BDIC3027J Software System Architecture

File Transmission System Document

Supervised By

Dr. Xiaobin Xu

Developer

Yuchen Wang 19372302

June 19, 2022

1 Introduction

The file transmission system is designed follow the SOLID principles to achieve the file transmission between multiple clients. After the system is started, each client will receive a graphic user interface to help them to manipulate the system. The system supports building and closing connections between every client. Each client is allowed to begin multiple file sending tasks and handle file receiving tasks at same time. For each task, it can contain multiple files and the file is transmitted one by one. The system also support the transmission query, which allows client to query the progress of tasks they sent and the task they received. Task suspend and continue are also supported in the system, both client who sends the task and the client who receives the task can suspend the task. However, the task continue decision should be made by the client who do the suspend. That is closer to the real scenario. The figure below shows the architecture of the system.

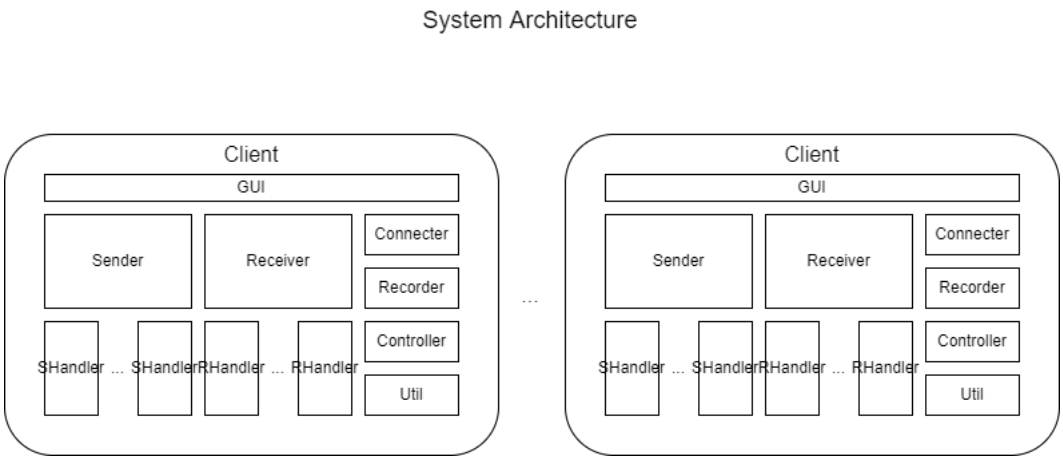


Figure 1: System Architecture

The system is realized in code. It follows most of the SOLID principles in most of parts. However, there are some parts still violate the principle. All of the design will be introduced in the following part with code included for the good design. The system contains 12 classes, 3 interfaces, and 2 abstract classes. The thumbnail below shows them basically.

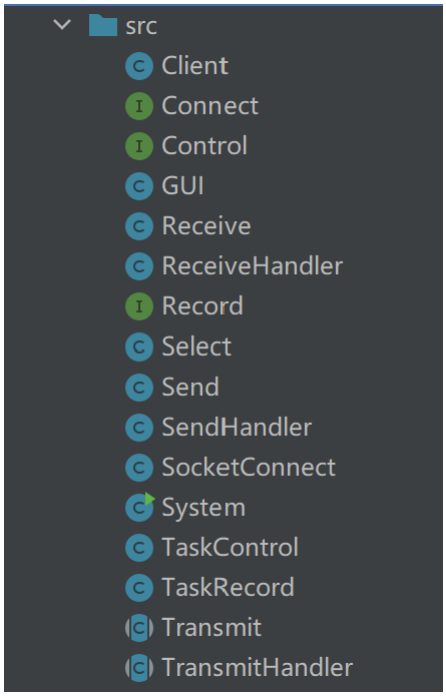


Figure 2: System Structure

The class diagram in the next page shows the relationship between the class and interfaces of the system.

