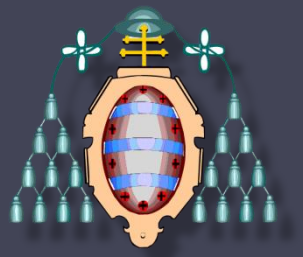
ES
Español



Universidad de Oviedo

Escuela de Ingeniería Informática

ARQUITECTURA DEL SOFTWARE

Arquitectura del Software

Clase de teoría 2

Escuela de Ingeniería Informática

2016-17

Jose Emilio Labra Gayo

Estilos arquitectónicos

Disposición: Construcción y distribución de software
Modularidad
Comportamiento
Integración
Negocio

Disposición (Allocation)

Relación del software con el entorno

Construcción, despliegue y distribución

Esquema clase

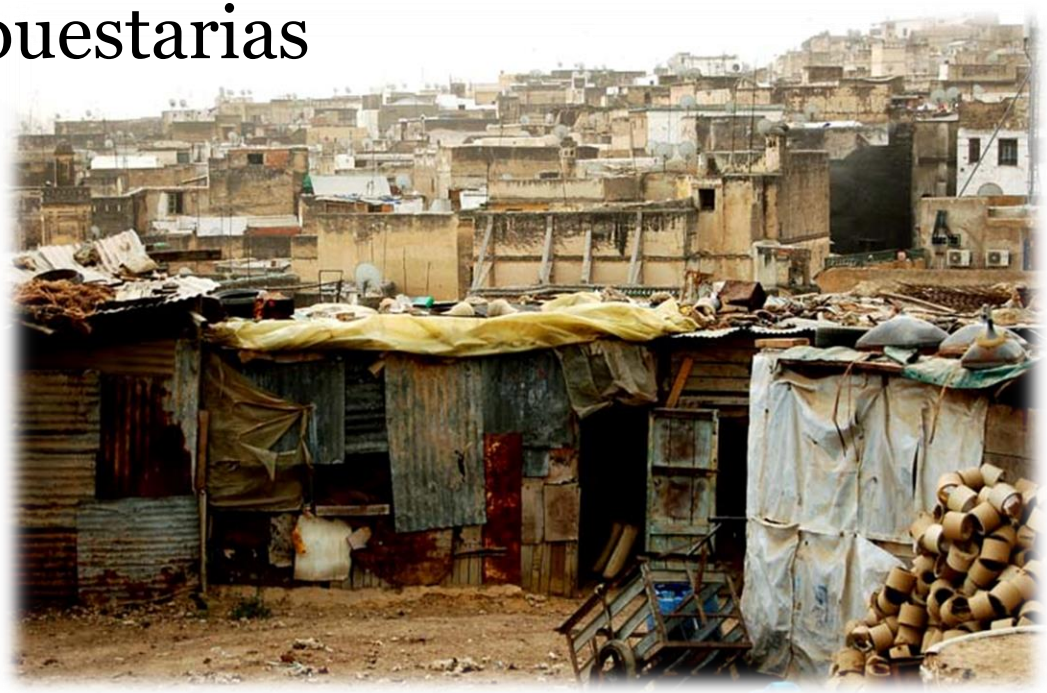
- Construcción
  - Estilos de desarrollo
    - Tradicionales, iterativos, ágiles
  - Gestión de configuraciones
    - Control de versiones
    - Gestión de dependencias
    - Despliegue e integración continua
- Distribución
  - Canales de distribución
- Repaso métodos ágiles y TDD

Estilos de desarrollo

Nota: Se ha incluido una selección. Puede consultarse una extensa lista en: http://en.wikipedia.org/wiki/List_of_software_development_philosophies

