Universidad de Oviedo

ES
Español

Taxonomías del software
Patrones, estilos y tácticas

ARQUITECTURA
DEL SOFTWARE

2024-25

Jose Emilio Labra Gayo

Taxonomías del software

- Construcción y mantenimiento
  - Gestión de configuraciones
- Modularidad
  - Descomposición en tiempo de desarrollo
- Tiempo de ejecución
  - Componentes y conectores
- Integración
- Disposición
  - Empaquetamiento, distribución, despliegue
- Entorno de negocio y empresa

Construcción y mantenimiento Software

Gestión de configuraciones

Software: ¿producto ó servicio?

- *Software as a Product* (SaaP):
  - Software entregable
    - Modelo comercial: software vendido a clientes
    - Distribuido o descargado
  - Ejemplo: Microsoft Office
- *Software as a Service* (SaaS):
  - Software desplegado
    - Modelo comercial: los clientes se suscriben
    - Normalmente disponible en alguna URL
  - Ejemplo: Google docs

Gestión de configuraciones software

- Gestión de la evolución del software
  - Gestionar aspectos de construcción de *software*
  - Evolución y cambio del *software*
- Aspectos:
  - Identificar líneas base (*baseline*) e ítems de configuración
    - Línea base = producto sujeto a gestión
      - Contiene ítems de configuración: documentos, ficheros de código, etc...
  - Control y auditoria de configuraciones: Sistemas control de versiones
  - Gestión y automatización de la construcción
  - Herramientas trabajo en equipo y colaboración
  - Seguimiento de incidencias, propuestas de cambios, etc.

Construcción de software

- Repaso a metodologías
  - Tradicionales, iterativas, ágiles,...
- Herramientas de construcción
  - Lenguajes, herramientas de construcción, etc.

