Question 1: Sockets + Strategy pattern

IT-SDJ2-A21

Software Engineering

VIA University College

Jordi Lazo

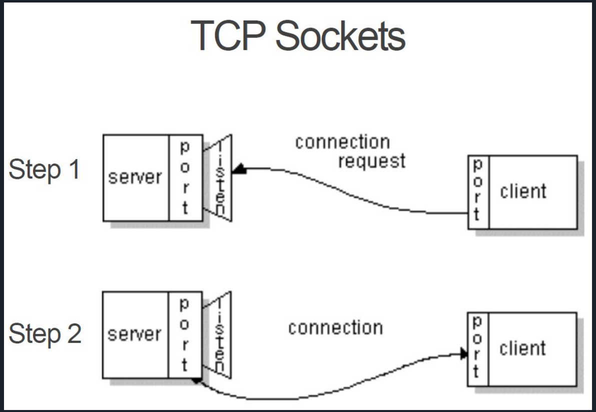
Sockets

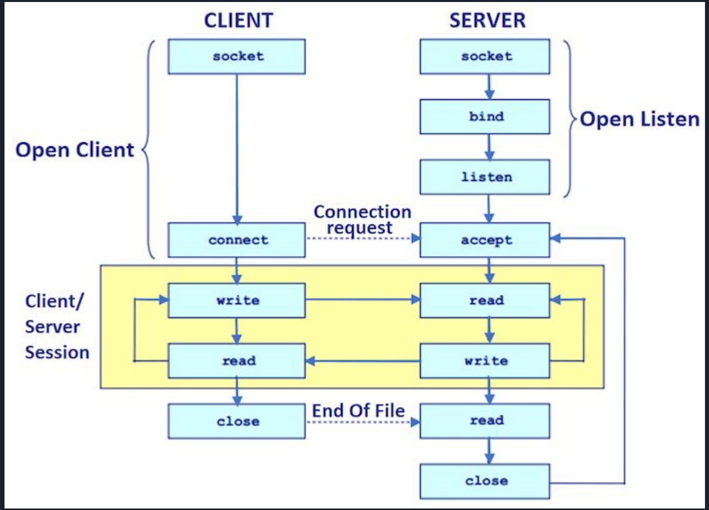
Main parts of a communication between computers using sockets

- ❖ A socket is an endpoint of a communication between two programs running on a network. Socket classes are used to create a connection between a client program and a server program. The Socket represents the client socket, and the ServerSocket the server socket.
- ❖ Client → Send a request
- ❖ Server → Respond a request
- ❖ *Socket* and *ServerSocket* are used for the TCP protocol (Stream Socket). The *DatagramSocket* is used for the UDP protocol (Datagram Socket).

Socket connection

- ❖ The client has to know two things about the server:
  - ➢ The server's IP address.
  - ➢ The port number.



