

Software de comunicaciones
Curso 2025-26

Práctica 1 Distribución usando RPCs



CONTENIDO

1	Objetivos de la práctica	3
2	Entrega de resultados	3
3	Recursos del alumno para la realización de la práctica	3
4	Creación del proyecto	3
5	La calculadora	3
6	El patrón de diseño <i>Adapter</i>	5
7	Fase 1	5
8	Fase 2	9
9	Preguntas a responder	20

1 - Objetivos de la práctica

- Entender los fundamentos de la distribución de aplicaciones utilizando middleware.
- Desarrollar una aplicación distribuida utilizando invocación remota de procesos de servicio (RPC) usando Google Remote Procedure Call (gRPC).

2 - Entrega de resultados

Se debe entregar un fichero *zip* en la tarea correspondiente de Moodle con los siguientes contenidos:

1. Carpeta completa con el proyecto Maven creado en la realización de la práctica. La carpeta entregada debe contener todos los ficheros fuente y los creados por las herramientas.
2. Archivo de texto con las respuestas a todas las preguntas numeradas realizadas en este enunciado.

La entrega en Moodle debe realizarse ANTES del XXX.

Los resultados entregados para esta práctica podrán ser necesarios para la realización del examen práctico del primer parcial de la asignatura.

3 - Recursos del alumno para la realización de la práctica

El alumno descargará desde la plataforma Moodle de la asignatura el archivo *softcom_p1.zip*, con los siguientes recursos:

- Este documento en formato *pdf*.
- Carpeta con el *javadoc* de la aplicación *CalculadoraGUI*.
- Fichero *pom.xml* del proyecto.
- Fichero *P1_Calculadora-1.0.jar* con la aplicación completa compilada para que el alumno pueda probarla y comprender su funcionamiento.

4 - Creación del proyecto

Crea un proyecto Maven en Eclipse con los siguientes parámetros:

```
Group Id    : es.upm.dte.softcom
Artifact Id: softcom_p1
Version     : 1.0
Packaging   : jar
```

Una vez creado el proyecto, sustituye su fichero *pom.xml* por el que se adjunta con los recursos de la práctica.

Ahora, abre un terminal en el directorio del proyecto y ejecuta *mvn clean install* o, desde eclipse, haz *Run as ->Maven install*.

Con esto, la estructura de directorios del proyecto estará creada y lista para seguir con el desarrollo de la práctica.

5 - La calculadora

Se dispone de una aplicación que implementa una calculadora con memoria, cuyo diagrama UML es el siguiente:

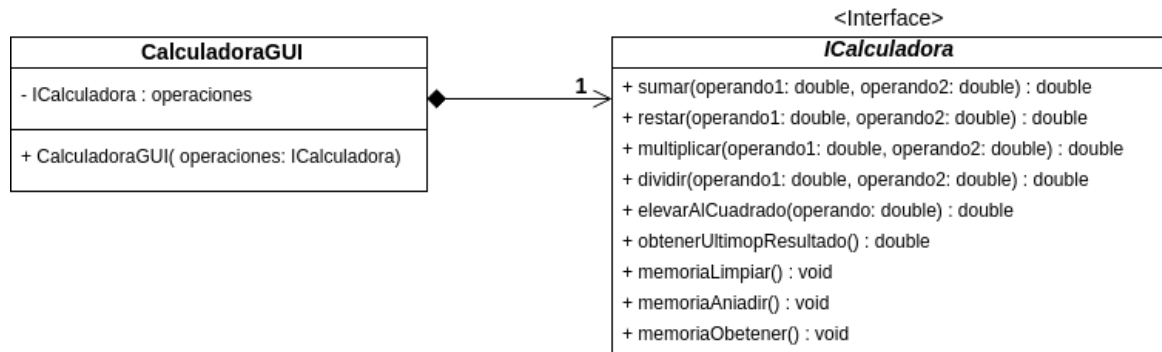


Figura 1: Diagrama UML de la clase *CalculadoraGUI* y el interface *ICalculadora*

La clase *CalculadoraGUI* dispone de una interfaz gráfica de usuario (Graphical User Interface, GUI), que se muestra en la Figura 2.



Figura 2: Interface gráfioco de la aplicación *Calculadora*

La aplicación calculadora dispone de teclas para realizar las cuatro operaciones aritméticas básicas, elevar un número al cuadrado, capacidad de mostrar el último resultado obtenido, operar con una memoria acumuladora y funciones programables. En esta práctica no se hará uso de las teclas para funciones programables.

La calculadora dispone de dos áreas diferenciadas: la de presentación de resultados en la parte superior y los botones en la parte inferior. El área de representación dispone de dos líneas: una para visualizar los resultados de las operaciones y los contenidos de las memorias y una segunda línea para la visualización de los números que se componen. Los botones, por su parte, son de tres clases: los que permiten componer un número, los operadores aritméticos y los de manipulación de las memorias. Las dos primeras clases permiten una operativa evidente, así que no se presenta su funcionalidad. En cuanto a los botones de manipulación de memorias, esta es su funcionalidad:

- **Último resultado (UR):** Permite visualizar el resultado de la última operación aritmética realizada.
- **Memoria-Añadir (MA):** Permite añadir a la memoria acumuladora el resultado de la última operación aritmética realizada.
- **Memoria-Obtener (MO):** Permite visualizar el contenido de la memoria acumuladora.
- **Memoria-Limpiar (ML):** Permite poner a cero el contenido de la memoria acumuladora.

El constructor de la clase *CalculadoraGUI* debe recibir como argumento una instancia de una clase que implemente el interfaz *ICalculadora*, que será la encargada de realizar la aritmética de las operaciones.

Para poder utilizar en la aplicación la clase *CalculadoraGUI* y el interface *ICalculadora*, no tienes

que hacer nada especial, ya están disponibles a través de la siguiente dependencia del fichero *pom.xml*:

```
<dependency>
  <groupId>es.upm.dte.softcom.CalculadoraGUI</groupId>
  <artifactId>calculadora-gui</artifactId>
  <version>5.0.0</version>
</dependency>
```

Asegúrate de leer el *javadoc* que se incluye entre los contenidos que se entregan con la práctica, para comprender perfectamente su funcionamiento.

Esta práctica se debe realizar siguiendo en secuencia cada una de sus fases. En la primera fase, el alumno debe implementar las clases que permitan ofrecer la funcionalidad completa de la aplicación. En la Fase 2, se realizará una distribución de la aplicación, de tal manera que la capacidad de realizar las operaciones, así como la gestión de las memorias de la calculadora, pasarán a ser hospedadas en un servidor específico.

6 - El patrón de diseño *Adapter*

Imagina una situación como la de la Figura 3. La salida de sonido que ofrece el tocadiscos no coincide con la toma de la que dispone el amplificador.



