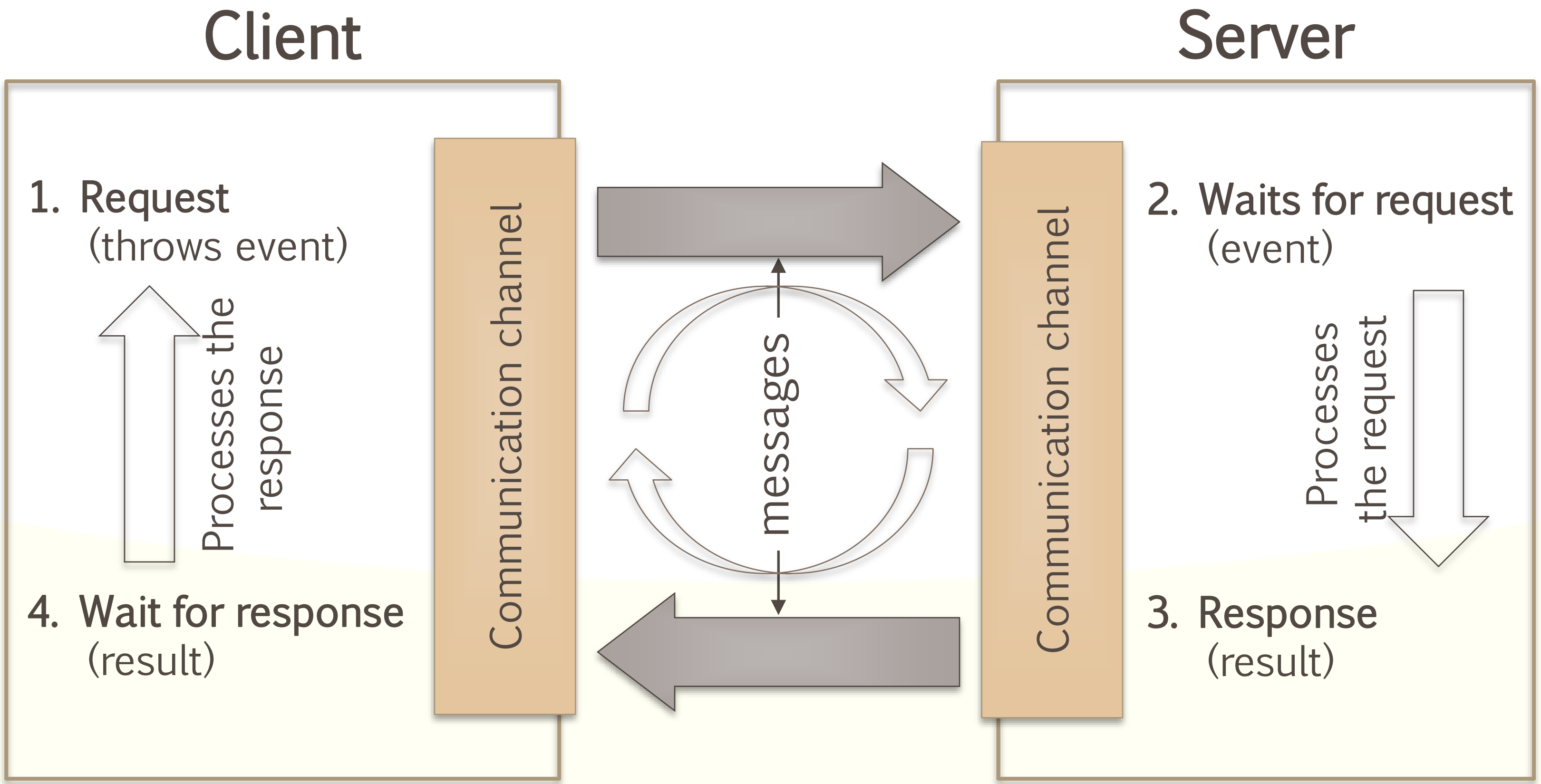
EVENT DRIVEN PROGRAMMING

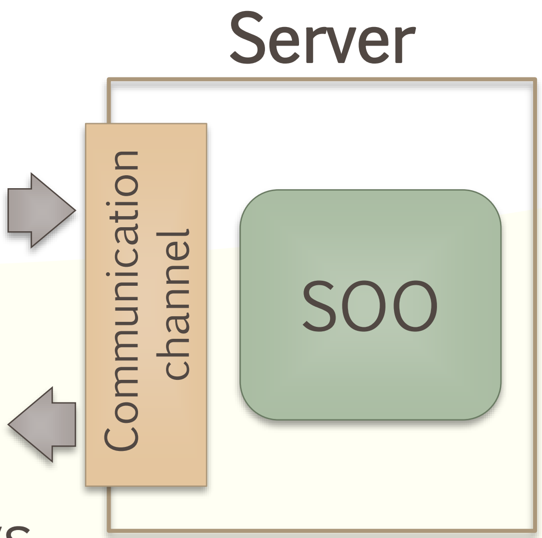
Lab sessions

Client – server app



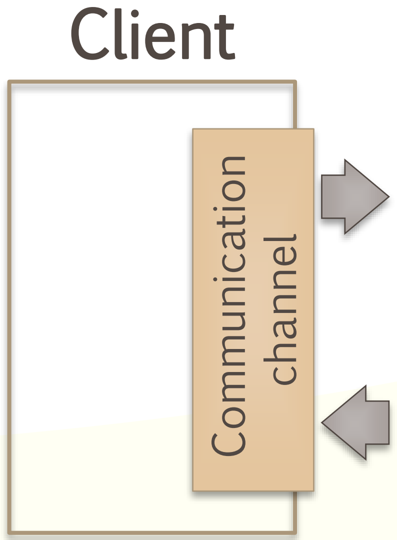
Server

- Program that offers a service to one or more clients.
  - The service itself is the *functionality* offered by the server, which can be encapsuled within an object. In that case, the object is referred to as **service operations object (SOO)**.
- It establishes a **communication channel** to receive service requests from the client and send responses to it
- It creates the **service operations object**
- It is continuously executing, always waiting for a **client** to connect



Client

- A program which connects to a **service** offered by a **server**
- Establishes the **communication channel** to send requests to the server and receive responses from it
- It interacts with the **SOO** through the network, by exchanging messages
  - The functionality specification of the service is needed: the **service interface**
- Its execution is not continuous. Once it receives the response, it is disconnected from the server.



Event – based systems

- The **clients** make calls to the service operations (throw events and send them to the server)
- The **server** executes those service operations received from the clients (equivalent to the event dealer)
- The methods from the SOO are the event controllers
- Library `EDP_library.jar` (Virtual Campus)
  - It implements client – server communication
  - You are going to need it for the lab sessions and the exam

Example: A service that greets the client

- Download the library from the VC and the zip file of this session

File	Description
`commun/IGreeter.java`	Service interface
`commun/ProhibitedActions.java`	Implements prohibited actions
`server/GreeterSOO.java`	Implements the service interface (only `greet()`)
`server/Server.java`	Server program
`client/Client1.java`	Client program (first version)
`client/Client2.java`	Client program (second version)

Example: A service that greets the client

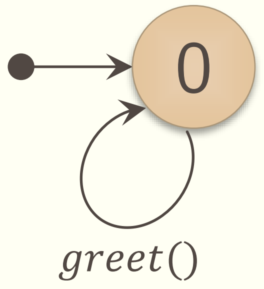
- In Eclipse, create a new Java project named $\mathrm{edp}-01$
- Copy the directories commun, server and client to src
  - They are going to be used as *packages*