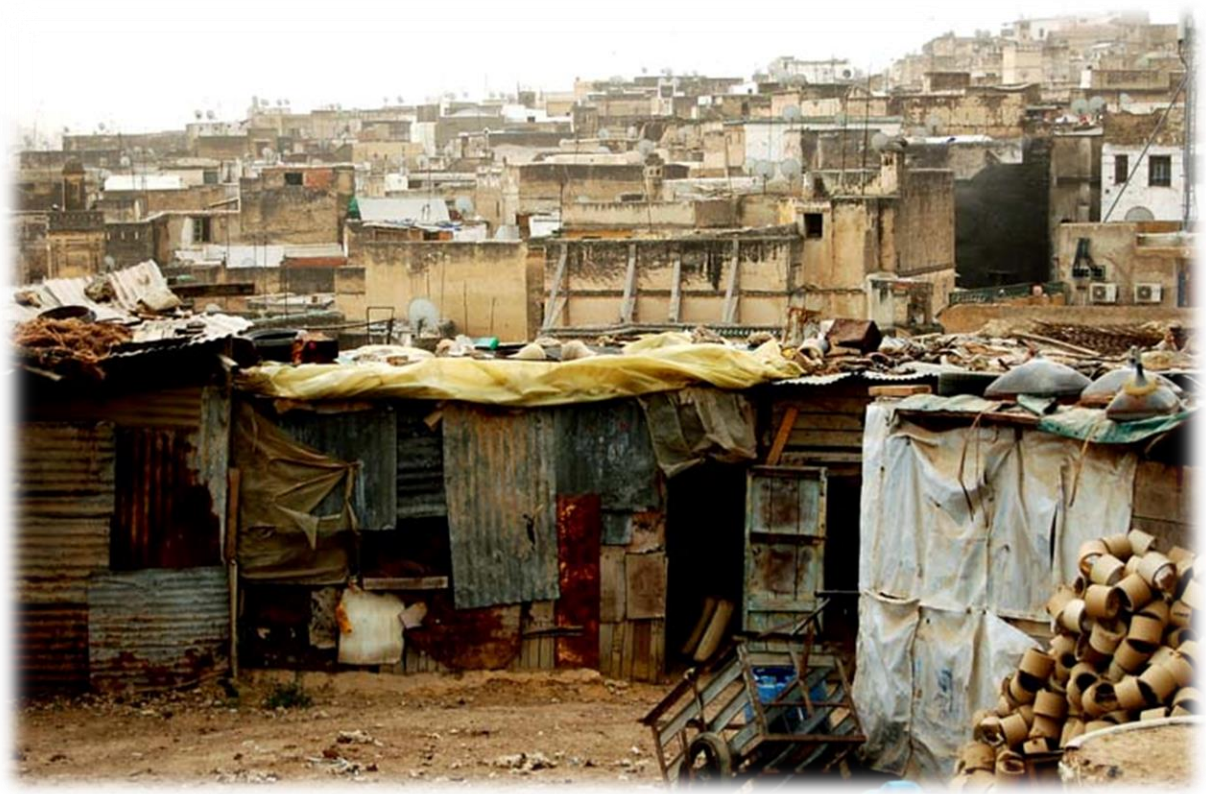
Incremental piecemeal

Crecimiento según necesidad

Codificar sin considerar la arquitectura

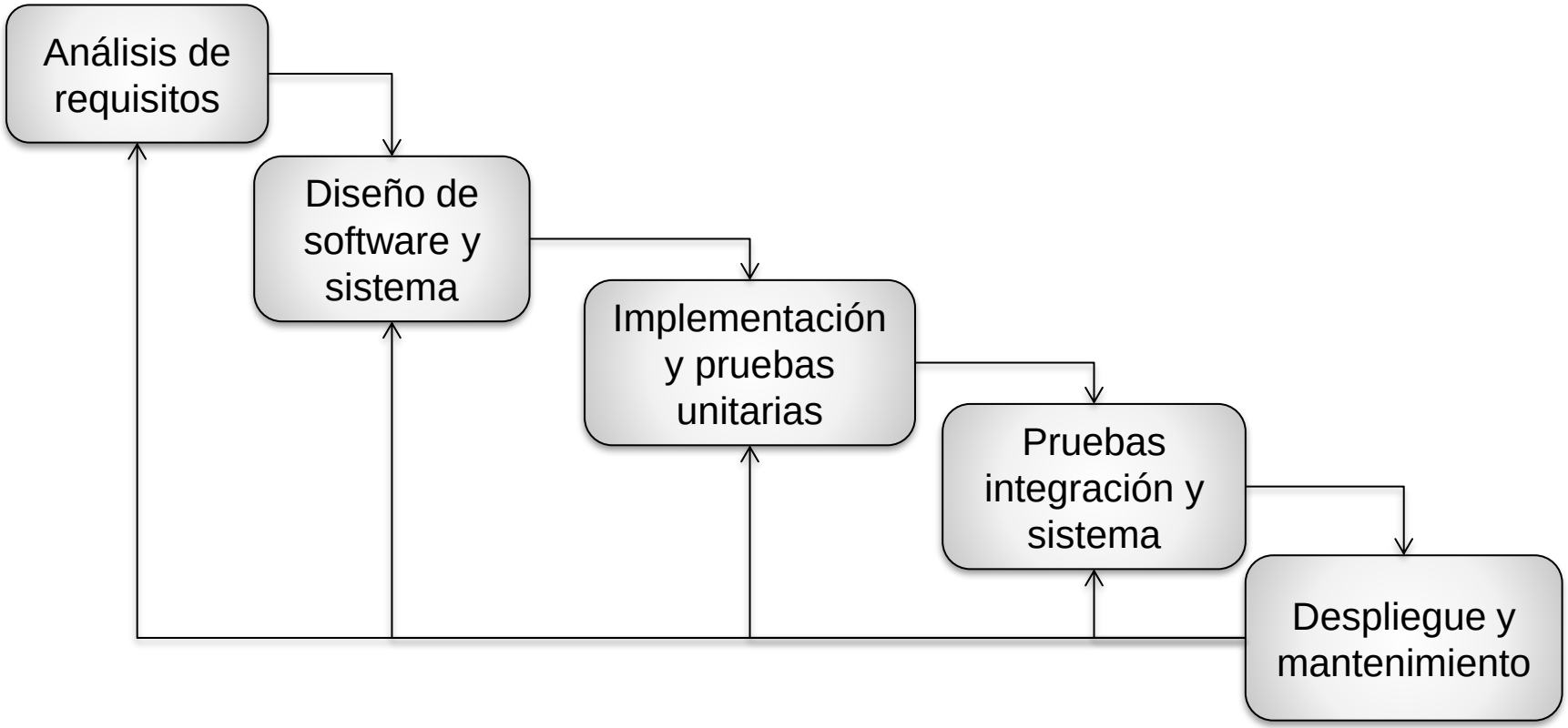
Software de usar y tirar

Limitaciones presupuestarias

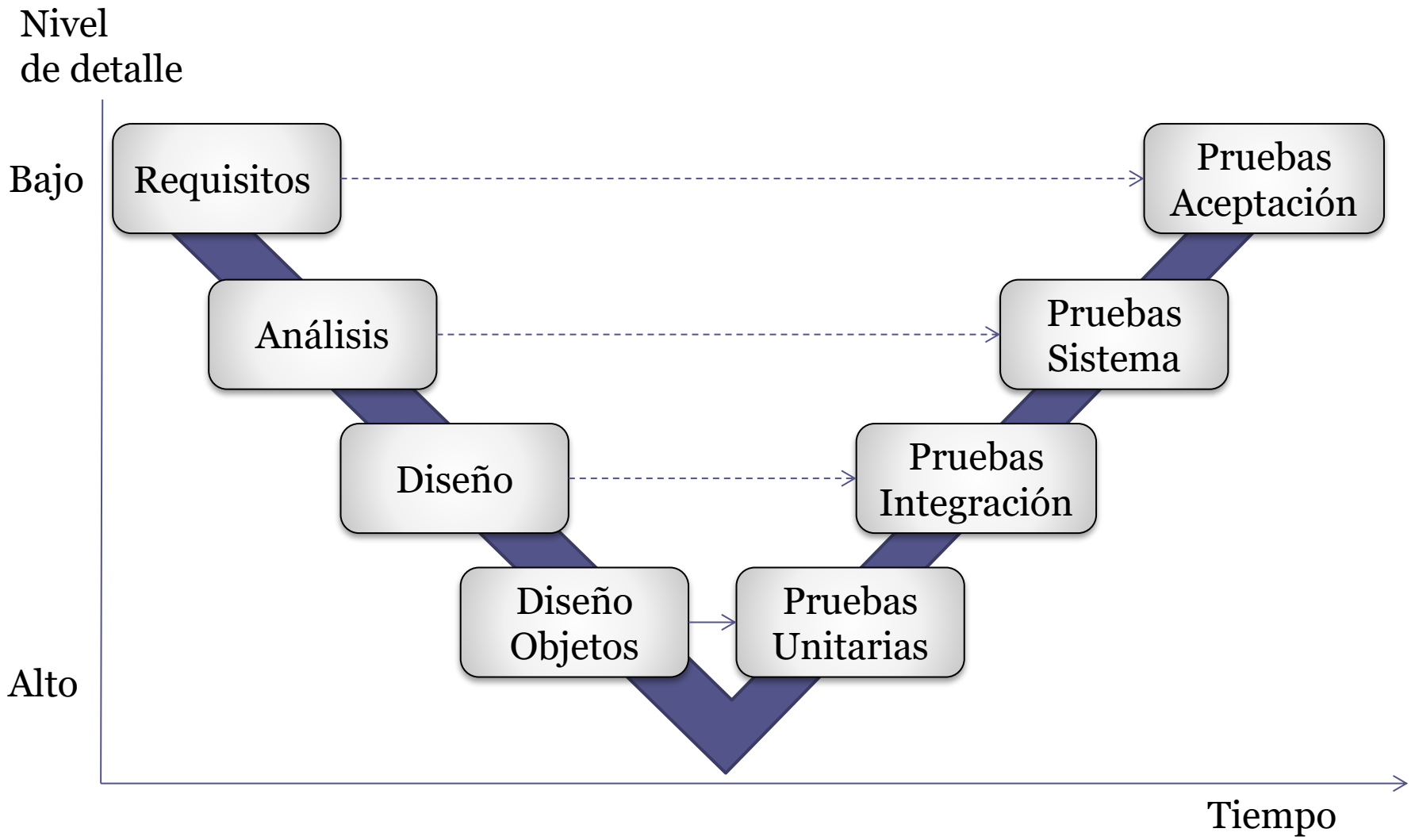


Cascada

Propuesto en años 70



Modelo en V

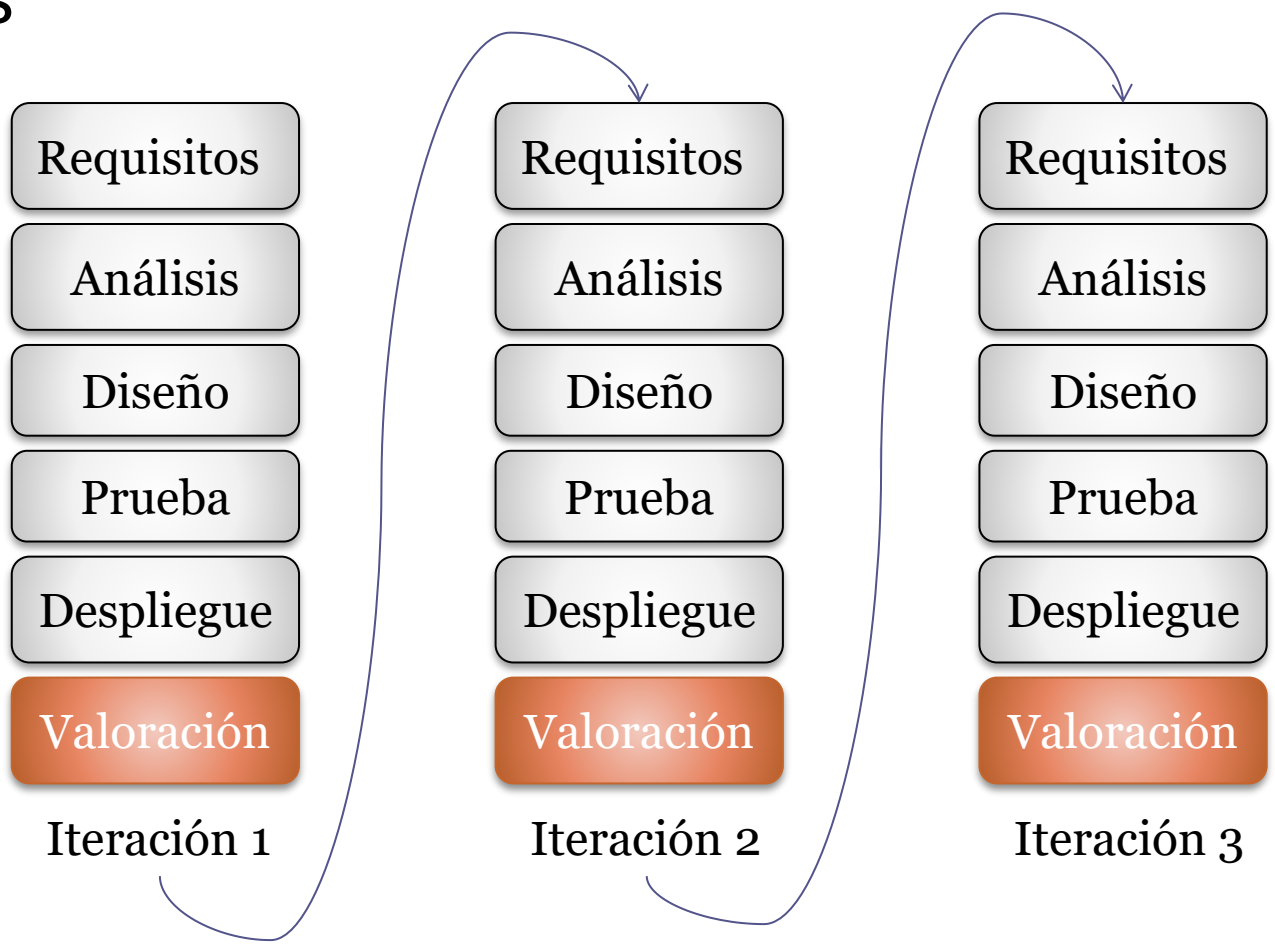

Nivel
de detalle
Bajo
Alto
Requisitos
Análisis
Diseño
Diseño
Objetos
Pruebas
Unitarias
Pruebas
Integración
Pruebas
Sistema
Pruebas
Aceptación
Tiempo

Big Design Up Front

- Antipatrón de modelos tradicionales
  - Demasiada documentación que nadie lee
  - Documentación diferente al sistema desarrollado
- Arquitectura degradada
- Sistemas que no son usados

Modelos iterativos

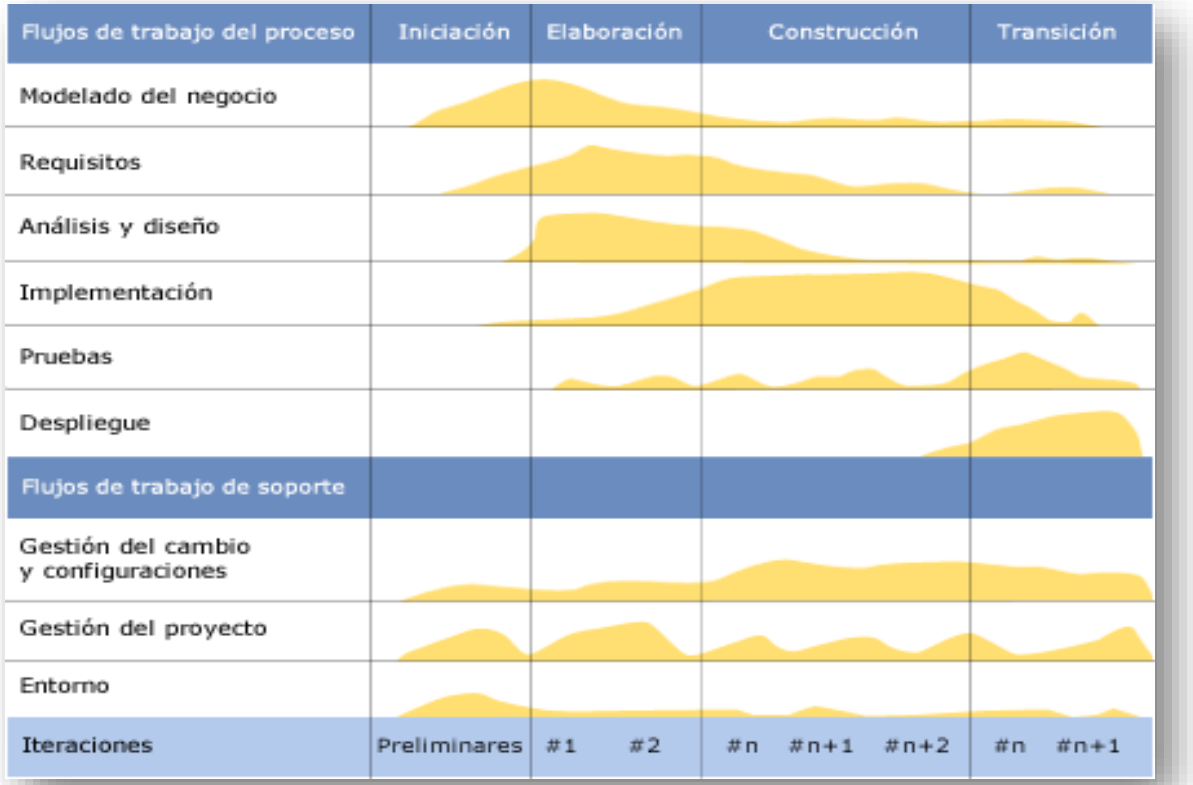
Basado en prototipos
Evaluación de riesgos



RUP (Rational Unified Process)

Uno de los modelos iterativos más utilizados

Proceso iterativo



Fuente: Wikipedia.
http://es.wikipedia.org/wiki/Archivo:Rup_espanol.gif

Métodos Ágiles

Numerosas variantes

- RAD (www.dsdm.org, 95)
- SCRUM (Sutherland & Schwaber, 95)
- XP - eXtreme Programming (Beck, 99)
- Feature driven development (DeLuca, 99)
- Adaptive software development (Highsmith, 00)
- Lean Development (Poppendieck, 03)
- Crystal Clear (Cockburn, 04)
- Agile Unified Process (Ambler, 05)
- . . .

