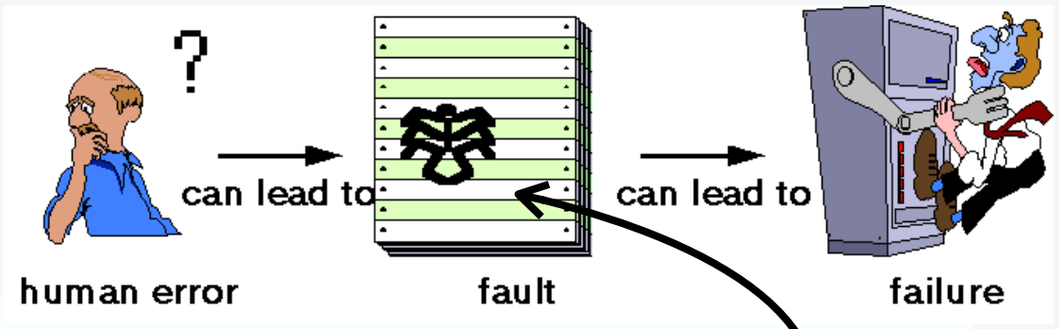
REPASO SESIONES S01..S05

SESIÓN S01



PROBAMOS PORQUE ...

- Cometemos errores

Las pruebas consumen mucho tiempo del desarrollo!!!

CÓMO PROBAMOS??

PROCESO

- PLANIFICACIÓN
- DISEÑO
- AUTOMATIZACIÓN
- EVALUACIÓN

PROBAMOS PARA ...

- Encontrar defectos (VERIFICACIÓN)
- Juzgar si la calidad del sw es aceptable (VALIDACION)
- Prevenir defectos (Pruebas estáticas)
- Toma efectiva de decisiones (Métricas)

TÉCNICAS

Pruebas DINÁMICAS (es necesario ejecutar el código):

- Pruebas UNITARIAS: encuentran DEFECTOS en las unidades. Necesitamos AISLAR nuestro SUT (controlando sus dependencias externas con DOBLES)
  - Diseño:
    - métodos de caja BLANCA (camino básico) (SESIÓN S01
    - métodos de caja NEGRA (particiones equivalentes) (SESIÓN S03)
  - Automatización
    - Drivers (verific basada en el estado, Usan STUBS para controlar las entradas indirectas de nuestro SUT) (S02) (S04)
    - Drivers (verif. basada en el comportamiento. Usan MOCKS para observar las salidas indirectas y registrar las interacciones de nuestro SUT con sus dependencias externas (S05)