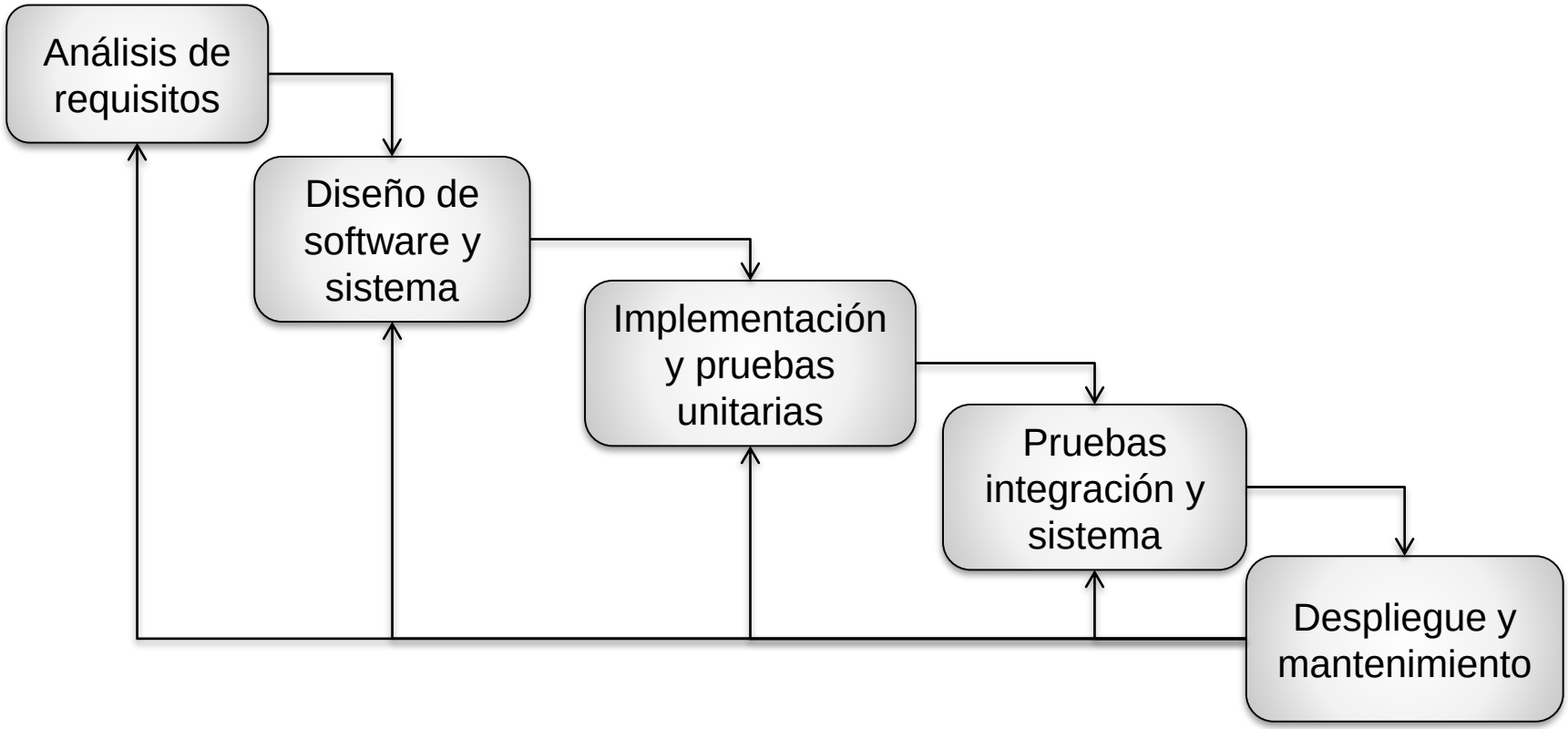
Incremental piecemeal

Crecimiento según necesidad
Codificar sin considerar la arquitectura
Software de usar y tirar
Limitaciones presupuestarias

