

# Diseño de Software

## Semana 1

# Agenda

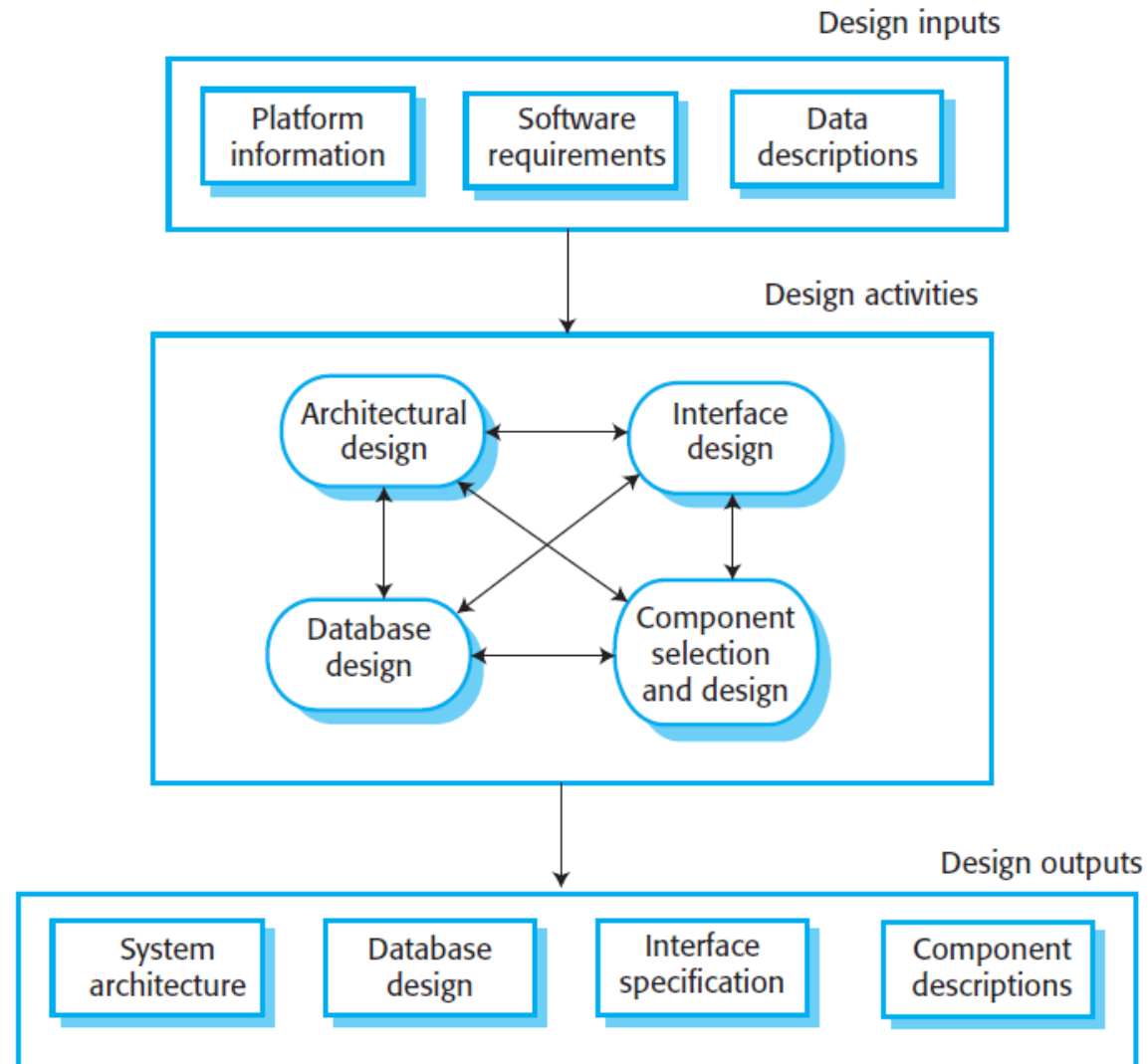
- Conceptos de diseño de software
- Diseño arquitectural vs. Diseño detallado
- Estilos arquitectónicos de software
- Control de versiones de software

# Conceptos de diseño de software

# ¿Qué es diseño de software?

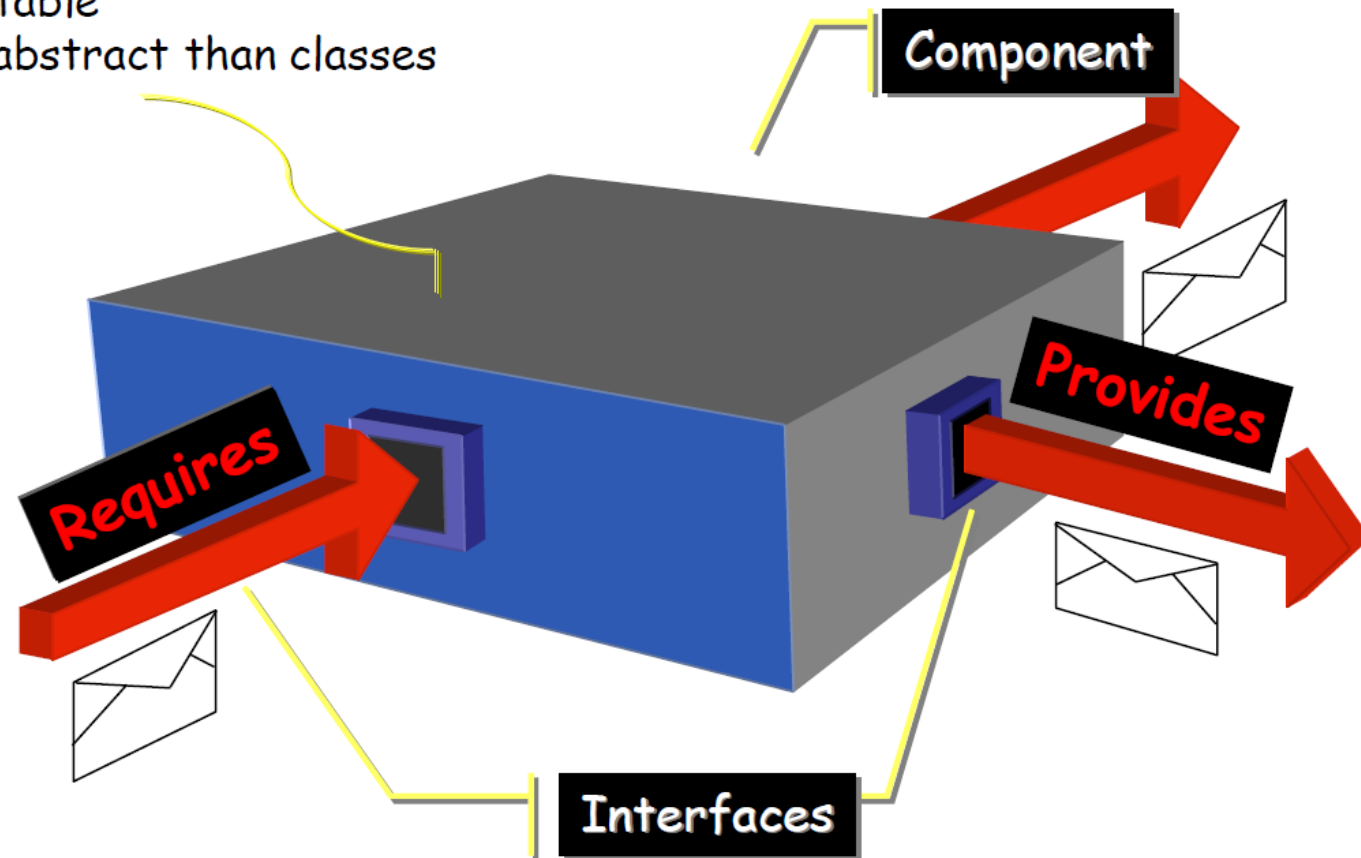
- Se puede referir a dos cosas: a un proceso o al resultado de este.
- “Diseño de software es un **proceso** que define la **arquitectura, componentes, interfaces** y otras características de un sistema o un componente”.
  - Software Engineering Body of Knowledge (SWEBOK)
- “Un diseño de software es una **descripción de la estructura del software** a ser implementado, **los modelos de datos** y estructuras usadas por el sistema, las **interfaces entre los componentes** del sistema y, algunas veces, los **algoritmos** utilizados”.
  - Ian Sommerville

# Un modelo general de proceso de diseño



# ¿Qué es un componente de software?

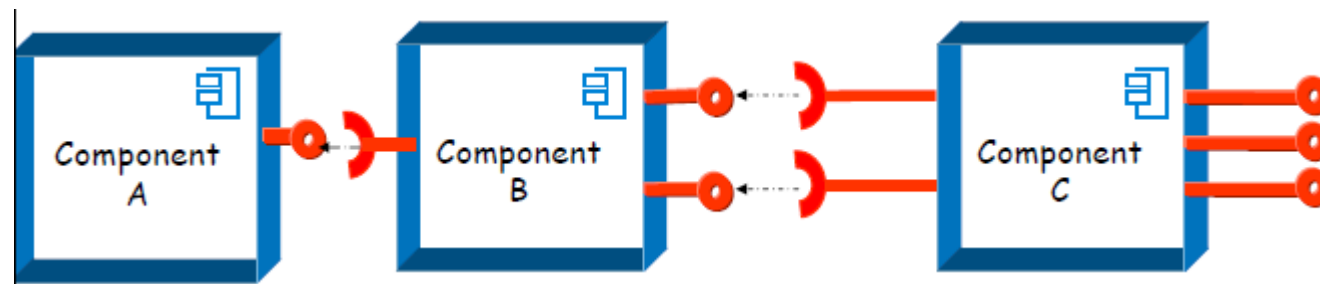
- Provides a service: implementation-independent
- Need not to be compiled
- Executable
- More abstract than classes