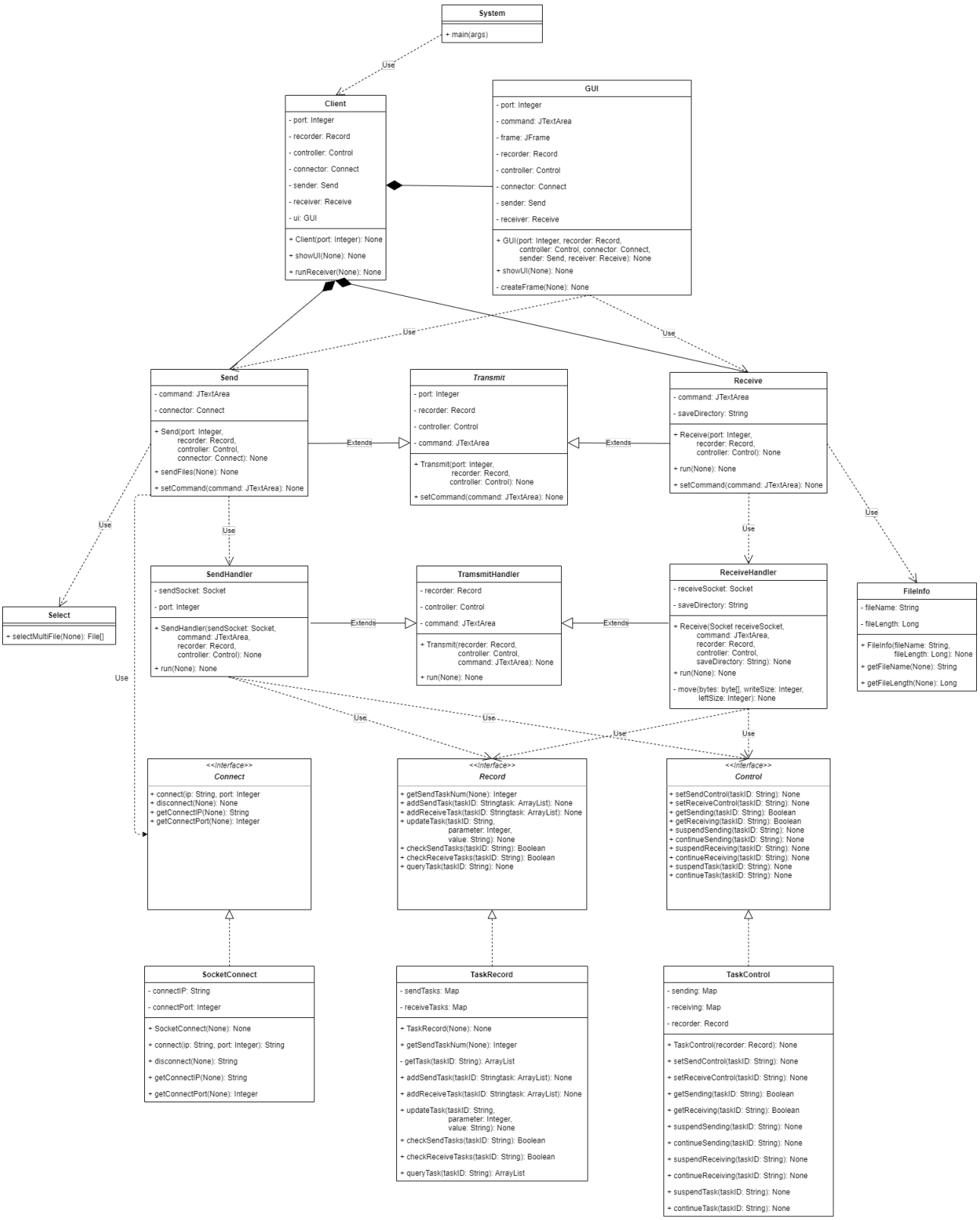


Figure 3: Class Diagram

2 SOLID Principles

This section will give some good design examples of code follow part of the SOLID principles and some bad designs that violate part of the SOLID principles.