Example: How to configure the edp–01 project

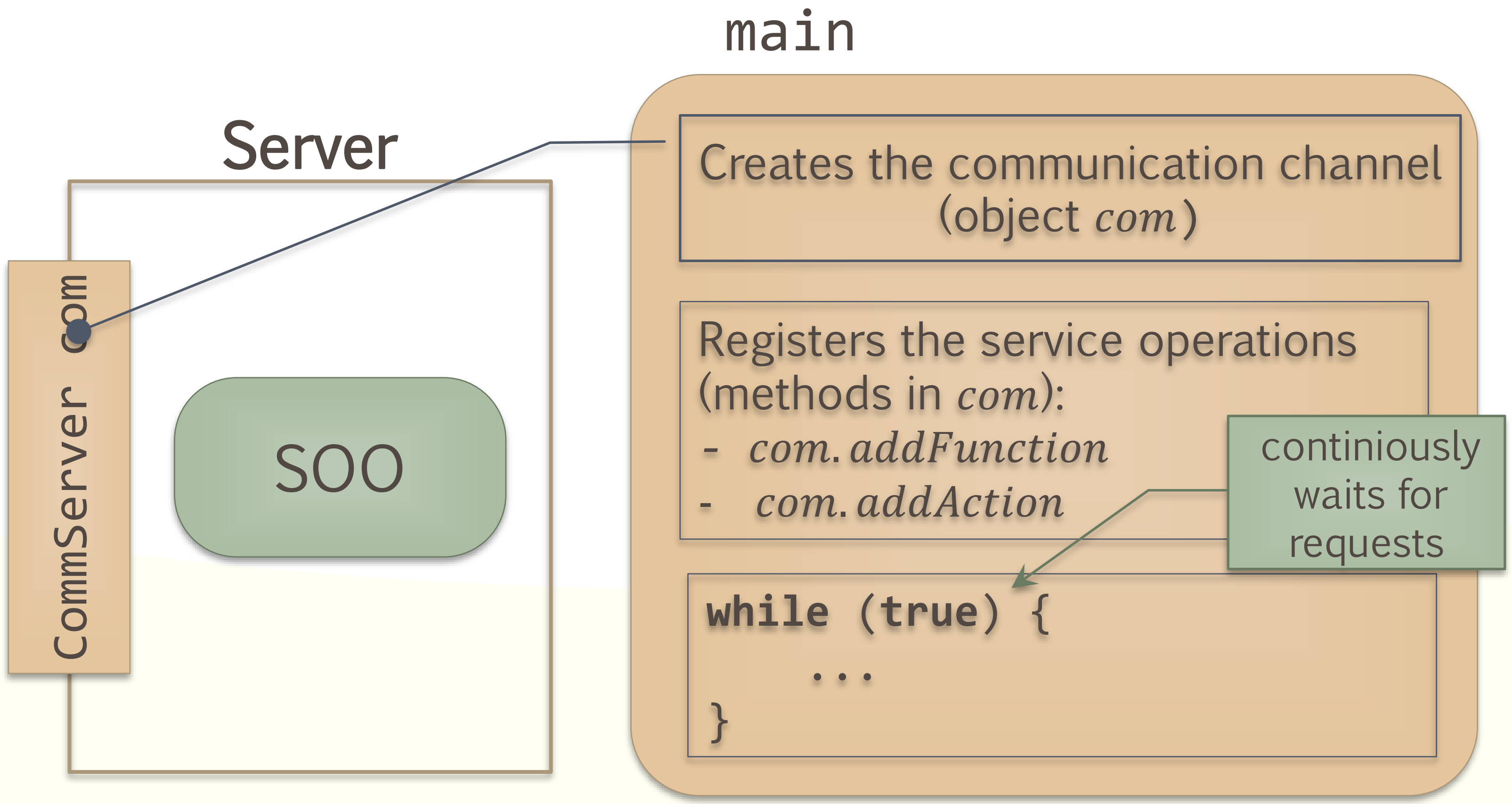
- Indicate that the project uses EDP_*library*.*jar*:
  - On the menu of the project (right click on its name), select the option `Build Path` and `Configure Build Path`...
  - On the dialog box, select the `Libraries` window and use the `Add External JARs` button to select the corresponding jar file. Finally, `Apply and Close`.
- There should not be any errors within the project now.