Métodos ágiles

Manifiesto ágil (www.agilemanifesto.org)

Individuos e interacciones	sobre	Herramientas y procesos
Software que funcione	sobre	Documentación
Colaboración con cliente	sobre	Negociación de contrato
Responder al cambio	sobre	Seguimiento de un plan

Métodos ágiles

- Realimentación
  - Ajustes constantes en el código
- Minimizar riesgo
  - Software en intervalos cortos
  - Iteraciones de horas o días
  - Cada iteración pasa todo el ciclo de desarrollo

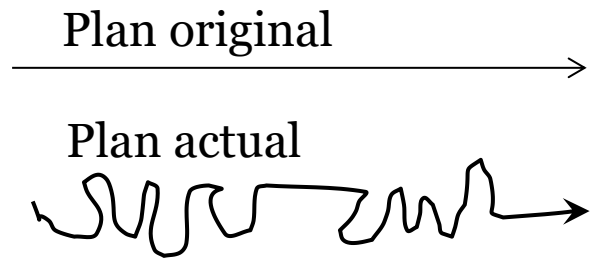
Métodos ágiles

Algunas prácticas (XP)

1. Planificaciones cortas
2. Pruebas
3. Programación en parejas (revisiones de código)
4. Refactorización
5. Diseño simple
6. Propiedad de código compartida
7. Integración continua
8. Cliente en lugar de desarrollo
9. Entregas pequeñas
10. Horarios *normales*
11. Estándares de codificación

Métodos ágiles

1. Planificaciones cortas
   - Después de cada iteración, volver a planificar
   - Requisitos mediante historias de usuario
     - Descripciones breves (Tamaño tarjeta)
       - Objetivos priorizados por clientes
       - Riesgo y recursos estimados por desarrolladores
     - Historias de usuario = pruebas aceptación
   - Preparación para el cambio



Métodos ágiles

2.- Utilización de pruebas

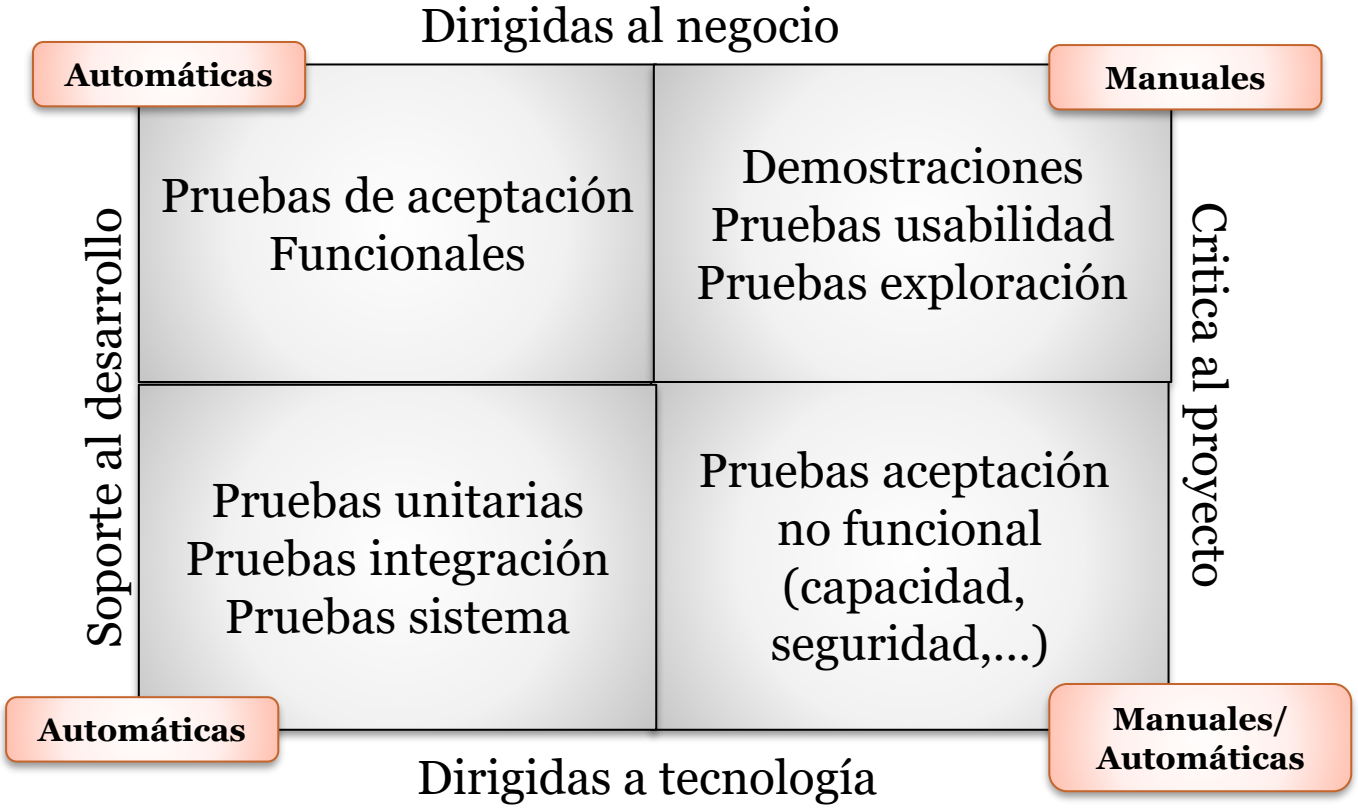
- Escribir pruebas incluso antes del código
- Inicialmente el código va a fallar
- Objetivo: pasar las pruebas
- Resultado:
  - Batería de pruebas automáticas (test-suite)
  - Facilita la refactorización

Métodos ágiles: tipos de pruebas

- Pruebas unitarias
  - Probar cada unidad separadamente
- Pruebas de integración
  - *Smoke testing*
- Pruebas de aceptación
  - Pruebas con historias de usuario
- Pruebas de capacidad/rendimiento
  - Pruebas de carga
- Pruebas de regresión
  - Chequear que los cambios nuevos no introducen nuevos errors, o regresiones

Métodos ágiles – Tipos de pruebas

Desarrollo vs proyecto
Negocio vs tecnología
Automáticas vs manuales



Métodos ágiles - Pruebas

Behaviour-driven development (BDD)

- Pruebas a partir de historias de usuario
- Deben escribirse junto con cliente
  - Herramientas: Cucumber, JBehave, Specs2,...
- Sirven como contrato
- Historias permiten medir progreso

```
Feature: Buscar cursos
  Para mejorar el uso de los cursos
  Los estudiantes deberían ser capaces de buscar cursos

  Scenario: Búsqueda por asunto
    Given hay 240 cursos que no tienen el asunto “Biología”
    And hay 2 cursos A001, B205 que tienen el asunto “Biología"
    When Yo busco el asunto “Biología"
    Then Yo debería ver los cursos:
      | Código |
      | A001   |
      | B205   |
```

Métodos ágiles - Pruebas

Principios FIRST

- F - Fast
  - La ejecución de pruebas debe ser rápida
- I - Independent:
  - Los casos de prueba son independientes entre sí
- R - Repeatable:
  - Tras ejecutarlos N veces, el resultado debe ser el mismo
- S - Self-checking
  - Se puede comprobar si se cumplen automáticamente, sin intervención humana
- T - Timely
  - Pruebas escritos al mismo (o antes) tiempo que código

Métodos ágiles - Dobles de pruebas

Objetos *Dummy:*

- Se pasan pero no se utilizan

Objetos *falsos (fake):*

- Tienen implementación parcial

Stubs:

- Respuestas precocinadas a ciertas preguntas

Espías: stubs que pueden registrar cierta información para depuración

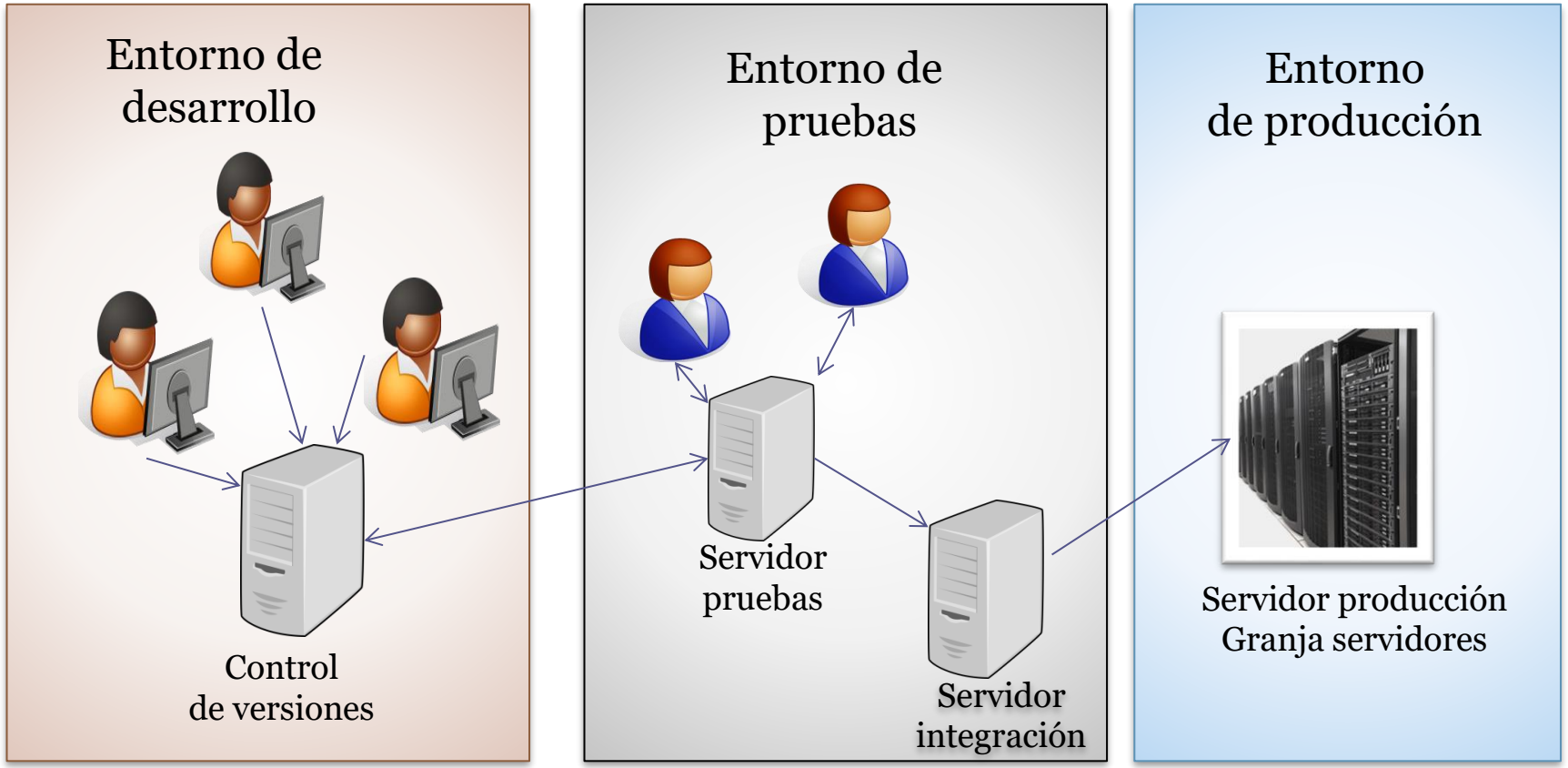
Mocks: simulan comportamiento de otros objetos

- Programados con ciertas expectativas sobre qué tipo de llamadas deben recibir

Fixtures. Elementos fijos de soporte a las pruebas

- *Ej. Bases de datos con ciertas entradas, determinados ficheros, etc.*

Entornos



Entorno de ensayo (*staging*) también se utiliza en ocasiones

Métodos ágiles

3. Programación en parejas
   - 2 ingenieros software trabajan juntos en un ordenador
     - El *conductor* maneja el teclado y crea implementación
     - El *observador* identifica fallos y da ideas
   - Los roles se intercambian cada cierto tiempo

Pull requests: antes de aceptar cambios, el código puede ser revisado

Métodos ágiles

4. Diseño simple
   - Reacción a Big Design Up Front
   - Diseño más simple que funcione

Documentación automatizada
   - JavaDoc y similares

Métodos ágiles

5. Refactorización

- Mejorar diseño sin cambiar la funcionalidad
  - Simplificar código (eliminar código duplicado)
  - Buscar activamente oportunidades de abstracción
- Pruebas de regresión
  - Se basa en la batería de pruebas

