PpSs Planificación y Pruebas de Sistemas Software

Curso 2019-20

Sesión S01: Pruebas del software

Pruebas: qué son, por qué y para qué probamos

Principios en los que se basan las pruebas

Pruebas y depuración

Tipos de pruebas en función de:

- Objetivos de las pruebas
- Instante en el que se realizan (niveles de pruebas)
- Técnica de pruebas usada

Casos de prueba: diseño y ejecución

Pruebas y comportamiento

Construcción de software y pruebas

Vamos al laboratorio...

María Isabel Alfonso Galipienso
Universidad de Alicante eli@ua.es

Definición del proceso de pruebas

"Testing is the process of executing a program with the intent of finding errors. If our goal is to show the absence of errors, we will discover fewer of them. If our goal is to show the presence of errors, we will discover a large number of them"

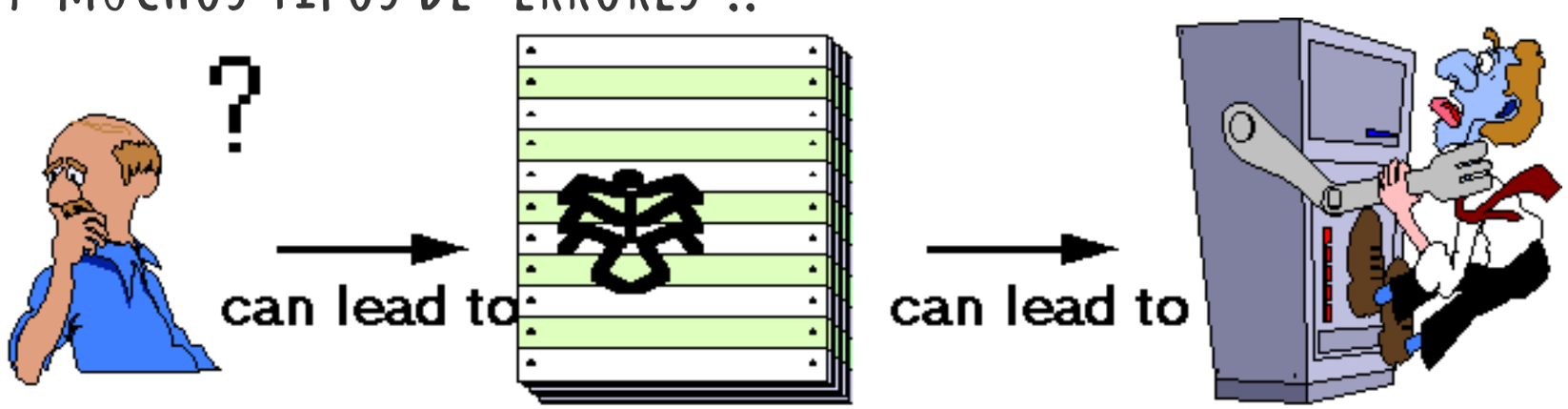
Glenford J. Myers (1979)

Las pruebas son un conjunto de actividades conducentes a conseguir alguno de estos objetivos:

* Encontrar defectos
* Evaluar el nivel de calidad del software
* Obtener información para la toma de decisiones
* Prevenir defectos

(ISQTB Foundation LevelSyllabus -2011)

Hay muchos tipos de "errores"!!



human error — mistake

fault — bug, defect

failure — error

Actividades del proceso de pruebas

según ISQTB Foundation Level Syllabus (2011)

PLANIFICACIÓN y control de las pruebas

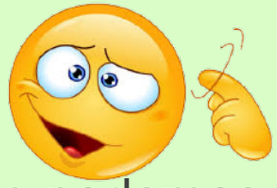
Definimos los objetivos de las pruebas, y en todo momento tenemos que asegurarnos de que cumplimos con esos objetivos (p.ej. queremos realizar pruebas sobre el 95% de código)

DISEÑO de las pruebas

Es el proceso más importante, si queremos efectivamente cumplir los objetivos marcados. Básicamente consiste en decidir con qué datos de entrada concretos vamos a probar el código, de forma que seamos capaces de detectar el máximo número de errores posibles, en el menor tiempo posible

IMPLEMENTACIÓN y ejecución de las pruebas

Creamos código, para probar nuestro código!!



...No, no nos hemos vuelto locos. La idea es que podamos ejecutar las pruebas pulsando un botón, en lugar de hacerlo de forma "manual". Lógicamente, hay que prestarle mucha atención al código de pruebas para que efectivamente nos ayude a conseguir nuestro objetivo

EVALUACIÓN del proceso de pruebas y emitir un informe

Aplicamos las métricas adecuadas para comprobar si hemos alcanzado los objetivos de pruebas planificados

Importancia de las pruebas

¿por qué probamos?

- Necesitamos realizar pruebas porque somos falibles (cometemos errores)
- Durante el desarrollo, aproximadamente un 30-40% de las actividades están relacionadas con las pruebas

A los usuarios no les gustan los errores

Uno de los objetivos de éxito del proyecto es que el software satisfaga las expectativas del cliente

Si las expectativas del cliente no se satisfacen, éste se sentirá justificadamente agraviado

¿para qué probamos?

- Es FUNDAMENTAL realizar un BUEN proceso de pruebas si queremos que nuestro proyecto tenga **ÉXITO** (se entregue a tiempo, con el coste previsto y satisfaga las expectativas del cliente)

Principios fundamentales de las pruebas

Las pruebas muestran la **PRESENCIA de defectos** (no pueden demostrar la ausencia de los mismos. Si no se encuentra un defecto, no significa que no los haya)

Las pruebas **exhaustivas** son **IMPOSIBLES**

La paradoja del pesticida (los tests deben **revisarse** regularmente para ejercitar diferentes partes del programa)

Hay que probar **TAN PRONTO** como sea posible (el coste de reparar un defecto es directamente proporcional al tiempo que transcurre desde que se incurre en él hasta que se descubre)

Clustering de defectos (normalmente los defectos se "concentran" en un reducido número de módulos o componentes del programa

Las pruebas son **dependientes del contexto** (no es lo mismo probar un sistema de comercio electrónico, o el del lanzamiento de un cohete espacial)

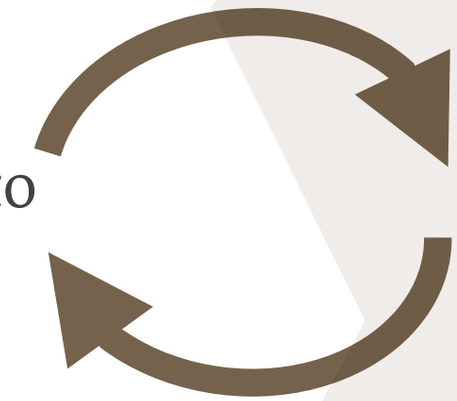
La **falacia de la ausencia de errores** (no es suficiente el encontrar defectos, el programa debe satisfacer las necesidades y expectativas del cliente)

TESTING Y DEBUGGING

Son procesos DIFERENTES!!