Cortesía: Dr. Rami Bahsoon, Component-Based Software Engineering Module

# ¿Qué es un componente?

- A component is a modular building block for computer software.
- More formally, a component is “a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”
  - OMG Unified Modelling Language Specification



Cortesía: Dr. Rami Bahsoon, Component-Based Software Engineering Module

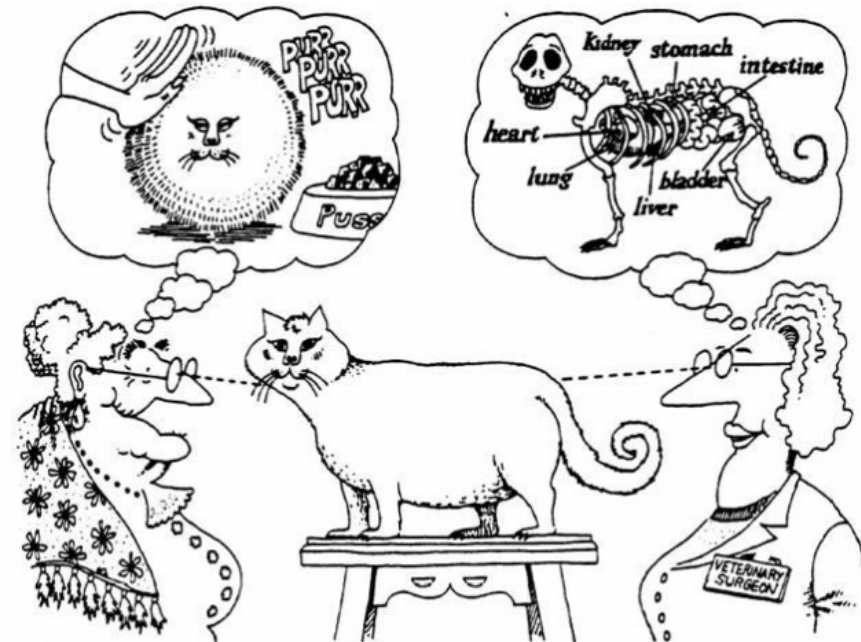
# Principios de diseño de software

- Los principios de diseño de software son nociones **fundamentales** que proveen las bases para **diferentes enfoques** de diseño de software. Son **siete**:
  - abstracción;
  - cohesión y acoplamiento;
  - descomposición y modularización;
  - encapsulamiento/ocultamiento de información;
  - separación de interfaz e implementación;
  - suficiencia, completitud y primitivismo;
  - y separación de preocupaciones.



# Principios de diseño de software

1. **Abstracción** se refiere a la vista de un objeto enfocándose en la información relevante a un propósito en particular e ignorando el resto de la información.



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

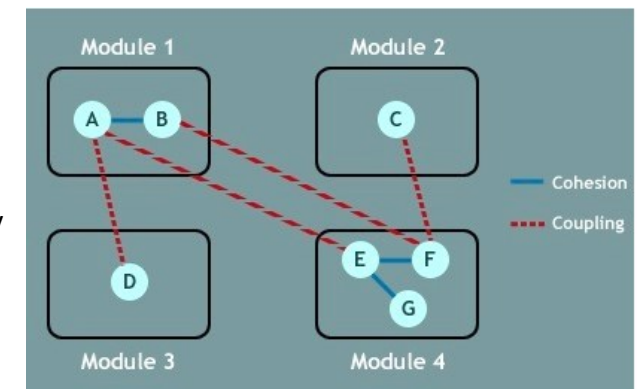
Cortesía: Hayim Makabee, The Cat as a Metaphor in Object-Oriented Software Development

# Principios de diseño de software

2. **Acoplamiento** se define como el **grado de interdependencia o interconexión entre los módulos** de un software, mientras que **cohesión** se define como el **grado de relación entre las responsabilidades asignadas a un módulo**.

- Si hay alta interdependencia, entonces los cambios en un módulo tendrán efectos significativos sobre el comportamiento de los demás.
- Si un módulo es responsable de un número de funciones no relacionadas, entonces las funcionalidades no han sido bien distribuidas entre los módulos.

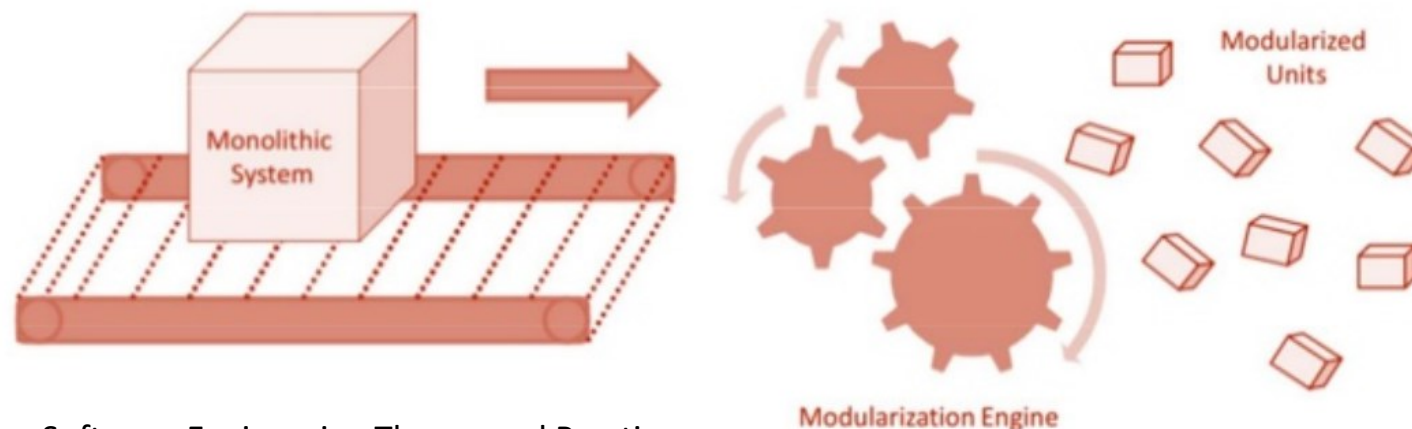
Cortesía: Masudur Rahman, Recommendation of Move Method Refactorings Using Coupling, Cohesion and Contextual Similarity



- Ideal: **alta cohesión y bajo acoplamiento**.

# Principios de diseño de software

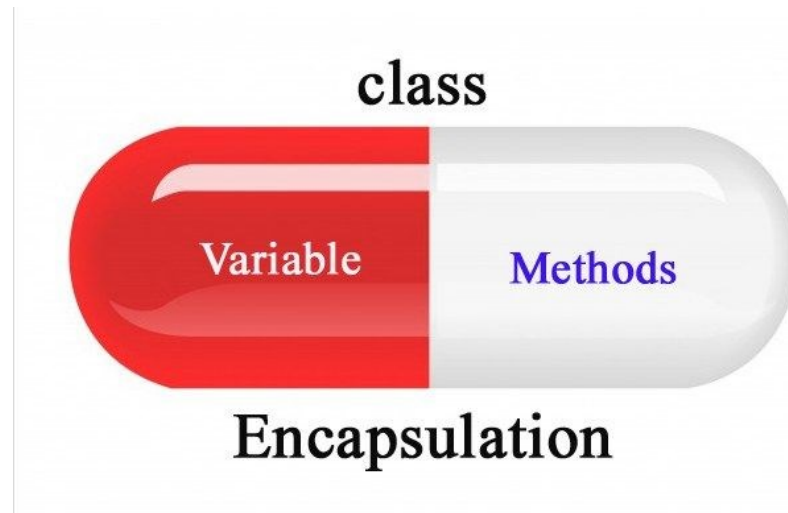
- 3. Descomposición y modularización** significa que un software es dividido en un número de componentes más pequeños con interfaces bien definidas que describen las interacciones entre los componentes.
- Usualmente, el objetivo es ubicar diferentes funcionalidades y responsabilidades en diferentes componentes.



