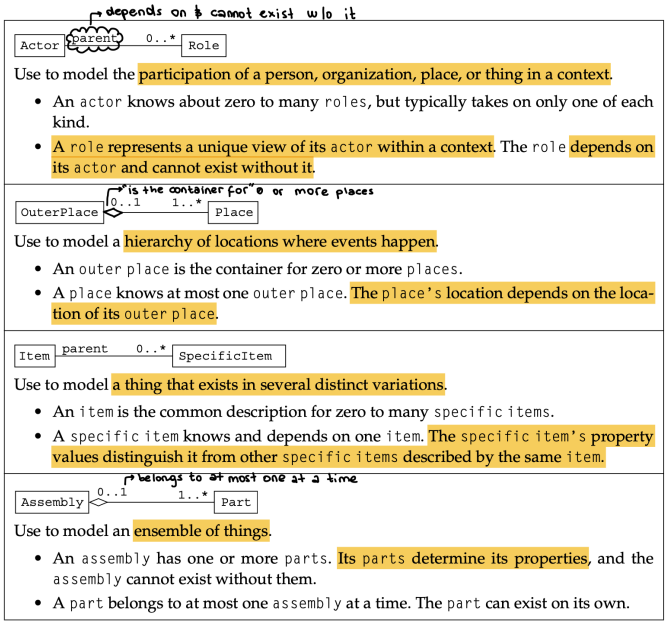
Modelo de Dominio



Use to model the participation of a person, organization, place, or thing in a context.

- An `actor` knows about zero to many `roles`, but typically takes on only one of each kind.
- A `role` represents a unique view of its `actor` within a context. The `role` depends on its `actor` and cannot exist without it.

Use to model a hierarchy of locations where events happen.

- An `outer place` is the container for zero or more `places`.
- A `place` knows at most one `outer place`. The `place's` location depends on the location of its `outer place`.

Use to model a thing that exists in several distinct variations.

- An `item` is the common description for zero to many `specific items`.
- A `specific item` knows and depends on one `item`. The `specific item's` property values distinguish it from other `specific items` described by the same `item`.

Use to model an ensemble of things.

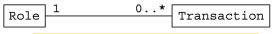
- An `assembly` has one or more `parts`. Its `parts` determine its properties, and the `assembly` cannot exist without them.
- A `part` belongs to at most one `assembly` at a time. The `part` can exist on its own.

Use to model a receptacle for things.

- A `container` holds zero or more `content` objects. Unlike an `assembly`, it can be empty.
- A `content` object can be in at most one `container` at a time. The `content` object can exist on its own.

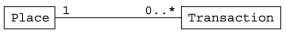
Use to model a classification of things.

- A `group` contains zero or more `members`. `Groups` are used to classify objects.
- A `member`, unlike a `part` or `content` objects, can belong to more than one `group`.



Use to record participants in events.

- A `transaction` knows one `role`, the doer of its interaction.
- A `role` knows about zero or more `transactions`. The `role` provides a contextual description of the person, organization, thing, or place involved in the `transaction`.

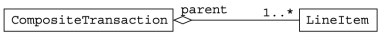


Use to record where an event happens.

- A `transaction` occurs at one `place`.
- A `place` knows about zero to many `transactions`. The `transactions` record the history of interactions at the `place`.

Use to record an event involving a single thing.

- A `transaction` knows about one `specific item`.
- A `specific item` can be involved in zero to many `transactions`. The `transactions` record the `specific item's` history of interactions.

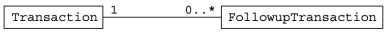


Use to record an event involving more than one thing.

- A `composite transaction` must contain at least one `line item`.
- A `line item` knows only one `composite transaction`. The `line item` depends on the `composite transaction` and cannot exist without it.

Use to record the particular involvement of a thing in an event involving multiple things.

- A `specific item` can be involved in zero to many `line items`.
- A `line item` knows exactly one `specific item`. The `line item` captures details about the `specific item's` interaction with a `composite transaction`.



Use to record an event that occurs only after a previous event.

- A `transaction` knows about some number of `follow-up transactions`.
- A `follow-up transaction` follows and depends on exactly one previous `transaction`.

Principios SOLID

Single Responsibility

📜 Una clase sólo debe tener una razón para cambiar.

Open-Closed

📜 Las entidades deben estar abiertas para la extensión pero cerradas para la modificación:

+ el comportamiento del módulo es extensible
- el código fuente es inviolable

Liskov Substitution

📜 Las sub-clases pueden ser utilizadas en lugar de clases base sin modificar el comportamiento de la clase base.

Interface Segregation

📜 Los clientes no deben depender de interfaces que no usan.

Dependency Inversion

📜 Módulos de alto nivel no deben depender de los de bajo nivel; ambos deben depender de abstracciones.

Las abstracciones no deben depender de detalles, los detalles deben depender de abstracciones.

