQPacket(request, socket, loggedInUser.getUsername());
  loggedInUser = requestByCode(opcode);

} else if (loggedInUser != null && !loggedInUser.isAuthenticated()) {
  loginPacket = new LoginPacket(opcode, message, request, socket);
  loggedInUser = requestByCode(opcode);
}
```

Figure 10 Server.java Server processes operation codes [GitHub Commit]

```
private static User requestByCode(short opcode)
        throws IOException, ClassNotFoundException {

  switch (opcode) {
    case ProtocolCode.LOGIN:
      return loginPacket.processAuthentication();

    case ProtocolCode.LOGOUT:
      return loginPacket.processAuthentication();

    case ProtocolCode.WRQ:
      if (writePacket.isValidRequest()) {
        LOGGER.info(ProtocolCode.WRQ + " Data upload has started");
        writePacket.writeDataOnServer();
      }
      return loggedInUser;

    case ProtocolCode.DATA:
      if (writePacket.isValidRequest()) {
        LOGGER.info(ProtocolCode.DATA + " Data download has started");
        writePacket.writeDataOnClient();
      }
      return loggedInUser;

    default:
      return loggedInUser;
```

Figure 11 Server.java Request is processed by opcode - switch case smell

In the implementation above, there is a clear sign of a switch case smell, ultimately breaking the Open Closed Principle. The server waits for an operation code to be received, and upon valid operation code receival, it calls another utility method i.e. requestByCode. The utility method processes the operation in another place.

Motivation

Client issues operation code to the server. Server is unaware of the type of request being received, but it needs to process every valid request.

After

```
43          operation = new Operation();
44          DataPacket dataPacket = new DataPacket(opcode, message, request, socket);
45
46          if (runCount == 0 ) {
47            auth = new Authentication(dataPacket);
48            runCount++;
49          } else if (auth.getUser().isAuthenticated()) {
50            fileTransfer = new FileTransfer(dataPacket, auth.getUser().getUsername());
51          }
52
53          requestByCode(opcode);
```

Figure 12 Server.java Operation context

I still have to use the requestByCode method and have a switch case in it. As request types are less likely to be changed or updated for an already established protocol, using the command pattern, the code looks a lot simpler and readable.

```
private void requestByCode(short opcode) throws IOException, ClassNotFoundException {

  switch (opcode) {
    case ProtocolCode.LOGIN:
      LoginCommand login = new LoginCommand(auth);
      operation.setOpcode(login);
      operation.operationCodeRequested();
      break;

    case ProtocolCode.LOGOUT:
      LogoutCommand logout = new LogoutCommand(auth);
      operation.setOpcode(logout);
      operation.operationCodeRequested();
      runCount = 0;
      break;

    case ProtocolCode.WRQ:
      FileUploadCommand fileUpload = new FileUploadCommand(fileTransfer);
      operation.setOpcode(fileUpload);
      operation.operationCodeRequested();
      break;

    case ProtocolCode.DATA:
      FileDownloadCommand fileDownload = new FileDownloadCommand(fileTransfer);
      operation.setOpcode(fileDownload);
      operation.operationCodeRequested();
      break;