HERRAMIENTAS

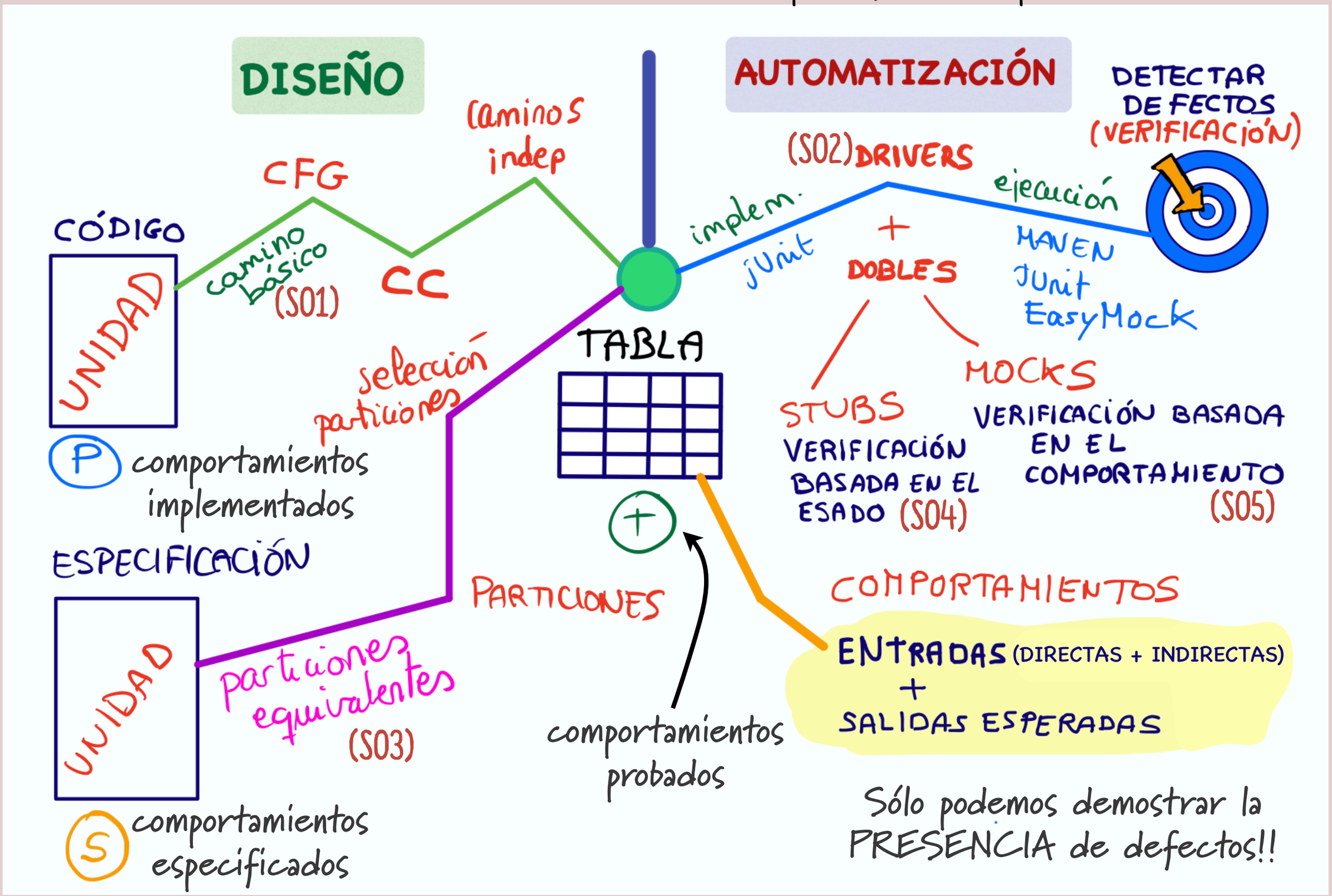
- Lenguaje JAVA
- Herram. construcción de proyectos: MAVEN
- Frameworks: JUnit, EasyMock

REPASO SESIONES S01..S05

El proceso de DISEÑO consiste en seleccionar, de forma sistemática, un conjunto de comportamientos a probar. El conjunto obtenido será efectivo y eficiente

No podemos hacer pruebas exhaustivas!!!

El proceso de AUTOMATIZACIÓN consiste en la ejecución de los comportamientos seleccionados, realización de las verificaciones correspondientes y obtención del informe de prueba, todo ello "pulsando un botón"



REPASO SESIONES P01..P05

Las sesiones prácticas nos permitirán comprender mejor los conceptos teóricos

SESIÓN P01A

Durante el curso vamos realizar pruebas DINÁMICAS. Para ello necesitaremos AUTOMATIZAR la ejecución de casos de prueba. Necesitaremos una herramienta de CONSTRUCCIÓN DE PROYECTOS. Usaremos MAVEN.
Maven proporciona 3 build scripts denominados ciclos de vida, y que podemos configurar usando el fichero pom.xml, en el que tendremos que reconocer 4 "zonas": coordenadas, propiedades, dependencias y build.
Todos los proyectos maven tienen una estructura de directorios predefinida y fija.
Durante el proceso de construcción se ejecutan las goals indicadas, o las asociadas a las fases del build script ejecutado. El proceso termina con BUILD SUCCESS o BUILD FAILURE, y genera, en el directorio target los resultados asociados a cada goal ejecutada.
Un artefacto es un fichero usado y/o generado por Maven y que se identifica por sus coordenadas.