Métodos ágiles

6. Propiedad colectiva del código

- El código pertenece al proyecto, no a un ingeniero particular
- A medida que los ingenieros desarrollan, deben poder navegar y modificar cualquier clase
  - Aunque no la hayan escrito ellos
  - Evitar fragmentos de una única persona

Métodos ágiles

7. Integración continua

- Cada pareja escribe sus propios casos de prueba y trabaja para satisfacerlos
  - Pasar 100% de casos de prueba
  - Integrar
- El proceso debe realizarse 1 ó 2 veces al día
  - Objetivo: evitar *integration hell*

Integración continua

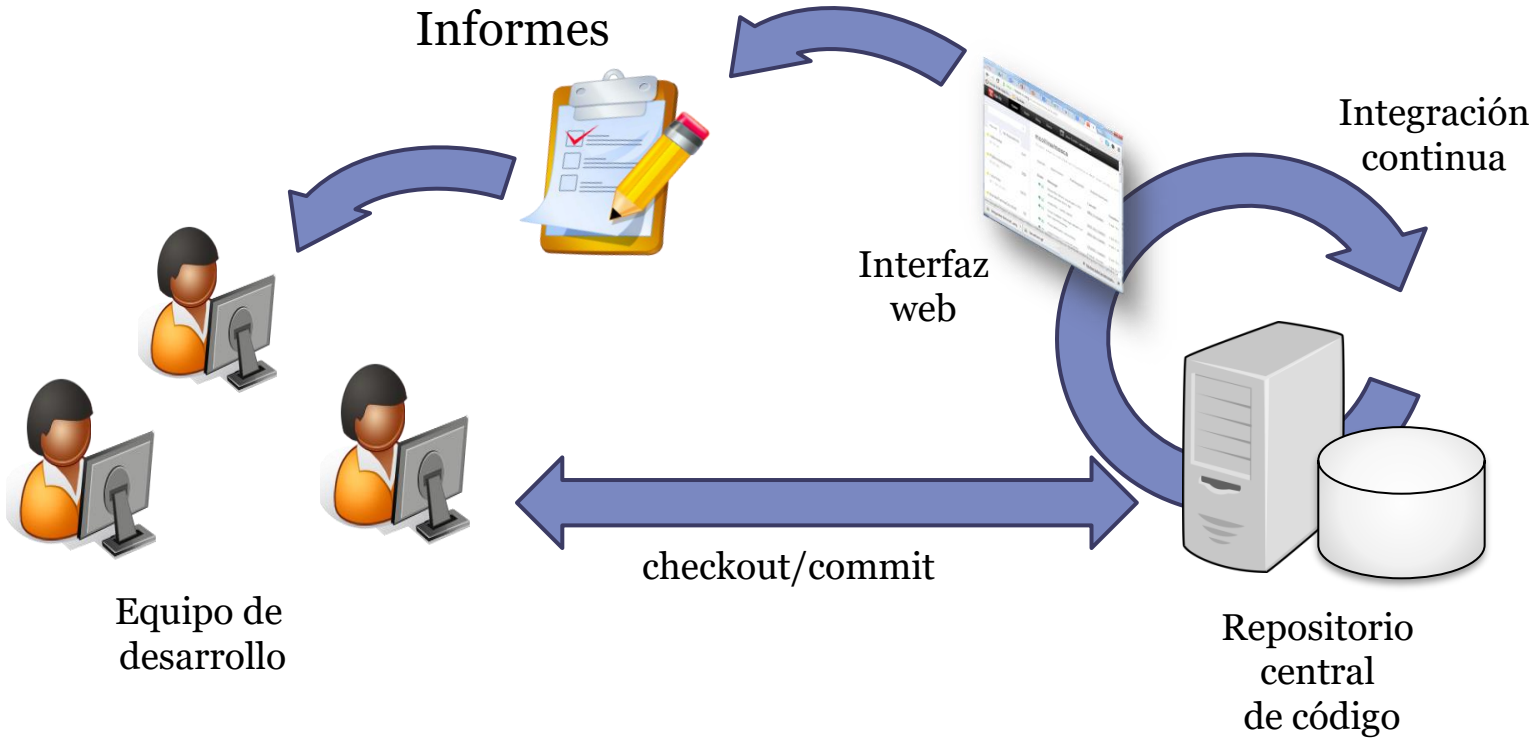
Mejores prácticas:

- Mantener repositorio de código
- Automatizar la construcción
- Hacer que la construcción pueda probarse
- Todo el mundo realiza commits a línea base
- Todo commit es construido
- Mantener la construcción rápida
- Probar en una replica del entorno de producción
- Facilitar la obtención de los últimos entregables
- Todo el mundo puede ver los resultados de la última construcción
- Automatizar despliegue

Integración continua

Herramientas

Hudson, Jenkins, Travis, Bamboo, Github Actions



Cliente en lugar de desarrollo

Cliente disponible para clarificar historias de usuarios y tomar decisiones críticas de negocio

Ventajas

- Desarrolladores no realizan suposiciones
- Desarrolladores no tienen que esperar para decisiones
- Mejora la comunicación

Entrega continua - continuous delivery

- Pequeñas *releases*
- Tan pequeñas como sea posible ofreciendo valor al usuario
  - Obtener realimentación temprana del cliente
- Modelos de entrega
  - Entregar algo cada cierto tiempo (noche/semana/...)
  - Entrega continua y automatizada
    - Delivery pipeline

Métodos ágiles

10. Horarios *normales*
    - 40h/semana = 40h/semana
- Evitar horas extra
- Programadores cansados escriben código pobre
    - A largo plazo ralentiza el desarrollo

Métodos ágiles

11. Código limpio

- Facilitar modificación de código por otras personas
- Utilizar buenas prácticas
  - Estilos y normas de codificación
  - Evitar *code smells*
- Manifiesto *software craftmanship*
- Libros *(Robert C. Martin)*
- *Clean Code*
- *Clean architecture*

https://manifesto.softwarecraftsmanship.org/#/es

Fuente: Clean Code. Robert Martin

Métodos ágiles

- Scrum
  - Gestión de proyectos/personas
  - División de trabajo en sprints
    - Reunión diaria de 15'
    - Backlog del producto
- Kanban
  - Modelo *lean* (esbelto)
    - Desarrollo Just in Time
    - Limitar cargas de trabajo

Gestión de configuraciones

Gestión de configuraciones

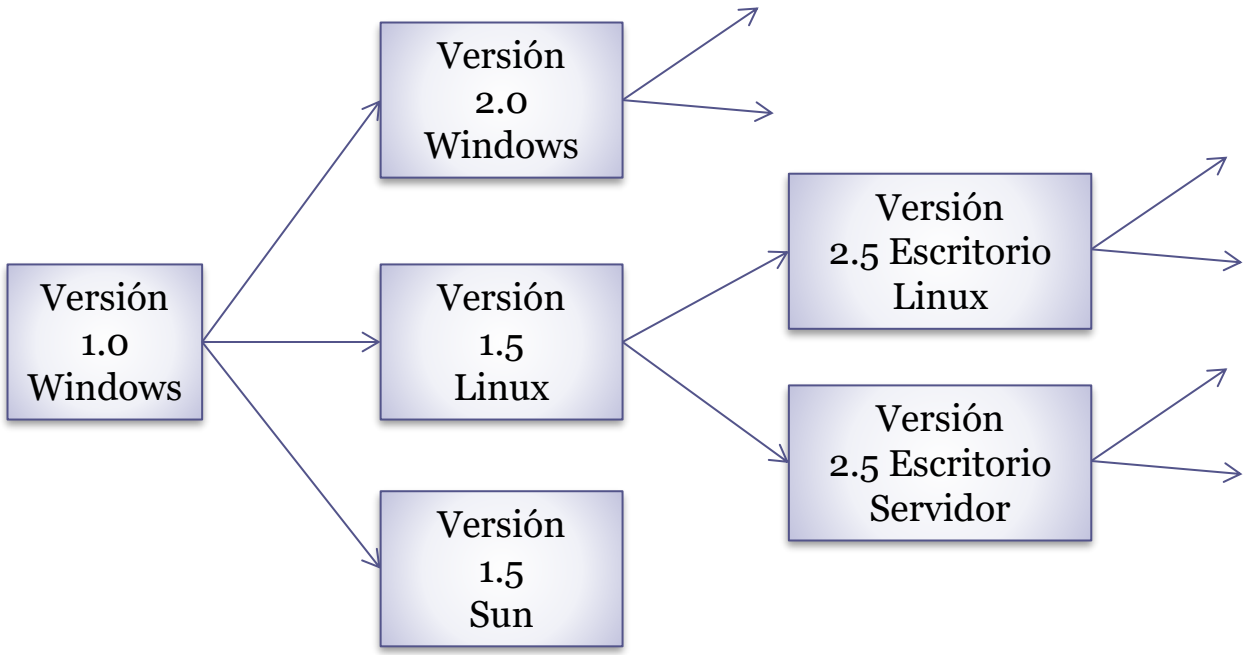
- Diferentes versiones de software
  - Funcionalidades nuevas o diferentes
  - Corrección de *bugs*
  - Nuevos entornos de ejecución
- Gestión de configuraciones: gestión de la evolución del software
  - Cambios del sistema = actividades en equipo
  - Costes y esfuerzo necesarios

Control de versiones

- Sistemas que gestionan las diferentes versiones del software
  - Acceso a todas las versiones del sistema
    - Facilidad para volver atrás
    - Diferencias entre versiones
  - Código colaborativo
  - Facilidad para gestión de ramificaciones
  - Metadatos
    - Autor de la versión, fecha actualización, etc.

Baseline

Baselines (línea de referencia): Software que es sometido a gestión de configuraciones.



Releases y versiones

Versión: instancia de un sistema funcionalmente distinta de otras instancias

Release (entregable): instancia de un sistema que es distribuida a usuarios externos al equipo de desarrollo.

