**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

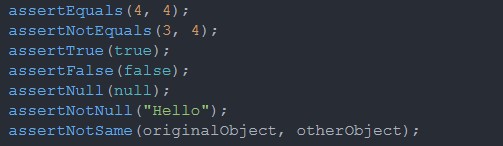
**@ParameterizedTest**Para corrrer el test con multiples argumentos, toma parametros de diferentes fuentes.

**@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**

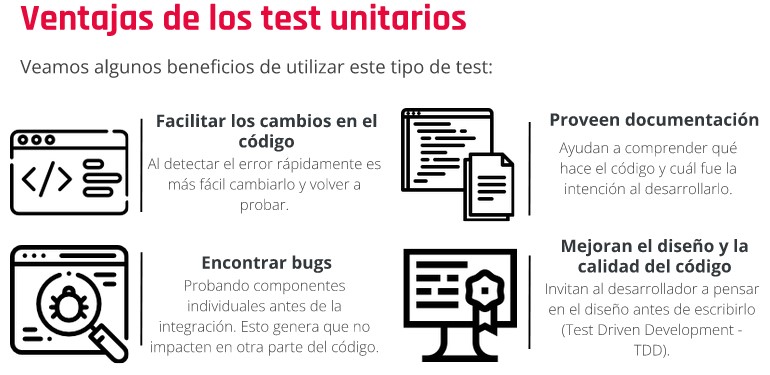
Tiene por objetivo testear una pequeña parte aislada del codigo, la cual se prueba por separada antes de integrarla con otras unidades. Cualquier dependencia del modulo bajo prueba se sustituye por un mock o un stub.

**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.

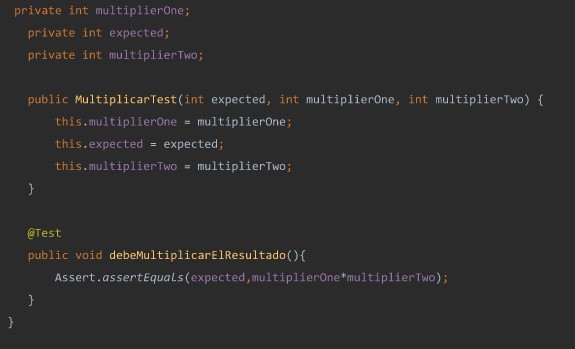
**T**horough: (Completos): Deben cubrir cada escenario de un caso de uso y no solo buscar una cobertura del 100%. Probar mutaciones, edge cases, excepciones, errores, entre otros.

**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.

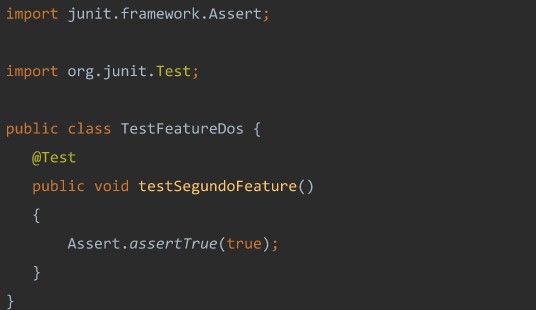
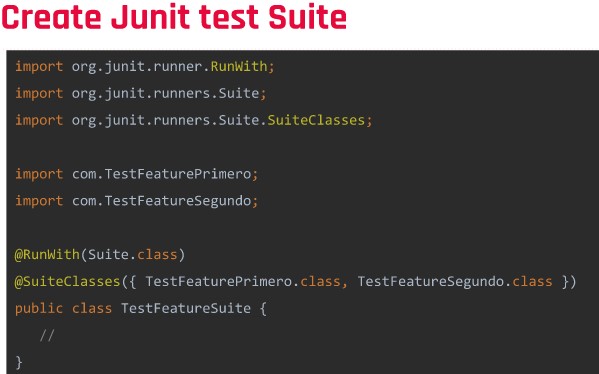
Finalmente, se ejecutara el test utilizando los datos que hemos recogido en el constructor.

**Test Suite**

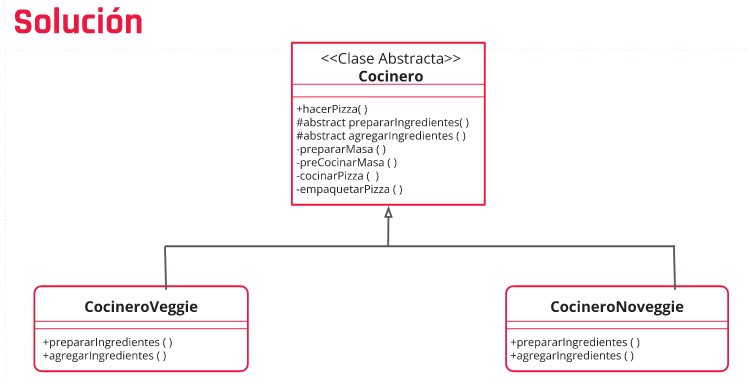
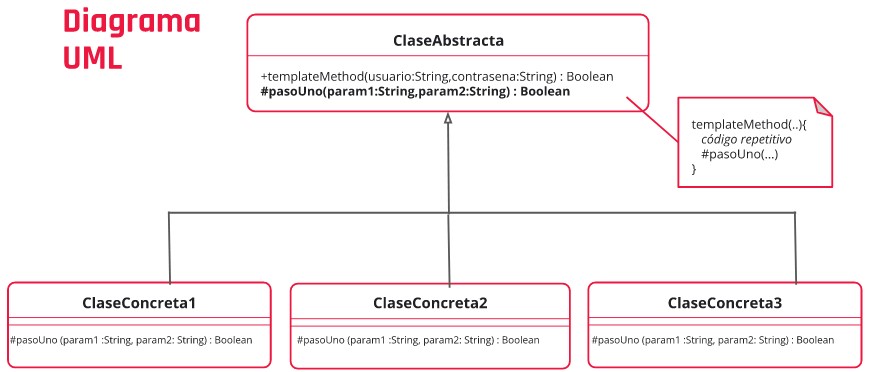
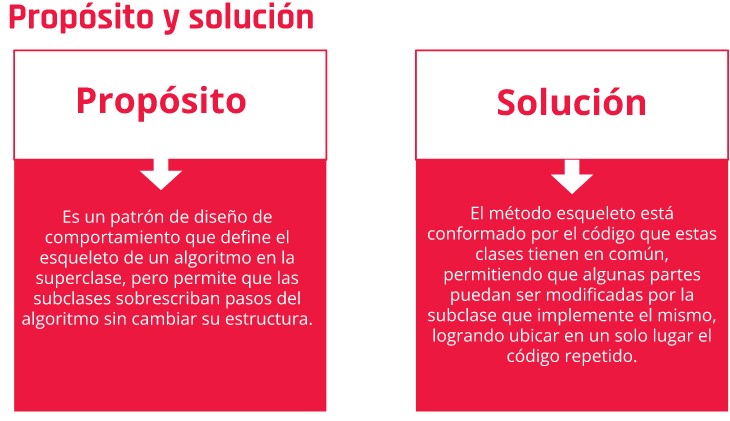
JUnit test suite nos permite agrupar y ejecutar los tests en grupo. Las suites de prueba se pueden crear y ejecutar con estas anotaciones

* @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