Figura 3: El cable con la salida de audio del tocadiscos no coincide con la toma de entrada de audio disponible en el amplificador. La solución: un adaptador

Una posible solución sería abrir el amplificador y añadirle una toma que coincida con la del tocadiscos. Esta solución es complicada, en algunos casos incluso podría no ser posible. Además, implica muchos riesgos de que rompamos algo en el amplificador.

Otra posible solución sería actuar en el tocadiscos, cambiando la salida de audio. Sucede igual que en el caso anterior, puede ser difícil o imposible, entraña riesgos y podría impedir la utilización del toadiscos con otros amplificadores.

La mejor solución suele ser intercalar un adaptador entre tocadiscos y amplificador, como se muestra en la figura.

En desarrollo de software, sucede frecuentemente una situación asimilable a la del ejemplo anterior. En esta práctica se va a hacer uso del patrón de diseño *Adapter*.

7 - Fase 1

Se dispone de la clase *OperadorAritmetico*, cuyo esquema UML es el de la Figura 4. Observa que la clase *OperadorAritmetico* resuelve todas las operaciones que necesitamos para usar la calculadora, pero no implementa el interfaz *ICalculadora*, por lo que no podemos utilizar directamente una instancia de esta clase en el constructor de *CalculadoraGUI*.

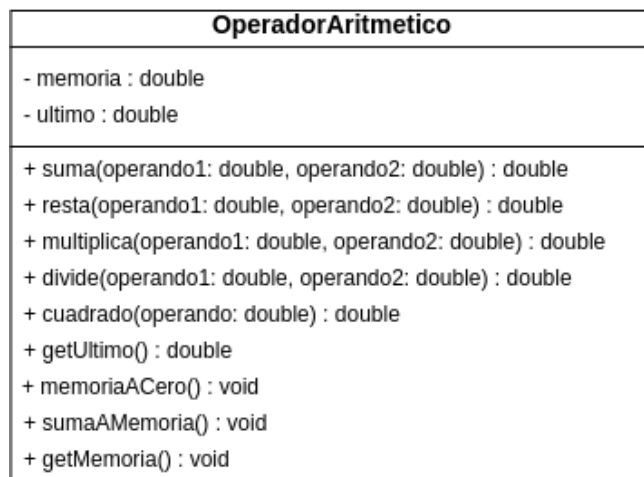


Figura 4: Esquema UML de la clase *OperadorAritmetico*

Queremos utilizar la clase *OperadorAritmetico* para realizar las operaciones de la calculadora. El código de *CalculadoraGUI* no lo podemos modificar, pues ni siquiera disponemos de él. Modificar el código de la clase *OperadorAritmetico* podría generar problemas, pues pudiera suceder que esa clase se estuviera utilizando en otros programas. La solución es desarrollar un adaptador que permita conectar las clases *CalculadoraGUI* y *OperadorAritmetico*, sin necesidad de modificar el código de ninguna de ellas.

La clase que va a hacer de adaptador la hemos llamado *AdaptadorCalculadora*. Esta clase implementa el interfaz *ICalculadora* y, al mismo tiempo, tiene un atributo que es una instancia de *OperadorAritmetico*, que le permitirá utilizar la implementación de las operaciones. El esquema sería el de la Figura 5.

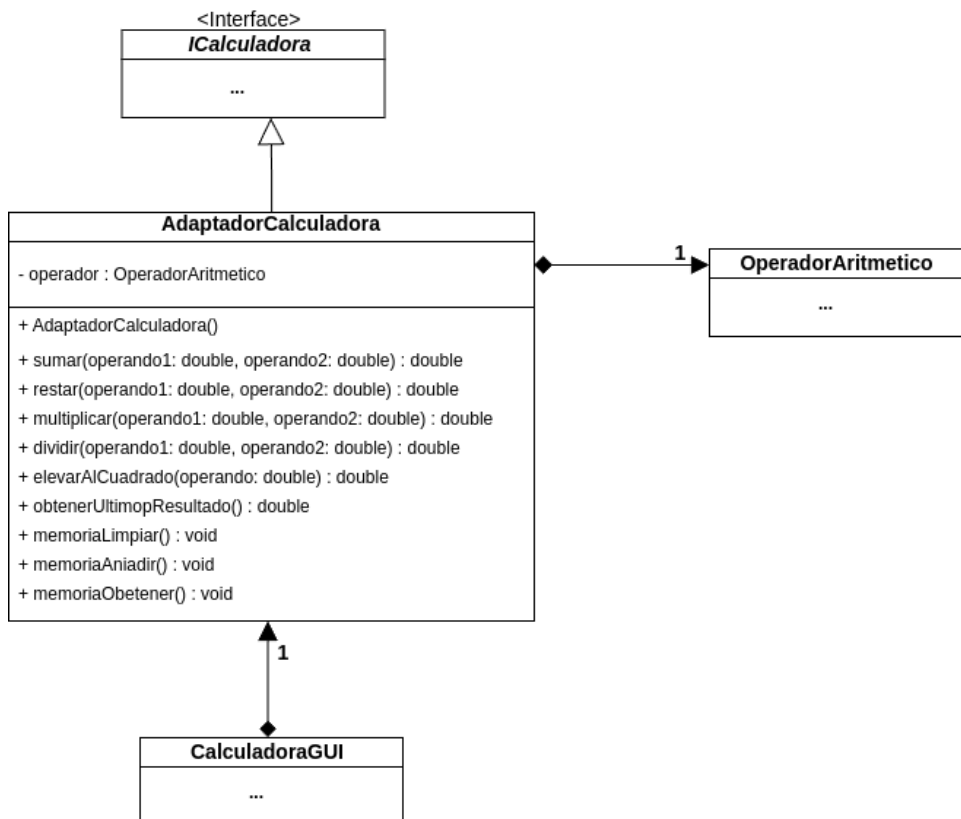


Figura 5: Esquema UML de la clase *AdaptadorCalculadora*: implementa el interfaz *ICalculadora* y tiene un atributo del tipo *OperadorAritmetico*.

Por ejemplo, el método *sumar()* de *AdaptadorCalculadora()* podría tener una codificación similar a la siguiente:

```
@Override
public double sumar(double operando1, double operando2) {
    double result = operador.suma(operando1, operando2);
    return result;
}
```

Como ves, el método implementa el interfaz *ICalculadora*, aprovechando la operación aritmética *suma()*, que ya está implementada en *OperadorAritmetico*. Ahora, al crear el objeto *CalculadoraGUI*, le pasaremos como argumento una instancia de *AdaptadorCalculadora*. De esta forma, aprovecharemos que las operaciones aritméticas ya están implementadas en *OperadorAritmetico* y las utilizaremos en nuestra calculadora a través de *AdaptadorCalculadora*.

Reproducimos a continuación el código de las clases *OperadorAritmetico* y *AdaptadorCalculadora*. El código no está completo. Como parte del trabajo de esta Fase 1, tendrás que completar la codificación de estas clases e incorporarlas al proyecto. La clase *OperadorAritmetico* la debes poner en un paquete llamado *operador_aritmetico* y la clase *AdaptadorCalculadora* en el paquete *calculadora*.