Clean Code

- ★ Nombres Significativos
  - revelar intenciones
  - distinciones significativas
  - pronunciables
  - buscables
  - clases → sustantivos
  - métodos → verbos
  - elegir una palabra por concepto
  - usar dominio de solución (algoritmos, patrones, términos matemáticos)
  - contexto significativo & reducido
  - evitar
    - desinformación
    - codificaciones
    - prefijos
    - mapeo mental
    - copiar palabras
- ★ Funciones
  - chicas → 2-4 líneas
  - bloques e identación → 1-2 líneas
  - una cosa (hacer algo o responder algo pero no las dos)
  - sin secciones
  - un nivel de abstracción
    - nombres descriptivos → mientras más chica más fácil
    - lectura de arriba hacia abajo → *The Stepdown Rule*
    - bloques de switch tolerados si
      - aparecen una vez
      - son usados para crear crear objetos polimórficos
      - están ocultos en una relación de herencia
  - argumentos

        - 0 == niládica → ideal
        - 1 == monádica → pregunta, operación, transformación
        - 2 == diádica → más difíciles de entender
        - 3 == triádica → muy difíciles de entender
        - > 3 == poliádica → justificación, no deberían usarse
        - flag → no usar
        - output → evitarlos, cambiar un objeto pero no devolverlo
    - excepciones en lugar de códigos de error
- ★ Comentarios
  Explicar el porqué y no el cómo
    - BUENOS
        - legales
        - informativos
        - explicativos
        - clarificadores
        - previsores de consecuencias
        - recordatorios de trabajo
        - amplificadores de importancia
    - MALOS
        - balbuceos
        - redundantes
        - confusos
        - obligatorios e innecesarios
        - periódicos
        - ruidosos
        - en reemplazo de una función / variable
        - marcadores de posición
        - cierre de corchetes
        - código viejo
        - información no local
        - conexiones no-obvias
        - cabeceras de funciones

Patrones de Arquitectura

Micro Kernel

Agrega funcionalidades adicionales como plug-ins al core de la aplicación. Provee
- + extensibilidad
- + separación

Se utiliza el core a través de los servicios externos, generalmente a través de un adapter.
La existencia de los plug-ins debe ser independiente al ‘core’; es decir, si un plug-in es modificado no es necesario hacer un re-deploy o recompilar el ‘core’

Componentes

- ➔ Core → lógica principal de la aplicación
- ➔ Plug-In Modules/Servers → funcionalidad adicional

Servicios

- Internos → extender la funcionalidad del core
- Externos → implementar abstracciones sobre el core mediante servicios

MVC

Desarrollo de interfaces de usuario:

- \+ divide las responsabilidades de la lógica de la aplicación de la vista
- \+ separa el dominio lógico de las vistas desacoplándolas

Componentes

- ➔ Modelo
  - ◆ componente central
  - ◆ maneja los datos y la lógica de la aplicación
- ➔ Vista
  - ◆ representa la información que contiene el modelo
- ➔ Controlador
  - ◆ controla las operaciones e interacciones entre el usuario y la información
  - ◆ modifica el controlador para que sea leído por la vista

Layers

Los componentes son organizados en capas horizontales, donde una sólo habla con la de abajo si tiene. La interacción entre las capas está estandarizada. Suelen ser:

- ➔ Presentación
- ➔ Servicios (>1 niveles)
- ➔ Dominio (>1 niveles)
- ➔ Persistencia
- ➔ Base de Datos (mismo nivel que persistencia)

Pipe & Filter

Patrón de arquitectura de datos, útil cuando se transforma y analiza la información en múltiples niveles. Se pueden agregar o quitar múltiples niveles de procesamiento de datos.

Broker

Intermediario entre dos partes diferentes de software que tienen que comunicarse entre sí sin saber de la existencia del otro.

Enterprise Architecture

Sistema fuertemente orientado a un negocio que cumple requerimientos no funcionales. Cada capa está aislada lógica y físicamente. Sistema distribuido.

Modelo de Arquitectura 4+1 Vistas

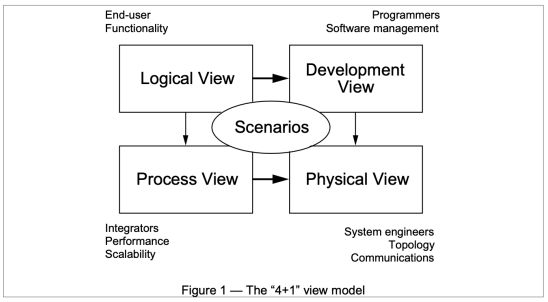


Figure 1 — The "4+1" view model

Vista Lógica

Descomposición orientada a objetos (requerimientos funcionales):