TESTING

El proceso de TESTING (pruebas) concluye cuando se **DETECTA un fallo** (failure) del programa (discrepancia entre el resultado esperado y el resultado real), lo cual es síntoma de que hay un defecto en el programa (bug). Los responsables de realizar las pruebas son los testers



DEBUGGING

El proceso de DEBUGGING (depuración) es una actividad del desarrollo que: **encuentra, analiza y elimina la CAUSA del fallo** de ejecución (failure), es decir, elimina el defecto que provoca el fallo de ejecución. Normalmente la depuración la realizan los desarrolladores

Estos dos procesos son necesarios (SIEMPRE) y complementarios. Siempre que depuramos algún defecto, hay que volver a repetir las pruebas (las veces que sea necesario) hasta asegurarnos de que hemos corregido la causa del fallo de ejecución.

OBJETIVOS DE LAS PRUEBAS

Tipos de pruebas dependiendo del OBJETIVO de las mismas

VERIFICACIÓN

- **Detectar problemas**

Se trata de buscar DEFECTOS en el programa que provocarán que éste no funcione correctamente (es decir ,según lo esperado, de acuerdo con los requerimientos especificados previamente)

VALIDACIÓN

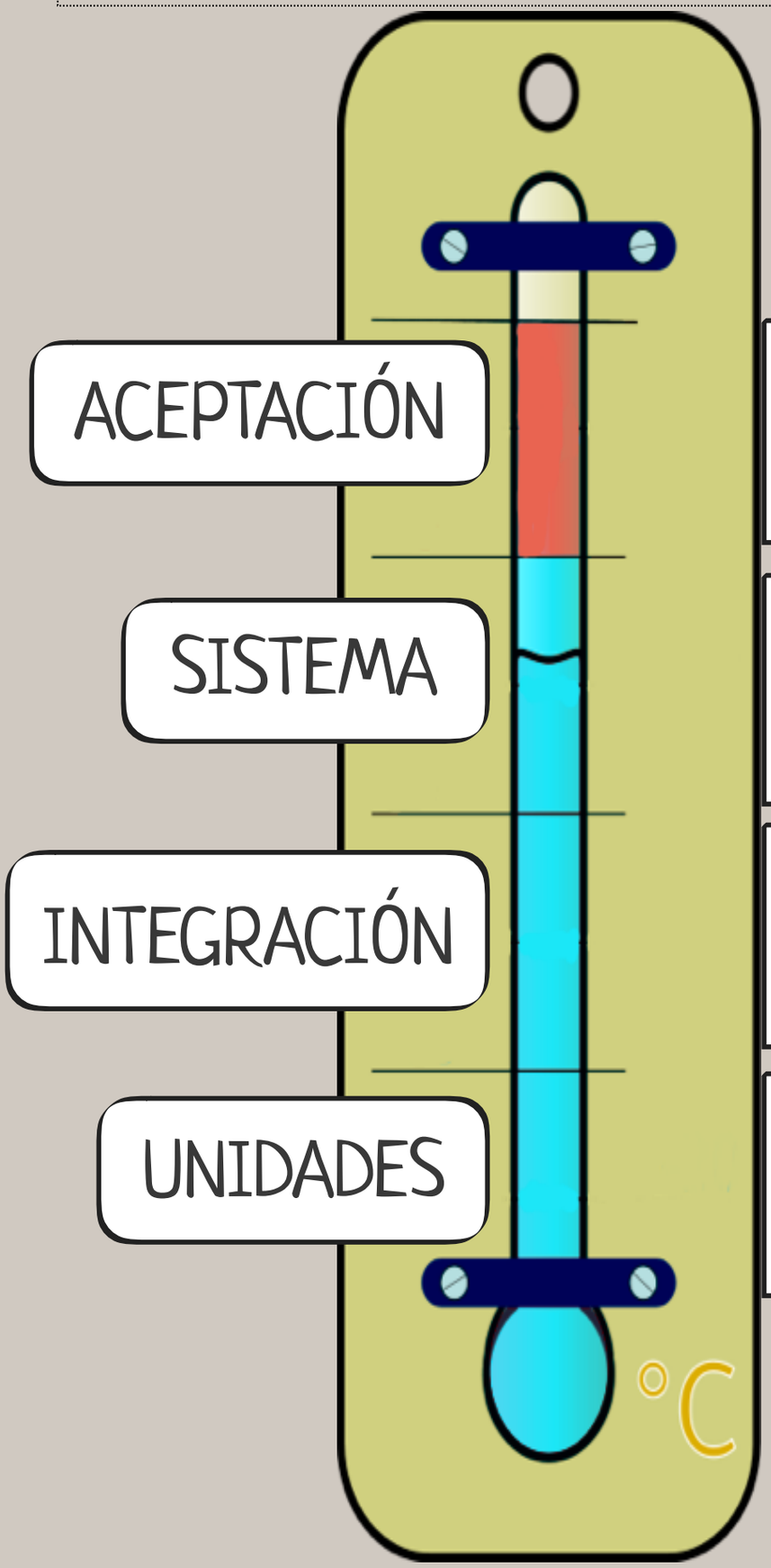
- **Juzgar la calidad del software o en qué grado es aceptable**

Se trata de ver si el producto desarrollado satisface las expectativas del cliente (comprobamos si lo que estamos construyendo es lo que el cliente quiere y/o necesita)

Estos dos procesos son necesarios (SIEMPRE) y complementarios (un producto puede funcionar correctamente, pero no satisfacer las expectativas del cliente (no es exactamente el producto que quería)