Código .1 Código de la clase *OperadorAritmetico*

```
package operador_aritmetico;

public class OperadorAritmetico {

    double memoria = 0.0;
    double ultimo = 0.0;

    public double suma(double operando1, double operando2) {
        double result = operando1 + operando2;
        this.ultimo = result;
        return result;
    }

    /*-- COMPLETAR: métodos resta y multiplica --*/

    public double divide(double operando1, double operando2) {
        double result;
        if(operando2==0) {
            result = Double.NaN;
        } else {
            result = operando1 / operando2;
        }
        this.ultimo = result;
        return result;
    }

    /*-- COMPLETAR: métodos cuadrado, getUltimo, memoriaACero, sumaAMemoria y getMemoria --*/

}
```

Código .2 Código de la clase *AdaptadorCalculadora*

```

package calculadora;

import CalculadoraGUI.ICalculadora;
import operador_aritmetico.OperadorAritmetico;

public class AdaptadorCalculadora implements ICalculadora {

    private OperadorAritmetico operador;

    public AdaptadorCalculadora() {
        this.operador = new OperadorAritmetico();
    }

    /*-- COMPLETAR: métodos sumar, restar y multiplicar --*/

    @Override
    public double dividir(double dividendo, double divisor) throws Exception {
        if(divisor == 0) {
            if(dividendo == 0) {
                throw new Exception("Indeterminación");
            } else {
                throw new Exception("Infinito");
            }
        }
        double result = operador.divide(dividendo, divisor);
        return result;
    }

    /*-- COMPLETAR: métodos elevarAlCuadrado, memoriaAniadir, memoriaLimpiar,
        memoriaObtener y obtenerUltimoResultado --*/
}

```

Además, se te entrega la clase *AppCalculadora* implementada. Esta clase es la que se utiliza para lanzar la aplicación. Esta clase la debes añadir al paquete *calculadora*.

Código .3 Código de la clase *AppCalculadora*

```

package calculadora;

import CalculadoraGUI.CalculadoraGUI;

public class AppCalculadora {

    public static void main(String[] args) {
        AdaptadorCalculadora adaptador = new AdaptadorCalculadora();
        new CalculadoraGUI(adaptador);
    }
}

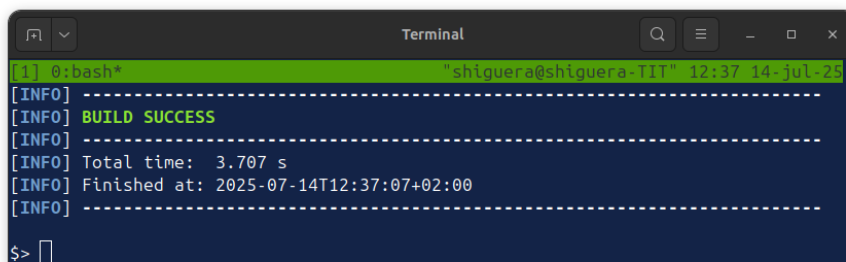
```

Una vez que hayas completado el código de estas tres clases, debes compilar la aplicación. Para ello, abre un terminal en el directorio del proyecto y teclea:

```
mvn clean install
```

También puedes hacerlo desde dentro de Eclipse pulsando con el botón derecho del ratón sobre el nombre del proyecto y eligiendo sucesivamente las opciones *Run as* → *clean* y *Run as* → *Install*.

Si todo ha ido bien, deberías ver en el terminal el mensaje *BUILD SUCCESS*, con una una imagen similar a la de la Figura 6.



```

Terminal
[1] 0:bash* "shiguera@shiguera-TIT" 12:37 14-jul-25
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.707 s
[INFO] Finished at: 2025-07-14T12:37:07+02:00
[INFO] -----
$>

```

Figura 6: Salida en el terminal tras compilar de manera satisfactoria el proyecto

El resultado de la compilación será el fichero *softcom_p1-1.0.jar*, que estará en el directorio *target* del proyecto. Para ejecutarlo y probar la aplicación, teclea la siguiente orden desde el terminal abierto en el directorio del proyecto:

```
java -cp target/softcom_p1-1.0.jar calculadora.AppCalculadora
```

Si estás trabajando en Windows, la barra para separar carpetas es la barra invertida \. En ese caso, la instrucción que debes teclear es:

```
java -cp target\softcom_p1-1.0.jar calculadora.AppCalculadora
```

Deberías ver en pantalla la calculadora y puedes probar las diferentes operaciones para comprobar que todo va bien.

Entre los recursos de la práctica se te entrega el programa completo compilado en el fichero *P1_Calculadora-1.0.jar*. Copia el fichero en el directorio del proyecto. También puedes probar ahí la aplicación con:

```
java -cp P1_Calculadora-1.0.jar calculadora.AppCalculadora
```

8 - Fase 2

En esta segunda fase de la práctica, vamos a distribuir la aplicación: la parte del interfaz gráfico de la calculadora permanecerá en un ordenador que vamos a denominar *cliente*, mientras que las operaciones aritméticas se calcularán en un segundo ordenador, que denominaremos *servidor*.

Por simplicidad, el servidor lo emularemos en el mismo ordenador en el que estemos trabajando, pero no habría ningún inconveniente en que se tratara de un ordenador físicamente independiente, pues realizaremos las conexiones utilizando HTTP.

Para conseguir nuestro objetivo, desarrollaremos un servicio gRPC, denominado *ServicioCalculadora*, que permite recibir peticiones con las operaciones de la calculadora y devuelve los resultados correspondientes.

Entre los recursos de la práctica se encuentra el fichero *P1_Calculadora-1.0.jar* con el programa completo y compilado, de forma que puedas probar y comprender su funcionamiento, antes de tratar de codificarlo por ti mismo, siguiendo las instrucciones de los próximos apartados.

Para probar el programa, debes copiar el fichero *P1_Calculadora-1.0.jar* en algún directorio de tu disco y abrir un terminal en ese mismo directorio. Lo primero, es arrancar el servidor para que quede en estado de escucha de peticiones, mediante la siguiente instrucción:

```
java -cp P1_Calculadora-1.0.jar service.server.ServidorCalculadora
```

Deberías ver que el servidor del *ServicioCalculadora* queda arrancado y en espera de recibir peticiones en el puerto 50051.

Ahora tienes que abrir un nuevo terminal situado en el mismo directorio y arrancar la calculadora, utilizando la siguiente instrucción:

```
java -cp P1_Calculadora-1.0.jar calculadora.AppCalculadoraRemota
```

Deberías ver en pantalla la calculadora. Prueba a hacer algunas de las operaciones que ofrece. El resultado debería ser el mismo que el que obtenías en la Fase 1 de esta práctica, con la salvedad de que ahora las operaciones se realizan a través del servidor.