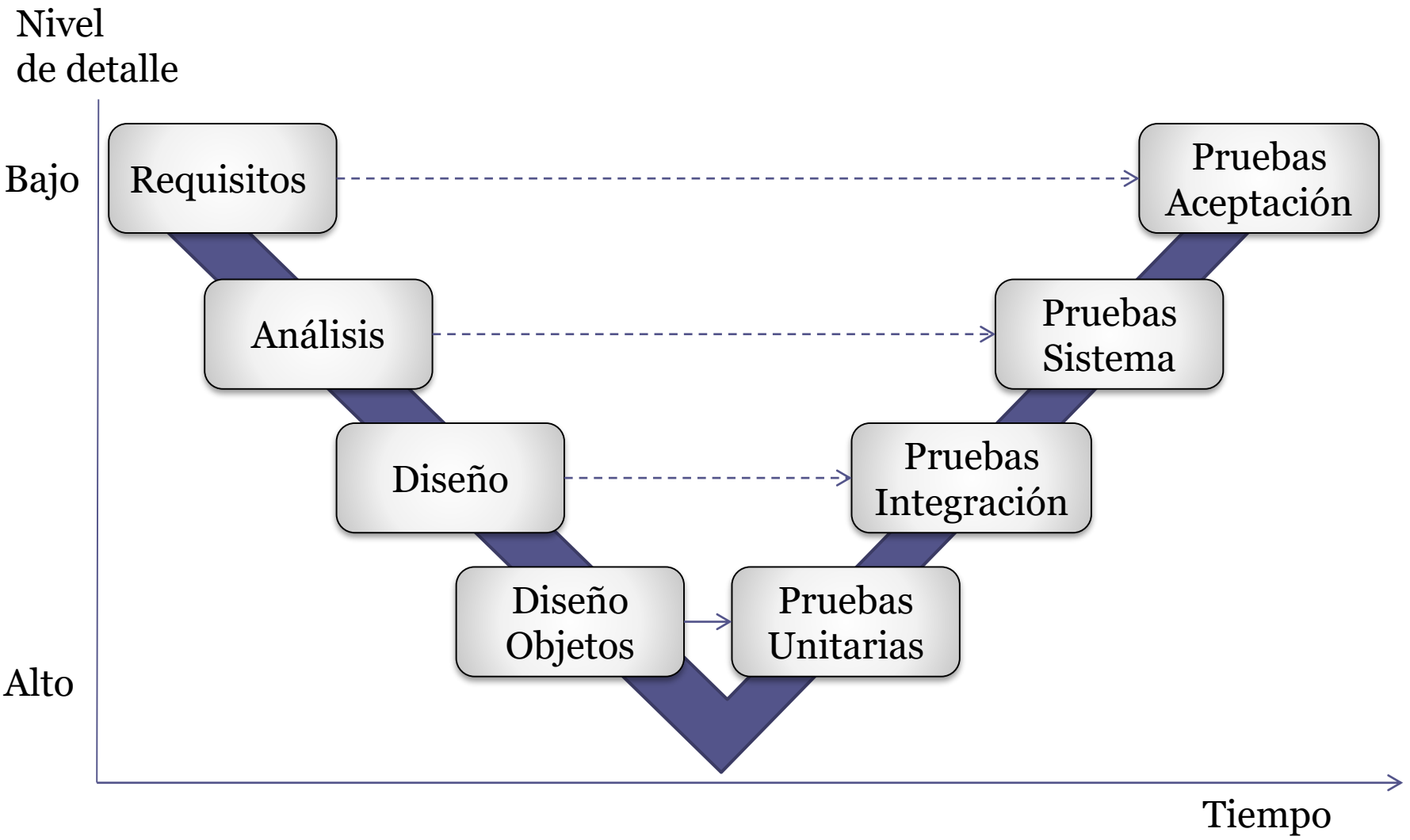


Cascada

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

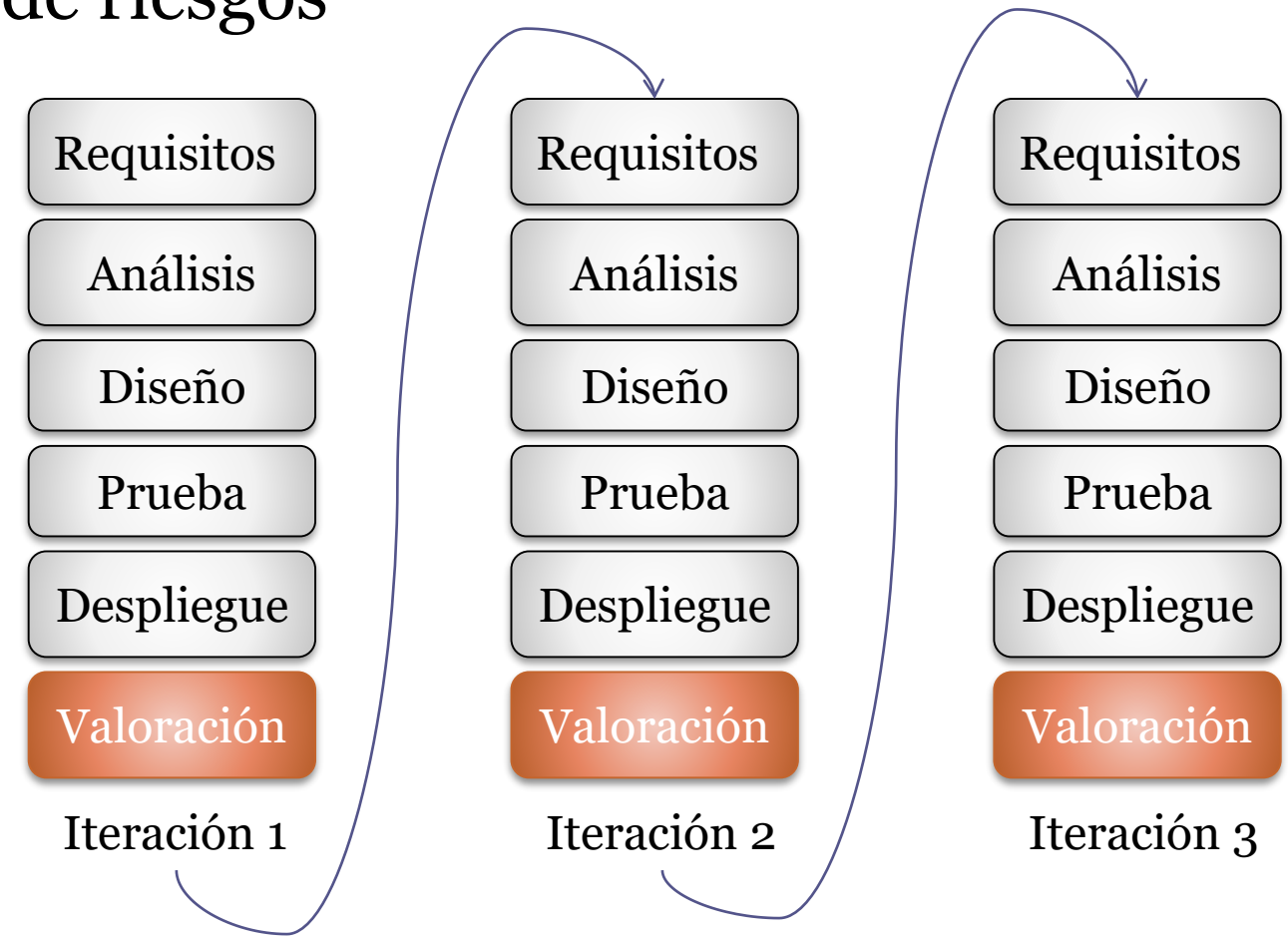
Big Design Up Front

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



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

Se repasan luego…

Gestión de configuraciones

Gestión de configuraciones

- Gestión de la evolución del software
  - Cambios del sistema = actividades en equipo
  - Software = algo continuo
    - Producto vs Servicio
  - Costes y esfuerzo necesarios para hacer cambios
- Diferentes versiones de software
  - Funcionalidades nuevas o diferentes
  - Corrección de *bugs*
  - Nuevos entornos de ejecución

Control de versiones

- Gestionar diferentes versiones 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.