Cortesía: Pfleeger, Software Engineering Theory and Practice

# Principios de diseño de software

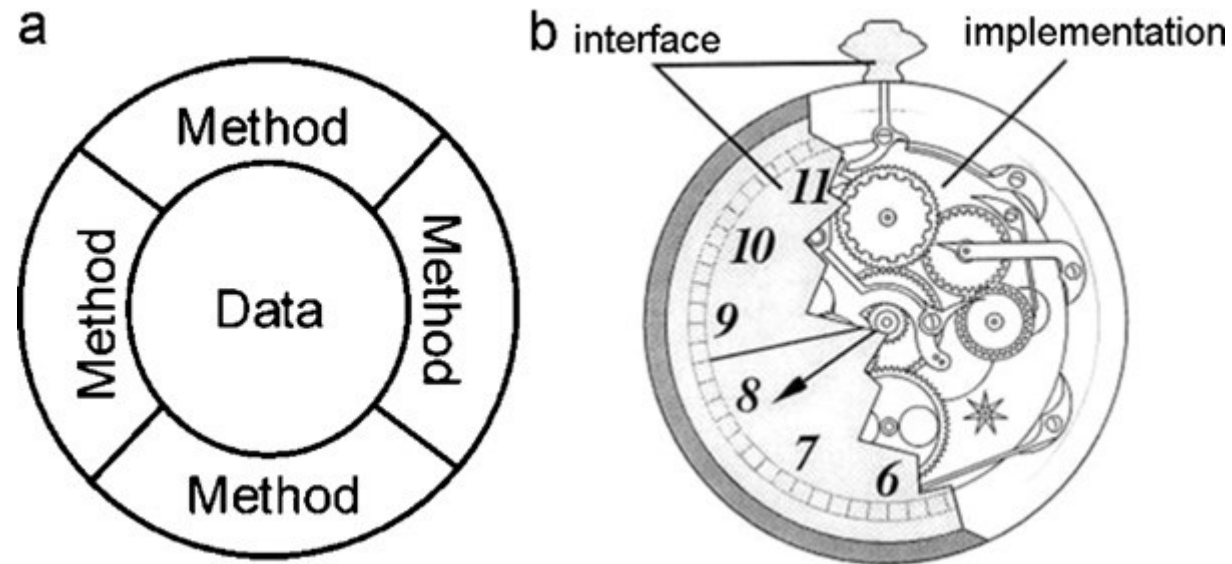
4. **Encapsulamiento y ocultamiento** de información significa agrupar y empaquetar los detalles internos de una abstracción y hacer esos detalles inaccesibles a entidades externas.



Cortesía: Sathasivam Karmehavannan, Encapsulation in Java programming language - Codeforcoding

# Principios de diseño de software

**5. Separación de interfaz e implementación** implica definir un componente especificando una interfaz pública (conocida para los clientes) que está separada de los detalles de cómo el componente está implementado.



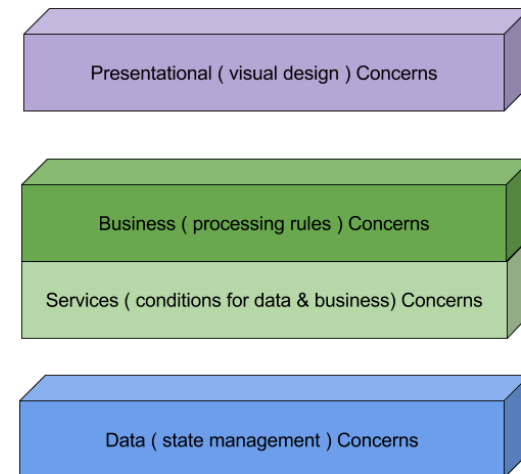
Cortesía: Li An, Modeling Human Decisions in Coupled Human and Natural Systems: Review of Agent-Based Models

# Principios de diseño de software

**6. Suficiencia, completitud y primitivismo:** Alcanzar suficiencia y completitud significa asegurar que un componente de software **captura todas las características** importantes para alcanzar su objetivo, no más y no menos. Primitivismo significa que el diseño debería estar **basado en patrones** que sean de fácil implementación.

# Principios de diseño de software

**7. Separación de preocupaciones:** Una preocupación (concern) es un área de interés con respecto al diseño de un software. La arquitectura de un software debe tener una o **varias vistas** de preocupaciones que permitan a los interesados **concentrarse en pocas cosas a la vez** a fin de reducir la complejidad.



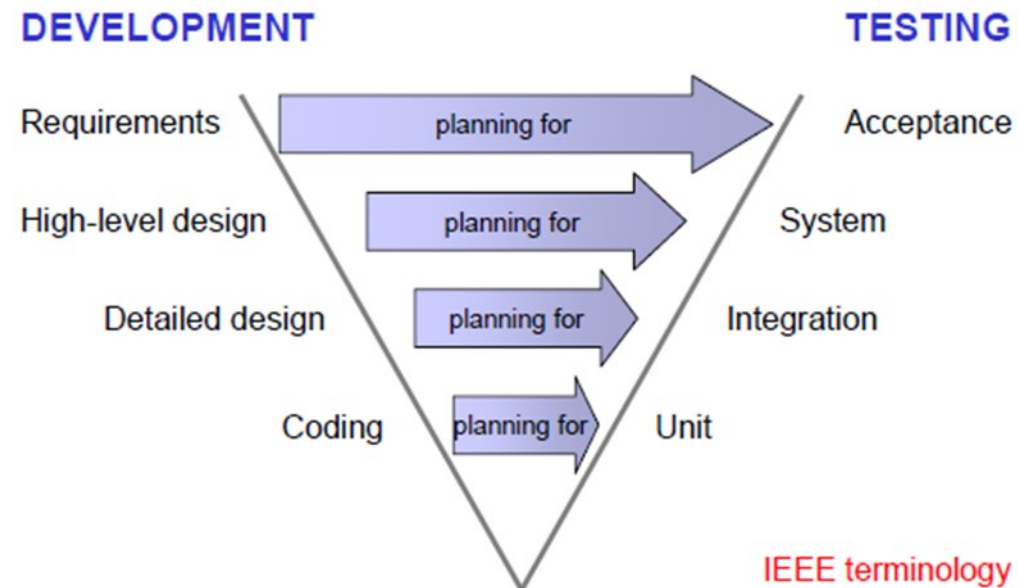
Cortesía: Willie Streeter, Practical Web Development and Architecture

# Diseño arquitectural vs. Diseño detallado



# Diseño de software

- Dos niveles de diseño de software:
  1. Diseño de alto nivel o arquitectural
  2. Diseño detallado

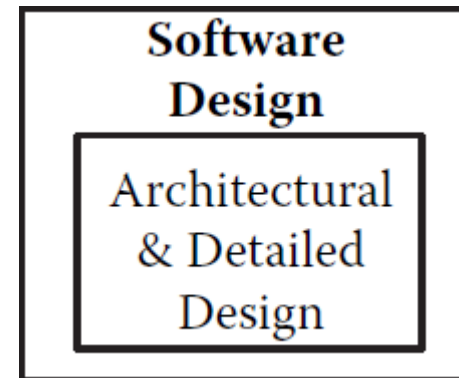


# Diseño de alto nivel o arquitectural

- La palabra **arquitectura** es a menudo utilizada en el contexto de algo a **alto nivel**, es decir, está separado de detalles a bajo nivel.
- Es un enfoque de **macro diseño** para crear modelos que ilustren aspectos de calidad y funciones del sistema de software.
- ¿Qué se decide en el diseño arquitectural de un software?
  - **Plataforma** tecnológica
  - **Despliegue** físico del sistema de software, incluyendo subsistemas ubicados en diferentes lugares.
  - Selección de los principales **componentes estructurales**
  - La forma en la que el sistema de software, como un todo, se va a **comunicar** (ej.: protocolos de comunicación)
  - Problemas de **concurrency**
  - Se evalúa **requerimientos** no funcionales (ej.: desempeño, seguridad, robustez, escalabilidad)
  - Entre otros

# Diseño detallado

- Se **refina** el diseño arquitectural a un punto en el que el **diseño** está lo suficientemente **completo** como para **empezar la construcción** del software.
- ¿Qué se decide en el diseño detallado de un software?
  - Se refina componentes en **clases**
  - Se implementan **interfaces**
  - Se especifica las **relaciones** entre clases
  - Se identifica y aplica **patrones** de diseño
  - Se diseña **componentes** y sus interfaces