    default:
      LOGGER.info("Invalid operation code");
  }
}
```

Figure 13 Server.java Command pattern

UML Class Diagrams

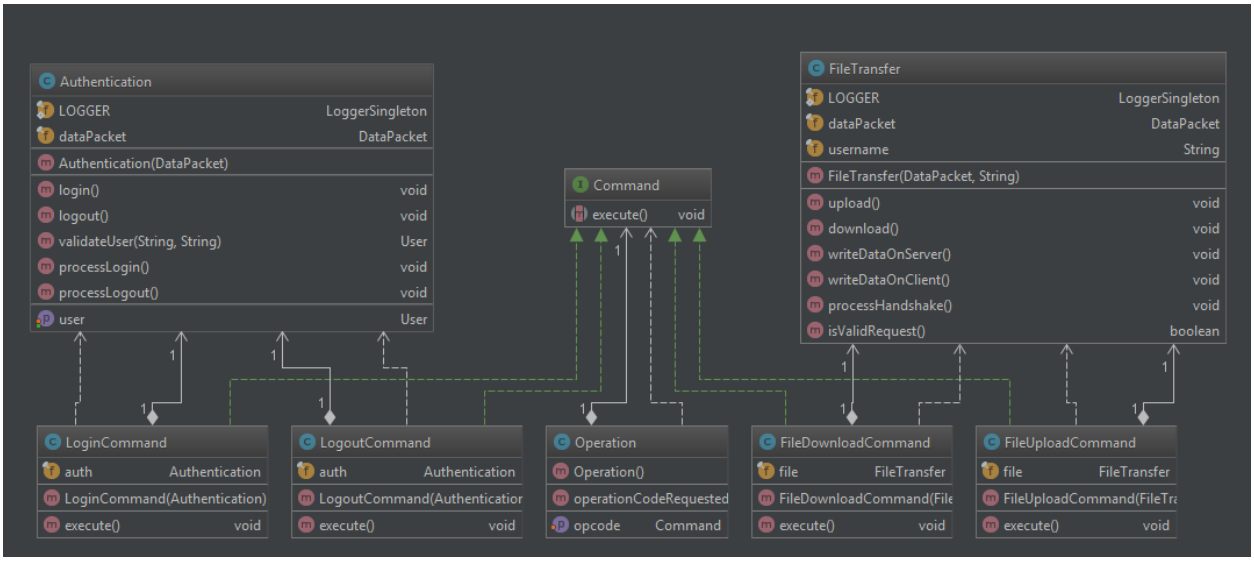


Figure 14 Command pattern to process operations [GitHub Commit]

Design

- Object per command
- Command interface
- execute method

Pros

- Possibly the second most used pattern after Singleton
- Decouple sender from processor
- Very few drawbacks
- Often used for undo functionality
- Allows to create a sequence of command

Cons

- Dependence on other patterns, thus requiring more knowledge of other patterns
- Increases the number of classes for each individual command

Strategy Pattern

Intent

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it (Shvets, n.d.).”

Before

```
} else if (event.getSource() == uploadChooser && auth.getState() instanceof LoggedInState) {
  try {
    String command = event.getActionCommand();
    uploadFile(command);

  } catch (InvalidArgException inval) {
    LOGGER.warn(inval.getMessage());
    logArea.append("Status: " + inval.getMessage() + "\n");
  }

} else if (event.getSource() == downloadChooser && auth.getState() instanceof LoggedInState) {
  try {
    String command = event.getActionCommand();
    downloadFile(command);

  } catch (InvalidArgException inval) {
    LOGGER.warn(inval.getMessage());
    logArea.append("Status: " + inval.getMessage() + "\n");
  }
}
```

Figure 15 UiWindow.java Upload and Download action events [GitHub Commit]

In the above implementation, uploadFile or downloadFile methods are called depending on the action performed. Both methods validate the request, and calls another method to process the file transfer.

```
if (command.equals(JFileChooser.APPROVE_SELECTION)) {
  LOGGER.info("Downloading " + file.getName() + " has started");
  downloadFileFromTheServer();

} else if (command.equals(JFileChooser.CANCEL_SELECTION)) {
  logArea.append("Status: Downloading cancelled\n");
}
```

Figure 16 UiWindow.java Download file

The file transfer for both upload and download requests are implemented within the UiWindow class.

Motivation

Encapsulate the file transfer algorithms in strategies, thus minimizing coupling. As the client is aware of the strategies that are available, it ultimately chooses the appropriate strategy it is going to use.

After

```
if (command.equals(JFileChooser.APPROVE_SELECTION)) {
  LOGGER.info("Uploading " + file.getName() + " has started");
  OperationContext context = new OperationContext(new UploadOperation());
  setOperationContext(context);
  context.executeOperation();

} else if (command.equals(JFileChooser.CANCEL_SELECTION)) {
  logArea.append("Status: Uploading cancelled\n");
}
```

Figure 17 UiWindow.java Upload operation strategy [GitHub Commit]

Now, instead of calling the method, it calls the UploadOperation strategy. In UploadOperation strategy, the upload processing algorithm is encapsulated. The setOperationContext method is called before the upload execution method is called. This is required as data from the UiWindow is needed to process the implementation.

I would imagine there is a better way to pass the values to the strategies.