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
- Puede haber resistencia de usuarios a nuevas *releases*
- Factores externos:
  - Marketing, clientes, hardware, …
- Modelo ágil: *releases* my 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
- Distribución: medios físicos (CDs, DVDs), Web (descargas), stores

Continuous delivery

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

Filosofía Wabi-sabi
Aceptar la imperfección
Software no finalizado: Suficientemente bueno (Good enough)

DevOps

Unir ***dev****elopment* y ***op****eration****s***

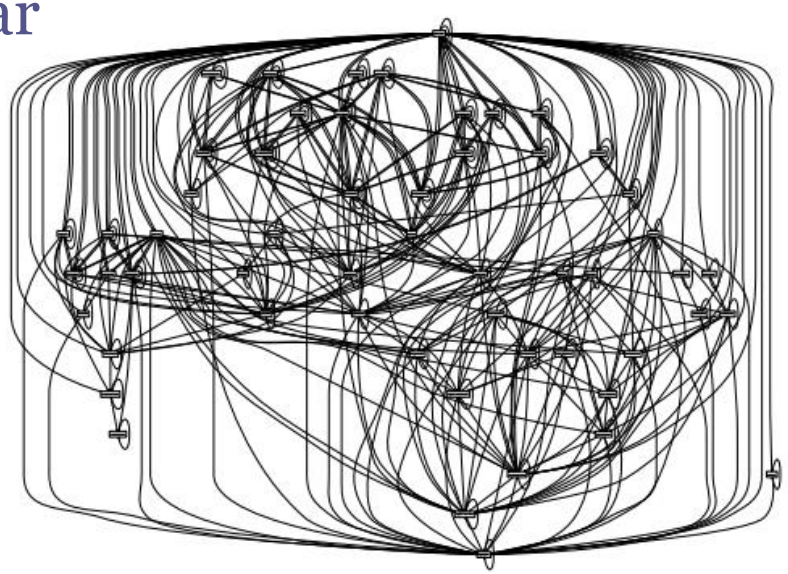
Cambio cultural donde el mismo equipo afronta las fases:

- 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

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

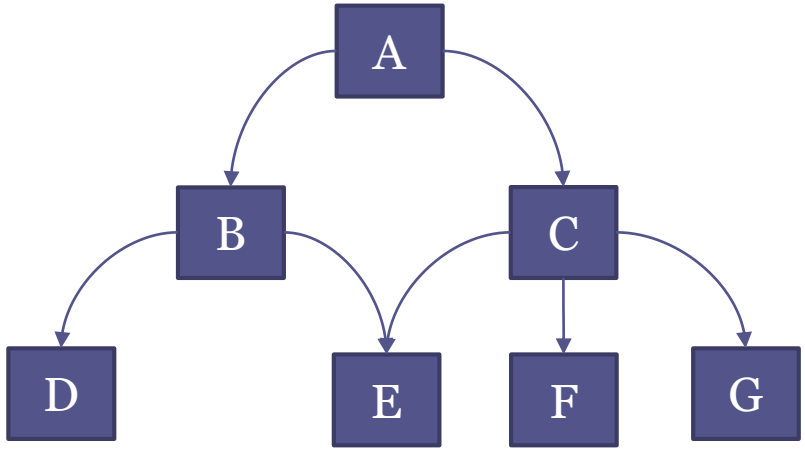
Grafo G = (V,E) donde

- 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



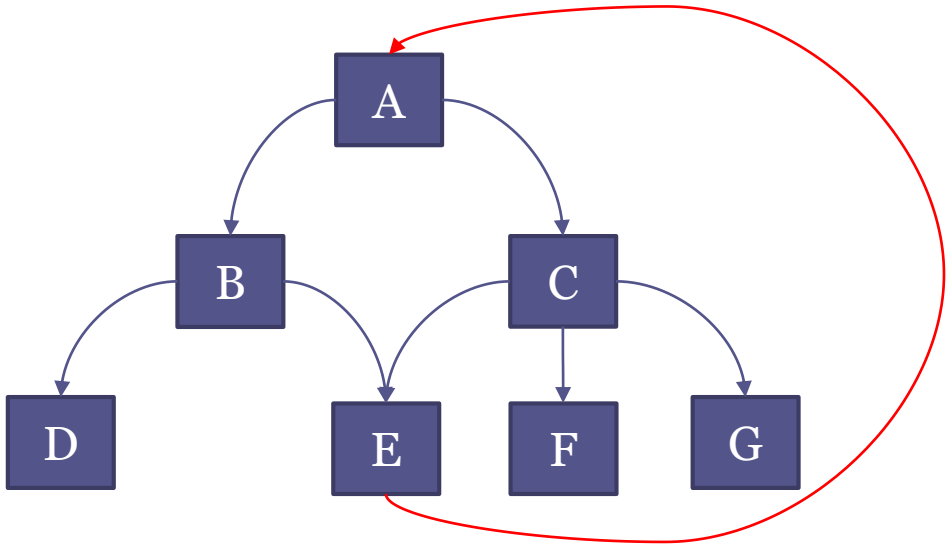
Principio de Dependencias cíclicas

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 la Construcción

Fases habituales del ciclo de vida del software

- Compilación
  - De código fuente a código binario
- Empaquetado
  - Gestión de dependencias e integración
  - También llamado enlace (linking)
- Ejecución de pruebas
- Despliegue
- Crear documentación/*release notes*