2.1 Single Responsibility Principle

- **Good Design**

  The **TaskControl** class is implemented follow the SRP. The responsibility of the class is to control the task suspend and continue for both client who send the task and the client who receive the task. In this class a boolean variable together with the task ID are stored in two different maps according to the kind of task. The responsibility of the variable is to control the task transmission. If it is true, the task is allowed to transmit. The TaskRecord object which relies on the Record Interface is also included to help find the kind of task. The class contains multiple methods from create the boolean variable, get the boolean variable, modify the boolean variable, etc. Other classes can use its methods to create a control boolean variable when a new task is begin, check its value when processing transmission, and modify its value when client choose to suspend or continue the task. The cohesion of this class is high because no other things can be done in the class. Some other classes like **TaskRecord**, **SocketConnect**, and **Select** also obey this principle.

```
1   public class TaskControl implements Control {
2
3       private static Map<String, Boolean> sending;
4       private static Map<String, Boolean> receiving;
5       private Record recorder;
6
7       public TaskControl(Record recorder){
8           sending = new HashMap<>();
9           receiving = new HashMap<>();
10          this.recorder = recorder;
11      }
12
13      public void setSendControl(String taskID){
14          sending.put(taskID, true);
15
16      }
17
18      public void setReceiveControl(String taskID){
19          receiving.put(taskID, true);
20      }
21
22      public Boolean getSending(String taskID){
23          return sending.get(taskID);
24      }
25
26      public Boolean getReceiving(String taskID){
27          return receiving.get(taskID);
28      }
29
30      public void suspendSending(String taskID){
31          sending.put(taskID, false);
32      }
33
34      public void continueSending(String taskID){
35          sending.put(taskID, true);
36      }
37
38      public void suspendReceiving(String taskID){
39          receiving.put(taskID, false);
40      }
41
```

```
42          public void continueReceiving(String taskID){
43              receiving.put(taskID, true);
44          }
45
46          public void suspendTask(String taskID){
47              if (recorder.checkSendTasks(taskID)){
48                  suspendSending(taskID);
49              } else if (recorder.checkReceiveTasks(taskID)){
50                  suspendReceiving(taskID);
51              }
52          }
53
54          public void continueTask(String taskID){
55              if (recorder.checkSendTasks(taskID)){
56                  continueSending(taskID);
57              } else if (recorder.checkReceiveTasks(taskID)){
58                  continueReceiving(taskID);
59              }
60          }
61      }
```

- **Bad Design**

  Due to the implement of GUI, a build-in command line is used for information display, which is created in the GUI class. The GUI class relies on other Interfaces and classes so its instance need to be created after the instance of other classes. It means that the variable **command** can not be passed to the instance of other classes that is created before the instance of GUI like **sender** and **receiver**. At the same time, the handler thread instances started by them need to give out the output during the transmission. One solution is to created a **setCommand** method inside the **Send** and **Receive** class to get the command variable after the create of GUI instance and pass it to the new beginning thread.

2.2 Open-Closed Principle

- **Good Design**

  The **Send** and **Receive** class extend the abstract class **Transmit**. The **SendHandler** and **ReceiveHandler** thread class extend the abstract thread class **TransmitHandler**. The Transmit class encapsulates some necessary variables like port, recorder, and controller to assist building transmission, making record, and control the transmission. The TransmitHandler class encapsulate them as well. By extending these classes, multiple transmission classes can be created to realize different transmit strategy if there are some further needs. In this scenario, the previous classes needn't be changed. That realized the principle of open for extension and close for modification.

```
1       public abstract class Transmit {
2
3           public int port;
4           public Record recorder;
5           public Control controller;
6           public JTextArea command;
7
8           public Transmit(int port, Record recorder, Control controller) {
9               this.port = port;
10              this.recorder = recorder;
11              this.controller = controller;
12          }
13
14          public void setCommand(JTextArea command){
15
16          }
17      }
```

```
public abstract class TransmitHandler extends Thread {

    public Record recorder;
    public Control controller;
    public JTextArea command;

    public TransmitHandler(JTextArea command, Record recorder,
        Control controller){
        this.recorder = recorder;
        this.controller = controller;
        this.command = command;
    }

    public void run(){}
}
```

