# Ticket to Ride Phase 0.5

Implementation of client/server communication. **This is an individual project.**

## Objectives

1. To learn how to construct a client and server that can communicate with each other using the HTTP protocol.
2. To reinforce your understanding of the command, proxy, and singleton patterns.

## Patterns

1. Command
2. Proxy
3. Singleton

## Requirements

You will create two implementations of this assignment. One without using the command pattern and one using the command pattern. For both implementations, you will create both a client and a server. Each will run in its own process, possibly on different machines. The client contains a '***StringProcessorProxy'*** class with the following three methods:

- `String toLower(String)`
- `String trim(String)`
- `Double parseDouble(String)`

Whenever a method of the ***StringProcessorProxy*** class is called, your program will proxy the method call to the server program which performs the actual logic of the method and returns the result back to the proxy in the client according to the following specification:

1. The `toLower(String)` method returns the input string value but with all upper-case letters converted to lower-case. For example, if the input string was "Go BYU", the resulting string would be "go byu". Your `toLower(String)` method must follow the semantics of the Java `String.toLowerCase()` method.

2. The `trim(String)` method converts the original string to a string where all leading and trailing space has been removed. None of the interior whitespace should be modified. For example, if the input string was "  Go BYU  ", then executing the command would return the string "Go BYU". Your `trim(String)` method must follow the semantics of the Java `String.trim()` method.
3. The `parseDouble(String)` method returns the double value of the input string. For example, if the input string was "-12.82", executing the command would return the value -12.82 as a Double object. Your `parseDouble(String)` method must follow the semantics of the Java `Double.parseDouble(String)` method. This includes the fact that if the input string is not a valid double, a `NumberFormatException` will be thrown. Keep in mind that the server is doing the processing so it will throw the original exception, but the error detected on the server must be propagated back to the ***StringProcessorProxy*** class on the client and thrown from the proxy's `parseDouble(String)` method.

Both implementations (with command pattern and without) should produce exactly the same result. The only difference is in the way the code is designed and implemented. We will describe the "without command pattern" implementation first, followed by the "with command pattern" implementation.

## Without Command Pattern Implementation

The "No Command Pattern" class diagram shows the classes and interfaces to be used for the "without command pattern" implementation. The top half of the diagram shows the client classes and the bottom half shows the server classes. The green classes are used for both the client and server. Lighter colored classes are classes you will use from the JDK. Darker colored classes are classes you will write. We will review the diagram in detail in class.

For this implementation, your program must follow the steps from the "No Command Pattern" client and server sequence diagrams for each method called from the ***StringProcessorProxy*** class. The client diagram shows the sequence of steps for the method calls to be proxied to the server and for the result to be retrieved from the server and returned by the proxy. The server diagram shows the sequence of steps for the server to receive and process the client's request and return the result to the client. See the "Constraints" section of this document for additional implementation requirements.

## With Command Pattern Implementation

The ”Command Pattern” class diagram shows the classes and interfaces to be used for the “with command pattern” implementation. This implementation should produce exactly the same results as the other implementation. However, the use of the Command pattern requires design differences for both the client and server as shown in the Command Pattern class and sequence diagrams. For this implementation, follow the steps from the “Command Pattern” client and server sequence diagrams to proxy the method calls form the client to the server, to process the request on the server, and to return the results.

# Constraints

The following constraints apply to both implementations of the project:

1. Only one instance of the ***ClientCommunicator***, ***StringProcessorProxy***, and ***StringProcessor*** classes is required. These classes must be implemented using the Singleton pattern.
2. The ***ClientCommunicator*** class must use Java’s ***java.net.HttpURLConnection*** class.
3. We will describe how to use the ***com.sun.net.httpserver.HttpServer*** class to implement your ***ServerCommunicator*** class. However, you can use a different server technology if you wish.
4. The client program and server program must be able to run in different processes on different machines. However, to test your programs during development, you should consider running both the client program and server program in separate processes on the same machine to simplify testing.
5. The encoding/decoding shown on the sequence diagrams should be to and from JSON, or to and from XML, or to and from a Java serialized object.

# Help

For JSON serialization, consider using the Gson or Jackson libraries. A jar for Gson can be found on the CS340 Group page at the bottom. If you will be serializing to and from XML, consider using the xstream library.

# Deliverables

You must appear in person and show a TA that your code works.

# Grading Rubric

A link to the grading rubric will be posted here before the due date.