Automatización de la construcción

Automatizar tareas de construcción

Objetivos:

- Evitar errores (minimizar "*malas construcciones*")
- Eliminar tareas redundantes y repetitivas
- Evitar complejidad
- Tener un histórico de construcciones y *releases*
- Facilitar la integración continua
- Ahorro de tiempo y dinero

Herramientas de automatización

Makefile (mundo C)
Ant (Java)
Maven (Java)
SBT (Scala, lenguajes JVM)
Gradle (Groovy, lenguajes JVM)
rake (Ruby)
etc.

Automatización de la construcción

`make`: Incluida Unix

- Orientado a un producto
- Lenguaje declarative basado en reglas
  - Difícil depurar en proyectos complejos
- Varias versiones: BSD, GNU, Microsoft
- Muy popular en C, C++, etc.

Automatización construcción

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

Automatización construcción

- `maven`: Platforma Java
  - Convención sobre configuración
  - Gestionar ciclo de vida del proyecto
  - Gestión de dependencias
  - Sintaxis XML (pom.xml)

Automatización construcción

- Lenguajes empotrados
  - Lenguajes de dominio específico empotrados en otros lenguajes de alto nivel
  - Mucha 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

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

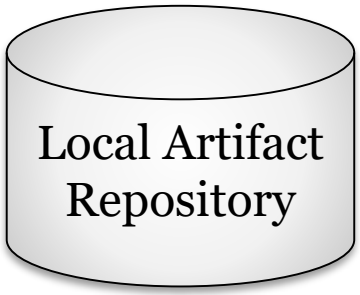
Configuración: fichero XML (Project Object Model)

- `pom.xml`

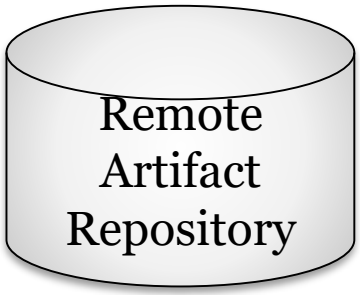
Maven

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



`<usuario>/.m2/repository`



Maven Central

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

Modelos de distribución

Líneas de producto
Canales de distribución

Líneas de producto

Línea de producto: productos que comparten una serie de características comunes satisfaciendo un segmento de mercado concreto

Objetivo:

- Reducción esfuerzo desarrollo
- Mejorar productividad
- Pasar de un único producto a una línea de productos

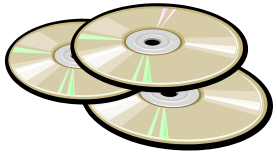
Líneas de producto

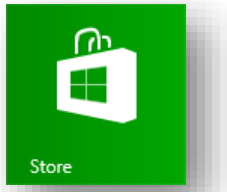
Requisitos

- Identificar soluciones genéricas a problemas comunes
- Desarrollo basado en componentes
- Plataformas genéricas
  - Reutilización de software
- Generación automática de sistemas

Canales de distribución

- Distribución tradicional
  - CDs, DVDs, etc.
- Distribución vía Web
  - Descargas, FTP, etc.
- Mercados de aplicaciones
  - Paquetes de aplicaciones Linux
  - Apple AppStore,
  - Google Play,
  - Windows Store





Entornos de ejecución

- Software on-premises
  - Se ejecuta en el inmueble de la persona/organización que lo va a utilizar
- SaaS (Software as a Service)
  - Se ejecuta remotamente
- Contenedores (Container as a Service)
  - Ejecución local
  - Se distribuye en contenedores
    - Fácil instalación
    - Mayor rendimiento que máquinas virtuales
  - Ejemplos: Docker, Kubernetes

Métodos ágiles

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 procesos y herramientas
- Software que funcione
  - sobre documentación
- Colaboración con el cliente
  - sobre negociación de contrato
- Respuesta 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