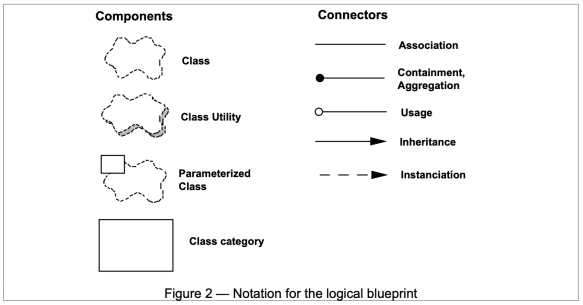


Figure 2 — Notation for the logical blueprint

Vista Procesos

Descomposición de procesos:

- requerimientos no funcionales (performance & disponibilidad)
- problemas de concurrencia, distribución, integridad, tolerancia de fallas
- muchos niveles de abstracción, en el más alto se comunican los procesos
- el software está particionado en tareas independientes
    - major → elementos de arquitectura
    - minor → tareas adicionales incorporadas localmente

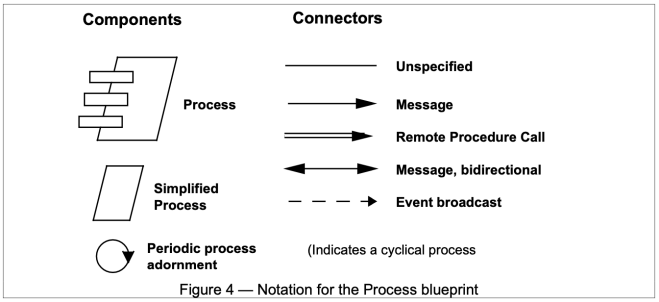


Figure 4 — Notation for the Process blueprint

Vista Desarrollo

Descomposición del subsistema:

- organización del software
- empaqueta porciones de software que puede ser desarrollado por un grupo de desarrolladores
- subsistemas desarrollados en una jerarquía de capas donde cada una provee de una interfaz bien definida a las capas de encima
- establece las bases para lanzar un producto

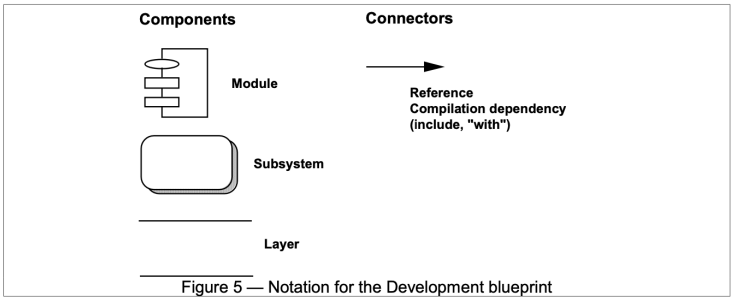


Figure 5 — Notation for the Development blueprint

Vista Física

Mapeo del software al hardware:

- tiene en cuenta requerimientos no funcionales (disponibilidad, confiabilidad, performance, escalabilidad)
- el software se ejecuta en una red de computadoras o nodos de procesamiento
- los elementos identificados deben ser mapeados en nodos

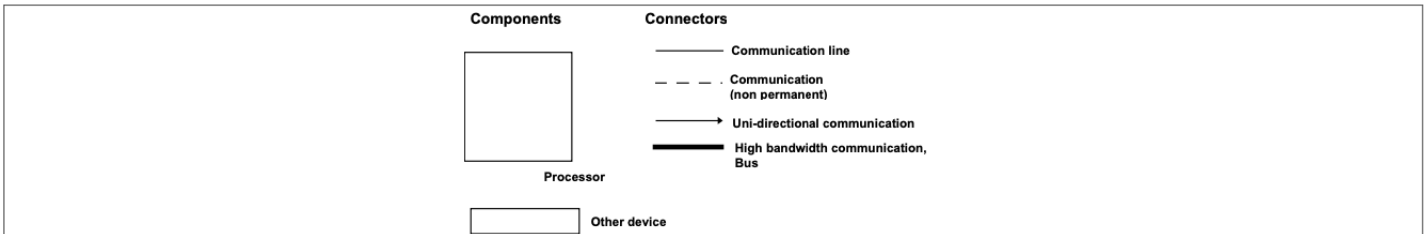


Figure 7 — Notation for the Physical blueprint

Escenarios

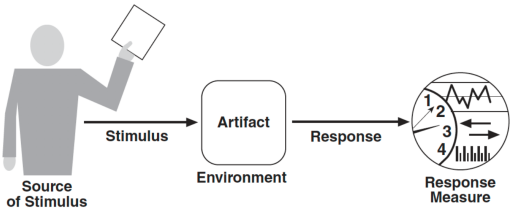
Junta todas las vistas y muestra una abstracción de los requerimientos más importantes.

Correspondencia entre Vistas

Vista lógica motiva el diseño de decisiones en vistas de: desarrollo & procesamiento.
Vistas de desarrollo & procesamiento afecta la vista física por las decisiones de cómo un proceso se comunica y la implementación de requerimientos.