Vamos a desarrollar la aplicación utilizando el mismo proyecto Eclipse que hemos utilizado en la Fase 1 de la práctica. En los siguientes apartados te vamos a ir dando pistas y explicaciones para que puedas ir codificando la aplicación paso a paso.

8.1 Creación del fichero *calculadora.proto*

Lo primero es crear el fichero de definición del servicio y de sus tipos de datos asociados, utilizando el lenguaje *Protobuf*, que es el que utilizan los servicios gRPC. Para ello, crea un fichero llamado *calculadora.proto* dentro del directorio *src/main/proto* del proyecto Eclipse y copia el siguiente código dentro de él:

Código .4 Código del fichero *calculadora.proto*

```
syntax = "proto3";

import "google/protobuf/empty.proto";

option java_package = "service";
option java_multiple_files = true;

service ServicioCalculadora {
    rpc sumar(DosOperandos) returns(Resultado);

    /*-- COMPLETAR: métodos restar, multiplicar, dividir y elevarAlCuadrado --*/

    rpc memoriaACero(google.protobuf.Empty) returns(google.protobuf.Empty);
    rpc memoriaRecuperar(google.protobuf.Empty) returns(Resultado);

    /* COMPLETAR: métodos memoriaAniadir y ultimoResultado --*/
}

message Operando {
    double number = 1;
}

message DosOperandos {
    double number1 = 1;
    double number2 = 2;
}

message Resultado {
    double resultado = 1;
}
```

Observa el código de definición del servicio y sus tipos. Se definen tres tipos de datos en sus correspondientes *message*: un tipo llamado *DosOperandos*, para los métodos que utilizan dos parámetros del tipo *double*, otro llamado *Operando*, para los que solo utilizan uno y un tipo para los resultados.

Observa también que hay métodos que no tienen parámetros o que no devuelven ningún valor. En estos casos hay que utilizar el tipo de datos *Empty*. Al principio del fichero hay una instrucción *import* para importar el tipo *Empty* y luego se utiliza en los métodos en los que es necesario hacerlo.

En la definición del servicio, que hemos llamado *ServicioCalculadora*, hemos dejado varios métodos sin implementar, para que los codifiques tú y puedas practicar cómo hacerlo.

Una vez que hayas completado el código de *calculadora.proto*, tienes que hacer un *mvn clean install* del proyecto. Si todo ha ido bien, se te deberían haber creado varias clases, como se muestra en la Figura 7.

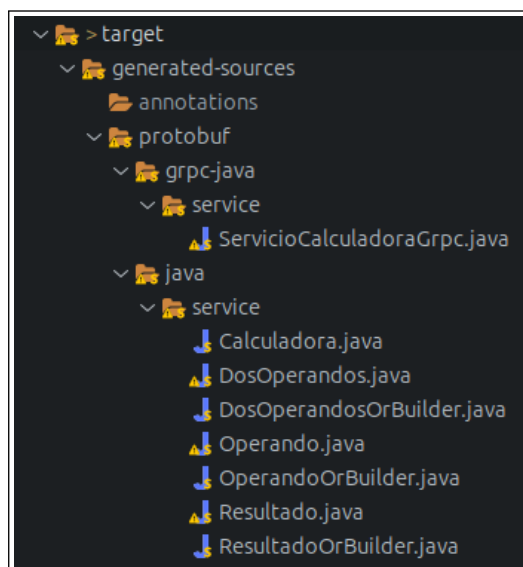


Figura 7: Clases creadas al compilar el fichero *calculadora.proto*

Observa que todas las clases se han creado dentro del paquete *service*, tal como hemos indicado en el fichero *calculadora.proto*.

La clase *ServicioCalculadoraGrpc.java* se genera automáticamente por el compilador de *Protobuf* y contiene el código necesario para conectar el servicio *Protobuf* con una aplicación Java que usa gRPC. Esto es, se trata del puente entre el archivo *.proto* y el código Java de nuestra aplicación.

Proporciona la clase base del servidor, *ServicioCalculadoraImplBase*. Esta clase abstracta define todos los métodos del servicio (sumar, restar, etc.) como métodos abstractos que tendremos que sobrescribir y es la clase base sobre la que construiremos el servicio real.

En el lado del cliente, esta clase proporciona los *stubs*, que son objetos que actúan como clientes del servicio remoto, y permiten hacer llamadas gRPC a los métodos del servicio. El desarrollador puede elegir el tipo de *stub*, según cómo quiera manejar las llamadas remotas:

- **BlockingStub:** llamadas síncronas (espera la respuesta antes de continuar). Es el que usaremos en esta práctica.
- **Stub:** llamadas asíncronas con *callbacks*.
- **FutureStub:** llamadas asíncronas que devuelven un *Future* (útil para programación concurrente).

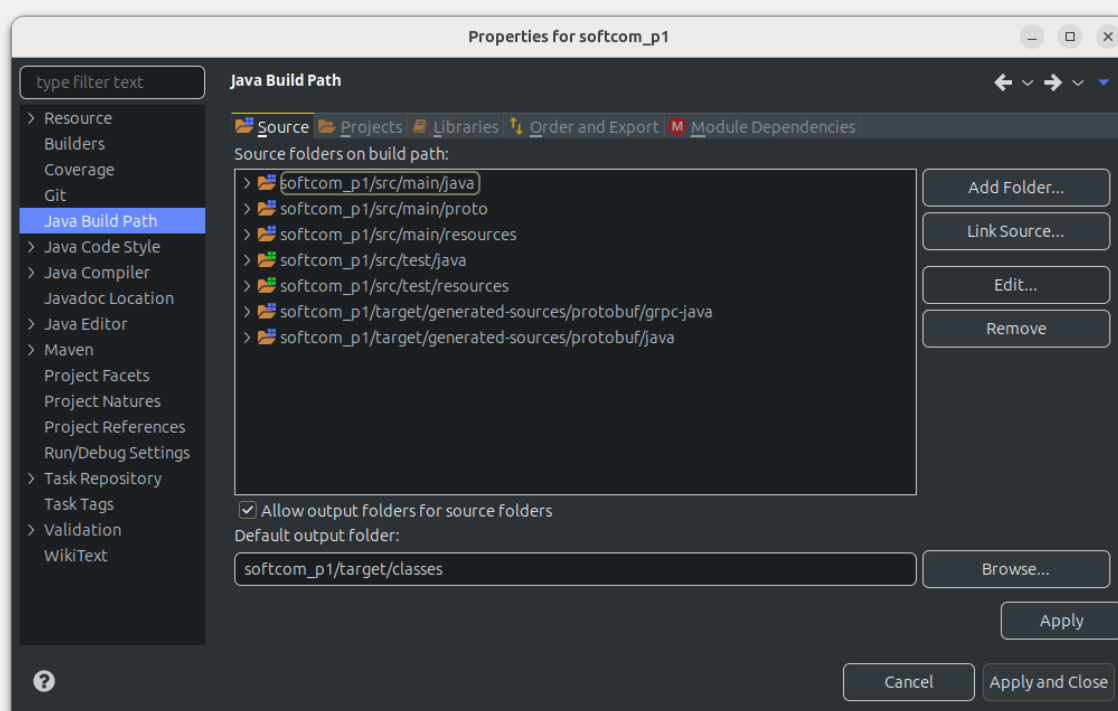
La clase *ServicioCalculadoraGrpc* también incluye métodos internos para construir el descriptor del servicio (información que gRPC necesita para manejar las llamadas), métodos para registrar el servicio en el servidor gRPC y métodos que hacen el *mapeo* de cada llamada RPC al código Java correspondiente.