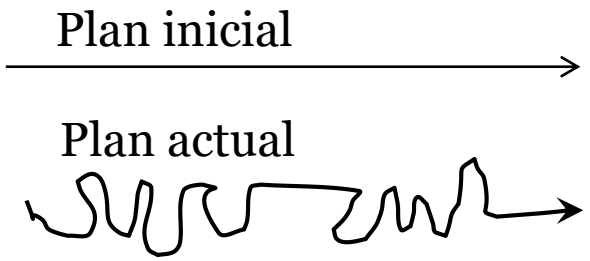
Métodos ágiles

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

Utilización de pruebas extensiva

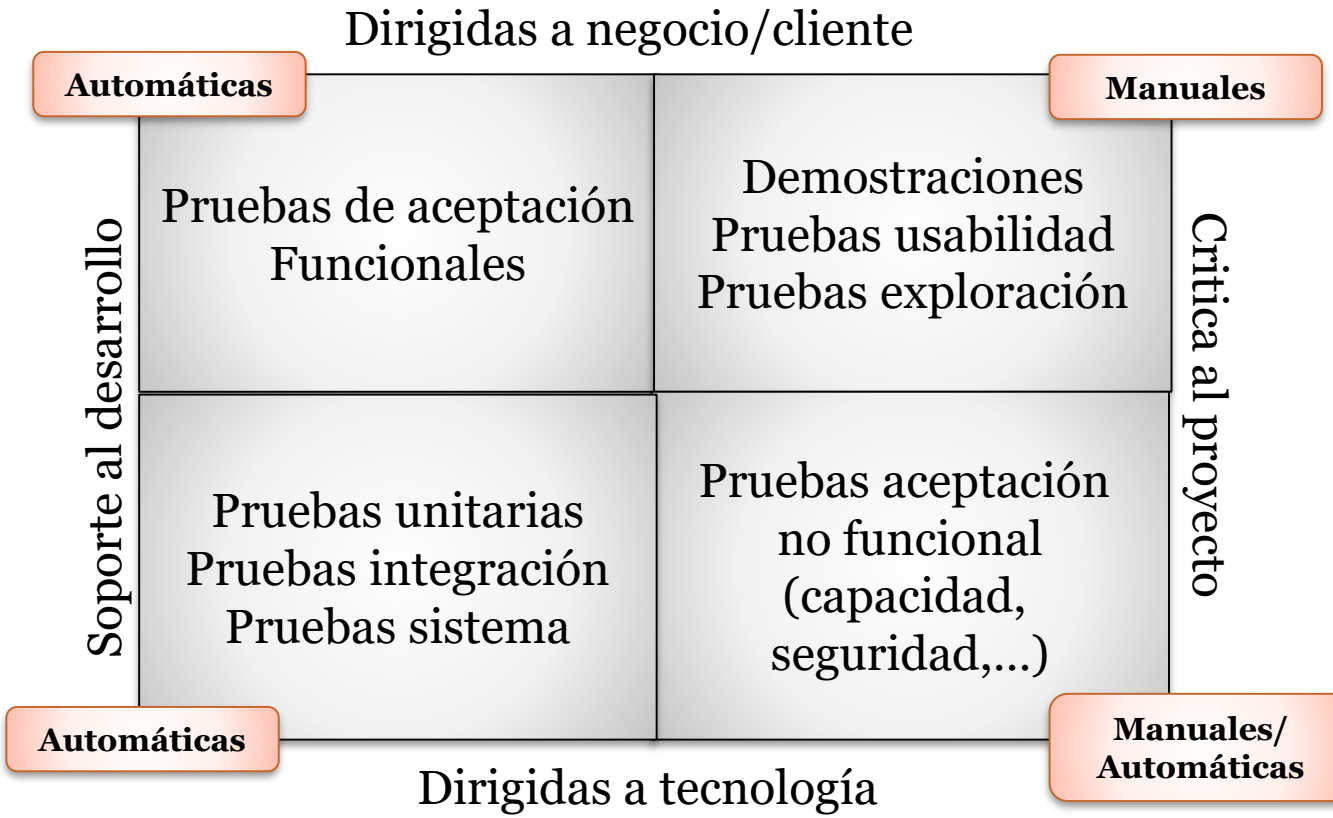
Objetivo: Desarrollo basado en pruebas

TDD (Test Driven Development)

Métodos ágiles

2.- Pruebas

Tipos de pruebas



TDD - Test Driven Development

Repetir para cada unidad de software (por orden)

1. Definir pruebas
2. Implementar
3. Verificar si la implementación pasa las pruebas

Ventajas:

- Código más limpio y seguro
- Batería de pruebas disponible
- Refactorización

Algunas herramientas:

- JUnit, assertJ

BDD Behaviour Driven Development

Behaviour-driven development (BDD)