Atributos de Calidad

Propiedad **medible o testeable** de un sistema que es utilizada para indicar qué tan bien un sistema satisface las necesidades de los posibles usuarios. No indica si el sistema



responde los requerimientos pero mide qué tan bien lo hace.

- Fuente de estímulo → entidad que interactúa con el sistema que está siendo medido
- Estímulo → acción que produce una respuesta apropiada del sistema
- Entorno → estado del sistema en el momento del estímulo
- Artefacto → el sistema (o una parte de él)
- Respuesta → actividad medible triggereada por el estímulo
- Medida → medida tomada por la respuesta, varía según el atributo de calidad que está siendo testeado.

Se deben definir qué atributos de calidad se deben priorizar. Cumplir con un atributo de calidad tiene un costo tanto de tiempo de desarrollo (ex: Performance) como económico (ex: Availability). Además, ciertos atributos de calidad son inherentemente opuestos (ex: Security vs Usability)

Disponibilidad

Cuánto tiempo el sistema está levantado (corriendo y andando bien).
Tener una buena disponibilidad requiere de redundancia en la infraestructura.

Confiabilidad

Habilidad para desarrollar sus funcionalidades sin fallas, errores, disrupciones y entregar resultados correctos y consistentes bajo condiciones esperables y no esperables.
Un sistema confiable está generalmente disponible.

Interoperabilidad

Qué tan fácil es para un sistema interactuar con otros.

Modificabilidad

Qué tan fácil es cambiar un sistema dado un cambio en los requerimientos.

Performance

Habilidad de alcanzar los tiempos de respuesta esperados.

Usabilidad

Qué tan fácil es para un usuario usar el software o la aplicación

Seguridad

Facultad de proteger la información contra procesos no autorizados y garantizar acceso a quienes lo tienen efectivamente.

GIT

Sistema de control de versiones centralizado y distribuido.

Servicios de Hosting

GitHub, GitLab, BitBucket, AWS CodeCommit, Azure DevOps, SourceForge.
Agregan features secundarios propios (issue trackers, pull requests, CI/CD, Docker Regisry, métricas).

Visualización del Repositorio

- ❖ git log --all --graph --oneline
- ❖ gitg
- ❖ gitk
- ❖ magit
- ❖ GitKraken
- ❖ SourceTree
- ❖ Tortoise Git
- ❖ Hosting UI (GitHub, GitLab, BitBucket, etc)
- ❖ IDEs

Commits

Algún estado del repositorio. Contiene ancestros, autor/fecha autoría, mensaje, archivos.

Atómicos

- \+ aplica un cambio completo → una razón para cambiar
- \+ no depende de cambios posteriores
- \+ facilita revertir, reordenar, buscar en la historia

Comandos Útiles

para inspeccionar el repositorio

- ➔ git log <archivo>
  - ◆ Historial de cambios de un archivo o carpeta
- ➔ git blame
  - ◆ ¿Cuándo fue la última modificación de cada línea?
- ➔ git bisect

- ◆ Búsqueda binaria en la historia

para manejar interrupciones

- ➔ git stash push -m <comment> / git stash pop
  - ◆ Guardar estado de archivos temporalmente
- ➔ git add -p
  - ◆ Commits de partes de archivos
- ➔ git reset
  - ◆ Deshacer commits

para agregar commits

- ➔ git cherry-pick <commit>
  - ◆ Copia un commit al branch actual
- ➔ git revert <commit>
  - ◆ Crea un commit que hace lo opuesto a <commit>

para ejecutar tests/linters/auto-formatters/etc

git hooks → antes o después de ciertos comandos se ejecutan programas dentro de .git/hooks (si existen):

- ➔ pre-commit
- ➔ pre-push

para manipular la historia local

- ➔ git commit --amend
  - ◆ Agregar cambios al commit actual
- ➔ git rebase --onto <new_root> <old_root>
  - ◆ Trasplanta una serie de commits de un lugar a otro
- ➔ git rebase -i <root>
  - ◆ Reordenar y combinar una serie de commits

Organización

Trunk Based

Todo cambio se agrega a un mismo branch:

- \+ simple
- \+ todo el código está en el último commit
- \- todo el código está en el último commit
- \- cuesta separar de líneas de trabajo

El control de versiones es naturalmente distribuido y concurrente.

Feature Branches

Una rama por funcionalidad de código independiente:

- \+ solo un feature incompleto por branch

+ menos gente trabajando en cada branch
+ el resto del equipo ve todos los cambios juntos
- administración de branches
- features necesitan ser independientes entre sí

Conflictos

Al hacer merge, hay conflicto cuando los cambios de cada rama interactúan de forma negativa.

Textual

Mismas líneas → git puede detectarlo.

Semántico

Misma lógica → requiere analizar la interacción.