El fichero *calculadora.proto* define el contrato del servicio y la clase *ServicioCalculadoraGrpc.java* proporciona el código Java que lo hace funcional, tanto en el servidor como en el cliente.

Además, se han creado clases para los tipos de datos definidos en el fichero *.proto*. En concreto, se han creado dos clases para cada uno de los tipos de datos, además de una clase auxiliar, *Calculadora.java*.

Nota:

Es posible que el editor de Eclipse no reconozca las clases que se han creado en el directorio *target*. Las clases están ahí y, si compilas el proyecto con Maven, verás que las reconoce y el programa se compila sin problema. No obstante, para que Eclipse reconozca esos directorios como parte del *path* del proyecto, es conveniente que añadas los directorios *grpc-java* y *java* que están en la carpeta *target/generated-sources/protobuf* al path del programa a través de la pestaña *source* del apartado *Java Build Path* de las propiedades del proyecto, como se ha hecho en la siguiente figura:



8.2 Creación del servidor

El siguiente paso es la creación del servidor que va a atender las peticiones que se hagan desde el cliente. Los métodos que debe atender son los especificados en el servicio definido en el fichero *calculadora.proto*.

Vamos a dividir la construcción del servidor en dos clases: la clase *ServicioCalculadoraImpl*, que es la que tendrá los métodos definidos en el servicio y la clase *ServidorCalculadora*, que servirá para lanzar un servidor HTTP.

La clase *ServicioCalculadoraImpl* deriva de la clase estática *ServicioCalculadoraImplBase*, definida en el fichero *ServicioCalculadoraGrpc* que se creo al compilar el fichero *calculadora.proto*. Puedes echar un vistazo al código de dicho fichero para comprobarlo. El esquema UML de la clase es el que muestra la Figura 8.

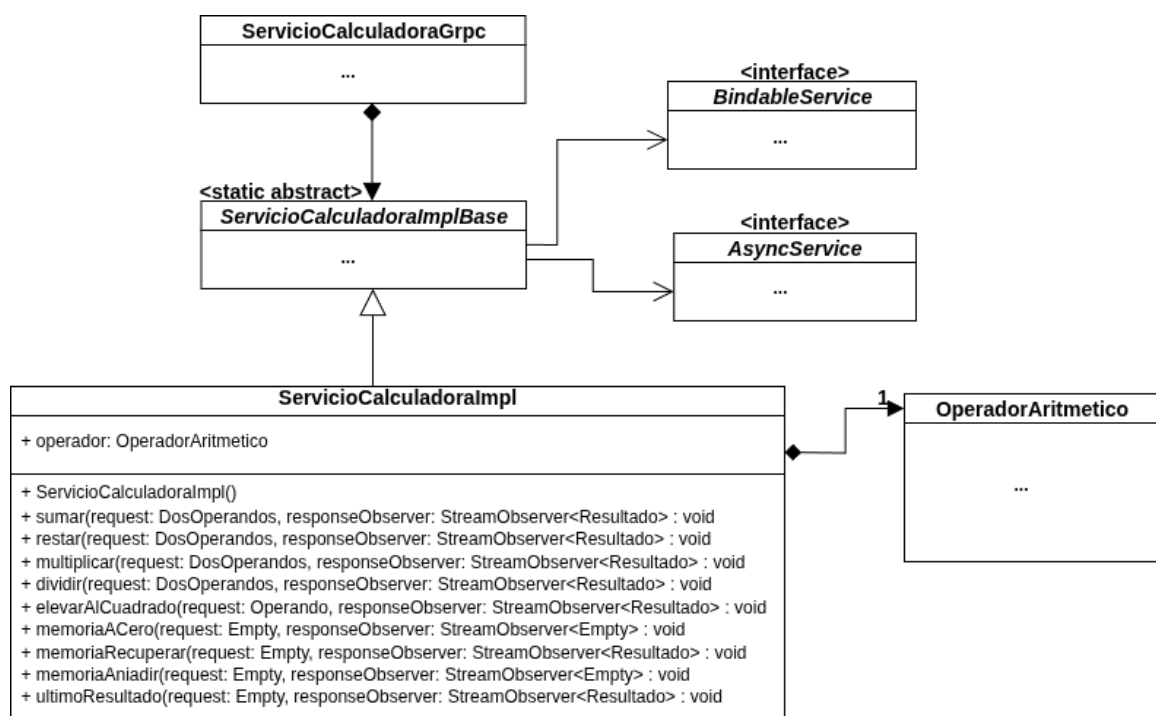


Figura 8: Diagrama UML de la clase *ServicioCalculadoraImpl*

Tienes que observar varias cosas en el diagrama de la clase *ServicioCalculadoraImpl* de la Figura 8. En primer lugar, utiliza la clase *OperadorAritmetico* que ya utilizamos en la Fase 1 de la práctica y que proporciona la operaciones que debe realizar la calculadora. Cuando codifiquemos los métodos aritméticos te podrá parecer que habría sido más fácil codificar directamente las operaciones en dichos métodos, sin recurrir a la clase externa *OperadorAritmetico*. Tienes que tener en cuenta que, en un caso real, pudiera ser que las operaciones fueran complejas y las proporcionara una clase de la que incluso no tuviéramos el código fuente. Es por eso que hemos preferido mantener la realización de las operaciones a través de la clase *OperadorAritmetico*.

La otra observación es que todos los métodos se llaman igual que los que se definieron en el fichero *calculadora.proto*. Además, estos métodos sobrescriben los métodos homónimos del *interface AsyncService* que implementa la clase *static abstract ServicioCalculadoraImplBase*.

La clase *ServicioCalculadoraImpl* la tienes que poner en el paquete *service.server*. Te ofrecemos a continuación el código de dicha clase, parcialmente completado. Debes codificar los métodos que faltan para que la clase esté operativa.