Niveles de pruebas

ver ISQTB Foundation Level Syllabus (2011)



Tipos de pruebas dependiendo del instante de TIEMPO del desarrollo que se realicen

Objetivo: valorar en qué grado el software desarrollado satisface las expectativas del cliente

REQUIERE los CRITERIOS DE ACEPTACIÓN

Objetivo: encontrar DEFECTOS derivados del COMPORTAMIENTO del sw como un todo

REQUIERE los REQUISITOS (funcionalidades) del sistema

Objetivo: encontrar DEFECTOS derivados de la INTERACCIÓN de las unidades probadas

REQUIERE establecer un ORDEN de las unidades a integrar

Objetivo: encontrar DEFECTOS en el código de las UNIDADES probadas

REQUIERE AISLAR el código de cada unidad a probar

En cada instante de tiempo tenemos que hacer tipos de pruebas DIFERENTES!!

TÉCNICAS DE PRUEBAS

ESTÁTICAS

- No requieren ejecutar código para detectar defectos en el software
- Pueden aplicarse en cualquier momento del desarrollo
- Ejemplos de defectos que pueden encontrarse: desviación de los estándares establecidos, defectos en los requerimientos, defectos en el diseño, mantenibilidad reducida, especificaciones de interfaces incorrectas…
- Reducen el coste de reparación de los defectos encontrados, ya que permiten detectarlos de forma temprana (en comparación con las pruebas dinámicas). Se centran en detectar las causas de los fallos de ejecución

En ambos casos, el objetivo es el mismo: encontrar DEFECTOS

Tipos de pruebas dependiendo de la TÉCNICA usada

DINÁMICAS

- Requieren ejecutar código para detectar defectos en el software
- Sólo pueden usarse si de dispone del código correspondiente

Trabajaremos fundamentalmente con técnicas DINÁMICAS

Estas dos técnicas son complementarias (hay defectos que sólo pueden descubrirse si se usa una de las dos)

Casos de prueba

Se usan en pruebas DINÁMICAS

Caso de prueba = dato concreto de entrada + resultado esperado

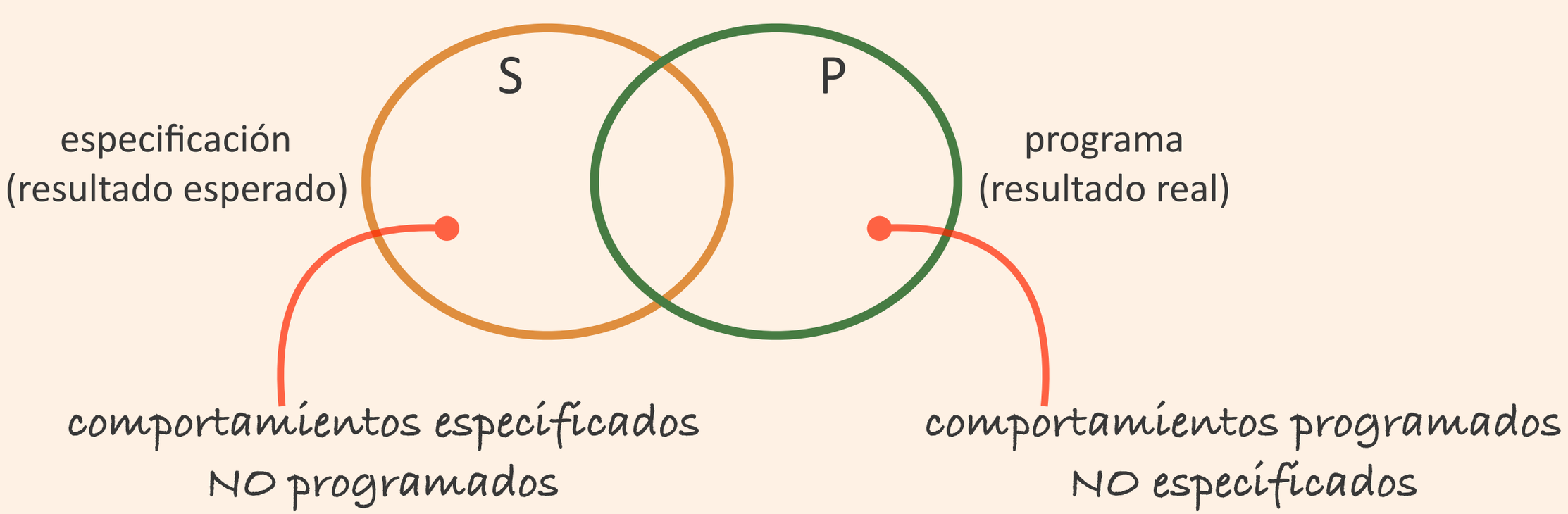
- Para realizar pruebas dinámicas, es ESENCIAL **determinar (diseñar)** un conjunto de CASOS DE PRUEBA para cada elemento a probar
- Un **caso de prueba** está formado por:
  - Datos CONCRETOS de entrada del elemento a probar
  - El resultado CONCRETO esperado, dadas las entradas anteriores
- La **ejecución** de un caso de prueba requiere:
  - Establecer las precondiciones sobre los datos de entrada (asunciones sobre lo que es cierto antes de ejecutar el caso de prueba)
  - Proporcionar los datos de entrada + el resultado esperado
  - Observar la salida (resultado real)
  - Comparar el resultado esperado con el resultado real
  - Emitir un informe (para poner de manifiesto si hemos detectado un fallo o no)

1. Piensa en algún ejemplo de caso de prueba
2. Indica algún ejemplo de precondición

Pruebas y comportamiento

S y P deberían ser idénticos!!!