Pull Requests

Antes de hacer merge, se verifica:

- calidad del código
- calidad de la aplicación
- cumplimiento de requerimientos
- si el cambio conviene

Debe:

❖ ser atómico
❖ explicar razones que no resultan obvias

Code Review

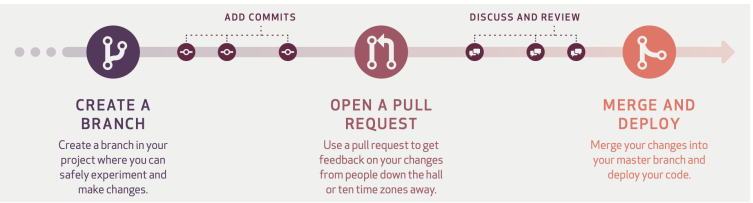
+ cooperativo
+ sin ego
+ prioridad
+ repartido

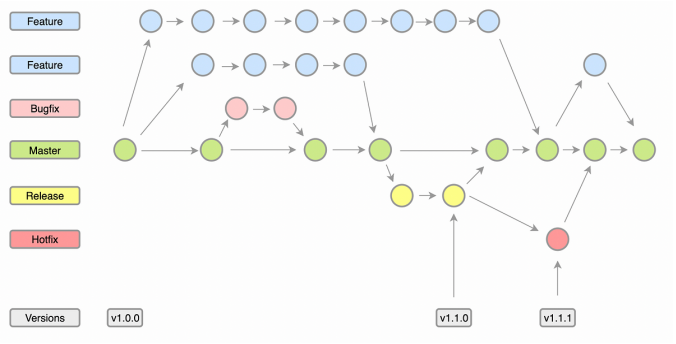
⇒ mayor calidad de código y difusión del conocimiento técnico y del proyecto

Branches a Largo Plazo

Conviene hacer merge intermedios.

Github Flow vs. Git Flow





Monorepo vs. Polirepo

- ➢ MONO
  - ○ un repositorio para todo el proyecto
  - \+ simple
  - \+ cambios coordinados a N componentes
  - \+ fácil filtrar para crear polirepo con historia (git filter-repo)
  - \- control de acceso más difícil de manejar

- ➢ POLI
  - ○ un repositorio por componente del proyecto
  - \+ control de acceso limitando autorización a ciertos repositorios
  - \+ modularizar componentes
  - \+ no hay forma fácil de unificar la historia para crear un monorepo equivalente

Conway's Law: Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Integración Continua

- ★ Continuous Integration
  - ○ integración frecuente del trabajo
  - ○ rápida detección de errores
  - ○ automático
- ★ Continuous Deployment
  - ○ producción de un entregable de forma rápida y certera
  - ○ desplegado a entornos de prueba
  - ○ automático

Docker

Contenedores: Utilizar 'linux namespaces' para crear un aislamiento de un ambiente del sistema operativo origen. Docker es el más famoso, pero existen otros; es un estándar.

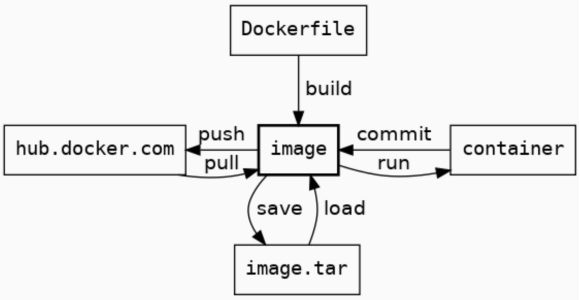
Herramienta que automatiza el despliegue de aplicaciones aisladas dentro de contenedores:

- ➢ container → grupo aislado de procesos
  - liviano
    - único propósito (1 comando sin dependencias)
    - mantiene el estado mínimo necesario
    - puede encender y apagarse rápidamente (útil para escalar horizontalmente)
- ➢ image → template para la creación de contenedores
- ➢ dockerfile → archivo con instrucciones para construir una imagen
- ➢ mounts → acceso a directorio(s) del host.
- ➢ volumes → Acceso a una sección del filesystem del host OS manejada por Docker, con el fin de persistir datos.
- ➢ networks → interfaces de red internas
- ➢ ports → forwardear puertos del hosts al contenedor

Brinda

- + **consistencia** (cualquier imagen se obtiene, inicia y configura de la misma manera)
- + **replicabilidad** (los contenedores basados en una misma imagen son inicialmente iguales)
- + **aislamiento** (se puede controlar la interacción entre la aplicación en un contenedor y el mundo exterior)

Sharing



Se comparten:

- imágenes en repositorios (*hub.dcker.com*)
- imágenes comprimidas (*image.tar*)
- el *dockerfile* que generó la imagen (no garantiza que sea una imagen idéntica)

Comandos

- docker logs
- Listados de objetos
  - docker image ls
  - docker container ls
  - docker volume ls