Código .5 Código de *ServicioCalculadoraImpl*

```

package service.server;

import com.google.protobuf.Empty;

import io.grpc.stub.StreamObserver;
import operador_aritmetico.OperadorAritmetico;
import service.DosOperandos;
import service.Operando;
import service.Resultado;
import service.ServicioCalculadoraGrpc;
  
```

```

public class ServicioCalculadoraImpl extends ServicioCalculadoraGrpc.ServicioCalculadoraImplBase {

    private OperadorAritmetico operador;

    public ServicioCalculadoraImpl() {
        operador = new OperadorAritmetico();
    }

    @Override
    public void sumar(DosOperandos request, StreamObserver<Resultado> responseObserver) {
        System.out.println("Invocado el método sumar()");
        double res = operador.suma(request.getNumber1(), request.getNumber2());
        Resultado resultado = Resultado.newBuilder().setResultado(res).build();
        responseObserver.onNext(resultado);
        responseObserver.onCompleted();
    }

    /*-- COMPLETAR: métodos restar, multiplicar, dividir y elevarAlCuadrado --*/

    @Override
    public void memoriaACero(Empty request, StreamObserver<Empty> responseObserver) {
        System.out.println("Invocado el método memoriaACero()");
        operador.memoriaACero();
        responseObserver.onNext(Empty.newBuilder().build());
        responseObserver.onCompleted();
    }

    /*-- COMPLETAR: métodos memoriaRecuperar, memoriaAniadir y ultimoResultado --*/
}

```

En el código anterior, observa que los métodos tienen dos parámetros: uno para la entrada, llamado *request* y del tipo de datos adecuado de entre los definidos en el servicio y el otro del tipo *StreamObserver*, adecuado a la respuesta que ofrece el servidor a través de dicho método. Todos estos elementos están definidos en la clase *ServicioCalculadoraGrpc* que se creó al compilar el fichero *calculadora.proto*.

La segunda clase que tenemos que codificar para tener el servidor operativo la hemos llamado *ServidorCalculadora*. Dispone de un único método, el método *main()*, que será el encargado de lanzar la aplicación servidor. Esta clase te la proporcionamos completamente codificada, para que puedas estudiarla:

Código .6 Código de la clase *ServidorCalculadora*

```

package service.server;

import io.grpc.Server;
import io.grpc.ServerBuilder;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

public class ServidorCalculadora {
    public static void main(String[] args) throws IOException, InterruptedException {
        final int port = 50051;
        final Server server = ServerBuilder
            .forPort(port)
            .addService(new ServicioCalculadoraImpl())
            .build();
    }
}

```



```

System.out.println("Servidor gRPC para ServicioCalculadora iniciado en el puerto " + port );
server.start();

// Shutdown hook: permite un apagado limpio del servidor,
// con CTRL+C o señal de terminación. Se puede aprovechar
// para liberar recursos.
Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        System.err.println("Apagando el servidor gRPC");
        try {
            // El tiempo de 30 segundos es arbitrario.
            // También se puede usar server.shutdownNow()
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace(System.err);
        }
    }
});

// Mantiene vivo el hilo principal hasta que el servidor se cierre
server.awaitTermination();
}
}

```

Hay varios detalles a destacar en el código anterior. En primer lugar, la forma de instanciar el *Server* de gRPC, encadenando varios métodos. Fíjate cómo se indica el puerto en el que se establecerá la escucha de peticiones. Se ha elegido el puerto *50051*, pero podría ser otro. También es interesante cómo se le indica la clase *ServicioCalculadoraImpl*, que es la que implementa los métodos del servicio.

Una vez que se ha creado el *Server*, se llama a su método *start()*, para que inicie la escucha.

A continuación, se crea un hilo para interceptar las peticiones de interrupción del servicio y dar tiempo a cualquier operación de liberación de recursos que se pudieran estar utilizando: cierre de ficheros, bases de datos u otros. Se podría haber hecho el programa sin este *hook* y, seguramente, en esta aplicación no tendría mucha importancia. Pero es una buena costumbre para aplicaciones más complejas, en previsión de errores al cerrar abruptamente el servidor. En nuestro caso, para cerrar el servidor una vez lanzado, podemos usar la combinación de teclas *CTRL+C*.

La última instrucción es importante, para que el hilo principal se mantenga activo hasta que se haya completado el hilo de cierre de la aplicación.

Ahora podemos probar nuestro servidor. Abre un terminal en el directorio principal del proyecto y compila todo utilizando la instrucción:

```
mvn clean install
```

Si todo ha ido bien, puedes lanzar el servidor tecleando en el terminal la siguiente instrucción:

```
java -cp target/softcom_p1-1.0.jar service.server.ServidorCalculadora
```

Si estás trabajando con un ordenador con sistema operativo Windows, la barra separadora de directorios es la barra invertida (**) y la instrucción la deberías escribir:

```
java -cp target\softcom_p1-1.0.jar service.server.ServidorCalculadora
```

El resultado debería ser similar al de la Figura 9.

```

[2] 0:java* "shiguera@shiguera-TIT" 21:56 13-jul-25
com/softcom_p1/1.0/softcom_p1-1.0.pom
[INFO] Installing /home/shiguera/nextcloud/AAA_Teleco/2025-26/3_SoftwareComunicaciones/w
s/softcom_p1/target/softcom_p1-1.0.jar to /home/shiguera/.m2/repository/es/upm/dte/softc
om/softcom_p1/1.0/softcom_p1-1.0.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.485 s
[INFO] Finished at: 2025-07-13T21:56:01+02:00
[INFO] -----
$> java -cp target/softcom_p1-1.0.jar service.server.ServidorCalculadora
Servidor gRPC para ServicioCalculadora iniciado en el puerto 50051

```

Figura 9: Servidor gRPC ejecutándose en el terminal, escuchando en el puerto 50051

8.3 Creación de un cliente para probar el servicio

Antes de conectar la calculadora con nuestro servidor de operaciones aritméticas, vamos a desarrollar un cliente del servicio que nos permita probar que los métodos funcionan como se espera, además de tener una primera aproximación a la creación de un cliente gRPC.

Crea el paquete *service.cliente* dentro del proyecto y copia el siguiente código en una clase llamada *PruebaClienteCalculadora.java*:

Código .7

```

package service.cliente;

import com.google.protobuf.Empty;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;

import service.ServicioCalculadoraGrpc;
import service.DosOperandos;
import service.Operando;
import service.Resultado;

public class PruebaClienteCalculadora {
    public static void main(String[] args) {

        // Crear canal hacia el servidor en localhost:50051
        ManagedChannel channel = ManagedChannelBuilder
            .forAddress("localhost", 50051)
            .usePlaintext() // sin TLS
            .build();

        // Crear un stub bloqueante (síncrono)
        ServicioCalculadoraGrpc.ServicioCalculadoraBlockingStub stub =
            ServicioCalculadoraGrpc.newBlockingStub(channel);

        // Ejemplo: sumar 10 + 5
        DosOperandos suma = DosOperandos.newBuilder().setNumber1(10).setNumber2(5).build();
        Resultado resultadoSuma = stub.sumar(suma);
        System.out.println("Suma: " + resultadoSuma.getResultado());
    }
}