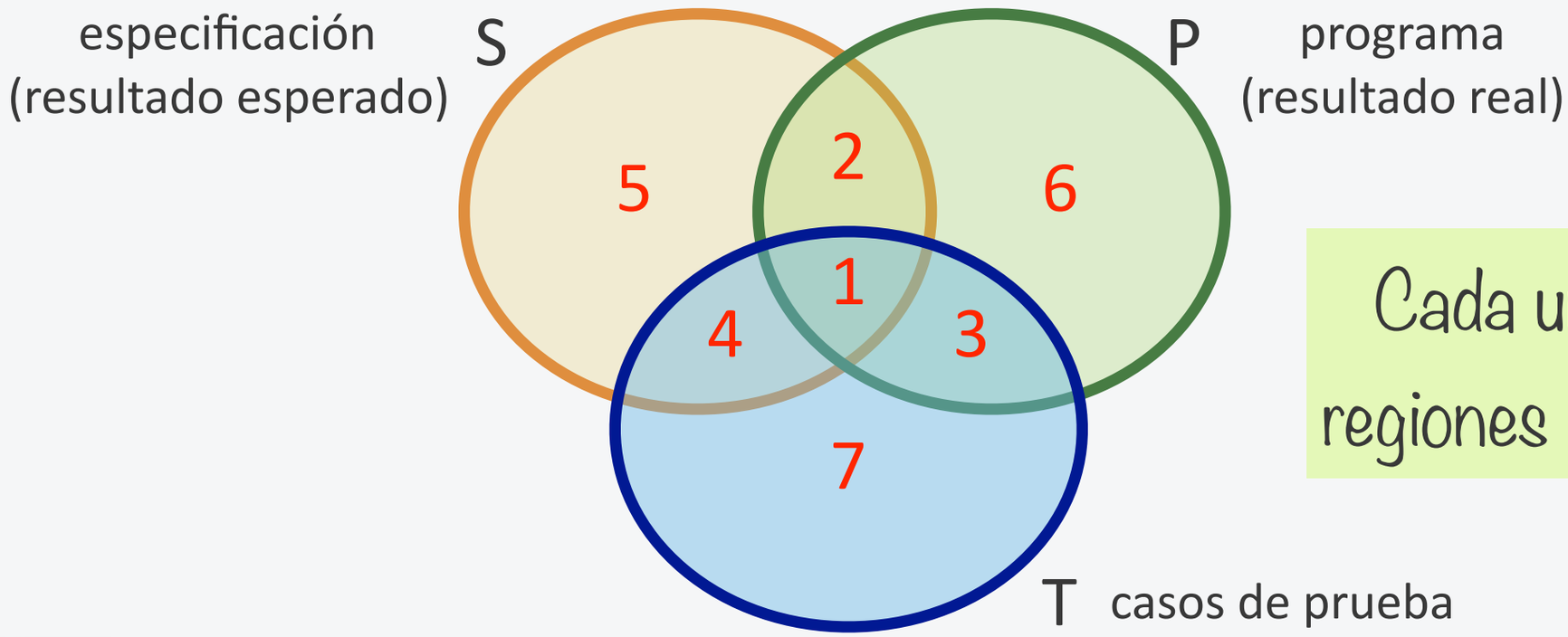
- Las pruebas conciernen fundamentalmente al comportamiento del elemento a probar: ¿qué debería hacer aquello que estoy probando?
- Supongamos un universo de comportamientos de programa. Dado un programa y su especificación, consideraremos:
  - el conjunto S de comportamientos especificados para dicho programa
  - el conjunto P de comportamientos programados



Estos son los problemas con los que se enfrenta un tester!!!

Comportamientos especificados, programados, y probados

- Incluyamos el conjunto T de comportamientos probados a la figura anterior:



Cada una de estas regiones es importante

Indica qué representan las regiones:

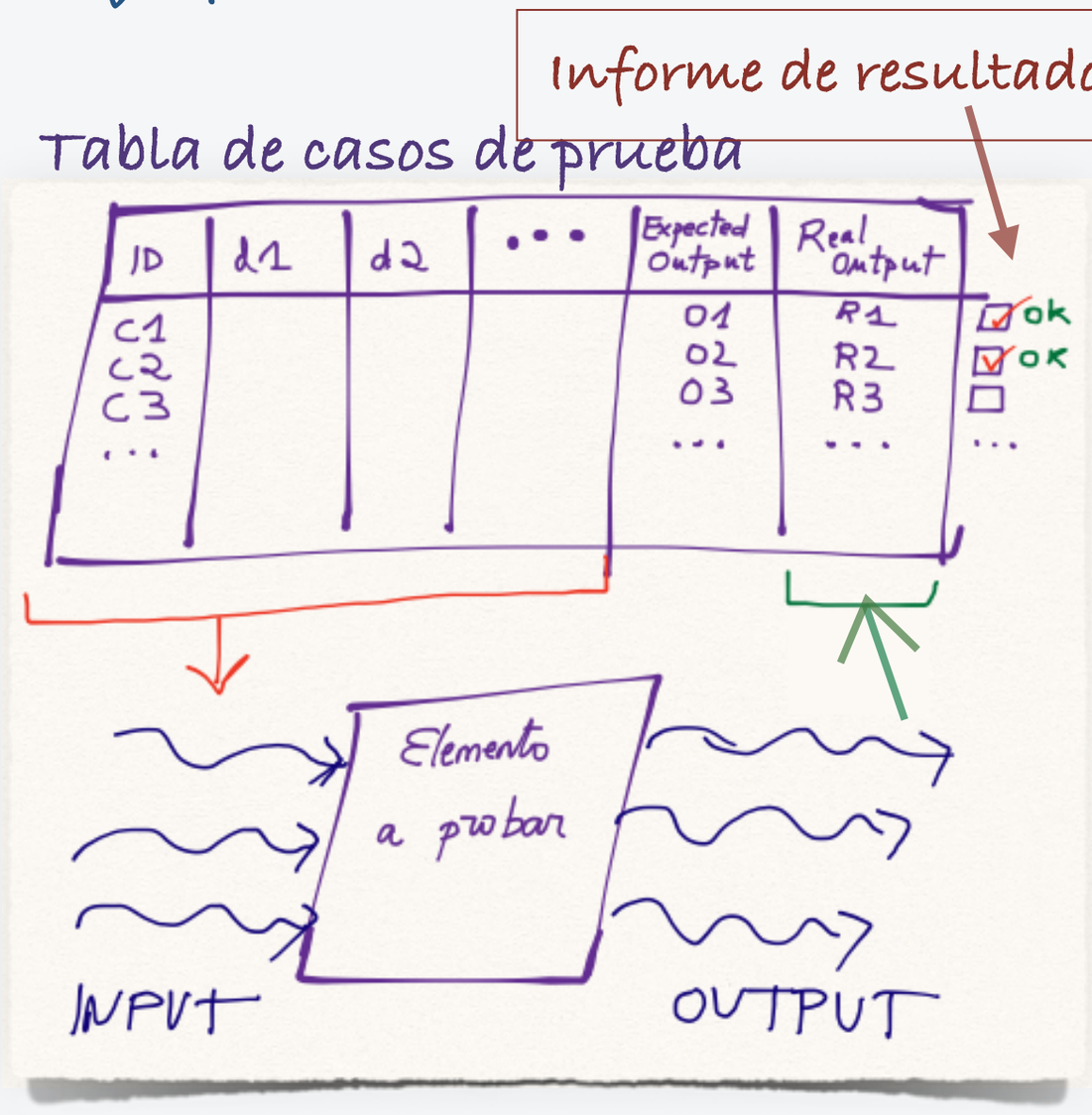
1. 2+5
2. 1+4
3. 3+7
4. 2+6
5. 1+3
6. 4+7