Example: service interface and implementation

- Every service interface should extend the interface $lib.DefaultService$ from the provided library.
  - It implements a default version for the method $close()$, allowing to properly close the SOO.
  - In some cases, it may be needed to modify this implementation.
- The service only implements the operation $greet()$.
  - The service has only got one state (the initial one), where the event $greet()$ can be triggered several times.
    - Data state of the SOO: the $String\ str$
    - In this case, the control state is unnecessary (as there is only one state), but it has also been included: $int\ state$



Server program



Creates the communication channel (object *com*)

Registers the service operations (methods in *com*):
- *com.addFunction*
- *com.addAction*

```
while (true) {
    ...
}
```

Server algorithm

1. Wait for a client request and get its identifier (*idClient*). Blocking operation.
2. Create the SOO object for the client *idClient* (one per client).
3. Exchange messages with the client: the request-response cycle.
4. When finishing communication and execution, close the SOO of the client

indefinitely

while **(true)**

Library `EDP_library`: Message exchange client ⇔ server

- Both the messages and the communication channels are objects.
- The **messages** are instances of the class `ProtocolMessages`
- The **communication channel** in the **client** is an instance of the class `CommClient`
- The **communication channel** in the **server** is an instance of the class `CommServer`
  - There is should be one instance per **service**
  - Frequently, the server offers just one service, but this is not mandatory as it may offer more

Library `EDP_library.jar`: Class ProtocolMessages

- The messages for the server (requests) are composed of:
  - An **identifier**: a `String` containing the name of the requested service operation, or the code for a disconnection request.
  - An **`Object array`**: holding the necessary arguments (the data) needed to execute the service operation.
- The messages for the client (responses) are composed of:
  - An **identifier**: a `String` to indicate that the service operation has been properly executed (OK), or that an exception has been triggered.
  - An **`Object array`**: if the event produces some result (or an exception), it will be sent within the first element of the array.

Library `EDP_library.jar`: Class ProtocolMessages

- `ProtocolMessages` is `Serializable`
  - The message exchange is carried out through the Internet network as byte sequences

- Methods:
  - `ProtocolMessages(String id, Object... args)`
    - Used for creating messages (as in the previous slide)
  - `String getID()`
    - Returns the message identifier
  - `Object[] getArgs()`
    - Returns the arguments of the message
  - `String toString()`
    - Returns the `String`: `ID(arg0, arg1,...)`, where `ID` is the value returned by `getID()` and `arg0`, `arg1`,`...` are the values returned by `getArgs()`

Service operations interface

- The interface specifies the provided functionalities
  - *Igreeter*
- Interface implementation:
  - Class *GreeterSOO*
    - Area of data
      - Control state: an integer (which is not used in this example).
      - Data state (system informatiob)
        - A *String*, the greeting message
    - The SOO is an instance of this class

Library `EDP_library.jar`: Class CommServer

- An instance of this class establishes the communication channel of the server and offers the service to the clients (through the SOO)

- **Methods**
  - `CommServer()`: creates the communication channel
  - `processEvent(id, obj, msg)`: process in the server the message `msg` received from the client `id` for the SOO `obj`
  - *`addFunction(idOp, f)`* y *`addAction(idOp, f)`:* register a service operation (function or action, respectively), associating its identifier (the function/action name) with a lambda function

Library `EDP_library.jar`: Class CommServer

- Each operation should be registered with its identifier an its evaluation procedure
- Example:

```
The method of the SOO:

    ResultType op(T0 arg0, T1 arg1, ...)

Can be registered as:

    addFunction("op", (o, x) -> o.op(x[0], x[1], ...))
```

  - Just remember that *castings* may be needed as no datatype are specified in the lambda definition

```
    addFunction("op", (o, x) -> (GreeterSOO o).op(x[0],
x[1], ...))
```

Library `EDP_library.jar`: Class CommServer

- `CommServer(idService)`: creates a communication channel for the specified service
- `waitForClient()`: **waits** until the connection of a client, returning its idenfiier (blocking operation)
- `closed(id)`: returns whether the client with identifier **id** has disconnected
- `waitEvent(id)`: the server waits for a message (an event) from the cliente **id**. Returns the received message.
- `sendReply(id, msg)`: sends the response message `msg` to the client `id`
- `activateMessageLog()`: activated the service messages log. By default, it is deactivated; this is an optional action.