```
private void setOperationContext(OperationContext context) {
  context.setServerInput(serverInput);
  context.setPortInput(portInput);
  context.setUserInput(userInput);
  context.setPasswordInput(passwordInput);
  context.setUploadChooser(uploadChooser);
  context.setDownloadChooser(downloadChooser);
  context.setLogArea(logArea);
  context.setHelper(helper);
}
```

Figure 18 UiWindow.java Setting operation context values

UML Class Diagrams

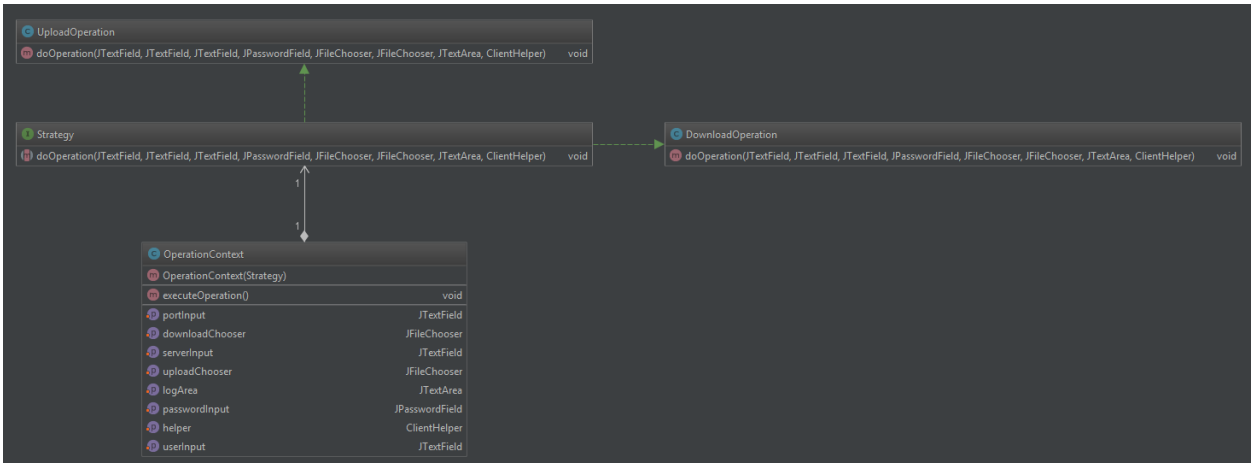


Figure 19 Strategy UML Diagrams

Design

- Interface / Abstract class
- Concrete class per strategy

Pros

- Externalizes algorithms
- Reduces conditional statements
- Algorithms can be reused and loosely coupled with the context entity
- Algorithms can be changed or replaced without changing the Context entity

Cons

- Client must be aware of the strategies
- Increased number of classes in the application

Composite Pattern

Intent

“Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly (Shvets, n.d.).”

