**JUnit y Test Unitarios**

**@BeforeAll**  
Se ejecuta antes de todos los test unitarios solo una vez.   
Para inicializar variables  
  
**@BeforeEach**  
Se ejecutan antes de acada test.  
Para inicializar variables comunes a todos los test

**@test**  
Para generar test unitario

**@test  
@disabled**  
Para que el test no se ejecute

**@AfterEach**  
Para correr despues de cada test

**@AfterAll**  
Para correr cuando terminan todos los test

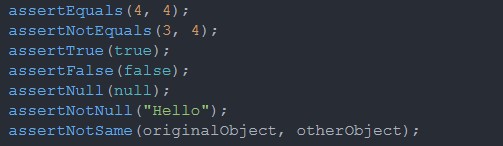
**@ParameterizedTest**Para corrrer el test con multiples argumentos, toma parametros de diferentes fuentes.

**@Tag**Permite lanzar conjuntos de test en funcion de las etiquetas que especifiquemos.

**Para concluir**

Tal como pudimos dar cuenta, **JUnit** provee una gran variedad de assertions que se encuentran ubicadas en org.junit.jupirter.api.Assertions, por ejemplo:

* assertArrayEquals
* assertEquals
* assertTrue and assertFalse
* assertNull and assertNotNull
* assertSame and assertNotSame
* assertAll
* assertNotEquals
* assertIterableEquals
* assertThrows
* assertTimeout and assertTimeoutPreemptively
* assertLinesMatch



**Test unitarios**

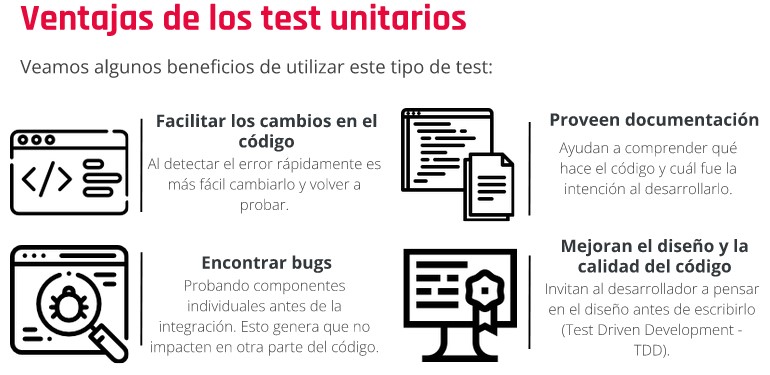
Tiene por objetivo testear una pequeña parte aislada del codigo, la cual se prueba por separada antes de integrarla con otras unidades. Cualquier dependencia del modulo bajo prueba se sustituye por un mock o un stub.

**Proceso 3A**

Arrange (organizar): Definen los requisitos que debe cumplir el codigo.

Act (Actuar): Ejecuta el test que dara lugar a los resultados que debemos analizar.

Assert (afirmar): Comprueban si los resultados obtenidos son los esperados. De ser asi, se valida y continua. Caso contrario, se corrige el error hasta que desaparezca.



**Test de integracion**

Las unidades individuales se integran para formar componentes mas grandes, si dos unidades fueron probadas, se combinan en un componente integrado y se prueba la interfaz entre ellas. Cubriendo un area mayor del codigo.

Tiene por objetivo validar la interaccion entre los modulos de software.

**Principio F.I.R.S.T.**

**F**ast (rapidos) Es posible tener milse de test en nuestro proyecto y deben ser rapidos de correr.

**I**solated/**I**ndependent (Aislados/independientes): Un metodo de test debe cumplir con los **3A**. Ademas, no debe ser necesario que sean corridos en un determinado orden para funcionar, es decir, deben ser independientes unos de los otros.

**R**epeatable (repetibles): Resultados deterministicos. No deben depender de datos del ambiente minetras estan corriendo. Ej: la hora del sistema.