Puede ser considerado un producto final

Nombres habituales de versiones

- Pre-alfa
  - Antes de las pruebas
- Alfa
  - En pruebas
- Beta (o prototipo)
  - Pruebas por usuarios
    - Beta-tester: usuario que hace pruebas
- Release-candidate
  - Versión beta que podría ser producto final

Otros esquemas de nombres

- Utilizar algunos atributos
  - Fecha, creador, lenguaje, cliente, estado,...
- Nombres reconocibles
  - Ganimede, Galileo, Helios, Indigo, Juno,...
  - Precise Pangolin, Quantal Quetzal,...
- Versioneado semántico (http://semver.org)
  - MAJOR.MINOR.PATCH (2.3.5)
    - MAJOR: cambios incompatibles con versión anterior
    - MINOR: nueva funcionalidad compatible con versión anterior
    - PATH: Reparación de bugs compatible con versión anterior
  - Versión 0 (inestable)
  - Pre-release: 2.3.5-alpha

Publicación de entregables

Una *release* supone cambios de funcionalidad

Planificación

- Publicar una *release* no es barato
- Los usuarios no suelen querer nuevas *releases*
- Factores externos:
  - Marketing, clientes, hardware, ...
- Modelo ágil: *releases* muy frecuentes
  - Utilizando integración continua se minimiza el riesgo

Publicación de entregables

Una release no es sólo software

- Ficheros de configuración
- Ficheros de datos necesarios
- Programas de instalación
- Documentación
- Publicidad y empaquetamiento

Continuous delivery

- Continuous delivery/entrega continua
  - Entregas rápidas para obtener feedback lo antes posible
    - Utilización de TDD e integración continua
    - Deployment pipeline (canal de despliegue)
  - Ventajas:
    - Afrontar el cambio
    - Minimizar riesgos de integración

Filosofía Wabi-sabi
Aceptar la imperfección
Software Suficientemente bueno (Good enough)

DevOps

Unir ***Dev****elopment* y ***Op****eration**s***

Cambio cultural en el que el mismo equipo afronta las fases:

- Codificar (code): Desarrollo y revisión de código, Integración continua
- Construir (build): Control de versiones, construcción
- Probar (test)
- Empaquetar: Gestión de artefactos
- Release: automatización de versiones
- Configurar y gestionar
- Monitorizar: Rendimiento, experiencia del usuario

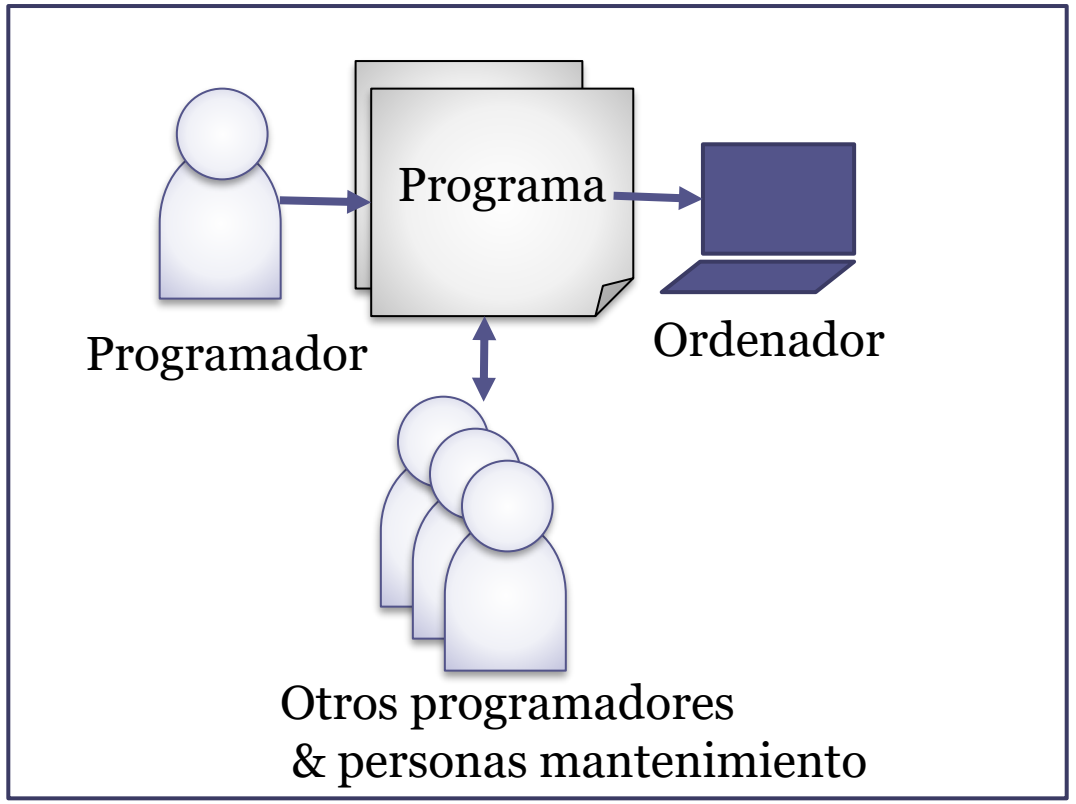
Herramientas de construcción

Lenguajes de construcción

- Lenguajes de configuración
  - Definiciones de recursos (JSON, XML, Turtle)
    - Ejemplos: .travis.yml, package.json, pom.xml
- Lenguajes de *scripting*
  - Escritos shell/batch
- Lenguajes de programación
  - Ejemplos: Java, Javascript,...
- Lenguajes visuales
  - Ejemplos: scratch, blender, ...
- Formales
  - Ejemplos: B-trees, Z language, OCL, ...

Aspectos de codificación

- Convenciones de nombres
  - Importantes para otros programadores, mantenimiento...
- Clases, tipos, variables, constantes con nombre...
- Gestión de errores
- Organización código fuente
  - Paquetes, directorios...
  - Dependencias
    - Librerías importadas
- Documentación del código
  - Javadocs, jsdoc...



Pruebas

Pruebas unitarias

Integración

Capacidad

Regresión

. . .

Recomendación:

- Separar código de pruebas y dependencias de código de producción

Construcción para reutilización

- Parametrización
  - Añadir parámetros
    - Bad smell: números mágicos en código
  - Ficheros y recursos de configuración
  - Variables de entorno
- Compilación condicional
- Encapsulación
  - Separar interfaz de implementación
    - *Bad smell*: partes internas públicas en librerías
- Empaquetamiento
  - Antipatrón: tareas manuales en empaquetamiento
- Documentación
  - Importante: Documentación API

Construir reutilizando

- Seleccionar unidades reutilizables
  - Componentes externos (COTS, FOSS)
- Gestión de dependencias
  - <ver más adelante>
- Gestión de actualizaciones
  - ¿Qué ocurre cuando otras librerías se actualizan?
- Temas legales
  - ¿Realmente puedo utilizar esa librería?
    - ¿Para productos comerciales?
      - Cuidado con librerías GNU
    - ¿Está la librería bien mantenida?

Herramientas de construcción

- Editores de texto
  - vi, emacs, Visual Studio Code, Sublime,....
- Integrated Development Environments (IDEs)
  - Ejemplos: IntelliJ, Eclipse
- Constructores de Graphical User Interface (GUI) Android Studio UI Editor, QtEditor,...
- Herramientas para asegurar la calidad (QA)
  - Test, analysis, ...<Ver siguiente trasparencia>

Herramientas para asegurar calidad

- Pruebas
  - xUnit, marcos de pruebas (mocha)
  - Lenguajes de aserciones (chai)
  - Herramientas de cobertura
- Aserciones
  - Pre-condiciones en métodos
- Inspecciones y revisiones de código
  - Pull requests con revisiones de código
- Herramientas de análisis de código
  - <<Ver siguiente>>

Herramientas de análisis de código

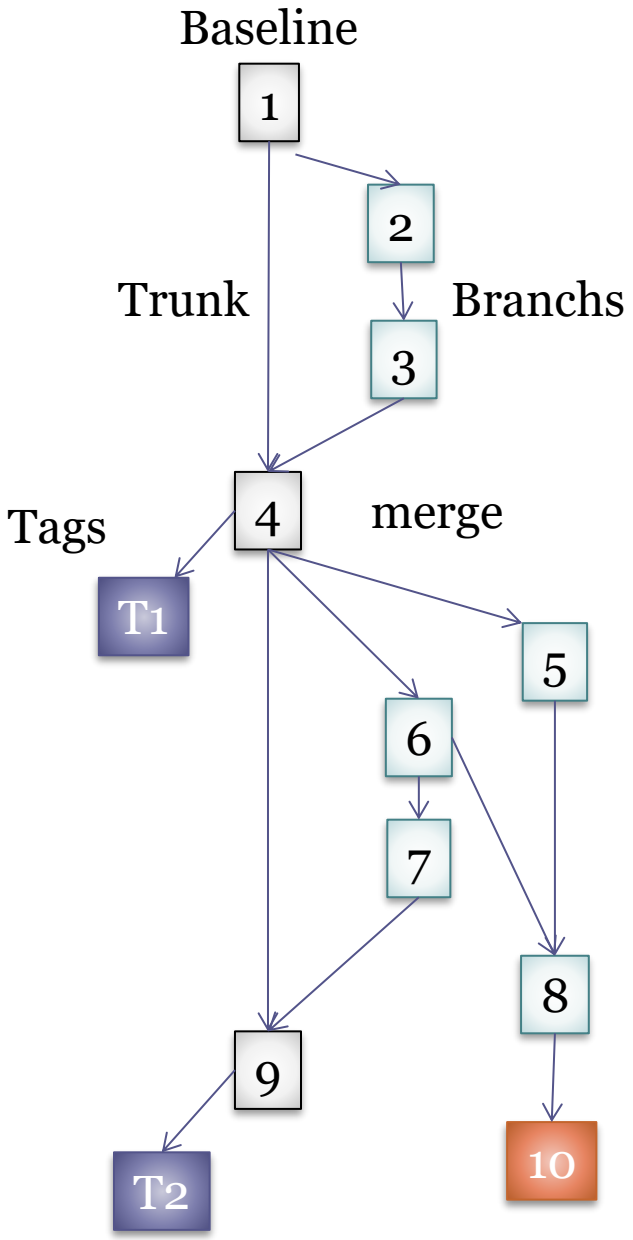
- Análisis estático vs dinámico
  - Sin ejecutar código/tras ejecutar código
  - Ejemplos: PMD, SonarCube,... (Codacy)
- Depuradores
  - Interactivos vs estáticos, logging
- Profilers
  - Información sobre uso de recursos
    - Memoria, CPU, llamadas a métodos, etc.
- Herramientas cobertura de código
  - Informan qué líneas de código se han ejecutado en pruebas
- Herramientas de *Program slicing*
  - Fragmento de programa (slice) que se ha ejecutado
    - Ejemplos: CodeSurfer, Indus-kaveri,...