# Estilos arquitectónicos

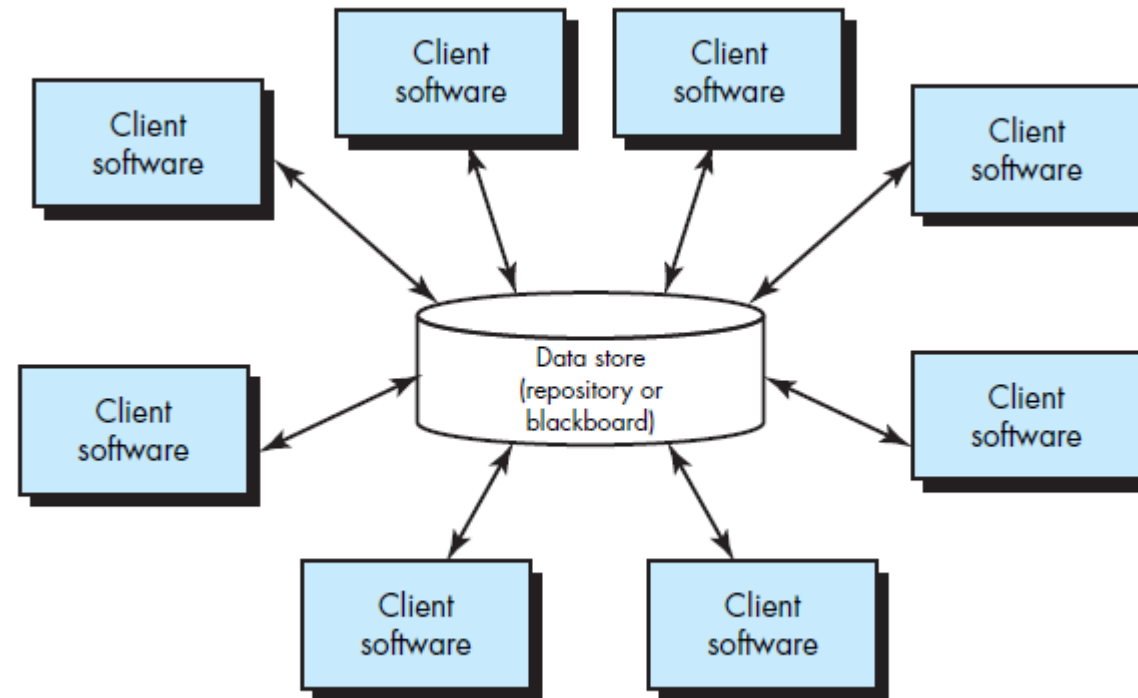
# Estilos arquitectónicos

- En construcción, se utiliza estilo arquitectónico como un mecanismo descriptivo para diferenciar los diferentes **estilos de construcciones**.
- Cada estilo arquitectónico de software incluye:
  1. un conjunto de **componentes** que llevan a cabo la función requerida por el sistema;
  2. un conjunto de **conectores** que permiten la **comunicación, coordinación y cooperación** entre componentes;
  3. **restricciones** que definen cómo se **integran** los componentes para formar el sistema; y
  4. modelos **semánticos** que permiten al diseñador entender las propiedades del sistema.

# Estilos arquitectónicos

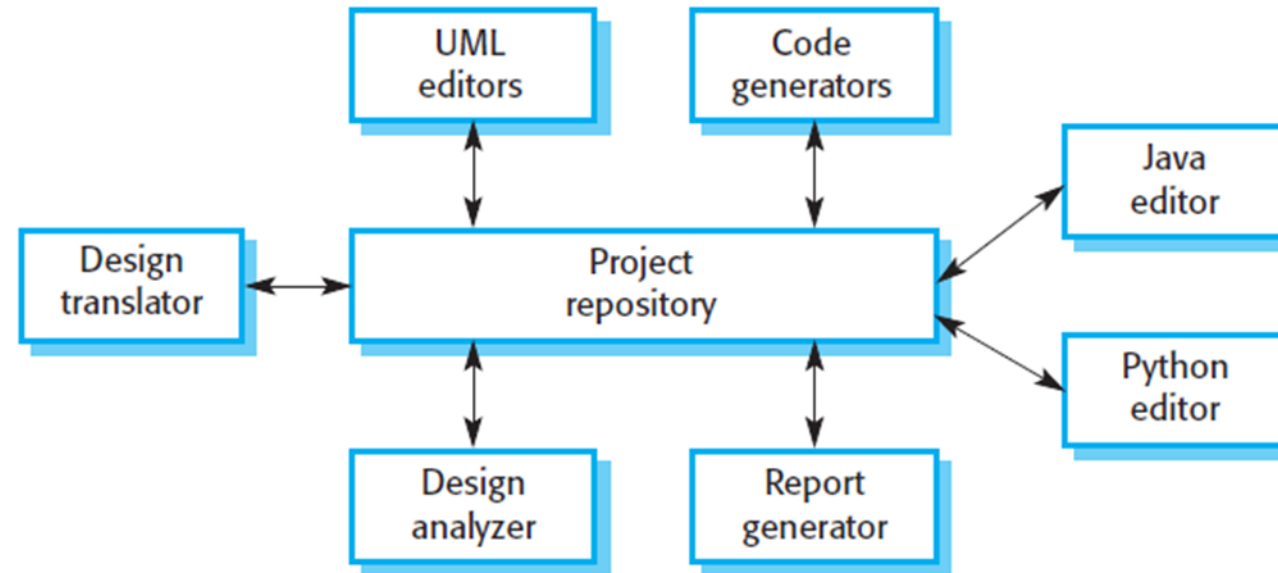
## 1. Arquitectura centrada en datos

- Un repositorio de datos es accedido frecuentemente por otros componentes para actualizar, agregar, eliminar o modificar datos en el repositorio.



# Estilos arquitectónicos

## 1. Arquitectura centrada en datos

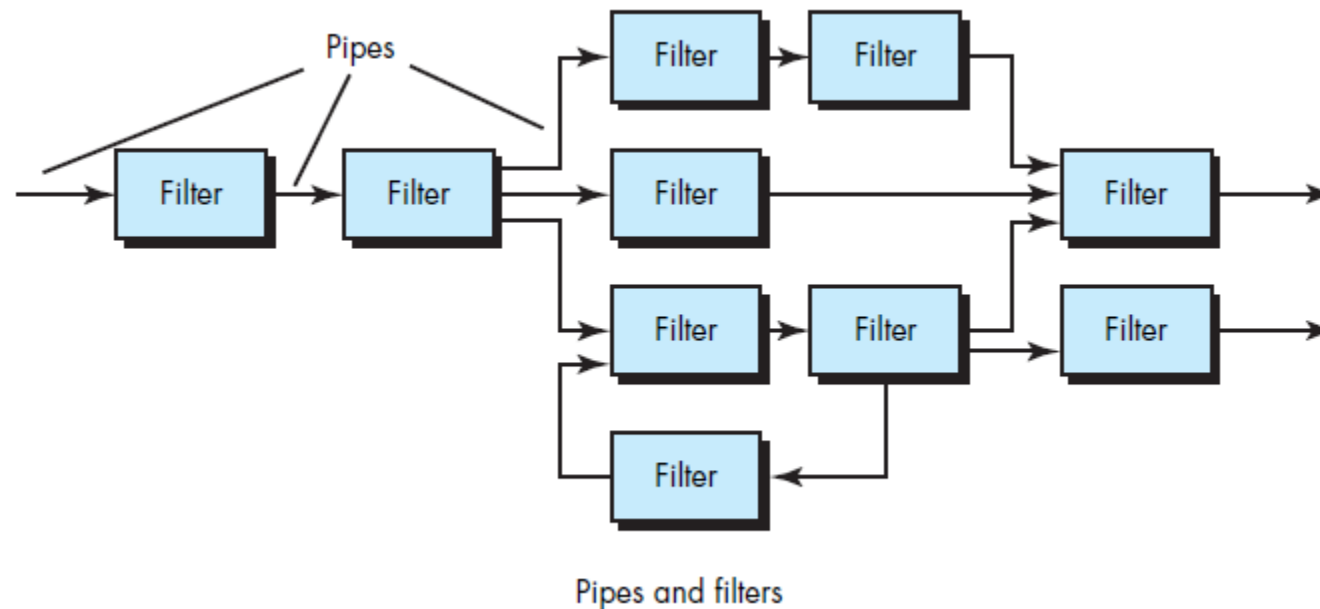


**Figure 6.11** A repository architecture for an IDE

# Estilos arquitectónicos

## 2. Arquitectura de flujo de datos

- Un dato de entrada es transformado en un dato de salida a través de una serie de componentes de manipulación o cálculo.