**S**elf-validating ( autovalidados): No debe ser requerida una inspeccion manual para validar los resultados.

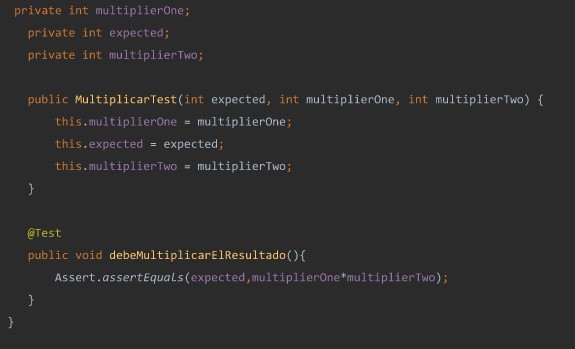
**T**horough: (Completos): Deben cubrir cada escenario de un caso de uso y no solo buscar una cobertura del 100%. Probar mutaciones, edge cases, excepciones, errores, entre otros.

**Test parametrizado**

Nos permite realizar multiples comprobaciones simplemente para probar diferentes casos.

Para utilizarlo, utilizamos un custom runner que es Parametrized. Con el cual podremos definir los parametros de varias ejecuciones en un solo test.





En la siguiente linea, estamos indicando que vamos a utilizar el runner de Parameterized, el cual se encargara de ejecutar el test las veces necesarias en funcion del numero de parametros configurado.



La anotacion **@Parameters** indica cual es el metodo que nos va a devolver el conjunto de parametros a utilizar por el runner.

Lo que necesitamos es un constructor que permita ser inicializado con los objetos que tenemos en cada elemento de la colección.

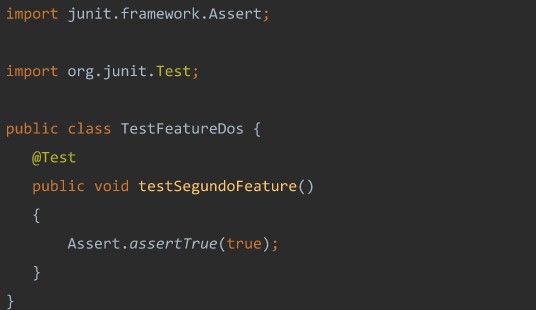
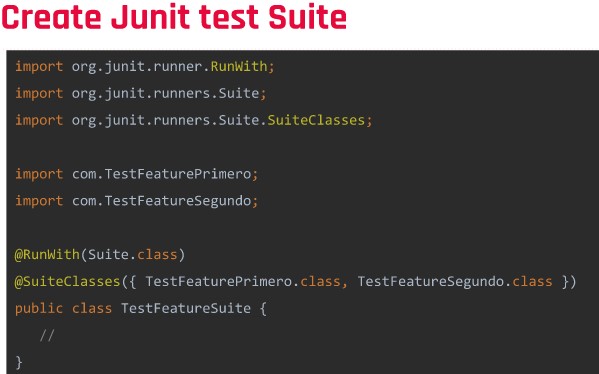
Finalmente, se ejecutara el test utilizando los datos que hemos recogido en el constructor.