SESIÓN P01B

Comportamiento = datos de entrada+resultado esperado

DISEÑO

Podemos seleccionar los casos de prueba a partir del código de nuestro SUT usando el método del camino básico.
Seleccionaremos un conjunto de comportamientos programados que garantizan la ejecución de todas las líneas de código (al menos una vez) de nuestro SUT y que se ejercitan todas las condiciones del SUT en su vertiente verdadera y falsa.

El conjunto obtenido es efectivo y eficiente

unidad=método java

PRUEBAS UNITARIAS

SESIÓN P02

AUTOMATIZACIÓN

Usaremos JUnit 5 para implementar drivers..
Un driver ejecuta un comportamiento seleccionado previamente (diseñado) Podremos parametrizar los drivers, reduciendo así la duplicación de código.
Disponemos de diferentes sentencias assert para verificar el resultado de los tests.
También podemos ejecutar los drivers de forma selectiva usando etiquetas.

PRUEBAS UNITARIAS

JUnit genera un informe con 3 posibles resultados.
Compilaremos y ejecutaremos los tests a través de Maven incluyendo la librería JUnit en el pom.

REPASO SESIONES P01..P05

Queremos probar cada UNIDAD por SEPARADO!!!

SESIÓN P03

DISEÑO

PRUEBAS UNITARIAS

Podemos diseñar los casos de prueba a partir de la especificación de nuestro SUT (método de particiones equivalentes).
Seleccionaremos un conjunto de comportamientos especificados que garantizan la ejecución de todas las particiones de entrada/salida (al menos una vez), y que las particiones inválidas se prueban de una en una.
El conjunto obtenido es efectivo y eficiente

SESIÓN P04

AUTOMATIZACIÓN

Implementamos drivers usando VERIFICACIÓN basada en el ESTADO.
Usaremos dobles, que sustituirán a las dependencias externas de nuestro SUT durante las pruebas.
Los dobles (STUBS) nos permiten controlar las entradas indirectas a nuestro SUT, para así poder AISLAR la unidad a probar (método java).
Para poder inyectar los dobles durante las pruebas puede que necesitemos refactorizar nuestra SUT.
La implementación del doble está físicamente separada del código del driver

Maven se encargará de ejecutar las pruebas de forma automática!!!

PRUEBAS UNITARIAS

SESIÓN P05

AUTOMATIZACIÓN

Implementamos drivers usando VERIFICACIÓN basada en el COMPORTAMIENTO.
Usaremos dobles, que sustituirán a las dependencias externas de nuestro SUT durante las pruebas.
Los dobles (MOCKS) nos permiten verificar la interacción de nuestro SUT con las dependencias externas, AISLANDO el código de la unidad a probar (método java).
Para poder inyectar los dobles durante las pruebas puede que necesitemos refactorizar nuestra SUT.
Usaremos la librería EasyMock para implementar los dobles "dentro" del driver.
La librería EasyMock permite implementar los dos tipos de verificaciones (estado y comportamiento).