- Pruebas a partir de historias de usuario
- Deben escribirse junto con cliente
  - Herramientas: Cucumber, JBehave, Specs2,...
- Sirven como contrato
- Miden el 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   |
```

Principios FIRST

Principios FIRST

- F - Fast
  - La ejecutación 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

Definiciones de pruebas

Definiciones

- Dobles de pruebas
  - Objetos *Dummy:* se pasan pero no se utilizan
  - Objetos *fake:* Tienen implementación parcial
  - *Stubs:* respuestas precocinadas a ciertas preguntas
  - *Espías:* son *stubs* que registran información para depuración
  - *Mocks:* objetos programados con ciertas expectativas sobre qué tipo de llamadas deben recibir
- *Fixtures*. Elementos de soporte a las pruebas
  - *Ej. Bases de datos con ciertas entradas, determinados ficheros, etc.*

Algunos tipos de pruebas

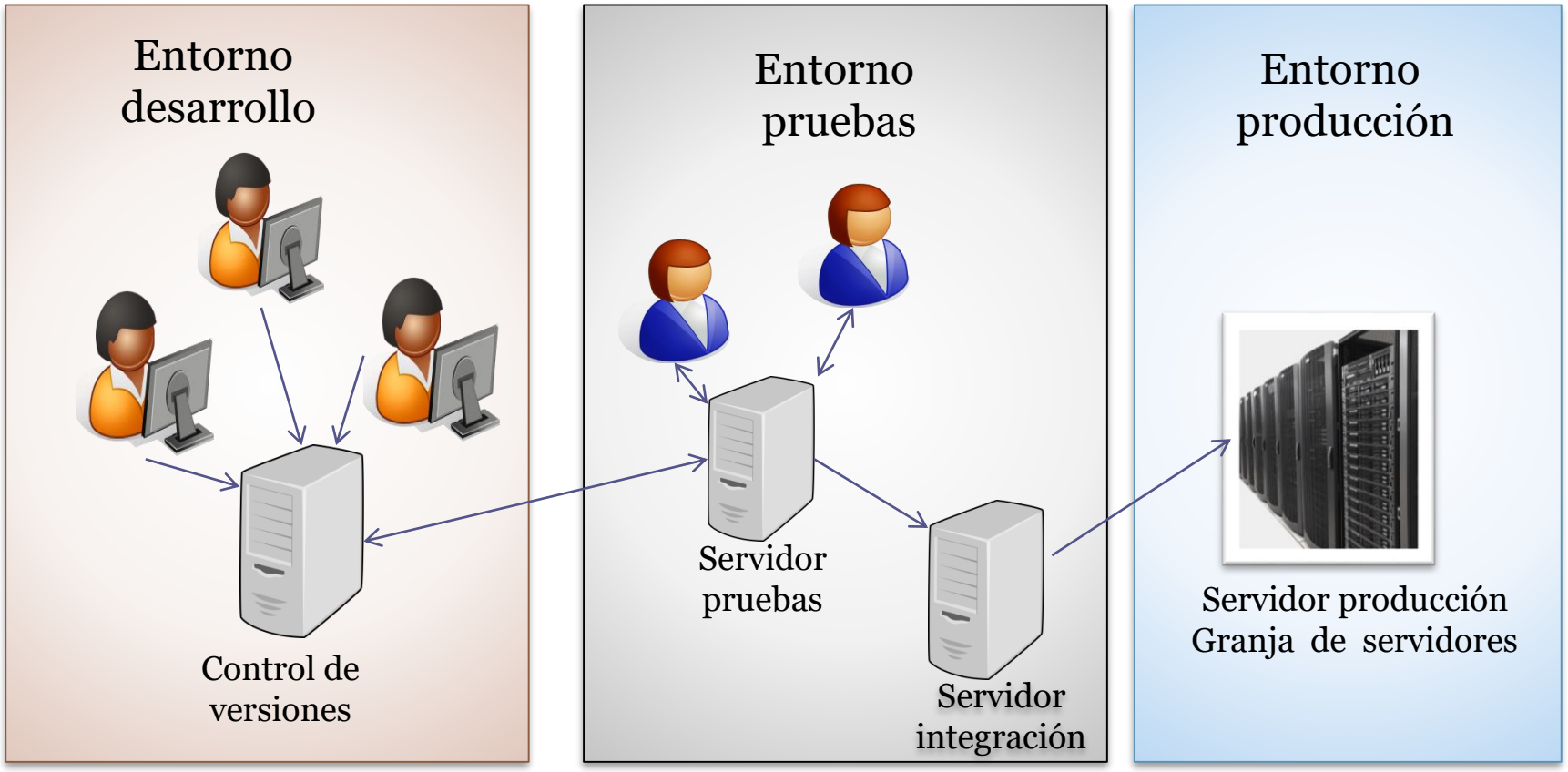
- Unitarias
  - Probar cada unidad por separado
- Integración
  - Probar el sistema
- Aceptación
  - Probar el sistema ante el cliente

¿Cuándo ejecutar pruebas?

- Bajo demanda
  - Un usuario ejecuta un script en línea de comandos
- Planificada
  - Se ejecuta automáticamente a ciertas horas
  - Servidor de integración
  - Ejemplo: *nightly builds*
- Lanzada (triggered)
  - En cada commit a sistema de control de versiones
  - Servidor de integración continua enlazado con sistema de control de versiones

Entornos de ejecución

Entornos habituales



También puede haber un entorno de ensayo (staging environment)

Revisiones de código y 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

Diseño simple

- Reacción a Big Design Up Front
  - Crear el diseño más simple que funcione
- Documentación automatizada
  - JavaDoc y similares

Refactorización

- Mejorar diseño sin cambiar la funcionalidad
  - Simplificar código (eliminar código duplicado)
  - Buscar activamente oportunidades de abstracción
- Pruebas de regresión
  - Probar un software de nuevo tras un cambio
  - Usar batería de pruebas
  - Detectar posibles problemas de rendimiento creados