```

```

// Ejemplo: elevar al cuadrado 7
Operando op = Operando.newBuilder().setNumber(7).build();
Resultado resCuadrado = stub.elevarAlCuadrado(op);
System.out.println("7 al cuadrado: " + resCuadrado.getResultado());

// Añadir el último resultado a la memoria
stub.memoriaAniadir(Empty.getDefaultInstance());

// Recuperar memoria
Resultado resMemoria = stub.memoriaRecuperar(Empty.getDefaultInstance());
System.out.println("Memoria actual: " + resMemoria.getResultado());

// Obtener último resultado
Resultado resUltimo = stub.ultimoResultado(Empty.getDefaultInstance());
System.out.println("Último resultado: " + resUltimo.getResultado());

// Dejar memoria a cero
stub.memoriaACero(Empty.getDefaultInstance());
Resultado memoriaCero = stub.memoriaRecuperar(Empty.getDefaultInstance());
System.out.println("Memoria tras poner a cero: " + memoriaCero.getResultado());

// Cerrar el canal
channel.shutdown();
}
}

```

En el código anterior, los pasos principales son:

1. Crear un *Channel* que conecte con el servidor que escucha en el puerto 50051:

```

ManagedChannel channel = ManagedChannelBuilder
    .forAddress("localhost", 50051)
    .usePlaintext()
    .build();

```

2. Crear el *stub* del cliente conectado a dicho *channel*:

```

ServicioCalculadoraGrpc.ServicioCalculadoraBlockingStub stub =
    ServicioCalculadoraGrpc.newBlockingStub(channel);

```

3. Llamar a los métodos que ofrece el servicio. En cada llamada a un método hay que seguir a su vez los siguientes pasos:

- a) Crear un objeto con los parámetros del tipo de datos adecuado de entre los definidos en el fichero de definición del servicio. Por ejemplo, para llamar al método *sumar()*, la creación del objeto para los parámetros es:

```

DosOperandos ops = DosOperandos.newBuilder().setNumber1(10).setNumber2(5).build();

```

En los métodos que no requieren parámetros, este paso no es necesario y llamaremos directamente al método pasándolo como parámetro una instancia del objeto *Empty*.

- b) Llamar al método que se quiere invocar, utilizando el *stub* del cliente, pasándole como argumento el objeto recién creado y recogiendo el resultado en un objeto de tipo de datos adecuado, en nuestro caso, un objeto del tipo *Resultado* que se definió en el fichero *calculadora.proto*. Por ejemplo, en la invocación al método *sumar()*, se hace:

```

Resultado resultadoSuma = stub.sumar(ops);

```

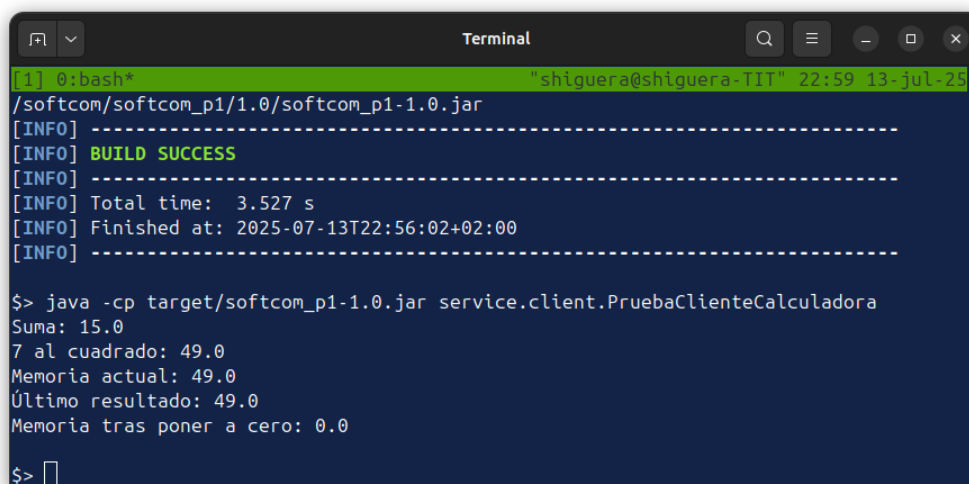
Observa cómo se hace en los métodos que no devuelven ningún resultado.

- c) Realizar la acción que se quiera hacer con el resultado del servicio, en este caso, mostrar el resultado en pantalla.

Tras crear la clase *PruebaClienteCalculadora*, deberás volver a compilar el proyecto con *mvn clean install*. Para probar la clase, el servidor tiene que estar arrancado y escuchando en el puerto 50051, tal como se explicó en el apartado anterior. Para ejecutar el cliente recién creado, tendrás que teclear en el terminal:

```
java -cp target/softcom_p1-1.0.jar service.client.PruebaClienteCalculadora
```

Recuerda utilizar la barra invertida (\) como separador de carpetas, si estás trabajando en Windows. Si todo va bien, deberías ver un resultado similar al de la Figura 10.



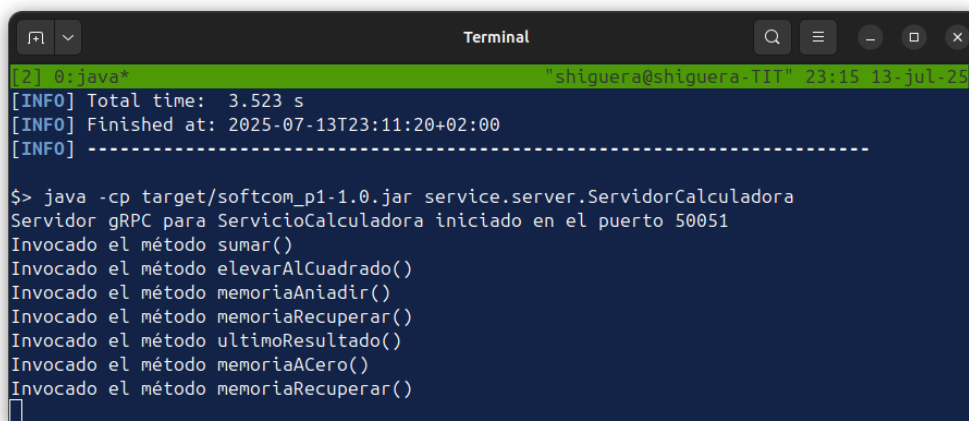
```
Terminal
[1] 0:~$ mvn clean install
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.527 s
[INFO] Finished at: 2025-07-13T22:56:02+02:00
[INFO] -----

$> java -cp target/softcom_p1-1.0.jar service.client.PruebaClienteCalculadora
Suma: 15.0
7 al cuadrado: 49.0
Memoria actual: 49.0
Último resultado: 49.0
Memoria tras poner a cero: 0.0

$>
```