Implementación de Dobles sin EasyMock

- La implementación de un stub tiene como objetivo CONTROLAR la/s entrada/s indirectas a nuestra SUT.
- La implementacion del doble SIEMPRE estará fuera del test (y de la clase que contiene los tests). Implentaremos un **UNICO** doble para toda la tabla (implementación **GENÉRICA**).
- Si un doble es invocado varias veces desde la SUT, dicho doble debe controlar más de una entrada indirecta. En lo posible, debemos **evitar usar bucles** para implementar los dobles. Por ejemplo:
  - el doble de operacionReserva(String socio, String isbn) devuelve valores diferentes cada vez que es invocado

```
@Override
public void operacionReserva(String socio,
                    String isbn) throws ... {
   if (falloConexion.contains(isbn)) {
       throw new JDBCException();
   } else if (!sociosValidos.contains(socio)) {
       throw new SocioInvalidoException();
   } else if (!isbnsValidos.contains(isbn)) {
       throw new IsbnInvalidoException();
   }
}
```

falloConexion es un ArrayList, con los isbns que generan una excepción JDBC
sociosValidos es un ArrayList, con los socios válidos
isbnsValidos es un ArrayList, con los isbns válidos

```
@Override
public void operacionReserva(String socio, String isbn)
                                                  throws ... {
  if (entradas!=null) {
    Entrada valor = entradas.remove(0);
    switch (valor) {
      case JDBC_EXC: throw new JDBCException(); break;
      case SOCIO_EXC: throw new SocioInvalidoException();break;
      case ISBN_EXC: throw new SocioInvalidoException();
    };
  }
}
```

Entrada es un tipo enumerado con valores {JDBC_EXC, SOCIO_EXC, ISBN_EXC}
entradas es un ArrayList de objetos de tipo Entrada

No uses un array si puedes usar un ArrayList!!
Si usamos una lista estamos asumiendo que vamos a iterar X veces. Un stub no puede ser el causante de que el test falle, por eso necesitamos comprobar que la lista no esté vacía!!

Ejercicio 1

https://easymock.org/api/

```java
public class Currency {
    private String units;
    private double amount;
    private int cents;
    private ExchangeRate converter;

    public Currency(double amount, String code) {
        this.units = code;
        this.amount = Double.valueOf(amount).intValue();
        this.cents = (int) ((amount * 100.0) % 100);
        converter=null;

    }

    public void setConverter(ExchangeRate converter) {
        this.converter = converter;
    }

    //comprobamos si el objeto es válido
    public boolean checkConverter() {
        throw new UnsupportedOperationException("Not supported yet");
    }

    public Currency toEuros() {
        if (checkConverter()) {
            if ("EUR".equals(units)) {
                return this;
            } else {
                double input = amount + cents / 100.0;
                double rate;
                try {
                    rate = converter.getRate(units, "EUR");
                    double output = input * rate;
                    return new Currency(output, "EUR");
                } catch (IOException ex) {
                    return null;
                }
            }
        } else return null;
    }
}
```

- Implementa un driver usando verificación basada en el comportamiento para la unidad Curency.toEuros(), teniendo en cuenta que queremos pasar a euros la cantidad de 2.50 dólares ("USD"), que usamos un objeto ExchangeConverter válido, que el resultado de invocar a ExchangeConverter.getRate() es 1.5 y que el resultado esperado es 3.75 "EUR"

En este caso el constructor tiene parámetros!!

Usaremos el método IMockBuilder.withConstructor( ), cuando creemos el doble

DRIVER

Versión sin chequear el orden de invocaciones entre mocks

- Usamos EasyMock para generar los dobles de forma automática

```
@Test
public void toEuros_when_2_50_USD_should_return_3_75_EUR() {
    //resultado esperado
    Currency expected = new Currency(3.75, "EUR");
    //creamos los dobles
    ExchangeRate mock = EasyMock.mock(ExchangeRate.class);
    Currency testable = EasyMock.partialMockBuilder(Currency.class)
                                .withConstructor(2.50,"USD")
                                .addMockedMethod("checkConverter")
                                .mock();
    //Programamos las expectativas
    Assertions.assertDoesNotThrow(()->
        EasyMock.expect(mock.getRate("USD", "EUR"))
                .andReturn(1.5)
    );
    EasyMock.expect(testable.checkConverter()).andReturn(true);
    testable.setConverter(mock);
    EasyMock.replay(mock,testable);
    Currency actual = testable.toEuros();
    assertEquals(expected, actual);
    EasyMock.verify(mock,testable);
}
```