- **Bad Design**

  If there is no abstract class **Transmit** and **TransmitHandler**, only Send, Receive, SendHandler, ReceiveHandler class is available. When there are some new transmission strategy need to be realized, a new transmission class and a new transmission handler class should be created from the beginning even if the new strategy is similar to the old one. Or the previous classes need to be modified to make them fit for the new requirement. That will increase needless complexity and can bring more work for the developer.

2.3 Liskov Substitution Principle

- **Good Design**

  The Liskov substitution principle emphasizes that the subtypes must be substitutable for their base types. The system is a small file transmission system with each class has its own duty. That means the coupling between classes is low. There are no normal class extends. However, the good design can be realized by extends the **FileInfo** class if there are some further needs. For example, a **VideoInfo** class is needed for more complex task. It need to extend the FileInfo class and add new method needed like **getTheme**. The methods of the FileInfo class are not needed to overwrite or overload because it only focuses on return the variances fileName and fileLength. Then the principle is realized because the subclass can substitute the super class.

```
public class FileInfo {

    private String fileName;
    private Long fileLength;

    public FileInfo(String fileName, Long fileLength){
        this.fileName = fileName;
        this.fileLength = fileLength;
    }

    public String getFileName(){
        return fileName;
    }

    public Long getFileLength(){
        return fileLength;
    }
}
```

- **Bad Design**

  Suppose the **FileInfo** class contains more variables like **transmissionSpeed** and **transmissionPredictTime** and more complex methods like **setTransmissionSpeed**, **predictTransmissionTime**. If the class is extended and the setTransmissionSpeed is overwritten and set the transmissionSpeed variable to zero. There will be problem when using the predictTransmissionTime method after the super class is substituted

by the subclass because the calculation result is infinite. That violate the principle due to the method from super class is edited.

2.4 Interface-Segregation Principle

- **Good Design**

  The **TaskRecord** and **TaskControl** class implement the **Record** and **Control** interface separately. They have different responsibilities. Actually, the instance of TaskControl needs the assistance of the instance of TaskRecord, which shows the dependency between these two class. The two interfaces is created to provide separate and limited responsibilities for classes to help them to achieve high cohesion and reduce the repeated implementation compared with a big and general interface is provided. The dependency of the two classes can be then optimized by considering the DIP.

```
1   public interface Record {
2
3       public int getSendTaskNum();
4
5       public void addSendTask(String taskID, ArrayList<String> task);
6
7       public void addReceiveTask(String taskID, ArrayList<String> task
            );
8
9       public void updateTask(String taskID, int parameter, String
            value);
10
11      public Boolean checkSendTasks(String taskID);
12
13      public Boolean checkReceiveTasks(String taskID);
14
15      public ArrayList<String> queryTask(String taskID);
16  }
```

```
1   public interface Control {
2
3       public void setSendControl(String taskID);
4
5       public void setReceiveControl(String taskID);
6
7       public Boolean getSending(String taskID);
8
9       public Boolean getReceiving(String taskID);
10
11      public void suspendSending(String taskID);
12
13      public void continueSending(String taskID);
14
15      public void suspendReceiving(String taskID);
16
17      public void continueReceiving(String taskID);
18
19      public void suspendTask(String taskID);
20
21      public void continueTask(String taskID);
22
23  }
```

- **Bad Design**

  Compared with using two interfaces to separate the responsibility, a bad solution is to use a general interface which provides combined methods from the two separated interfaces. For example, use **Manage** interface instead of **Record** and **Control** to provide the transmission management functions. In this

scenario, a **TaskManage** class is created to implement the Manage interface, which needs to achieve transmission progress record and transmission progress control. That leads to the low cohesion of the class.