Before

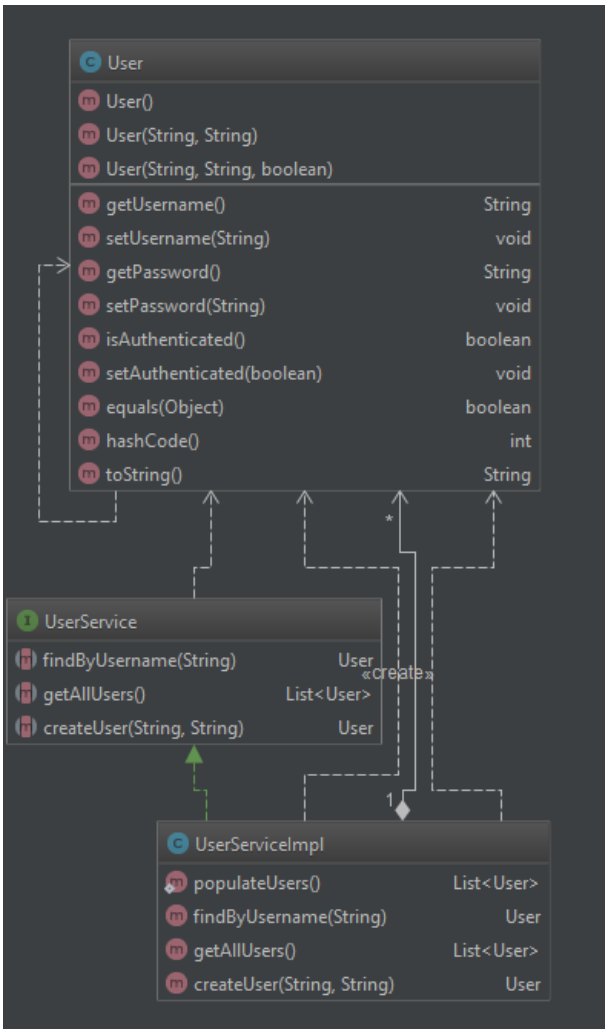


Figure 20 User Service UML Diagram [GitHub Commit]

User service implementation contains a list of registered users. However, it does not have any hierarchy.

Motivation

Organizational hierarchy.

After

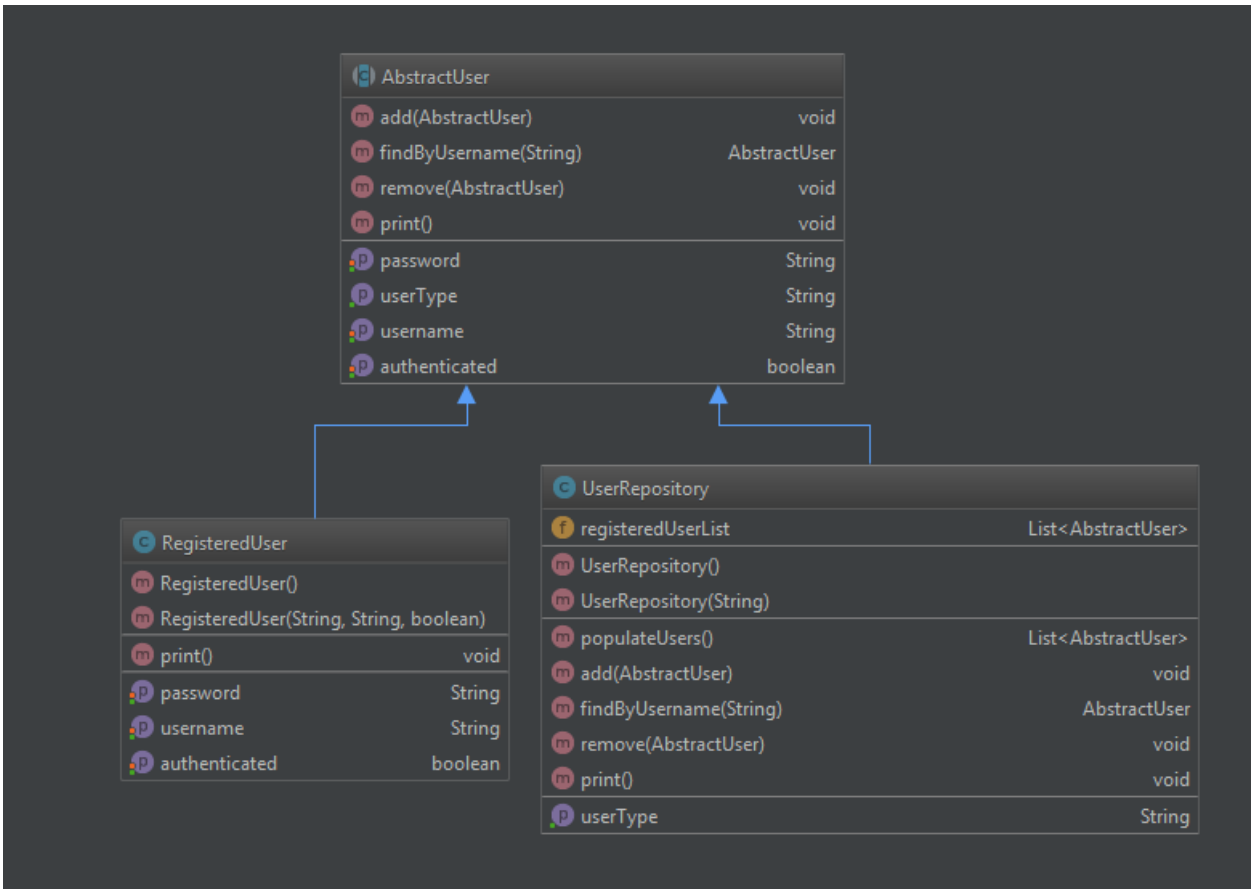


Figure 21 Composite pattern [GitHub Commit]

Design

- Tree structured
- Component
- Leaf or Composite, same operations
- Composite knows about child objects

Pros

- Generalizes a hierarchical structure
- Easier for clients

Cons

- Can overly simplify the system
- Difficult to restrict what we want to add to it
- Implementation can possibly be costly dealing with large composites
- Easy to confuse with composite and composition

End Notes

I have successfully implemented six design patterns. However, if time permitted, I would have implemented the other four patterns I have suggested in “Possible Implementations” section.

`Refactoring to pattern started from` `fcd53a8fc5438b8cf34854dd15bfeca2c3d0e3e8`