- ¿Qué puede hacer un tester para conseguir que la región 1 sea lo más grande posible?
  - ➡ Identificar el conjunto de casos de prueba utilizando algún método de DISEÑO de pruebas

Ejecución de casos de prueba

- Una vez identificados los casos de prueba, hay que implementar y **automatizar** su ejecución, para ello utilizaremos diferentes herramientas: JUnit, DbUnit, Selenium, Jmeter,…
- La **ejecución** de los casos de prueba nos permite obtener un "informe" con el resultado de los tests. Por ejemplo, si utilizamos JUnit, este informe será, : Pass, Failure, o Error, para cada test
- La ejecución de los casos de prueba forma parte del proceso de **construcción del sistema**. La construcción del sistema es una actividad que SIEMPRE va a estar presente en el proceso de desarrollo de un proyecto software, por lo que es importante entender en qué consiste dicho proceso

La columna de Resultado Real se "rellenará" cuando EJECUTEMOS los casos de prueba que previamente hemos diseñado



Para detectar errores necesitamos ejecutar los tests (y por lo tanto, el código a probar debe estar DISPONIBLE -> pruebas DINÁMICAS)

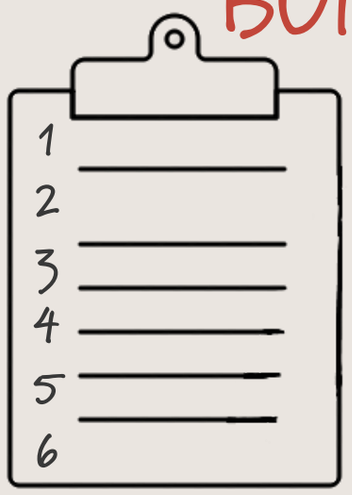
Construcción del sistema

Las pruebas formarán parte NECESARIAMENTE de nuestro proceso de construcción

- ¿Qué es una construcción del sistema (**build**)?
  - Es el proceso realizado para "reunir" todo el código fuente ("the process for putting source code together") y verificar que dicho software funciona como una unidad cohesiva.
  - El proceso de construcción de un sistema está formado por una **secuencia de acciones** definidas en uno o más “build scripts”. Un **build script** puede consistir en una secuencia de compilación, pruebas, empaquetado y despliegue (entre otras cosas)
- Las herramientas de construcción del sistema nos permiten automatizar el proceso de construcción a partir de build scripts
- Ejemplos de herramientas utilizadas para automatizar las construcciones del sistema:
  - Make: para lenguaje C
  - Ant, Maven, Graddle: para lenguaje Java

Nosotros usaremos MAVEN !!!

BUILD SCRIPT

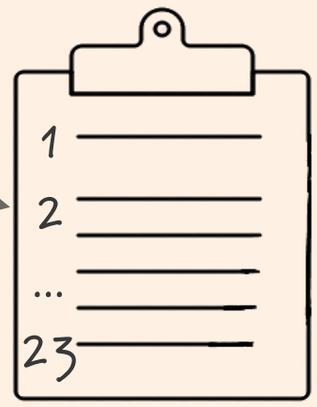


Contiene la SECUENCIA de acciones que se ejecutarán de forma automática (pulsando un botón)

MAVEN HERRAMIENTA PARA AUTOMATIZAR LA CONSTRUCCIÓN DE PROGRAMAS Java

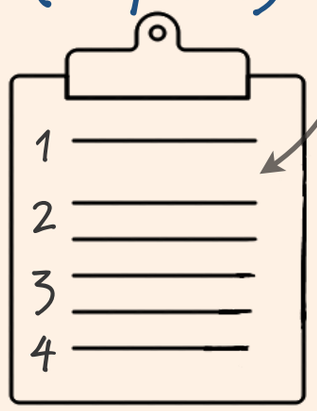
Maven estandariza la secuencia de **procesos lógicos** (FASES) de un build script. A dicha secuencia la denomina CICLO DE VIDA. Maven proporciona 3 ciclos de vida. Cada fase sólo pertenece a un único ciclo de vida.

ciclo de vida por DEFECTO (23 fases)



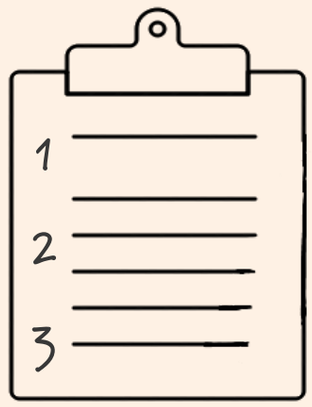
Cada ciclo de vida tiene unas goals asociadas por defecto a ALGUNA de sus fases

ciclo de vida SITE (4 fases)