**Test Suite**

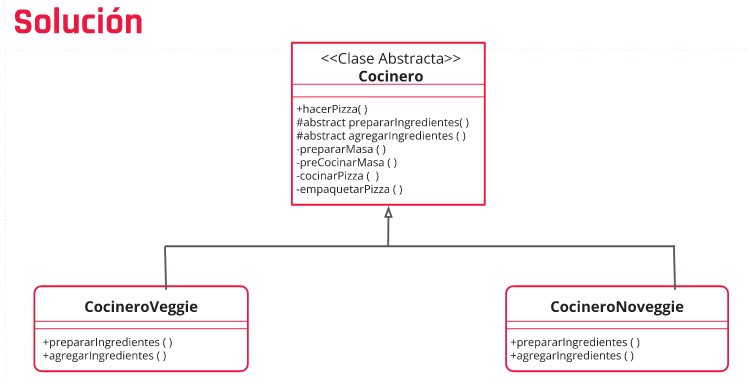
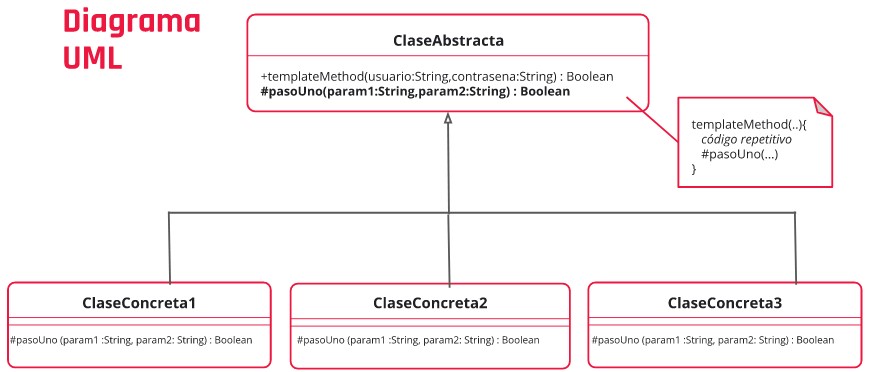
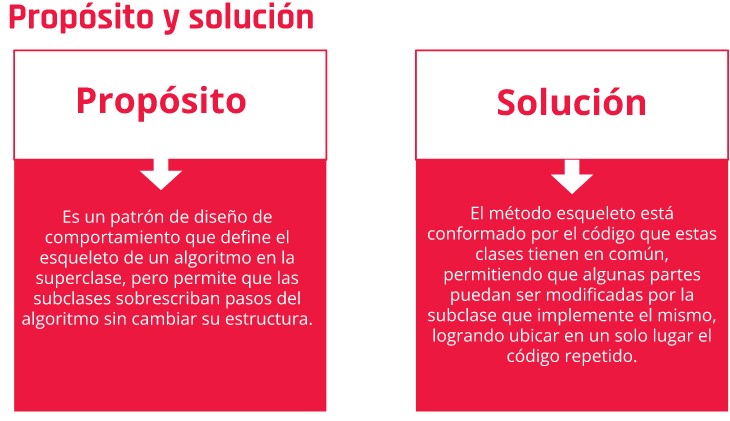
JUnit test suite nos permite agrupar y ejecutar los tests en grupo. Las suites de prueba se pueden crear y ejecutar con estas anotaciones

* @RunWith
* @SuiteClasses





**Patron Template Method**



**Log4j**

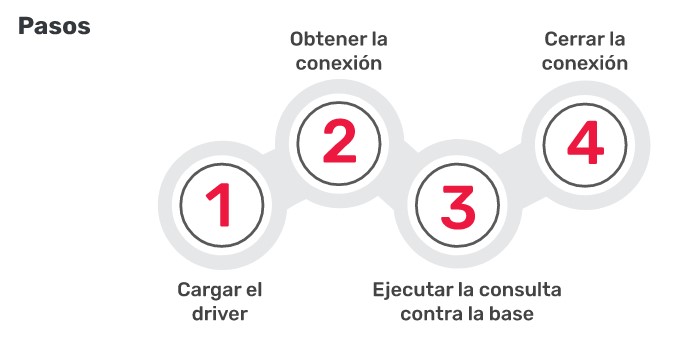
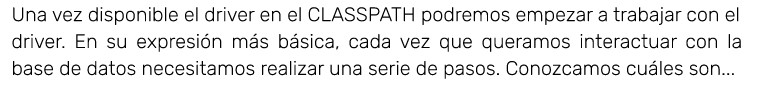
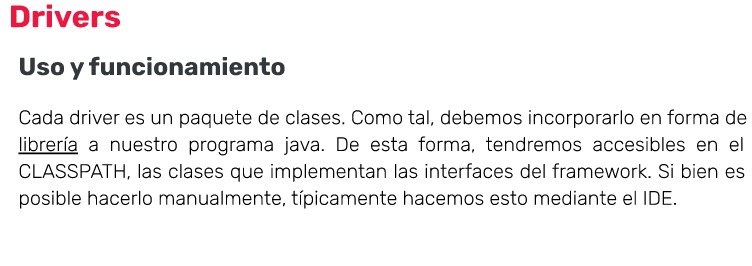
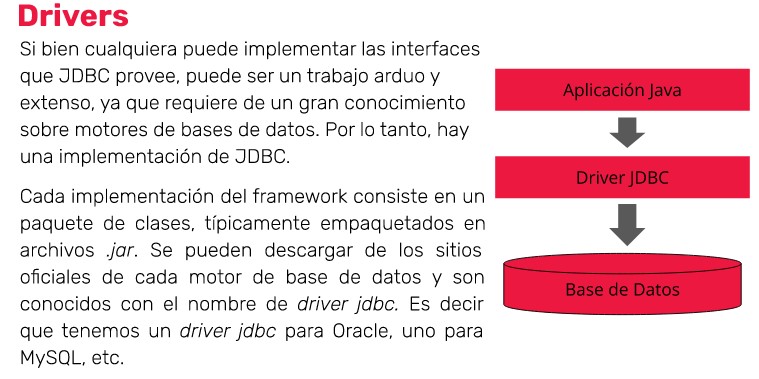
Crear log4k.properties como un file.