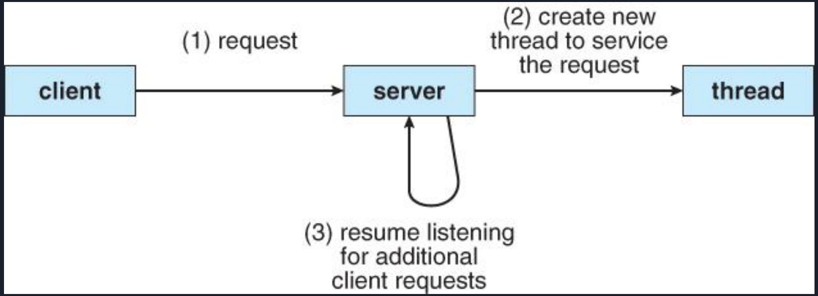


Multithreading

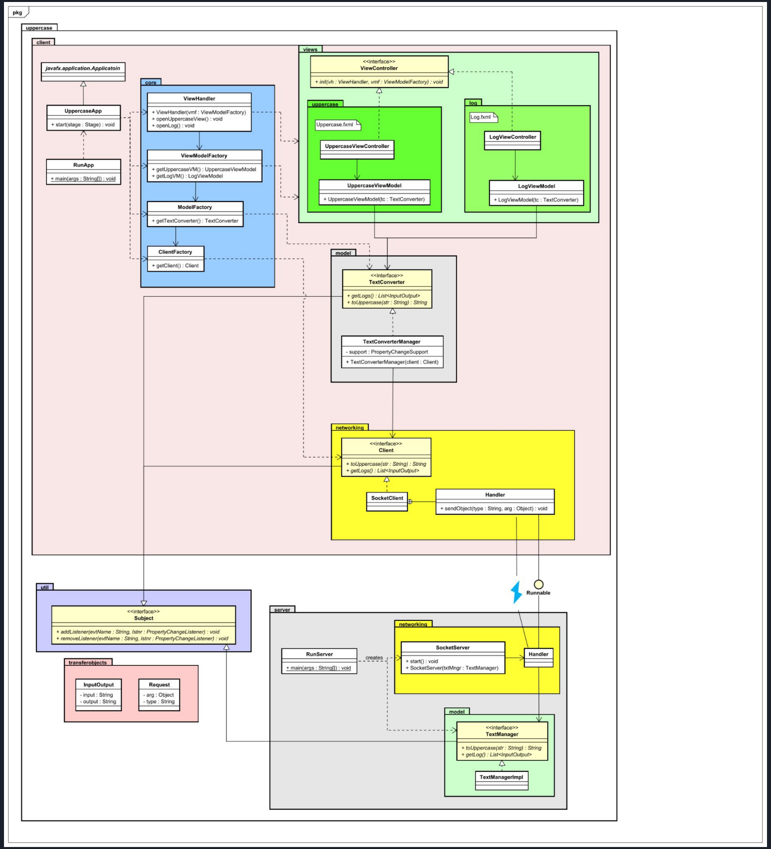
- ❖ It allows simultaneous connections.
- ❖ For each client connection, the server starts a child thread to process the request independent of any other incoming requests.
- ❖ *Client* class extends Thread.
- ❖ Server can take multiple client requests and start the processing. So individual threads will be started and they will work in parallel.

Multithreading

- ❖ Why is multithreading needed on the server side?
    - ➢ In a single threaded server requests may make the server unresponsive for a long period.
    - ➢ Support multiple connections concurrently.
    - ➢ Scalability.
- ❖ Why is multithreading needed on the client side?
    - ➢ Allows different users to connect to the same server.
    - ➢ Clients do not overlap information sent to the server.


client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

UML + Java example


pkg
uppercase
client
javafx.application.Applicatoin
UppercaseApp
+ start(stage : Stage) : void
RunApp
+ main(args : String[]) : void
core
ViewHandler
+ ViewHandler(vmf : ViewModelFactory)
+ openUppercaseView() : void
+ openLog() : void
ViewModelFactory
+ getUppercaseVM() : UppercaseViewModel
+ getLogVM() : LogViewModel
ModelFactory
+ getTextConverter() : TextConverter
ClientFactory
+ getClient() : Client
views
<<interface>>
ViewController
+ init(vh : ViewHandler, vmf : ViewModelFactory) : void
uppercase
Uppercase.fxml
UppercaseViewController
UppercaseViewModel
+ UppercaseViewModel(tc : TextConverter)
log
Log.fxml
LogViewController
LogViewModel
+ LogViewModel(tc : TextConverter)
model
<<interface>>
TextConverter
+ getLogs() : List<InputOutput>
+ toUppercase(str : String) : String
TextConverterManager
- support : PropertyChangeSupport
+ TextConverterManager(client : Client)
networking
<<interface>>
Client
+ toUppercase(str : String) : String
+ getLogs() : List<InputOutput>
SocketClient
Handler
+ sendObject(type : String, arg : Object) : void
Runnable
util
<<interface>>
Subject
+ addListener(evtName : String, lstnr : PropertyChangeListener) : void
+ removeListener(evtName : String, lstnr : PropertyChangeListener) : void
transferobjects
InputOutput
- input : String
- output : String
Request
- arg : Object
- type : String
server
RunServer
+ main(args : String[]) : void
creates
networking
SocketServer
+ start() : void
+ SocketServer(txtMngr : TextManager)
Handler
model
<<interface>>
TextManager
+ toUppercase(str : String) : String
+ getLog() : List<InputOutput>
TextManagerImpl