2.5 Dependency-Inversion Principle

- **Good Design**

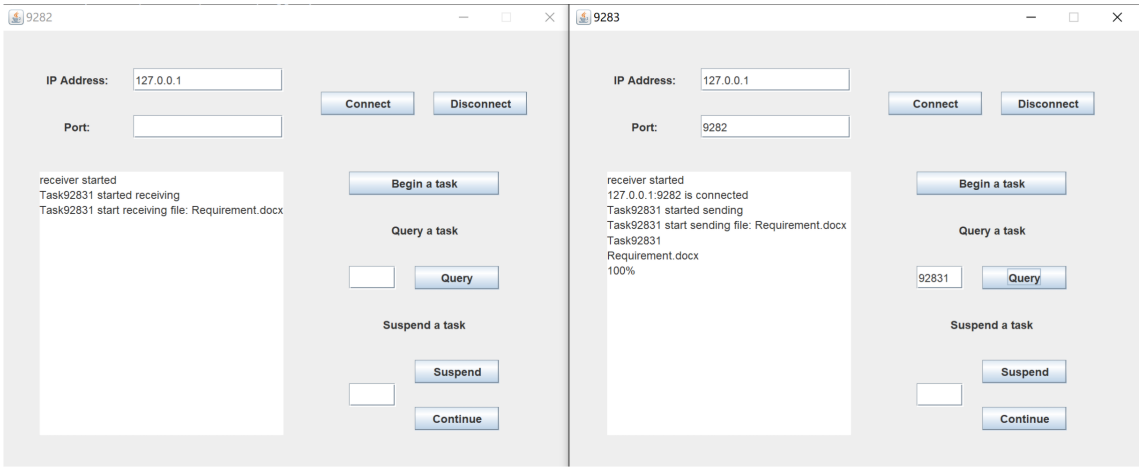
  This GUI needs to listen to client's operation then gives out some responses, which makes it inevitably depends on other classes that provide different service in the system. Instead of using the type of classes that provide the service, the type of the interface implemented by them is used like **Record**, **Control**, and **Connect**. That allows the **GUI** class does not depend on normal classes, but depend on the abstractions, which means the low-level modification will not affect high-level. The figure below shows part of the code of GUI class.

```
1     private int port;
2     private JTextArea command;
3     private JFrame frame;
4     private Record recorder;
5     private Control controller;
6     private Connect connector;
7     private Send sender;
8     private Receive receiver;
9
10    public GUI(int port, Record recorder, Control controller, Connect
          connector, Send sender, Receive receiver){
11        this.port = port;
12        this.recorder = recorder;
13        this.controller = controller;
14        this.connector = connector;
15        this.sender = sender;
16        this.receiver = receiver;
17        this.command = new JTextArea(20, 50);
18        this.sender.setCommand(this.command);
19        this.receiver.setCommand(this.command);
20        createFrame();
21    }
```

- **Bad Design**

  The bad design is very obvious, which is use the type of the class itself instead of the type of the interface it implemented. For example, the **GUI** class uses the instance of **TaskRecord** and **TaskControl** to help handle the input of client. That may face some risks like the change made in low level can affect the high level because the high level is not built on the abstraction. These risks can reduce the system reliability and can add difficulty to the system maintainability.

3 Instruction



1. You can modify the number of client started in the `system` class.
2. After the client started, you need to type the `port` of another client than click `connect` to connect.
3. You can click `disconnect` to disconnect, but it only works after all tasks are finished.
4. Then you can click the `Begin a task` and select the file you want to transmit.
5. After selection, the file will be transmitted and another client will receive it automatically.
6. The received file will be saved to the `/save` directory and you can edit it in the `receive` class.
7. You can type the task ID and click `query` to query the task progress on both side.
8. You can type the task ID and click `suspend` to suspend the transmission and click `continue` to continue the transmission.

PS. The `continue` function sometimes can't work.

The system implementation use the idea from https://blog.csdn.net/fx1ts/article/details/38702603

Figure 4: Instruction