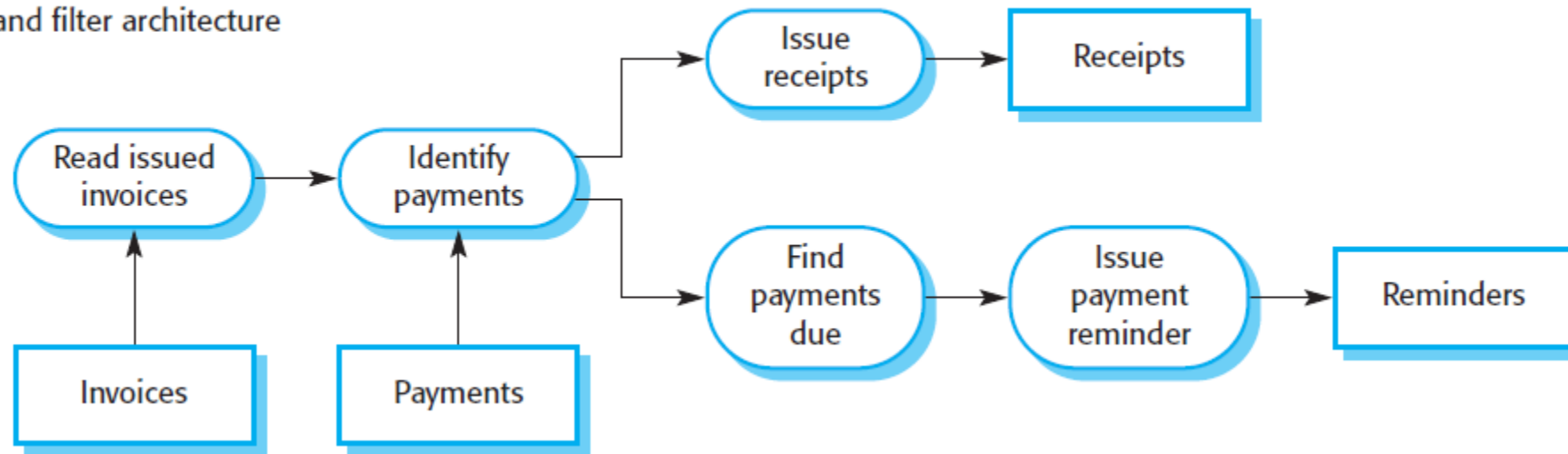




# Estilos arquitectónicos

## 2. Arquitectura de flujo de datos

**Figure 6.15** An example of the pipe and filter architecture



# Estilos arquitectónicos

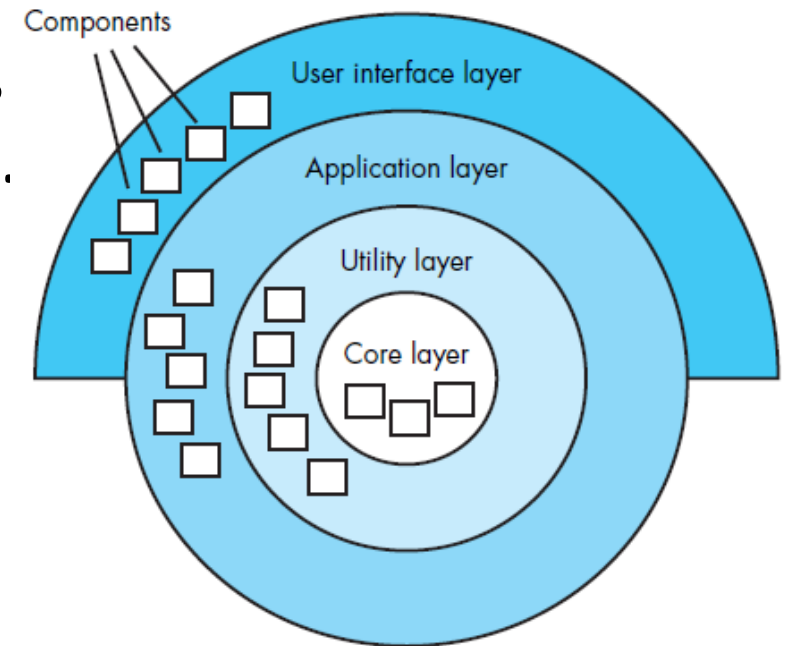
## 2. Arquitectura de flujo de datos

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.15 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse architectural components that use incompatible data structures.

# Estilos arquitectónicos

## 3. Arquitectura en capas

- Se define varias capas y cada una de ellas cumple operaciones que progresivamente se acercan al conjunto de instrucciones de máquina.
- En la capa **más exterior**, los componentes hacen operaciones de **interfaz de usuario**.
- En la capa **más interior**, componentes llevan a cabo operaciones de **interfaz hacia el sistema operativo**.
- Las capas **intermedias** proveen **servicios** utilitarios y funcionales.



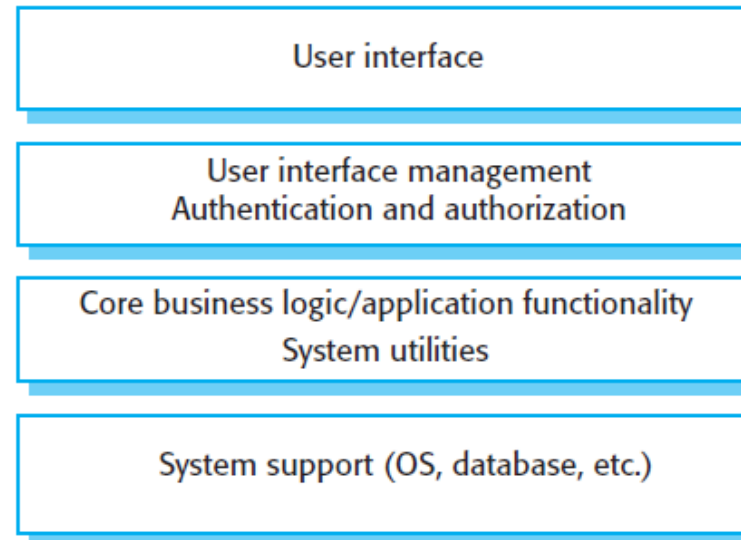
# Estilos arquitectónicos

## 3. Arquitectura en capas

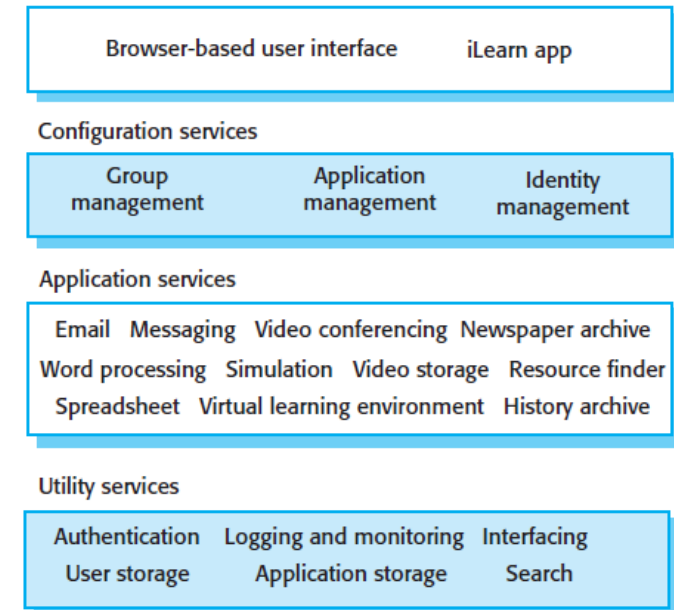
Name	Layered architecture
Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system. See Figure 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

# Estilos arquitectónicos

## 3. Arquitectura en capas



**Figure 6.8** A generic layered architecture

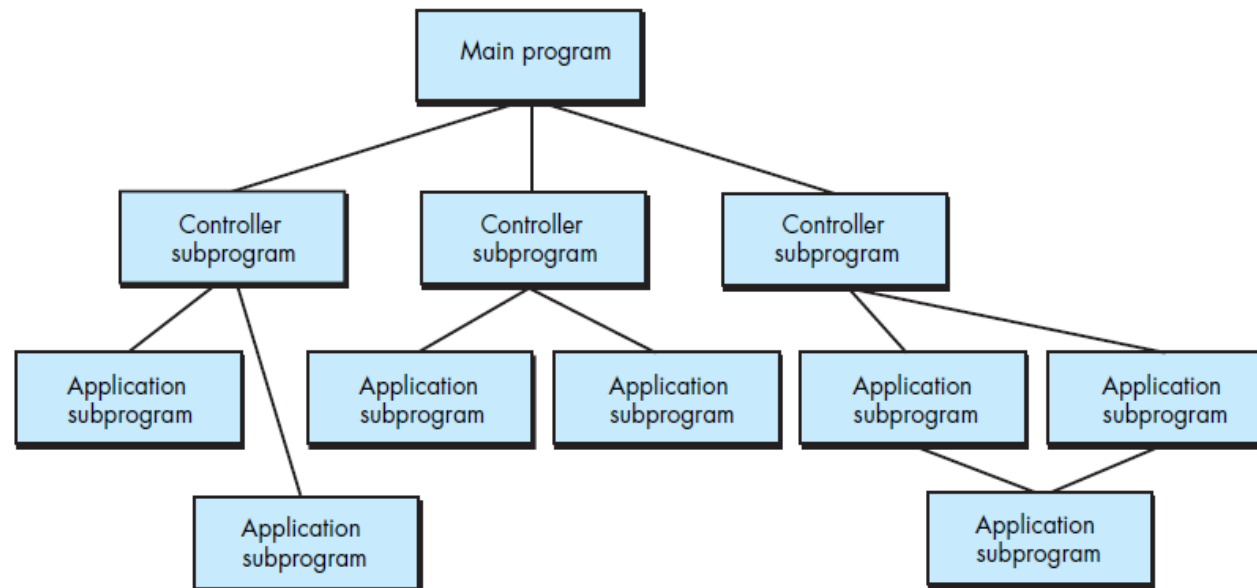


**Figure 6.9** The architecture of the iLearn system

# Estilos arquitectónicos

## 4. Arquitectura de llamada y retorno

- Se centra en la creación de una **estructura** de programa que sea fácilmente **modificable y escalable**. Principales **subcategorías**:
  - Arquitectura de **programa principal /subprograma**
  - Arquitectura de **llamada a procedimiento remoto**



# Estilos arquitectónicos

## 5. Arquitectura orientada a objetos.

- Los componentes de un sistema **encapsulan los datos y las operaciones** que se deben aplicar para manipular los datos.
- La **comunicación y coordinación** entre componentes debe ser realizada **por medio de paso de mensajes**.