Figura 10: Salida ofrecida por la ejecución del cliente *PruebaClienteCalculadora*

Si ahora miras la ventana del servidor, deberías ver una línea de información escrita por cada llamada que hayas hecho al servicio, como muestra la Figura 11.



```
Terminal
[2] 0:~$ mvn clean install
[INFO] Total time: 3.523 s
[INFO] Finished at: 2025-07-13T23:11:20+02:00
[INFO] -----

$> java -cp target/softcom_p1-1.0.jar service.server.ServidorCalculadora
Servidor gRPC para ServicioCalculadora iniciado en el puerto 50051
Invocado el método sumar()
Invocado el método elevarAlCuadrado()
Invocado el método memoriaAnadir()
Invocado el método memoriaRecuperar()
Invocado el método ultimoResultado()
Invocado el método memoriaACero()
Invocado el método memoriaRecuperar()

$>
```

Figura 11: El servidor muestra una línea informativa para cada petición que recibe

8.4 Creación del cliente para la calculadora

Ahora llega el momento de crear el cliente que dará servicio a la calculadora. Te proponemos crearlo en dos pasos. Primero, crea una clase llamada *ClienteCalculadora*, en el paquete *service.cliente*. Esta clase es similar al cliente que hemos desarrollado en el apartado anterior pero, en vez de invocar directamente los métodos del *ServicioCalculadora*, ofrece métodos públicos que permite invocarlos desde una clase externa.

La segunda clase que tienes que crear es *AdaptadorCalculadoraRemota*, en el paquete *calculadora*. Esta clase sigue el patrón *Adapter* y permite conectar la clase *ClienteCalculadora* con la clase *CalculadoraGUI*, de manera similar a lo que se hizo en la Fase 1.

El diagrama de clases de la aplicación se muestra en la Figura 12.

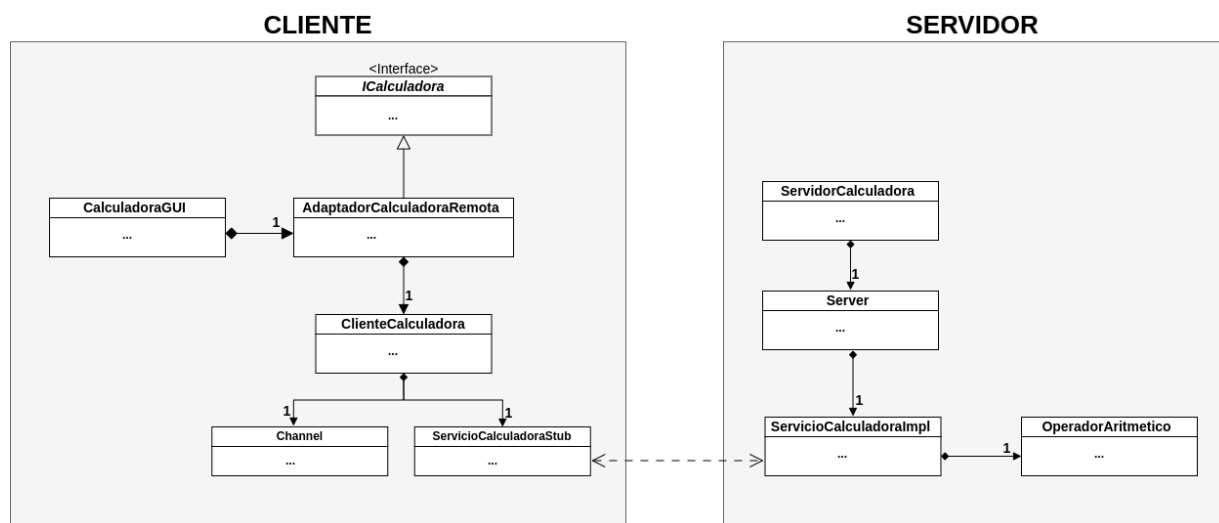


Figura 12: Diagrama de clases para el ServicioCalculadora

La Figura 13 muestra el explorador de proyecto en el Eclipse del autor, con todas las clases creadas, dentro de sus paquetes.

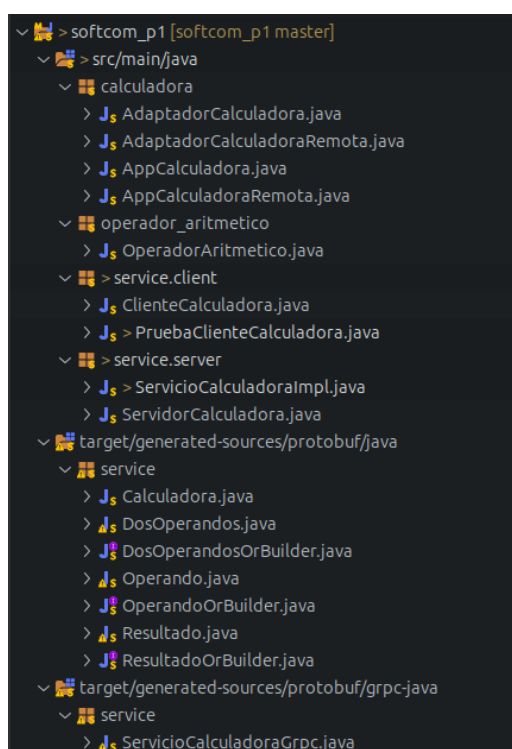


Figura 13: Vista del explorador del proyecto en Eclipse, con todas las clases de la aplicación, en sus respectivos paquetes

9 - Preguntas a responder

En este apartado te vamos a plantear algunas preguntas, en relación con la aplicación de calculadora remota desarrollada en la fase 2, que deberás responder por escrito en un fichero de texto. Estas respuestas, junto con la totalidad de las carpetas del proyecto que hayas creado, deberán formar parte de la entrega que tienes que hacer.

1. ¿Es posible que dos clientes estén activos al mismo tiempo e intenten realizar operaciones en la calculadora a través de un único servidor? Haz la siguiente prueba y responde a las preguntas que se plantean:
 - a) Arranca el servidor en un terminal.
 - b) Arranca la aplicación *AppCalculadoraRemota* en otro terminal.
 - c) Arranca una segunda vez la aplicación *AppCalculadoraRemota* en un tercer terminal.
 - d) En una de las calculadoras haz la operación 9×8 .
 - e) En la otra calculadora haz la operación *UR* (*Ultimo Resultado*).
 - f) En la segunda calculadora haz la operación *MA* (*Memoria Añadir*).
 - g) En la primera calculadora haz la operación *MR* (*Memoria Recuperar*).

Indica los resultados que has obtenido en los pasos anteriores y explica brevemente por qué sucede así.

2. Explica brevemente de qué manera intentarías evitar este comportamiento del servicio calculadora y cómo harías para conseguir que cada cliente que use la calculadora tenga su propio espacio de trabajo, independiente de otros clientes que pudieran estar utilizando el servicio.