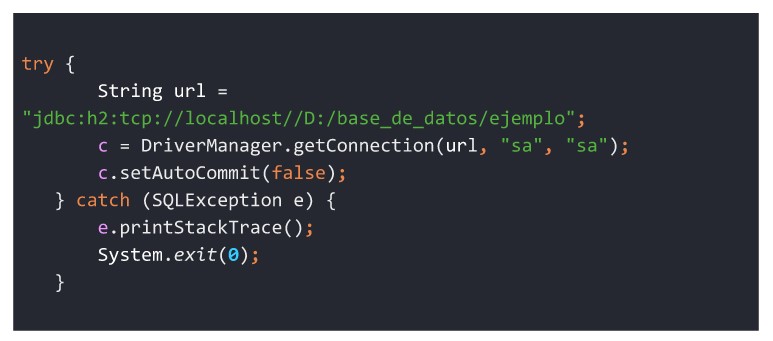
# indicamos el nivel minimo de logging y los appenders que vamos a emplear. Establecemos el nivel en DEBUG y creamos 2 appenders, stdout y file.  
log4j.rootLogger=DEBUG, stdout, file  
  
# Nivel se empezaran a mostrar las advertencias tanto por consola como a almacenarse en el fichero.  
log4j.logger.infoLogger=DEBUG  
  
# Evitamos que los appenders hereden la configuracion de sus appenders padres, en caso de que los hubiera.  
log4j.additivity.infoLogger=false  
  
# Configuracion para imprimir mensajes por consola, primero indicamos el tipo de looger refiriendo la clase que imprimira los mensajes  
# En la segunda linea le decimos que queremos imprimirlo directamente por consola.  
log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.Target=System.out  
  
# Plantilla que tendra cada mensaje.  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n   
  
# Configuramos Appender para que salga a traves de un fichero.  
# configuramos clase como RollingFileAppender para crear distintos ficheros al cumplirse determinadas condiciones.  
log4j.appender.file=org.apache.log4j.RollingFileAppender  
  
# nombre con ruta que tendra el fichero de log  
log4j.appender.file.File=avisos.log  
  
# establecemos el tamaño maximo del fichero con MaxFile Size, y con MaxBackupindex indicamos cuantos archivos podemos tener usando el mismo log.  
# a partir de llegar al maximo, comenzaran a sobreescribirse empezando por el mas antiguo.  
log4j.appender.file.MaxFileSize=5MB  
  
# indicamos que plantilla tendran nuestros mensajes  
log4j.appender.file.layout=org.apache.log4j.PatternLayout  
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