# Control de versiones de software



# Control de versiones de software

- **Manejo de versiones es el proceso de mantener un registro de las diferentes versiones de los componentes de un software y los sistemas en los cuales estos componentes se usan.**
- **El manejo de versiones implica asegurar que los cambios hechos por diferentes desarrolladores a estas versiones no interfieren con la anterior.**
- **Los sistemas de control de versiones automatizan el manejo de versiones.**

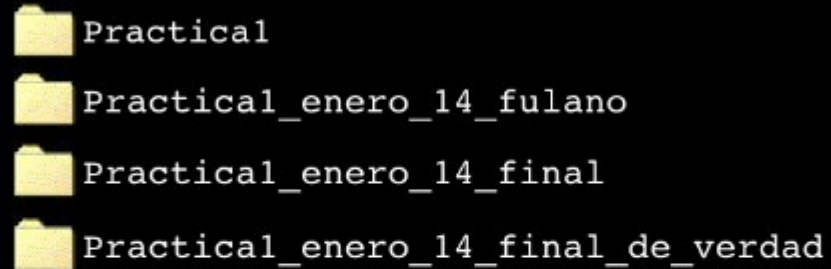
# Control de versiones de software

- **Ventajas:**

- Almacenamiento y backup
- Control de acceso mediante permisos
- Deshacer 'ilimitado'
- Combinar aportaciones de distintos colaboradores
- Sincronización de desarrolladores
- Histórico de cambios
- Versiones en paralelo

# Control de versiones de software

Copiar y Pegar archivos



- Practical
- Practical\_enero\_14\_fulano
- Practical\_enero\_14\_final
- Practical\_enero\_14\_final\_de\_verdad

**NO**

es Control de Versiones

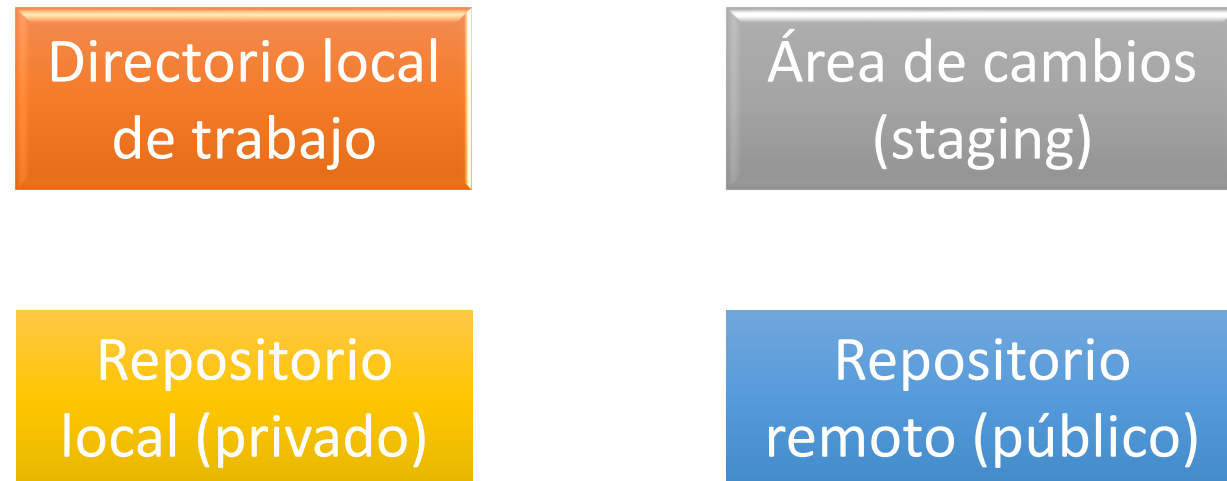
# Control de versiones de software

- **Vocabulario básico:**

- **Repositorio:** Almacén de datos que guarda cada versión de nuestro proyecto, incluyendo los datos asociados a cada commit.
- **Commit:** Cambio de una versión a otra.
- **Stage:** Lugar donde se guardan los cambios que serán parte de un commit.
- **Directorio:** Dirección dentro de mi computador donde tengo archivos, pero estos pueden o no estar versionados por git.

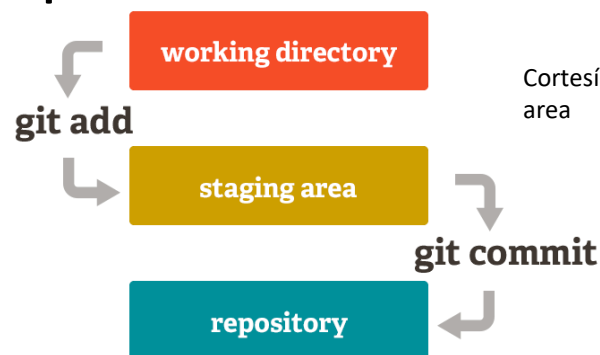
# Git

- Es un sistema de versionamiento distribuido.
- Aunque se trabaje con un repositorio remoto, siempre se tiene uno local.

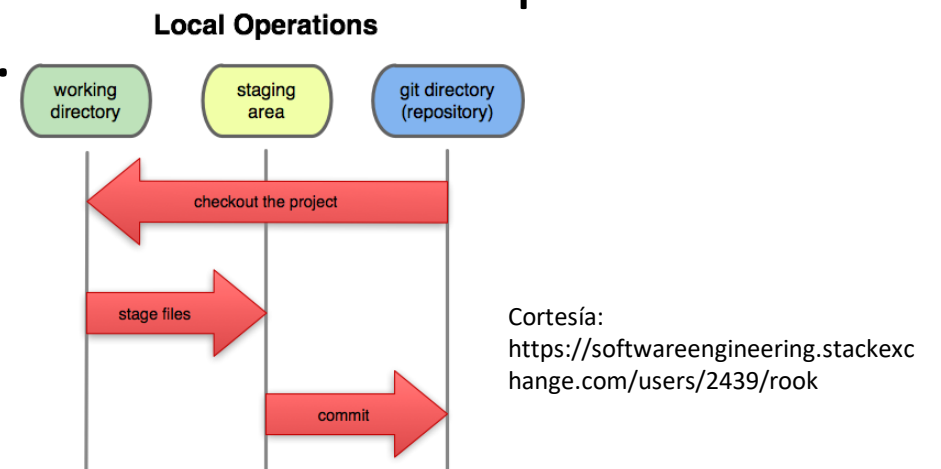


# Git: Staging Area

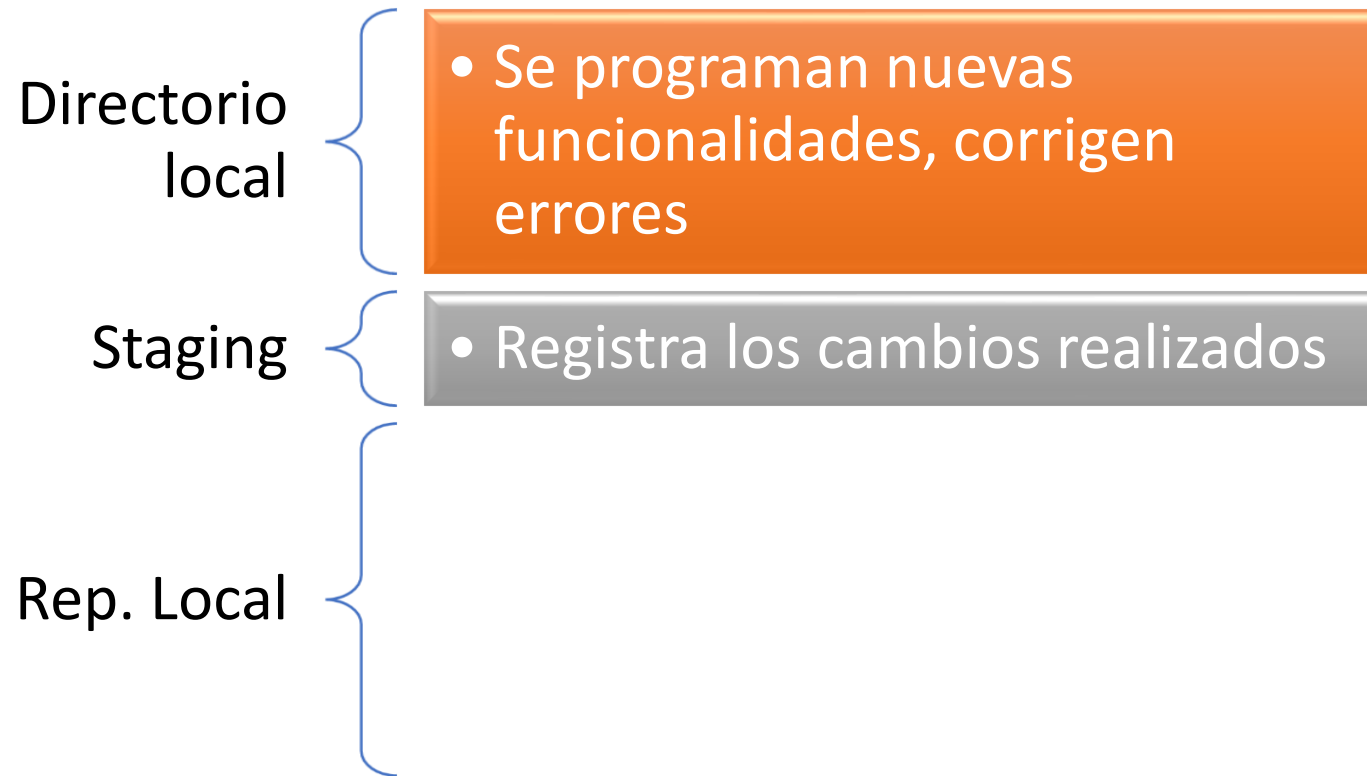
- Es un área intermedia en donde los cambios pueden ser formateados y revisados antes de hacer el commit.
  - Digamos que estamos trabajando en dos características: una está terminada y la otra en progreso. Usted quisiera hacer commit e irse a casa a las 17.00, pero sin hacer commit de lo concerniente a la segunda característica.
  - Usted hace stage de las partes que usted sabe corresponden a la primera característica, y hace commit.