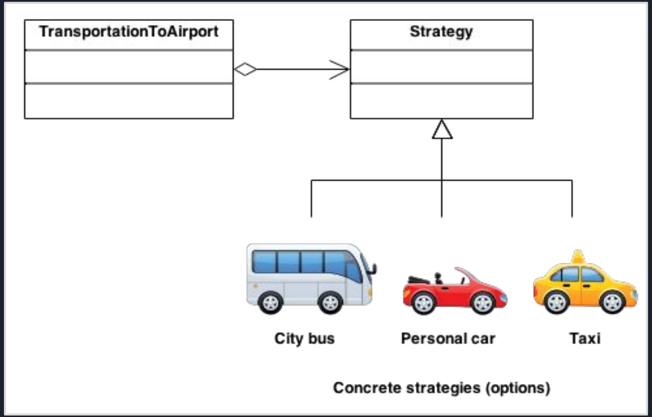
Strategy pattern

What is the purpose?

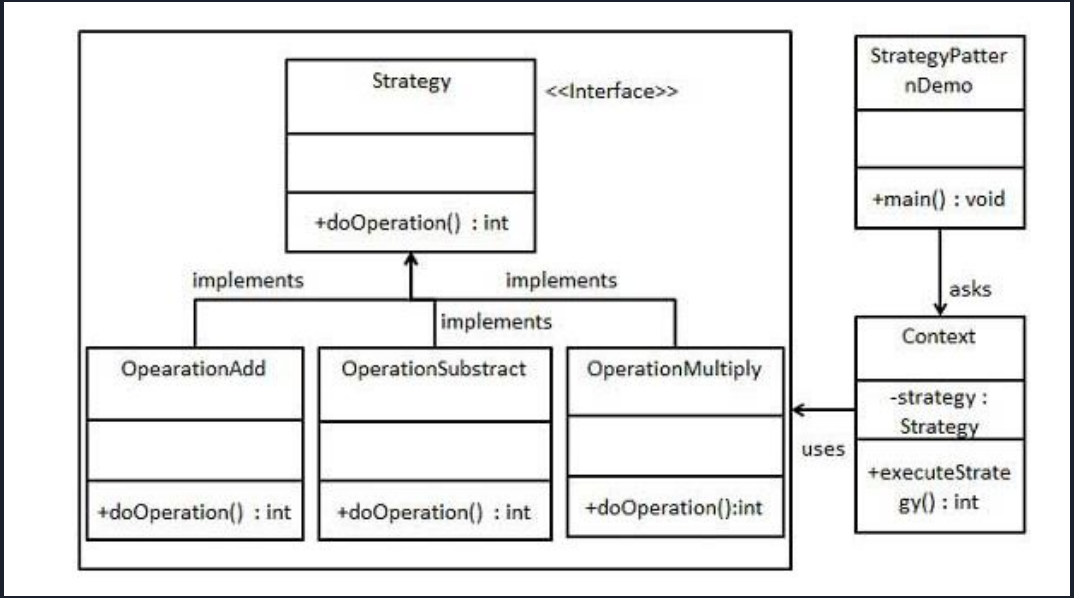
- ❖ Strategy pattern is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- ❖ It can be changed at run time.
- ❖ It isolate the code, internal data, and dependencies of various algorithms from the rest of the code.
- ❖ The original object delegates execution to one of these objects, instead of implementing all variants of the algorithm.

What are the different parts involved? How do they interact?

- *Context* class delegates the work to a linked strategy object instead of executing it on its own. (TransportationToAirport)
- *Strategy* interface have the common methods to all variants of the algorithm. (Strategy)
- Extract all algorithms into their own classes. (CIty bus, Personal car, Taxi)



UML + Java example


Strategy
<<Interface>>
+doOperation() : int
implements
implements
implements
OpearationAdd
+doOperation() : int
OperationSubstract
+doOperation() : int
OperationMultiply
+doOperation():int
StrategyPatternDemo
+main() : void
asks
Context
-strategy : Strategy
+executeStrategy() : int
uses