Library `EDP_library.jar`: Class Trace

- It is available in the *package* Optional
- It has got class methods (`static`) which allow to track the different actions performed in the server.
  - The trace is directly shown in the standard output
  - By default, tracking is disabled.
- Method `void activateTrace(CommServer com)`
  - Activates tracking in the server.

Activating the action tracking is recommended (after creating its communication channel), because the library includes several tracking protocols that will inform you about the various actions being executed in the server.

Library `EDP_library.jar`: Class CommClient

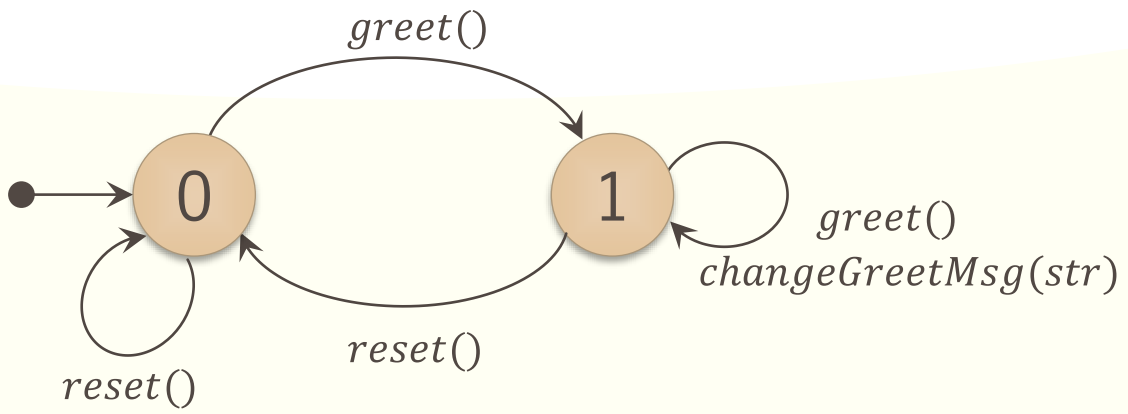
- It allows to establish communication fromt he client to a service in a server
- **Methods:**
  - `CommClient()`: creates a connection channel with the default service address in the local machine
  - `CommClient(server, idService)`: creates a connection channel with the provided server and service
  - `sendEvent(msg)`: sends a messages through the communication channel
  - `waitReply()`: waits for a response from the server, through the communication channel (blocking operation)
  - `processReply(msg)`: process the response received from the server, returning the actual result (it could be an exception)

Library `EDP_library.jar`: Class CommClient

- Methods:
  - `disconnect()`: disconnect the client from the server.
  - `activateMessageLog()`: activate the message log in the client.

Exercise: developing the Greeter server

- Introduce the necesary changes for the server service to offer also the following optional operations: $changeGreetMsg(str)$ and $reset()$.
- The evento $changeGreeterMsg(str)$ can only be thrown if the event $greet\ ()$ has been previously thrown. The state graph should be the following one:



Exercise: develope a Client program

- Throw 3 times the event $greet()$ and show the received message in the console.
  - Activate tracing in the server and execute it
  - On another Eclipse console, execute the client
- Add two functions in the client to throw the events $changeGreetMsg(str)$ and $reset()$. Process the responses (if there are any).
- From the client's main, throw the following events, in this order and show the responses in the console:
  - $changeGreetMsg(\text{“Changing the message”})$, $greet()$, $greet()$, $changeGreetMsg(\text{“Another message change”})$, $greet()$, $reset()$, $greet()$
  - Before executing, what should the last greeting message be?