Cortesía: <https://git-scm.com/about/staging-area>

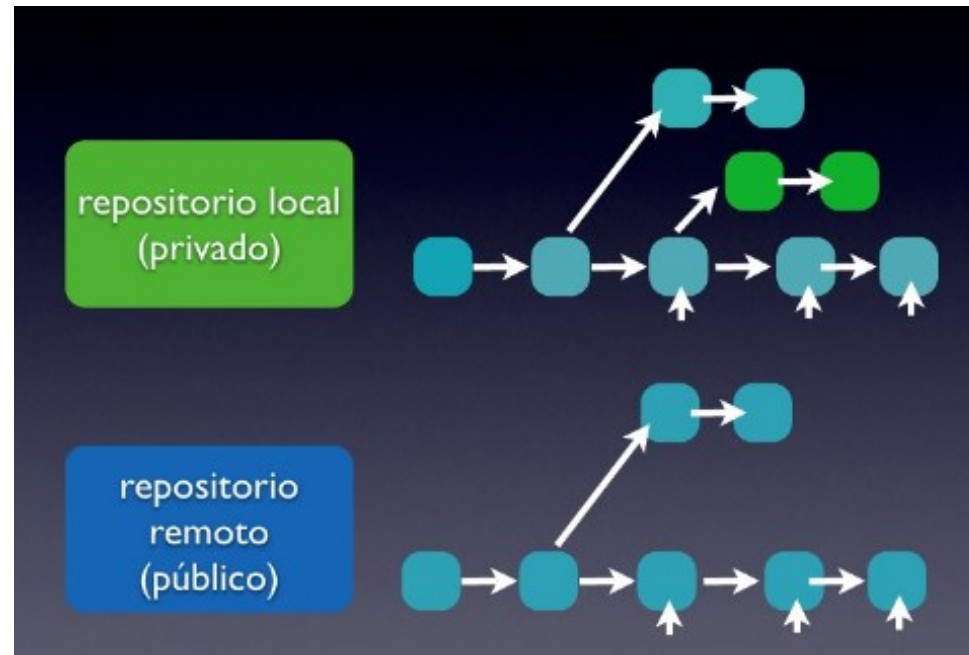


Cortesía: <https://softwareengineering.stackexchange.com/users/2439/rook>



# Ramas (Branches)

- Se pueden llevar ramas para desarrollar en local, otras en remoto para manejar las versiones del programa, sincronizarlas entre sí...

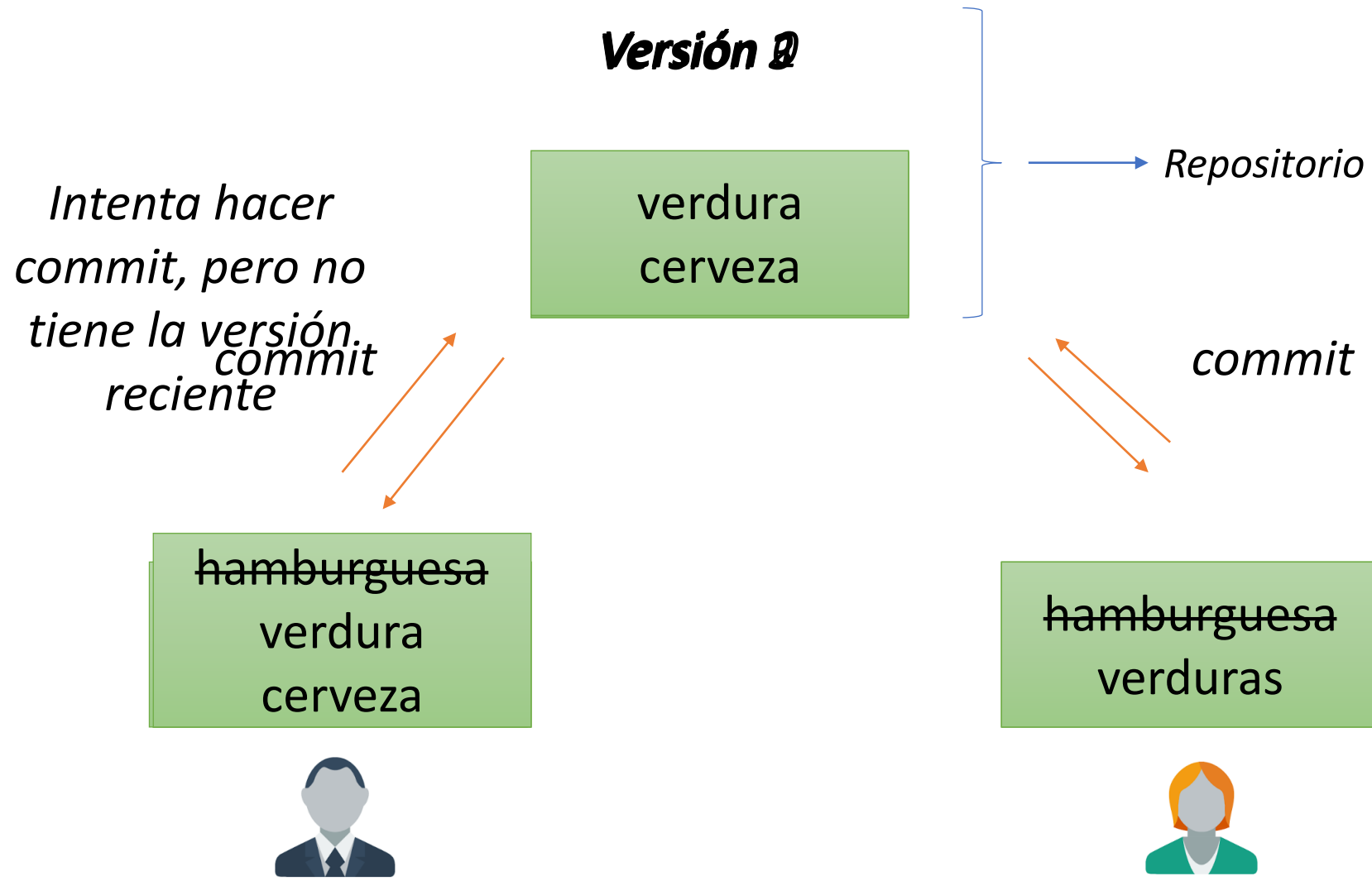




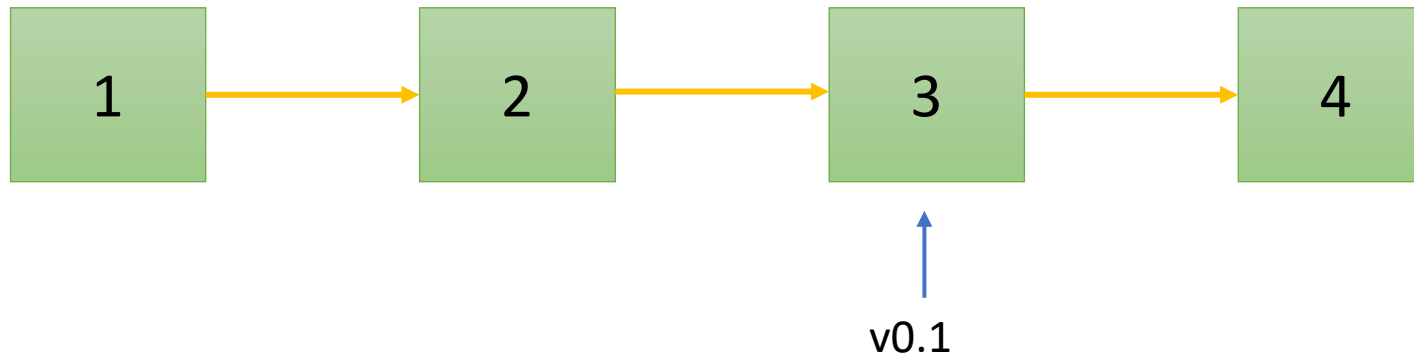
# GitHub

- Acceso web a repositorios Git.
- Gratuito para proyectos libres.
- Antepasados: sourceforge, Savannah, gforge
- “Red social” de programadores.
- Currículo para las empresas.
- Acceso mediante la web con usuario y contraseña.
- Acceso por consola con clave pública y privada SSH-Key.