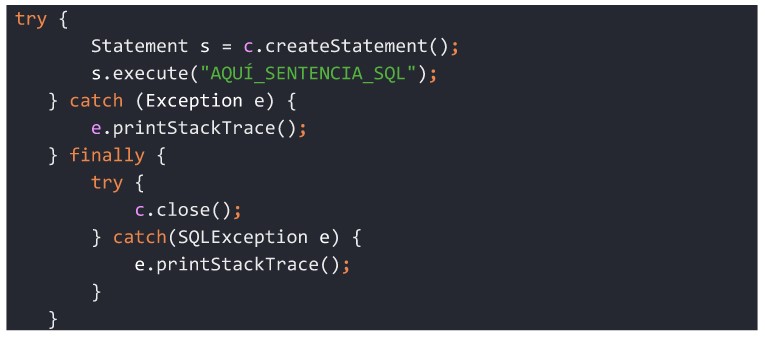
**Acceso a base de datos**

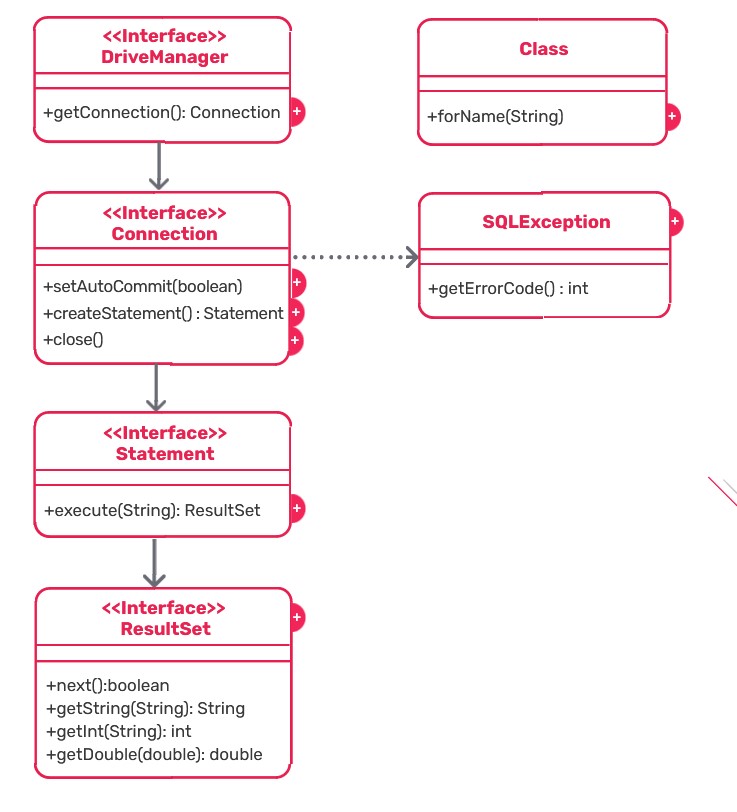
**¿Qué es JDBC?**

JDBC, por las siglas de Java DataBase Conectivity, es un framework que consiste en múltiples interfaces y solo algunas clases de soporte. Esto se debe a que la idea detrás de JDBC es que cualquiera pueda crear su propia implementación del framework y adaptarla a sus necesidades. Dado que se trata de un conjunto de interfaces, cualquier código que interactúe con el framework no se verá afectado si se altera la implementación.  
  
Es por esto que JDBC define interfaces que solo declaran el comportamiento que debe llevarse a cabo para conectarse e interactuar con una base de datos. Así, nos encontramos con interfaces tales como: Connection (abstracción del comportamiento de una conexión), Statement (define el comportamiento para realizar sentencias contra una base de datos, sean queries y otras instrucciones), ResultSet (que abstrae el comportamiento para extraer resultados de las consultas), entre otras. Todas estas clases e interfaces están dentro del paquete java.sql.\*. Entonces, para interactuar con los diferentes motores de base de datos, debemos tener una implementación de estas y otras interfaces, es decir, una clase concreta que implemente cada interfaz.







****

**Desgloce del UML de arriba**

**Drive Manager**

Con la clase disponible, hacemos uso de la clase DriverManager que es el registro de los drivers jdbc que tenemos configurados y obtenemos una conexión a la base de datos.