Sistemas de control de versiones

Control de versiones

Definiciones

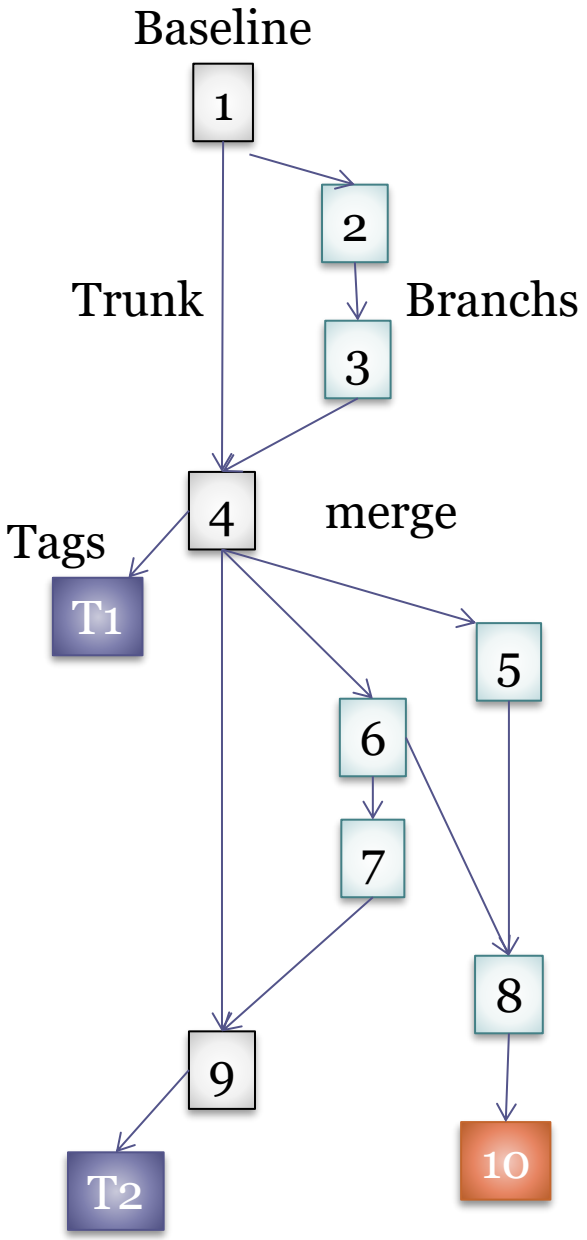
- Almacén (repositorio): Lugar en el que se almacenan los cambios.
- Baseline: Producto inicial en control de versiones
- Delta: cambios de una versión respecto a la anterior
- Trunk: Tronco o rama principal de un producto. Rama master en Git
- Branch (Rama): desviación de la rama principal
- Tag: Etiqueta de una línea de versiones



Control de versiones

Definiciones

- Check-out: Copia local de trabajo de una determinada rama o revisión.
- Commit: Comprometer los cambios locales en el sistema de control de versiones.
- *Merge* (fusión) : Combinación de dos conjuntos de cambios en uno.
- Estilos de *ramificación*: por característica, por equipo, por versión



Control de versiones

- 2 tipos
- Centralizados
  - Repositorio centralizado de todo el código
  - Administración centralizada
  - CVS, Subversion, ...
- Distribuidos
  - Cada usuario tiene su propio repositorio
  - Git, Mercurial

Git

Sistema de control de versiones distribuido

Diseñado por Linus Torvalds (Linux)

Objetivos:

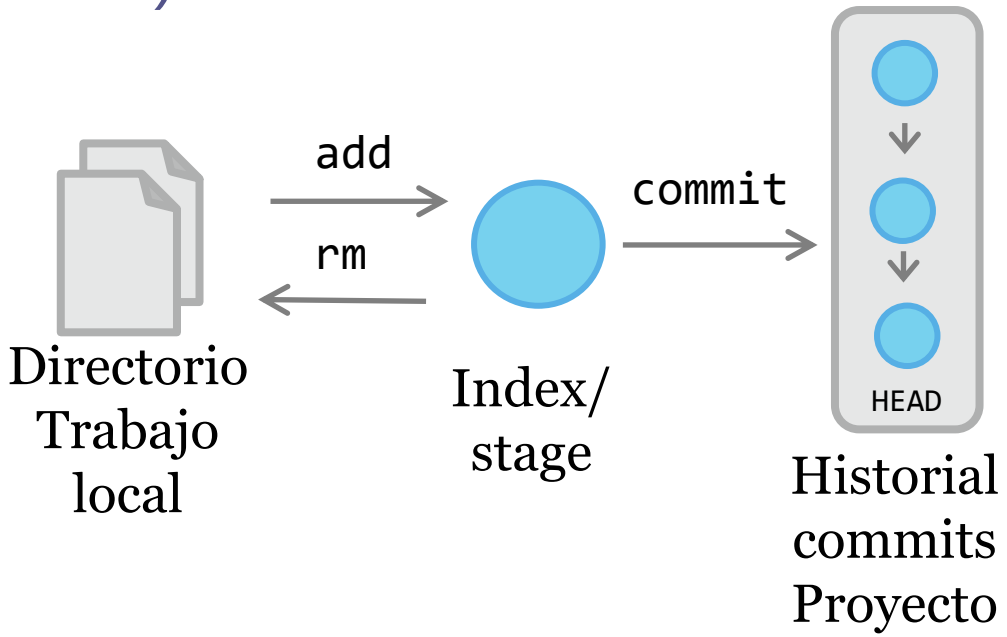
- Aplicaciones con gran nº de archivos de código
- Trabajo distribuido
- Apoyo a desarrollo no lineal (ramificaciones)

Más información:
http://rogerdudler.github.com/git-guide/

Componentes locales

3 components locales:

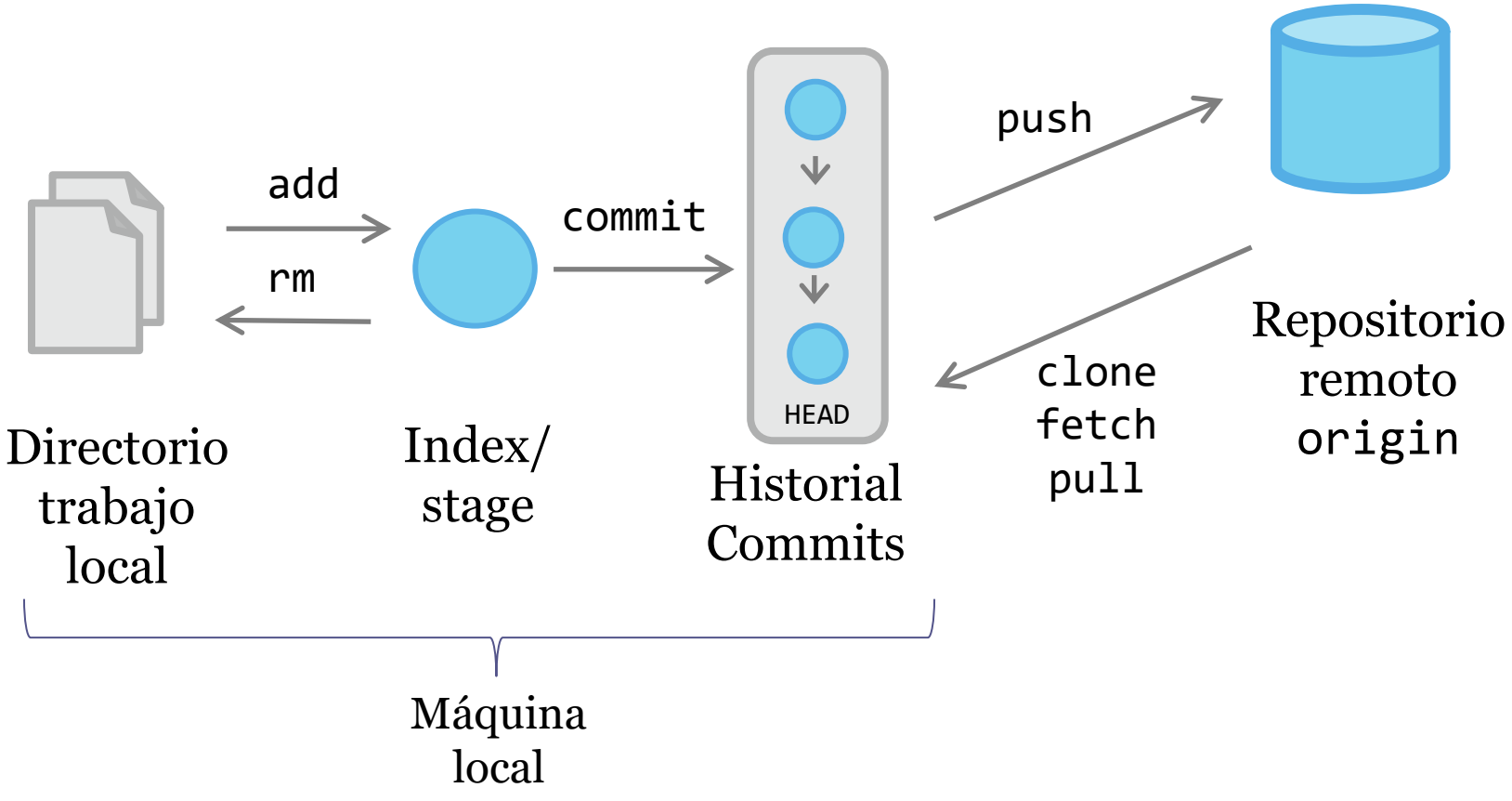
- Directorio trabajo local
- Index (stage area). También llamada cache
- Historial del proeycto: Almacena versiones o commits
- HEAD (versión más reciente)



Repositorios remotos

Conectar con repositorios remotos

`origin` = inicial



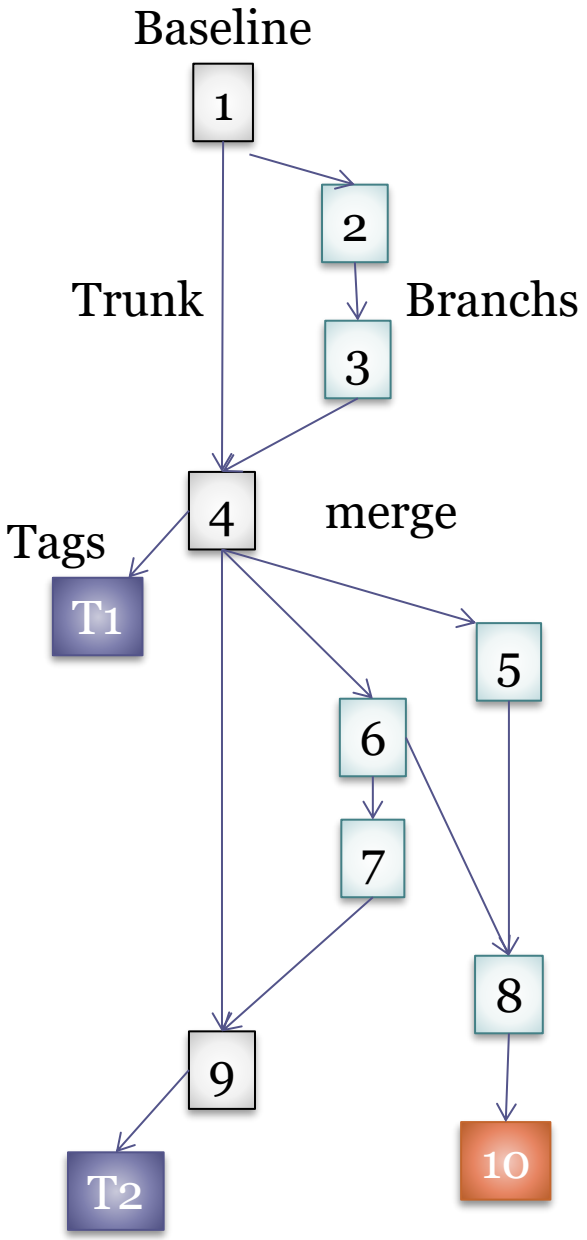
Ramas (branches)