# Ejemplo: Lista de Compras



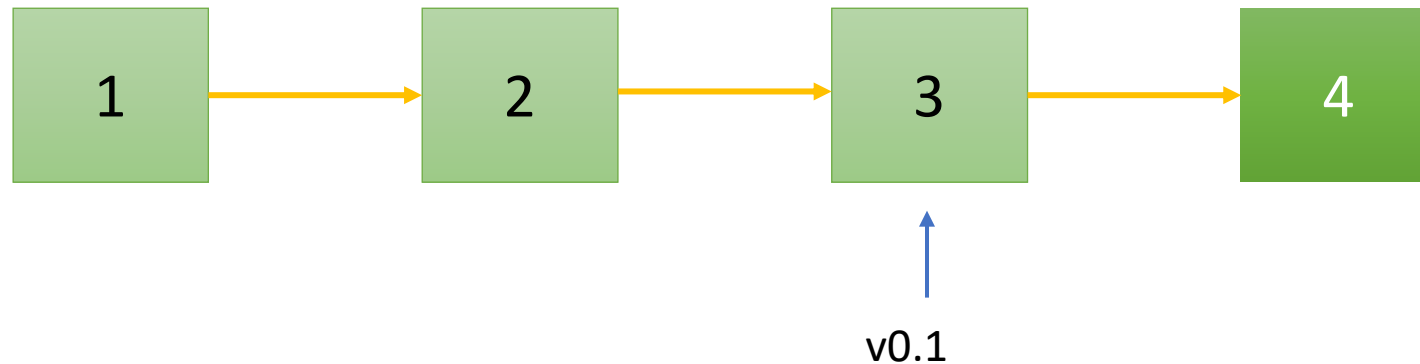
# Control de versiones de software



Se pueden etiquetar versiones concretas para localizarlas fácilmente en la historia del repositorio.

# Control de versiones de software

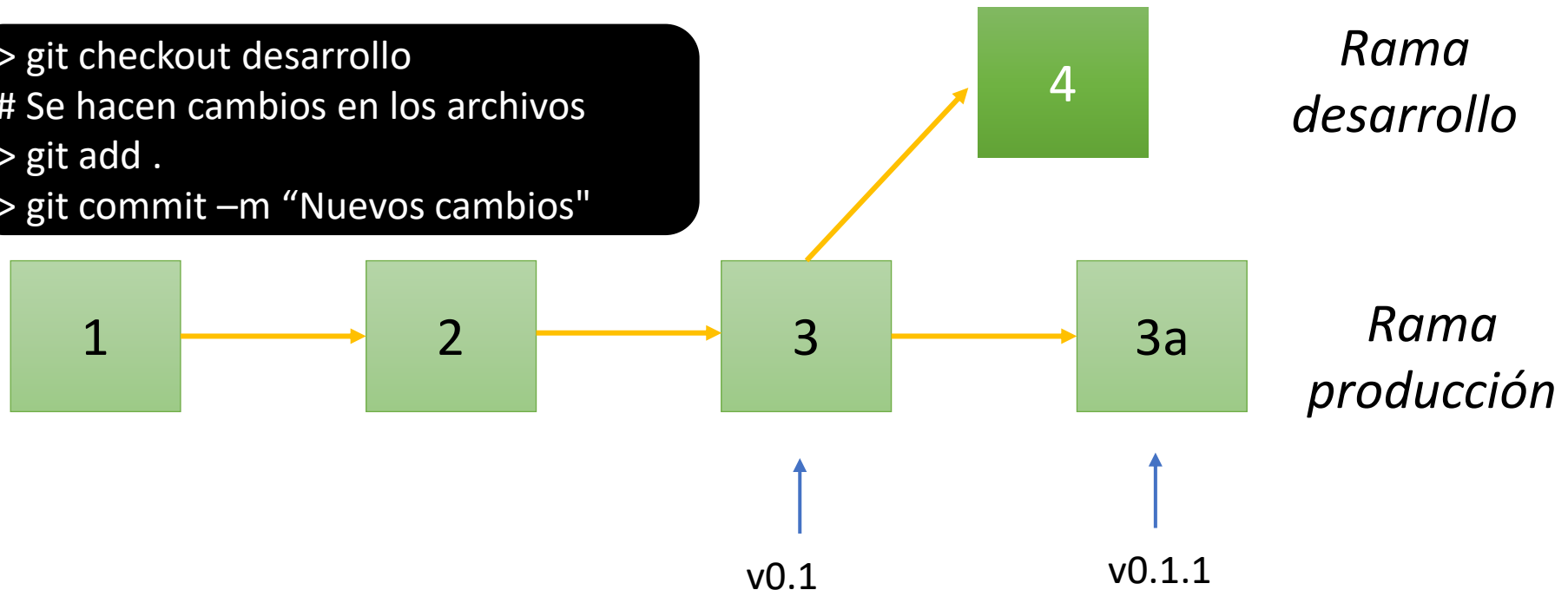
- ¿Qué ocurre si estamos desarrollando la funcionalidad 4 y se descubre un bug en alguna de las anteriores?



# Control de versiones de software

- Para evitar esto se puede trabajar en “ramas”

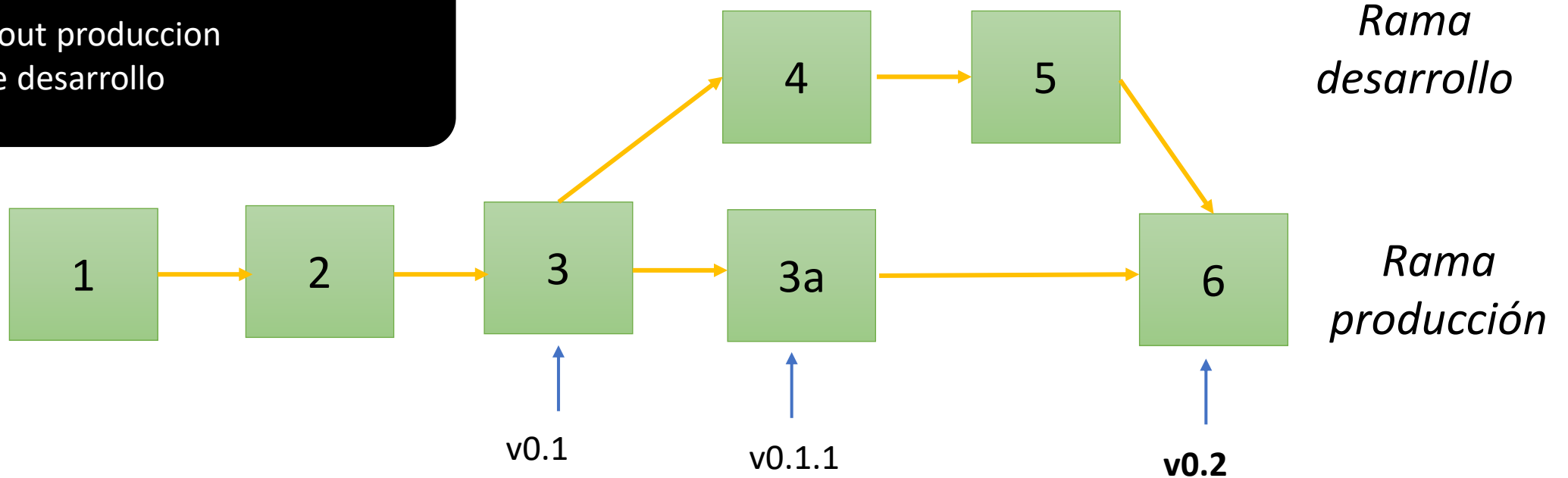
```
> git checkout desarrollo  
# Se hacen cambios en los archivos  
> git add .  
> git commit -m "Nuevos cambios"
```



# Control de versiones de software

- Cuando la rama de desarrollo sea estable, la fusionaremos con la de producción

```
> git checkout produccion  
> git merge desarrollo
```



# GitHub

- La rama principal antes se llamada **master**. (quizá lo vean en ejemplo de Internet)
- La rama principal ahora se llama **main**.
- En Windows y Mac se puede usar **GitHub Desktop** y usarlo sin necesidad de clave SSH.
- Desde **consola** se deben generar las **claves pública y privada SSH** y luego agregar en nuestro perfil de GitHub la clave pública.

Más información en: <https://docs.github.com/es/authentication>

# Antes de finalizar



# Puntos para recordar

- ¿Qué es diseño de software?
- ¿Qué se decide en el diseño de alto nivel y en el detallado?
- ¿Cuáles son los principios de diseño de software?
- Descripción de los principales estilos arquitectónicos
- Conceptos básicos de control de versiones.
- ¿Cómo se crea una nueva versión de mi código en Git?
- ¿Qué son las ramas?

# Lectura adicional

- IEEE, “Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK)”
  - Chapter 2: Software Design
- Pressman and Maxin, “Software Engineering”
  - Chapter 12: Design Concepts
  - Chapter 13: Architectural Design
- Robert Martin, Clean Architecture
  - Chapter 1: What is Design and Architecture
- Gui and Scott, “Measuring Software Component Reusability by Coupling and Cohesion Metrics”
- Ian Sommerville, “Software Engineering”
  - Chapter 2: Software Processes
  - Chapter 6: Architectural Design

# Lectura adicional

- Carlos E. Otero, “Software Engineering Design: Theory and Practice”
  - Chapter 1: Introduction to Software Engineering Design
- ➔ • LinkedIn Learning “Resolución de conflictos en GitHub”
  - <https://www.youtube.com/watch?v=H34vxQeQkgg>
- Bourke, “Introduction to Git”
  - <http://cse.unl.edu/~cbourke/gitTutorial.pdf>
- Tutorials Point, “Git Tutorial”
  - [https://www.tutorialspoint.com/git/git\\_create\\_operation.htm](https://www.tutorialspoint.com/git/git_create_operation.htm)
- Loeliger and McCullough, “Version Control with Git”
- Atlassian, “Become a Git Guru”
  - <https://www.atlassian.com/git/tutorials>

# Próxima sesión

- DevOps y Entrega Continua