- Eliminar objetos
    - docker image rm
    - docker container rm
    - docker volume rm
- Control de contenedores
    - docker run <image>
    - docker stop <container>
    - docker start <container>
    - docker kill <container>

Dockerfile

Se almacenan como una referencia a la imagen anterior y una lista de diferencias, llamada layer:

➔ Cada layer es inmutable

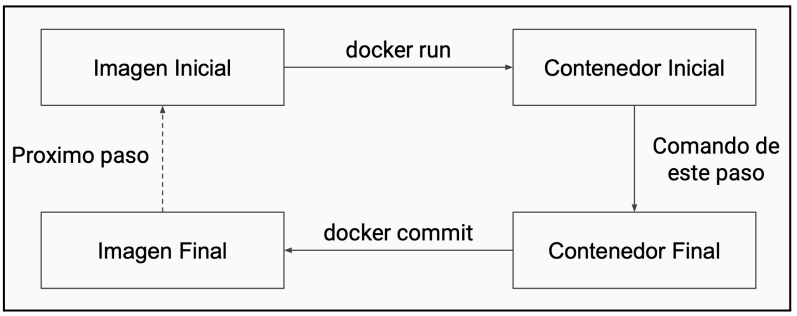
➔ Cada layer referencia a su predecesor:

- ◆ Dos builds comparten layers hasta la primera diferencia
- ◆ Layers ya construidas sirven como caché

Cada layer depende de:

- Imagen anterior
- Comando ejecutado
- Archivos usados del directorio del Dockerfile

Se trata de usar Layers cacheadas lo más posible: aquellos pasos que tengan menos cambios en el tiempo deben ser los primeros que se deben construir (por ejemplo, instalar dependencias). En caso de necesitar instalar paquetes OS, hacerlo de manera tal que se realice pocas veces.



docker history <imagen> → Muestra cada paso que produjo una imagen.

Multi-etapas

Separa la construcción de una imagen donde cada etapa:

+ parte de una nueva imagen base
+ se puede construir en paralelo con otras etapas
+ permite copiar archivos a otra etapa

→ Se usa para no incluir compiladores/linters/etc en runtime

```
# Dockerfile
# Stage 0: Create js bundle

FROM node:12-slim AS build
WORKDIR /app

COPY ./package.json ./package-lock.json ./
RUN npm ci

COPY . .
RUN npm run build



# Stage 1: Web server

FROM nginx:1.23 AS deploy
COPY --from=build /app/build/
                  /usr/share/nginx/html/
```

Instrucciones

FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]

Inicializa una etapa de buildeo y setea la imagen base para las instrucciones subsiguientes. Pueden usarse múltiples imágenes o una etapa de buildeo como dependencia de otra. El alias sirve para referenciar la imagen en la etapa de copia.

ARG <name>[=<default value>]

Variable que se pasan los usuarios durante el buildeo

RUN <command> ó RUN ["executable", "param1", "param2"]

Ejecuta el comando en una nueva capa sobre la imagen. La salida se utiliza en la siguiente capa.

CMD ["executable","param1","param2"] ó CMD ["param1","param2"] ó CMD command param1 param2

Sólo puede haber una de estas instrucciones, y si hay más sólo se ejecutará la última. Provee las bases para un container en ejecución. Setea el comando que va a ser ejecutado en la imagen.

LABEL <key>=<value> <key>=<value> <key>=<value>

Agrega metadata a una imagen.

EXPOSE <port> [<port>/<protocol>...]

Informa a docker que el container en ejecución escucha del puerto de red especificado. El protocolo por defecto es TCP.

ENV <key>=<value> …

Setea el valor de una variable de entorno que va a vivir para todas las instrucciones siguientes.

ADD [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]

Copia nuevos archivos/directorios y los agrega al filesystem de la imagen.

COPY [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]

Copia archivos/directorios y los agrega al filesystem de la imagen.

VOLUME ["/data"]

Crea un punto de montaje externo accesible por el host u otros containers.

WORKDIR /path/to/workdir

Setea el directorio de trabajo para las instrucciones RUN, CMD, ENTRYPOINT, COPY, ADD.

Infraestructura como código

Trata la infraestructura de un servicio como código → infraestructura repetible (reemplazable). Como es código, cualquier cambio queda guardado en un sistema de control de versiones.

Seguridad

- el control de acceso de archivos es por userId
- el filesystem sólo almacena userId's
- un contenedor puede ejecutar con cualquier userId
- el único control de acceso a un volumen es que sólo sea para lectura
- un usuario con permiso de crear contenedores puede escalar a ser root
    - docker run -it -v /:/host ubuntu

Orquestradores

Coordina el uso de múltiples contenedores (DB, load balancer, etc) para implementar una única aplicación, con todas las dependencias y configuraciones necesarias entre ellos. Algunos: Docker-Compose, Kubernetes.