Git facilita gestión ramas

- Master/trunk = rama inicial

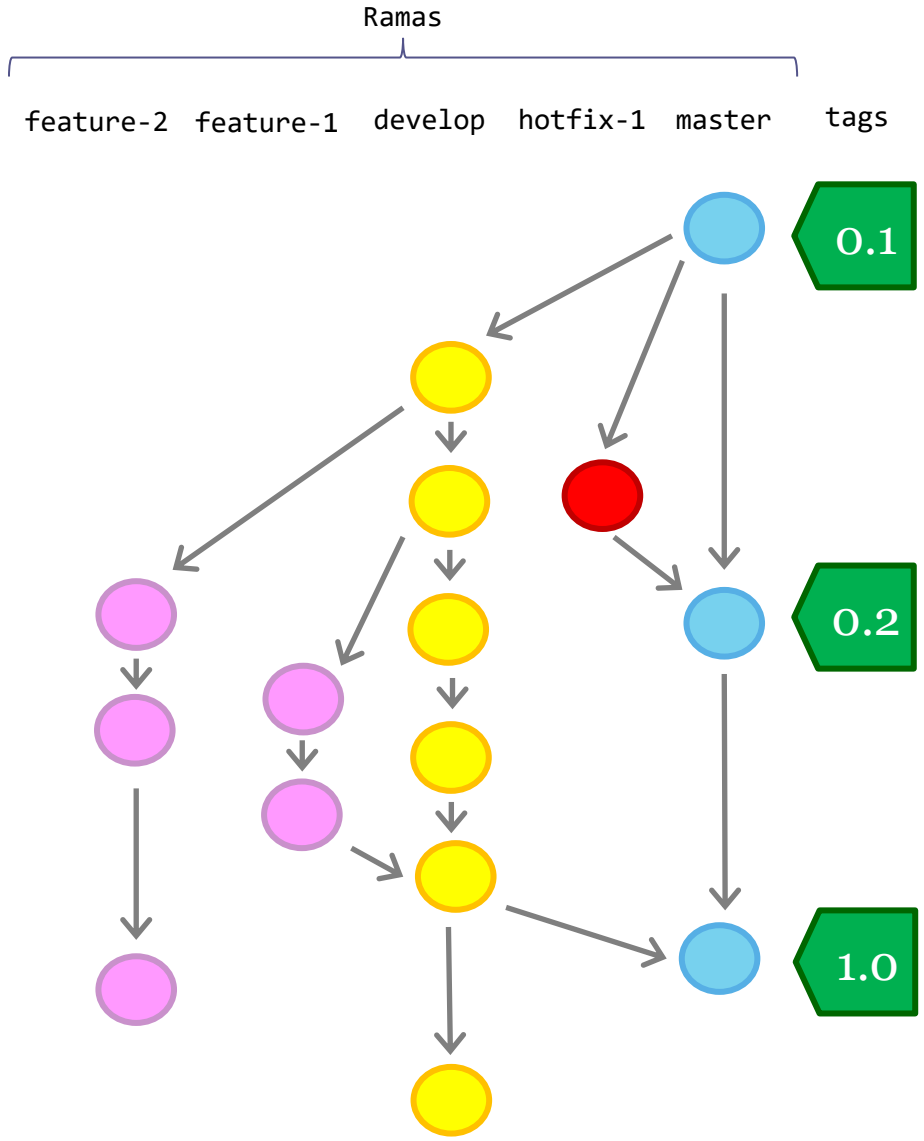
Operaciones:

- Crear ramas (*branch*)
- Cambiar rama (*checkout*)
- Combinar (*merge*)
- Etiquetar ramas (*tag*)



Patrones de ramificación

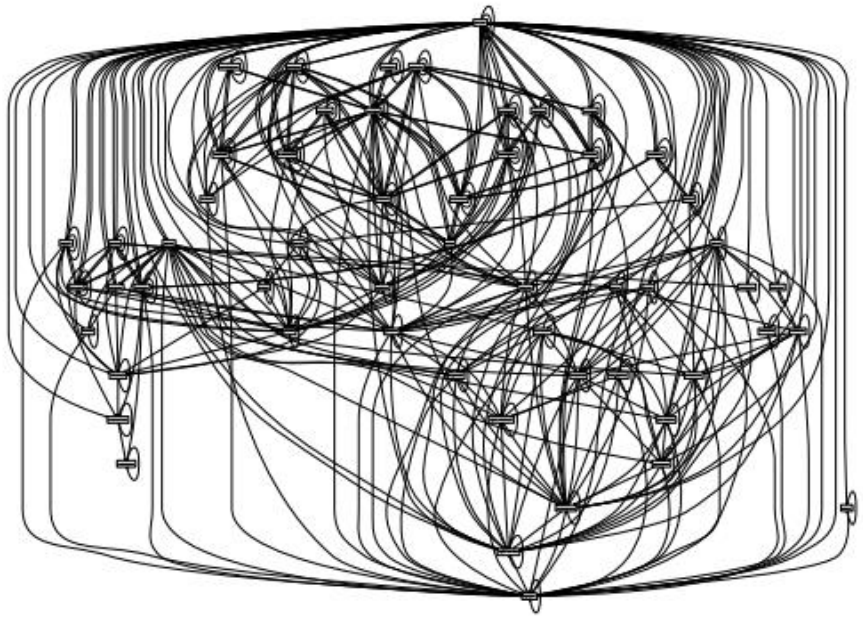
- Git-flow
  - Rama develop como principal
  - Ramas por características y hotfix
- Github-flow
  - Todo en master es desplegable
  - No se necesita rama hotfix
  - Promueve pull-requests
- Trunk-based development
  - Todo en rama principal (master)
  - Ramificación por características de poca duración (días)



Gestión de dependencias

Gestión de dependencias

- Librería: Colección de funcionalidades utilizadas por el sistema que se desarrolla
  - El sistema depende de dicha librería
    - La librería puede depender de otras librerías
    - La librería puede evolucionar
      - Versiones incompatibles
  - Grafo de dependencias



Grafo de dependencias de Mozilla Firefox
Fuente: The purely functional deployment model. E. Dolstra (PhdThesis, 2006)

Grafo de dependencias

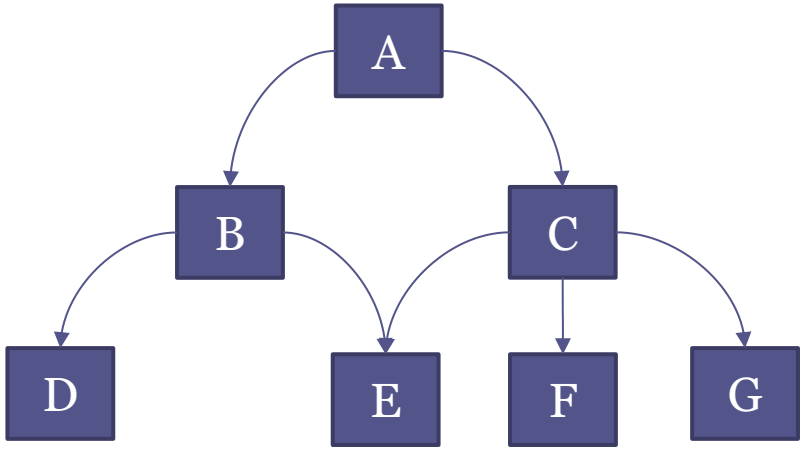
Grafo G = (V,E) donde

- V = vértices (componentes/paquetes)
- E = aristas (u,v) que indican que u depende de v

Métrica CCD (cumulative component dependency)

- Suma de dependencias de todos los componentes
  - Cada componente depende de sí mismo

En ejemplo:
CCD=7+3+4+1+1+1+1=18

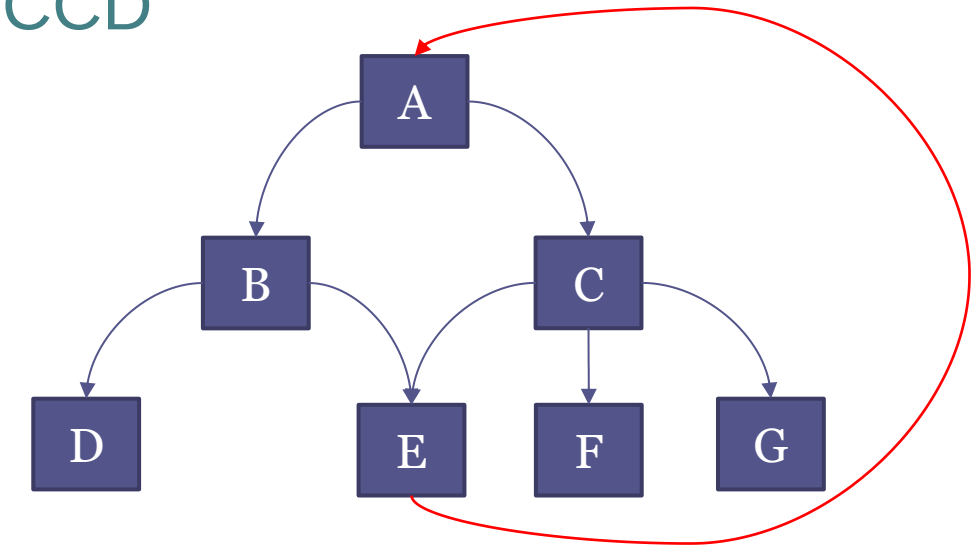


Principio de Dependencias cíclicas

El grafo de dependencias no debería tener ciclos

Añadir un ciclo puede hacer crecer la CCD

Ejemplo:



CCD = 7+7+7+1+7+1+1=31

Gestión de dependencias

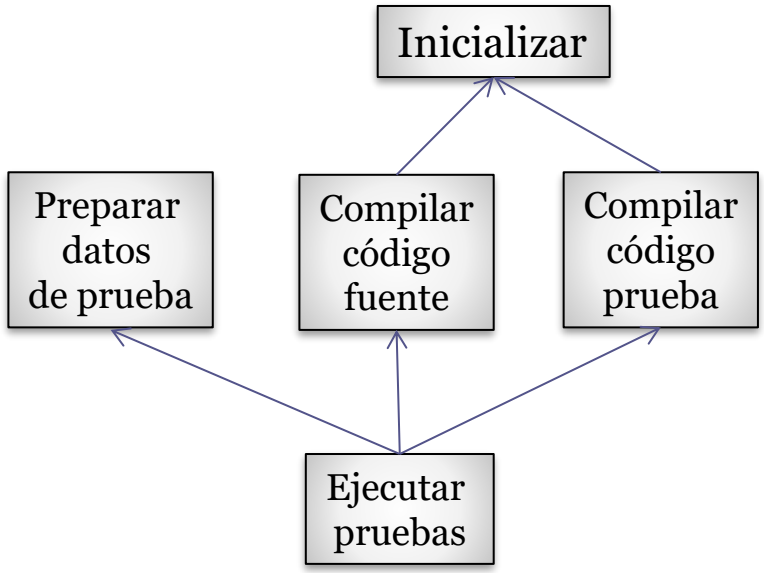
Modelos

- Instalación local: las librerías se instalan para todo el sistema.
  - Ejemplo: Ruby Gems
- Incluir solamente en proyecto (control de versiones)
  - Garantiza versión adecuada
- Enlace externo
  - Repositorio con librerías
  - Dependencia de Internet y evolución de la librería

Automatización de construcción

Herramientas de automatización de la construcción y el despliegue

- Organizar las diferentes tareas
  - Compilar, empaquetar, instalar, desplegar, etc.
  - Dependencias entre tareas
  - Deben garantizar:
    - Ejecutar todos los prerrequisitos
    - Ejecutarlos una sola vez



Automatización de la construcción

Atributos de calidad:

- Corrección
  - Evitar errores comunes (minimizar "*bad builds*")
  - Eliminar tareas redundantes o repetitivas
- Simplicidad: Gestionar complejidad
- Automatización y facilidad de publicación
  - Disponer de histórico de releases y construcciones
  - Integración continua
- Coste
  - Ahorrar tiempo y dinero

> "Nunca envíes a un humano a realizar el trabajo de una máquina"
> G. Hohpe

¿Cuándo construir?