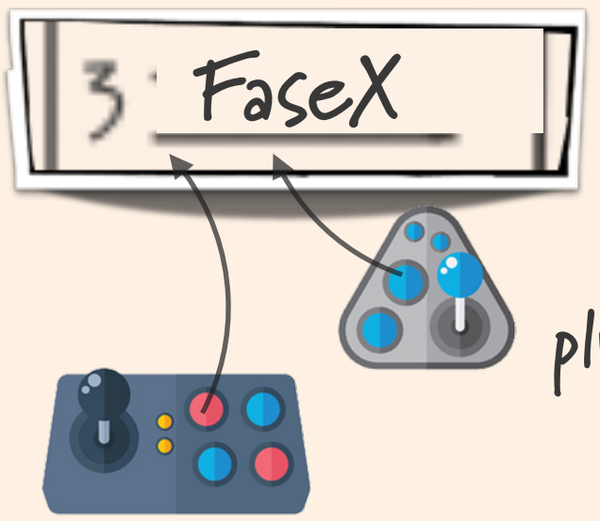


Un ciclo de vida EJECUTA siempre las fases de forma SECUENCIAL a través de sus goals asociadas

ciclo de vida CLEAN (3 fases)



EJEMPLOS de fases: compile, test, package, clean, install, deploy…

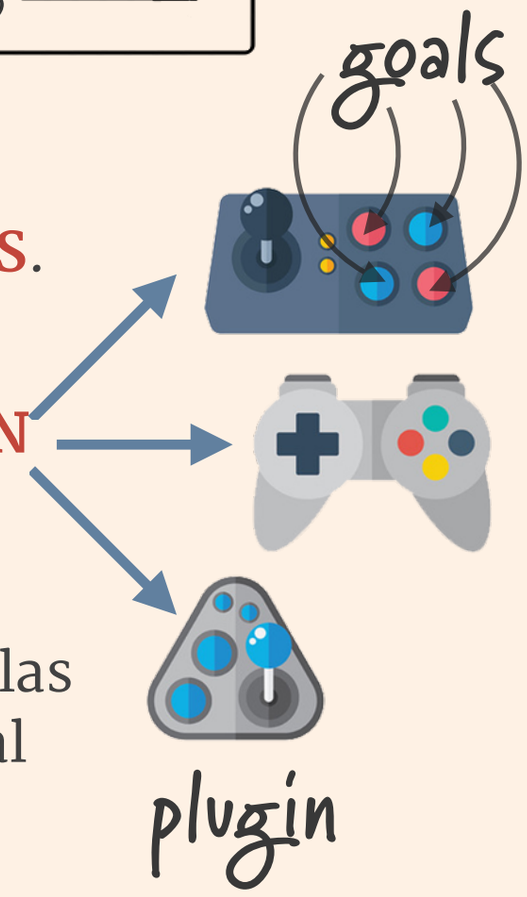


Una FASE PUEDE asociadas una o más **GOALS**. una goal es una ACCIÓN EJECUTABLE

Una goal SIEMPRE pertenece a algún **PLUGIN**

pluginP1:goalZ

En este caso la fase **"FaseX"** tiene asociadas las goals **"pluginP1:goalZ"** (la goalZ pertenece al plugin pluginP1), y **"plugin P2: goalK"**