Propiedad colectiva de 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

Integración continua

- Integración y verificación de pruebas automatizada
  - Sistema externo se lanza automáticamente
  - Integrar frecuentemente los cambios propios del código en el repositorio de código central
    - Ejecutar todas las pruebas unitarias y de integración
    - Juntar las copias de todos los desarrolladores
- Herramientas:
  - Hudson, Jenkins, Travis, Bamboo

Integración continua

Objetivos

- Desarrollo basado en pruebas
- Evitar "infierno de integración" (*integration hell)*
- Mantener a todos los desarrolladores al día
  - Todo el mundo debe poder ver la última construcción
- Facilitar la obtención de los últimos entregables

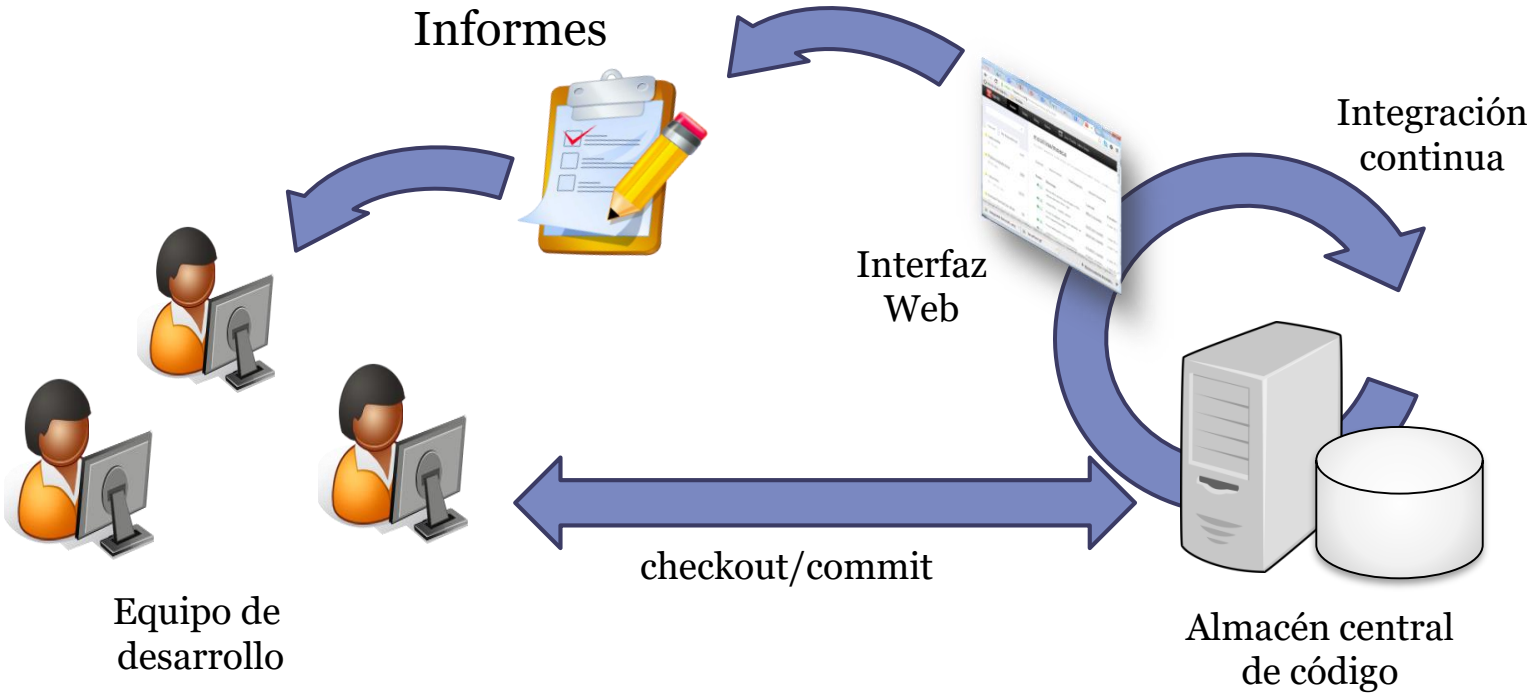
Integración continua

Buenas prácticas:

- Mantener repositorio de código
- Automatizar la construcción
- Hacer la construcción que se pruebe automáticamente
- Todos el mundo hace *commits* a línea base
- Todo *commit* debe ser construido
- Mantener la construcción rápida
- Probar en un clon del entorno de producción
- Automatizar despliegue

Integración continua

Esquema



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
    - Mejor comunicación

Entregas pequeñas

- Tan pequeñas como sea posible pero que ofrezcan valor al usuario
  - Entregas no son, por ejemplo, implementar BD
- Obtener realimentación temprana del cliente
- Planificar siguiente entrega tras cada iteración
  - Entregar algo cada semana
    - Cuidado:
      - No todo el mundo puede trabajar así

Horarios normales

40h/semana = 40h/semana

Evitar horas extra

Programadores cansados escriben código pobre

- A largo plazo ralentiza el desarrollo

Código limpio & Estándares de codificación

Facilitar modificación de código por otras personas

- Utilizar buenas prácticas
  - Estilos y normas de codificación
  - Evitar *code smells*
- Manifiesto *software craftmanship*
- Libro *Clean Code (Robert C. Martin)*

Fuente: Clean Code. Robert Martin

Ejemplos de metodologías

- 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