- Bajo demanda
  - Usuario lanza un *script* en línea de commandos
- Planificada
  - Automáticamente en ciertas horas
    - Ejemplo: *nightly builds*
- *Triggered*
  - Cada *commit* al Sistema control de versions
  - Servidor de integración continúa enlazado con Sistema de control de versiones

Herramientas

Makefile (C world)
Ant (Java)
Maven (Java)
SBT (Scala, JVM languages)
Gradle (Groovy, JVM languages)
rake (Ruby)
npm, grunt, gulp (Javascript)
cargo (Rust)
etc.

Automatización de construcción

`make`: Incluido en Unix

- Orientado a producto
- Lenguaje declarativo basado en reglas
  - Cuando el proyecto es complejo, los ficheros de configuración pueden ser difíciles de depurar
- Varias versiones: BSD, GNU, Microsoft
- Muy popular en C, C++, etc.

Automatización de construcción

- `ant`: Plataforma Java
  - Orientado a tareas
  - Sintaxis XML (build.xml)

Automatización de construcción

- `maven`: Plataforma Java
  - Convención sobre configuración
  - Gestionar ciclo de vida del proyecto
  - Gestión de dependencias
    - Descarga automática y almacenamiento local
  - Sintaxis XML (pom.xml)

Automatización de construcción

- Lenguajes empotrados
  - Lenguajes específicos empotrados en lenguajes interpretados de alto nivel
  - Gran versatilidad
  - Ejemplos:
    - `gradle` (Groovy)
    - `sbt` (Scala)
    - `rake` (Ruby)
    - `Buildr` (Ruby)
    - …

Nuevas herramientas

- Pants (Foursquare, twitter)
  - https://pantsbuild.github.io/
- Bazel (Google)
  - http://bazel.io/
- Buck (Facebook)
  - https://buckbuild.com/

Maven

Maven

Herramienta de automatización de construcción
Describe cómo construir el software
Describe dependencias del software
Principio: Convención sobre configuración

Jason van Zyl
Creador Maven

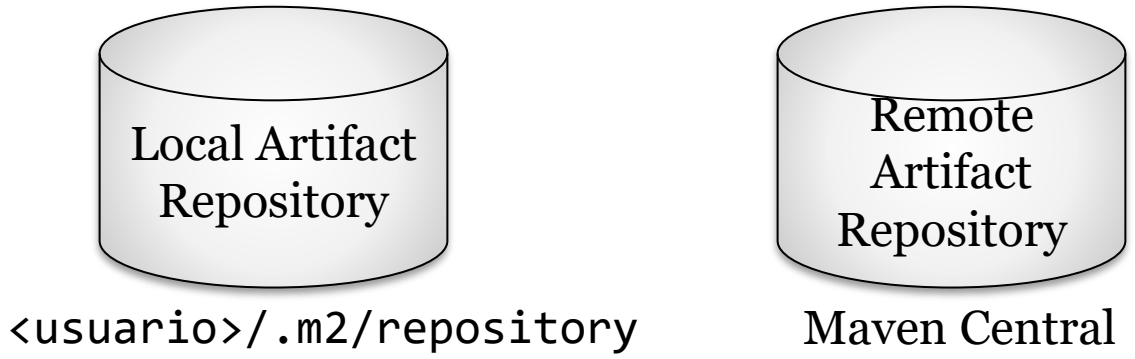
Maven

- Fases típicas de construcción:
  - `clean, compile, build, test, package, install, deploy`
- Idenfiticación de módulo
  - 3 coordenadas: Grupo, Artefacto, Versión
  - Dependencias entre módulos
- Configuración: fichero XML (Project Object Model)
  - `pom.xml`

Maven

Almacenes de artefactos

- Guardan diferentes tipo de artefactos
  - Ficheros JAR, EAR, WAR, ZIP, plugins, etc.
- Todas las interacciones a través del repositorio
  - Sin caminos relativos
  - Compartir módulos entre equipos de desarrollo



Maven Central

Repositorio público de proyectos

Más de 1 mill de GAV

≈ 3000 proyectos nuevos cada mes (GA)

≈ 30000 versiones nuevas al mes (GAV)*

http://search.maven.org/

Otros repositorios:

https://bintray.com/

* Fuente: http://takari.github.io/javaone2015/still-rocking-it-maven.html

POM - Project Object Model

- Sintaxis XML
- Describe un proyecto
  - Nombre y versión
  - Tipo de artefacto (jar, pom, ...)
  - Localización del código fuente
  - Dependencias
  - Plugins
  - Profiles
    - Configuraciones alternativas para la construcción
- Estructura basada en herencia
- Referencia: `https://maven.apache.org/pom.html`

POM - Project Object Model

- Estructura basada en herencia
  - Super POM
    - POM por defecto de Maven
    - Todos los POM extiende el Super POM salvo que se indique de formaexplícita
  - parent
    - Declara el POM padre
    - Se combinan las dependencias y propiedades

Maven

Identificación de proyecto

- GAV (Grupo, artefacto, versión)
  - Grupo: Identificador de agrupamiento
  - Artefacto: Nombre del proyecto
  - Versión: Formato {Mayor}.{Menor}.{Mantenimiento}
    - Se puede añadir "-SNAPSHOT" (en desarrollo)


```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>es.uniovi.asw</groupId>
<artifactId>censusesN</artifactId>
<version>0.0.1</version>
<name>censusesN</name>
...
```

Maven

Estructura de directorios

- Maven utiliza una estructura convencional
  - src/main
  - src/main/java
  - src/main/webapp
  - src/main/resources
  - src/test/
  - src/test/java
  - src/test/resources
  - . . .
- Directorio de salida:
  - target

Maven

- Ciclo de vida
  - 3 ciclos de vida por defecto
    - clean
    - default
    - site
  - Cada ciclo de vida tiene sus fases

Ciclo de vida "clean"

Borrar código compilado

3 fases

- pre-clean
- clean
- post-clean

Ciclo de vida "default"

Compilación y empaquetado de código

Algunas fases

```
validate
initialize
generate-sources
generate-resources
compile
test-compile
test
package
integration-test
verify
install
deploy
```

Ciclo de vida "site"

Generar documentación proyecto

```
pre-site
site
post-site
site-deploy
```

Maven

- Gestión automática de dependencias
  - Identificación mediante GAV
  - Ámbito
    - `compile`
    - `test`
    - `provide`
  - Tipo
    - jar, pom, war,...

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
        . . .
    </dependencies>
</project>
```

Maven

- Gestión automática de dependencias
  - Las dependencias son descargadas
  - Alojadas en repositorio local
  - Pueden crearse repositorios intermedios (proxies)
    - Ejemplo: artefactos comunes de una empresa
  - Transititivdad
    - B depende de C
    - A depende de B $\Rightarrow$ C también se descarga

Maven

- Múltiples módulos
- Proyectos grandes pueden descomponerse
- Cada proyecto crea un artefacto
  - Tiene su propio fichero pom.xml
- El proyecto padre agrupa los módulos

```
<project>
    ...
    <packaging>pom</packaging>
    <modules>
        <module>extract</module>
        <module>game</module>
    </modules>
</project>
```

Maven Plugins

Maven tiene una arquitectura basada en plugins

2 tipos de plugins

- Build
  - Se identifican en `<build/>`
- Reporting
  - Se identifican en `<reporting/>`

Lista de plugins: `https://maven.apache.org/plugins/index.html`

Maven

Algunas fases habituales

`archetype:generate` - Genera esqueleto de un proyecto

`eclipse:eclipse` - Genera proyecto eclipse

`site` - Genera sitio web del proyecto

`site:run` - Genera sitio web y arranca servidor

`javadoc:javadoc` - Generar documentación

`cobertura:cobertura` - Informe del código ejecutado en pruebas

`checkstyle:checkstyle` - Chequear el estilo de codificación

npm

npm

- Node.js Package Manager
- Creado inicialmente por Isaac Schlueter
  - Posteriormente, empresa Npm inc.
- Gestor de paquetes por defecto de NodeJs
  - Otro gestor para NodeJs: Yarn
  - Gestiona las dependencias
  - Permite *scripts* para tareas comunes
- Almacén de software
  - Paquetes públicos o de pago
- Fichero de configuración: package.json

Configuración npm: package.json

Fichero configuración: package.json

npm init crea un esqueleto inicial

Campos:

```
{
 "name":              "...obligatorio...",
 "version":           "...obligatorio...",
 "description":       "...opcional...",
 "keywords":          "...",
 "repository":        {... },
 "author":            "...",
 "license":           "...",
 "bugs":              {...},
 "homepage":          "http://. . .",
 "main":              "index.js",
 "devDependencies":   { ... },
 "dependencies":      { ... }
 "scripts":           { "test": " ... " },
 "bin":               {...},
}
```

Nota: Yeoman proporciona esqueletos completos

Paquetes npm

Base de datos: http://npmjs.org

Instalación de paquetes:

2 opciones:

Local

```
npm install <packageName> --save (--save-dev)
```

Global

```
npm install -g <packageName>
```

Dependencias npm

Gestión de dependencias

- Paquetes locales en caché, directorio: `node_modules`
- Acceso a módulos mediante: `require('...')`

Paquetes globales (instalados con opción --global)

- Cacheados en: `~/.npm` folder

Paquetes con ámbito marcados con @

Comandos y *scripts* npm

- Npm contiene numerosos comandos
  - start ≈ node server.js
  - test ≈ node server.js
  - ls: muestra lista de paquetes instalados
  - ...
- Scripts creados por el usuario:
  - run-script <name>
  - Definidos en sección "scripts"
- Para gestionar tareas más complejas
  - gulp, grunt

https://docs.npmjs.com/cli-documentation/