Generalmente permiten separar infraestructura de variables de configuración en diferentes archivos (.ENV)

Compose

Anteriormente se usaba un comando, ahora se guarda la configuración en un yml:

→ before:
```
$ docker run -v "./nginx_html:/usr/share/nginx/html:ro" -p 12345:80 nginx:1.23
```

→ now:
```
version: "3"
services:
    webserver:
        image: nginx:1.23
        volumes:
            - "./nginx_html:/usr/share/nginx/html:ro"
        ports:
            - "12345:80"
```

Para una DB:
- ➢ indicar a docker-compose que gestione un volumen "db_persist"
- ➢ la imagen mysql recibe dos variables de entorno (usr & pass), que provienen de docker-compose
- ➢ docker-compose completa las variables con valores que recibe (variables de entorno propias o en un .ENV)

.ENV

Separa configuración:

- del contexto donde se despliega
  - puertos, URLs, API keys
  - variables en .env
- interna al sistema
  - objetos configurables para los contenedores que lo componen, pero que no son visibles desde fuera del sistema
  - puertos/hosts para comunicación interna

CI/CD

Formas de determinar el entorno de ejecución de un build:

- Build crea un imagen docker pusheada a un registro
- Artefactos creados dentro de una imagen

De esta manera se evita dependencias del runner de la pipeline.

Una vez buildeado, se puede subir la imagen a un registro privado, que permita usar las imágenes en diferentes entornos, según archivos de environment.

Paradigmas de Programación

Computabilidad

Cualquier formalismo que pueda simular una máquina de Turing es **Turing-Completo**:

- una computadora es una máquina de Turing con memoria limitada
- un lenguaje de programación es la implementación de un formalismo

Conceptos

➢ Concepto: idea que se puede expresar o prohibir en un lenguaje

➢ Modelo de Programación: conjunto de técnicas de programación y principios de diseño aplicados a un lenguaje

→ Dado un problema, queremos un lenguaje con conceptos y modelo de programación que faciliten resolverlo.

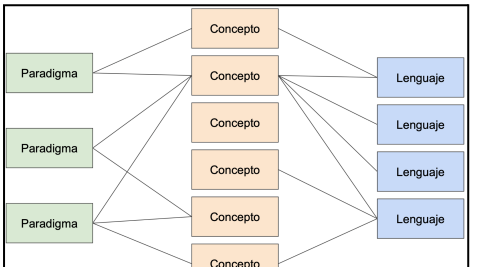
Paradigma

Teoría o conjunto de teorías cuyo núcleo central se acepta sin cuestionar y que suministra la base y modelo para resolver problemas y avanzar en el conocimiento. El paradigma newtoniano.

de Programación

➔ conceptos centrales

➔ mecanismos de



        - ◆ razonamiento (conceptos favorecidos / prohibidos)
        - ◆ comunicación (vocabulario / notación)

Combinaciones

- \+ múltiples puntos de vista para hallar soluciones
- \+ para componentizar sistemas, cada uno se expresa en su paradigma natural
- \- si mezclamos sin criterio, los conceptos podrían no interactuar bien

Imperativos vs. Declarativo

- Imperativo
    - describe el cómo, paso a paso, de lo que se desea obtener
    - el estado final debería ser la solución al problema
- Declarativo
    - describe el problema a solucionar, el sistema se encarga de hallar cómo hacerlo

Semántica

Significado de un problema:

- ❖ Operativa → estado & transformaciones de un problema
    - ➢ programador como intérprete humano
- ❖ Denotacional → linkeo con objetos matemáticos
    - ➢ problema matemático complejo
- ❖ Axiomática → pre & post condiciones

Programación Estructurada

Todo control de flujo expresado como: secuencia, alternativa o repetición.

Programación Modular

Conjunto de código con una interfaz definida que cumple un propósito discreto. Permite limitar acceso a estados/operaciones.

Programación Orientada a Objetos

Objeto

- ➢ estado y comportamiento agrupado en una entidad
    - prototipos
        - ■ un objeto es replicado para crear otro
        - ■ los objetos se modifican para describir el estado/comportamiento deseado
    - clases
        - ■ molde que se usa para crear objetos nuevos que pueden no ser modificables
- ➢ encapsula su estado, ocultándolo a otros

- ➢ aísla comportamientos/estados y analiza sin considerar otros
- ➢ comunicación
  - mensajes
    - asincrónicamente
    - típico en sistemas concurrentes/distribuidos
  - métodos
    - sincrónicamente
    - típico dentro de un thread

Programación Funcional

Basado en funciones matemáticas; dada una misma entrada, la salida siempre será la misma. Esto implica que se puede sustituir una llamada a la función por su resultado (transparencia referencial).

Una función no tiene efectos secundarios (no modifica estado del programa).

Estructuras de datos persistentes: Al modificar un ADT se crea una copia de él.