Identificación de los artefactos Maven

Un artefacto Maven es cualquier FICHERO generado o usado por Maven

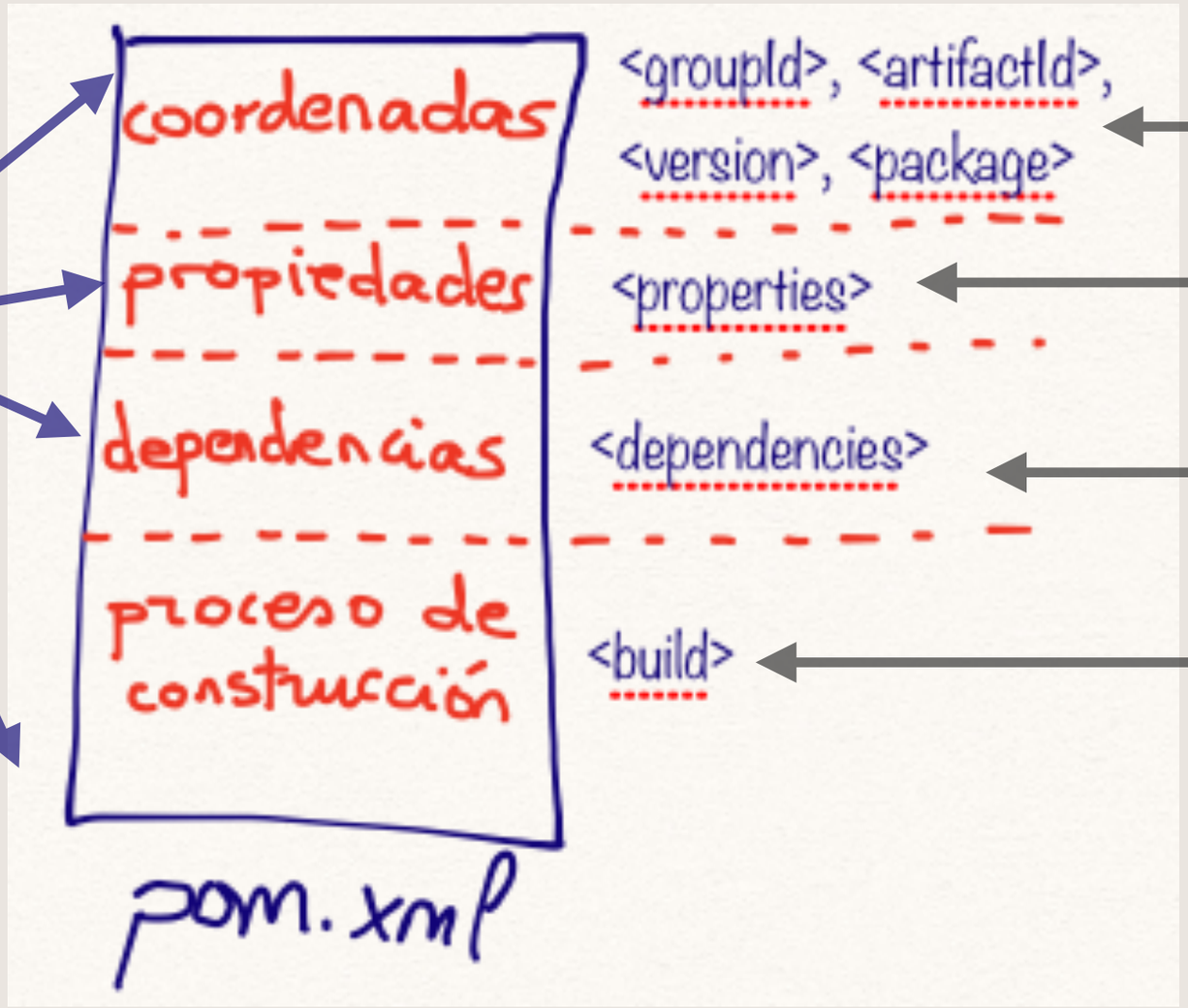
`groupId:artifactId:packaging:version`

- Cualquier artefacto utilizado por Maven se identifica por sus **COORDENADAS**:
  - **groupId**: es el identificador de Grupo (<groupId/>)
  - **artifactId**: es el identificador del artefacto (nombre del artefacto) (<artifactId/>)
  - **packaging**. es el empaquetado del artefacto (<package/>), por defecto el empaquetado es “jar”
  - **version**: es la versión del artefacto (<version/>)
- Ejemplos:
  - **org.apache.maven.plugins:maven-compiler-plugin:jar:3.2**
    fichero maven-compiler-plugin-3.2.jar que contiene la goal "compile" asociada por defecto a la fase con el mismo nombre "compile"
    (ver https://maven.apache.org/plugins/maven-compiler-plugin/plugin-info.html)
  - **mi.practica:practica7:jar:1.0.SNAPSHOT**
    fichero practica7-1.0.SNAPSHOT.jar con todos los .class de nuestro proyecto maven, se genera en la fase "package" cuando se construye el proyecto
  - **junit:junit:4.12**
    fichero junit-4.12.jar que contiene los .class necesarios para implementar los tests (importamos las clases desde el código fuente)
- Los artefactos Maven residen en **REPOSITORIOS**:
  - **remotos** (p.ej. http://mvn.repository.com)
  - **locales** (p.ej. **$HOME/.m2/repository**)
  - La identificación del artefacto se utiliza para almacenar y localizar físicamente el artefacto en el repositorio correspondiente
    - P.ej. $HOME/.m2/repository/junit/junit/4.12/junit-4.12.jar

Información relevante de la configuración

TODOS los proyectos Maven tienen un fichero pom.xml en su directorio raíz

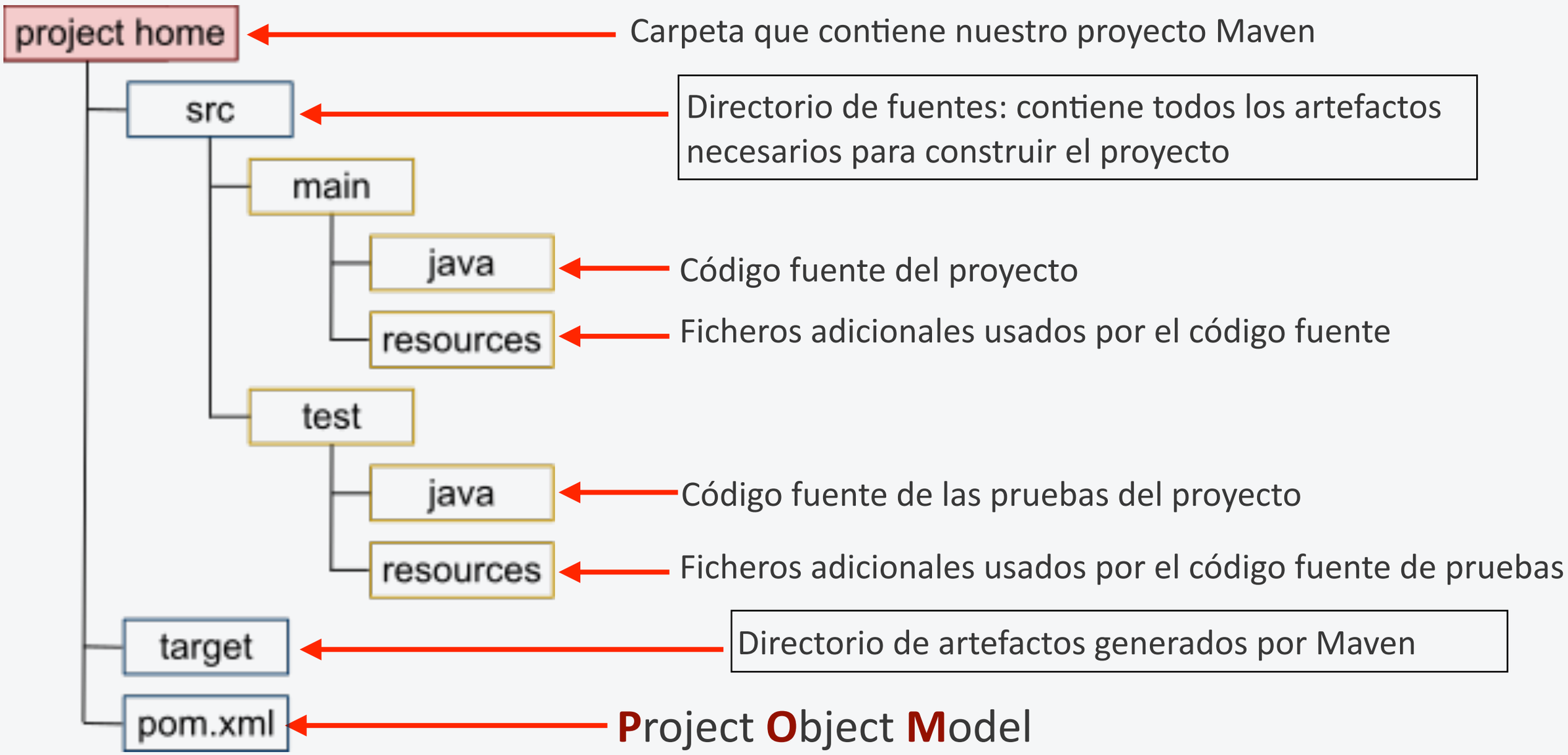
En el fichero pom.xml configuraremos, de forma declarativa, el build script utilizado por maven para construir el proyecto. Para ello utilizamos diferentes etiquetas xml



Estructura de un proyecto Maven

Maven estandariza la estructura de directorios de cualquier proyecto java. Un proyecto Maven no tiene por qué tener todos los directorios de la estructura estándar. Por ejemplo, si no necesitamos ningún fichero adicional al código podemos omitir la carpeta src/main/resources