En lugar de invocar al constructor por defecto, podemos invocar a cualquier otro constructor de la clase

```
.withConstructor(double.class, String.class)
.withArgs(2.50,"USD")
```

Si hay más de un constructor con parámetros compatibles tenemos que indicar primero el tipo de los parámetros

- Si, por ejemplo la clase ExchangeRate tuviese un constructor con un parámetro de tipo X, deberíamos usar:

```
object = new X();
ExchangeRate mock = EasyMock.createMockBuilder(ExchangeRate.class)
                            .withConstructor(object)
                            .mock();
```

DRIVER

Versión que comprueba el orden de invocaciones entre mocks

```
1. @Test
2. //versión con verificación basada en el comportamiento ESTRICTA!!!
3. public void toEuros_when_2_50_USD_should_return_3_75_EUR() {
4.     //resultado esperado
5.     Currency expected = new Currency(3.75, "EUR");
6.     //creamos los dobles
7.     IMocksControl ctrl = EasyMock.createStrictControl();
8.     ExchangeRate mock = ctrl.mock(ExchangeRate.class);
9.     Currency testable = EasyMock.partialMockBuilder(Currency.class)
10.             .withConstructor(2.50,"USD")
11.             .addMockedMethod("checkConverter")
12.             .mock(ctrl);
13.     //Programamos las expectativas
14.     EasyMock.expect(testable.checkConverter()).andReturn(true);
15.     Assertions.assertDoesNotThrow(()->
16.             EasyMock.expect(mock.getRate("USD", "EUR"))
17.                     .andReturn(1.5)
18.     );
19.     //EasyMock.expect(testable.checkConverter()).andReturn(true);
20.     testable.setConverter(mock);
21.     ctrl.replay();
22.     Currency actual = testable.toEuros();
23.     assertEquals(expected, actual);
24.     ctrl.verify();
25.}
```

Creamos los dos dobles en el contexto de un objeto IMocksControl. El objeto "ctrl" chequea el orden de invocaciones entre los dobles.

Si descomentamos esta línea y comentamos la línea 14, el test devolverá FAILURE

Cuando implementemos un driver usando verificación basada en el comportamiento SIEMPRE tendremos que verificar el ORDEN de las invocaciones entre los dobles!!!

EJERCICIO PROPUESTO

Dado el siguiente código, implementa un driver usando verificación basada en el comportamiento, suponiendo que en la BD tenemos los alumnos indicados en la siguiente tabla antes de ejecutar el método. El método en cuestión obtiene un listado de alumnos después de acceder a una base de datos (tabla alumnos), o bien devuelve una excepción de tipo SQLException

```
public class Listados {
  public String porApellidos(Connection con, String tableName) throws SQLException {
   Statement stm = con.createStatement();
  //realizamos la consulta y almacenamos el resultado en un ResultSet
   ResultSet rs =
      stm.executeQuery("SELECT apellido1, apellido2, nombre FROM " + tableName);
   String result = "";
   //recorremos el ResulSet
    while (rs.next()) {
      String ap1 = rs.getString("apellido1");
      String ap2 = rs.getString("apellido2");
      String nom= rs.getString("nombre");
      result += ap1 + " " + ap2 + ", " + nom + "\n";
   }
   return result;
  }
}
```

Nota: Connection, Statement y ResultSet son Interfaces Java, cuyos métodos pueden devolver una excepción de tipo SQLException, al igual que el método a probar.

tabla alumnos

Apellido1	Apellido2	Nombre
Garcia	Planelles	Jorge
Pérez	Verdú	Carmen
González	Alamo	Eva
Martínez	López	Roque

Resultado esperado:

"Garcia Planelles, Jorge\n
Pérez Verdú, Carmen\n
González Alamo, Eva\n
Martínez López, Roque\n"