DriverManager.getConnection necesita tres parámetros:

* **URL de conexión:** es una URL para conectarse al servidor, el formato varía ligeramente de proveedor a proveedor, pero siempre se puede encontrar en la documentación oficial de cada uno.
* **Usuario:** si el motor soporta acceso multiusuario, acá se pone el nombre de usuario.
* **Password:**si el motor soporta acceso multiusuario, acá se pone la contraseña de usuario.

**Connection**

**+setAutoCommit :** Con este método indicamos si esta conexión debe manejar las transacciones automáticamente (true) o manualmente (false). Por el momento, lo pondremos en true.

**+createStatement:** Con este método, preparamos el camino para ejecutar una sentencia contra la base de datos. Aquí nuevamente, **se obtiene una instancia de una clase que implementa la interfaz** Statement.

**+close:** Para cerrar la conexión. Siempre que usamos algún recurso deberemos liberarlo y, según vimos, es uno de los usos típicos del bloque finally en un esquema try-catch-finally.

**Statement**

Lo que hacemos es ejecutar la sentencia contra la base de datos. Toma como argumento un String con la sentencia SQL propiamente dicha. Cabe aclarar que hay diferentes maneras de ejecutar sentencias:

1. **boolean execute(String):** se utiliza para ejecutar **cualquier** *sentencia* a la base de datos.
2. **int executeUpdate(String):**se utiliza para ejecutar sentencias DML, o sea, sentencias que manipulen datos (insert, update, delete). Devuelve un entero con la cantidad de filas afectadas por la sentencia
3. **ResultSet executeQuery(String):** se utiliza para ejecutar consultas a base de datos.

**ResultSet**

Se utiliza para obtener resultados de una consulta a la base de datos. Para recorrer los resultados ResultSet tiene una serie de operaciones fundamentales:

* **next():** devuelve true o false, indicando si hay una tupla siguiente para analizar o no. **Si la hay, avanza el puntero una posición**.
* **getXYZ(nombreCampo: String):**son métodos para obtener los valores de cada campo, dependiendo de su tipo. Por ejemplo:
  + getInt(“id”) devolvería el entero que corresponde al campo “id”.
  + getString(“nombre”) devolvería el String con el valor del campo “nombre”.
  + getDate(“fecha”) devolvería un java.sql.Date almacenado en el campo fecha
  + Y la lista sigue: getBoolean, getLong, getDouble, getByte, etc.
  + Cabe aclarar que getXYZ también acepta un número de columna como un entero —es decir, podríamos hacer getString(2) para obtener el valor del nombre del usuario, si el índice de columna está basado en 1 y no en 0—.
  + Si se utilizan “alias” en la consulta, el String que se pase al getXYZ será el alias.

**Class**

Con este método cargamos o ‘’registramos’’ la clase principal del driver. Esto hace disponible al DriverManager para poder administrar los drivers jdbc e ir instanciando las clases —que implementan las interfaces— del driver. Como lo que estamos haciendo es buscar una clase por su nombre, mediante un parámetro String, el compilador puede tomar esa sintaxis como válida, pero al momento de la ejecución, puede que la clase no se encuentre en el CLASSPATH. Por eso, este método arroja una ClassNotFoundException que debe manejarse adecuadamente.

**SQLException**

Es una checked exception como cualquier otra y tiene un atributo muy útil cuyo valor podemos consultar con el método getErrorCode(). Este método nos da el código de error que arrojó la base de datos, es decir que, además de la excepción, podemos saber a más bajo nivel qué está ocurriendo.

**¿Qué es H2?**

H2 es una base de datos open source escrita en Java que permite integrar aplicaciones en Java o ejecutarse en modo cliente-servidor. Principalmente, se puede configurar para que se ejecute como una base de datos en memoria. Entonces, los datos no persistirán en el disco, debido a que la base de datos no se utiliza para el desarrollo de producción, sino principalmente para el desarrollo y las pruebas.