Aquí mostramos PARTE de la estructura de directorios de Maven



Configuración por defecto y ejecución de Maven

Podemos utilizar la configuración por defecto (secuencia de goals asociadas por defecto a las fases de un ciclo de vida), o bien alterar dicha secuencia (añadir, quitar, modificar goals)

Para lanzar el proceso de construcción se utiliza el comando mvn:

mvn faseM (Se ejecutan TODAS las goals asociadas a las fases, desde fase1 hasta faseM)

mvn pluginX:goal5 (Se ejecuta sólo la goal goal5 del plugin pluginX)

Las GOALS que están asociadas por defecto a las fases de DEFAULT LIFECYCLE (cuando el empaquetado de nuestro proyecto es jar) son las siguientes:

Fase	plugin : goal	acciones
process-resources	`maven-resources-plugin: resources`	Copia *.* de /src/main/resources en target
compile	`maven-compiler-plugin: compile`	Compila *.java de /src/main/java
process-test-resources	`maven-resources-plugin: testResources`	Copia *.* de /src/test/resources en target
test-compile	`maven-compiler-plugin: testCompile`	Compila *.java de /src/test/java
test	`maven-surefire-plugin: test`	Ejecuta los tests unitarios
package	`maven-jar-plugin:jar`	Empaqueta *.class + recursos en un jar
install	`maven-install-plugin: install`	Copia el fichero jar en reposit. local
deploy	`maven-deploy-plugin: deploy`	Copia el fichero jar en reposit. remoto

Las fases se ejecutan siempre en el mismo orden comenzando desde la PRIMERA !!!

	FASES	PLUGIN : GOALS
1	validate	
2	initialize	
3	generate-sources	
4	process-sources	
5	generate-resources	
6	process-resources	resources:resour
7	compile	compiler:compile
8	process-classes	
9	generate-test-sources	
10	process-test-sources	
11	generate-test-resources	
12	process-test-resources	resources:testResources
13	test-compile	compiler:testCompile
14	process-test-classes	
15	test	surefire:test
16	prepare-package	
17	package	jar:jar
18	pre-integration-test	
19	integration-test	
20	post-integration-test	
21	verify	
22	install	install:install
23	deploy	deploy:deploy

EJEMPLO

mvn test-compile

Ejecuta las goals de las fases 1..13

Genera los .class de src/test/java

mvn compile

Ejecuta las goals de las fases 1..7

Genera los .class de src/main/java

```
ProyectoMaven
|____src
|  |____main ...
|  |____test ...
|
|____target
|  |____generated-sources ...
|  |____classes ...
|  |____maven-status
|  |  |____maven-compiler-plugin
|  |  |  |____compile...
|  |  |  |
|  |  |  |____testCompile ...
|  |____generated-test-sources
|  |____test-classes ...
|
|  |____surefire-reports ...
|  |
|  |____maven-archiver ...
|  |____Proyecto-Ejemplo-1.0-
|       SNAPSHOT.jar
|____pom.xml
```

mvn test

Ejecuta las goals de las fases 1..15

Ejecuta los .class de src/test/classes y genera un informe

mvn package

Ejecuta las goals de las fases 1..17

Genera el .jar del proyecto

Y AHORA VAMOS AL LABORATORIO...

Bitbucket git Maven IJ

P01-IntelliJ 〉 pom.xml — Add Configuration...

Project — P01-IntelliJ ~/Dropbox/PPS

- .idea
- src
  - main
    - java
      - ppss
        - DataArray
        - Matricula
        - Triangulo
    - resources
  - test
    - java
      - ppss
        - DataArrayTest
        - MatriculaTests
        - TrianguloTest
  - target
  - P01-IntelliJ.iml
  - pom.xml
- External Libraries
- Scratches and Consoles

Vista de proyectos

P01-IntelliJ

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
4           xsi:schemaLocation="http://maven.apache.org/POM/4
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>practica1.ppss</groupId>
8      <artifactId>P01-IntelliJ</artifactId>
9      <version>1.0-SNAPSHOT</version>
10
11     <properties...>
16
17     <dependencies>
18         <dependency>
19             <groupId>org.junit.jupiter</groupId>
20             <artifactId>junit-jupiter-engine</artifactId>
21             <version>5.3.1</version>
22             <scope>test</scope>
23         </dependency>
24     </dependencies>
25
26     <build>
27         <plugins>
28             <!-- JUnit 5 requires Surefire version 2.22.0
29
30             <plugin>
```

Editor

Maven

- P01-IntelliJ
  - Lifecycle
    - clean
    - validate
    - compile
    - test
    - package
    - verify
    - install
    - site
    - deploy
  - Plugins
  - Dependencies

Ventana Maven

Run: P01-IntelliJ [test]

```
INFO]
INFO] ------------------------------------------------------------------------
INFO] BUILD SUCCESS
INFO] ------------------------------------------------------------------------
INFO] Total time: 8.451 s
INFO] Finished at: 2019-01-24T23:24:23+01:00
INFO] ------------------------------------------------------------------------

rocess finished with exit code 0
```

Ejecución Maven (texto)

Maven test results: P01-IntelliJ-1.0-SNAPSHOT

- P01-IntelliJ (practica1.ppss:P01-IntelliJ:1.0-SNAPSH — 158 ms
  - ppss.DataArrayTest — 49 ms
  - ppss.MatriculaTests — 101 ms
  - ppss.TrianguloTests — 8 ms
    - testTipo_trianguloC1 — 2 ms
    - testTipo_trianguloC2 — 2 ms
    - testTipo_trianguloC3 — 1 ms

Ejecución Maven (gráfico)

Maven test results | 4: Run | 6: TODO | Terminal | Event Log

1: Project | Learn | 7: Structure | 2: Favorites | Ant Build | Maven | Database

Referencias bibliográficas

- Software Testing. A craftsman's approach. 4th edition. Paul C. Jorgensen (2014) ISBN-13: 978-1-4556-6068-0
  - Capítulo 1. A perspective on testing

- ISTQB. Foundations level syllabus (http://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html)

- Maven:
  - Sitio oficial: http://maven.apache.org