Generalmente, las funciones que tienen efectos secundarios (IO por ejemplo) están marcadas y se conoce lo que van a hacer.

Lenguajes multiparadigma dicen soportar programación funcional cuando:

+ soportan funciones como componente de primera clase → existan funciones de orden superior (funciones que operan sobre otras funciones como argumento i.e.: map, filter)
+ no requieren funciones matemáticas

Utiliza evaluación perezosa, solo se computan valores cuando es necesario

Modelo de ejecución (probablemente salteable):

- Orden aplicativo: se evalúan los parámetros antes de llamar a la función hasta llegar a una primitiva.
  - Aplicación predecible de efectos colaterales
- Orden Normal: Se evalúan los parámetros dentro del cuerpo de la función, hasta llegar a una primitiva.
  - Control sobre cuántas veces se evalúa un argumento.
  - Puede evaluar funciones que son ciclos infinitos en orden aplicativo

Programación Lógica

Basado en predicados, un programa es axiomas/base de datos (X siempre ocurre), reglas de inferencia (Si X se cumple, entonces A se cumple), consulta.

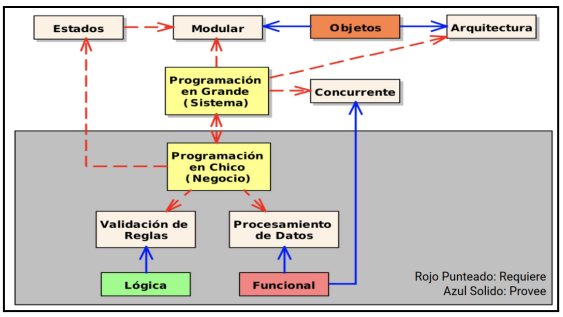
Modelo Relacional

Esquema de bases de datos, conceptualmente similar a programación lógica.

	Base de Datos	Reglas	Turing-Completo
Lógico	Pequeña	Muchas	Intencional
Relacional	Grande	Pocas	Workarounds

Combinaciones de Paradigmas

Los diferentes paradigmas son combinables en un sistema mayor, con diferentes resultados entre ellos, algunos malos, otros buenos.



Objetos - Funcional → Functional Core - Imperative Shell

El núcleo de la aplicación es funcional, sin acceso al mundo exterior, donde solo se aplica la lógica de dominio.
Existe una capa imperativa que se encarga de hacer de puente con el mundo exterior y el núcleo funcional.
Típico para sistemas distribuidos donde cada nodo debe conocer el estado de una simulación.

Lógico - Funcional

Se buscan soluciones dentro de reglas lógicas y se analiza/manipula el flujo de datos mediante programación funcional.
Puede ser búsqueda eficiente dentro de estructuras simbólicas complejas.

Funcional - Objetos

Se prohíbe la comunicación por estado global y se utiliza un grafo de datos en los llamados a funciones. Se encapsula el objeto y el contrato se cumple sin importar cómo sucede por dentro.

Objetos - Relacional

Un grafo de entidades interactuando de a pares con un conjunto de hechos interactuando en conjuntos.
Pueden ocurrir conflictos de representación: Object–relational impedance mismatch.

Lenguajes específicos de dominio

Lenguajes especializados para un dominio, que sacrifican generalidad por expresividad/usabilidad.

- **Embebidos** → lenguaje de propósito general (frameworks)
- **Generados** → código de un lenguaje anfitrión (transpilar)
- **Interpretados** → intérprete generalmente construido en lenguaje anfitrión, se implementan etapas de parseo y ejecución

REST

Representational State Transfer: estilo de arquitectura para proveer estándar entre sistemas informáticos, haciendo que la comunicación entre ellos sea fácil. Utilizada con el fin de desacoplar el comportamiento entre cliente y servidor. Con tan solo el conocimiento de cómo se van a transferir los datos es suficiente.

Resources (URIs)

Uniform Resource Identifier → identifica unívocamente un recurso mediante sustantivos (en plural) y por ser recursos principales y subordinados (se manejan cosas).

Los mensajes intercambiados entre cliente y servidor son sobre recursos que el servidor controla y que el cliente quiere afectar. No existen comandos que dependan de otros anteriores.

Stateless

Cualquier parte del sistema, cliente o servidor, no necesita saber el estado actual de su contraparte para funcionar. Es decir, cualquier mensaje es interpretable sin necesidad de conocer la historia de los mensajes.

Cacheable

Utilizar los mecanismos de HTTP para reducir el ancho de banda usado, la latencia y la carga en los servidores.

HTTP

Utilizar las bondades de HTTP, como el buen uso de los verbos ‘GET’, ‘POST’, ‘DELETE’, ‘PUT’, etc. o el uso de los códigos de retorno, a fin de proporcionar una interfaz uniforme.

Responses

- ➢ 1xx: Informational → *hold on*
- ➢ 2xx: Success → *here you go*

- ➢ 3xx: Redirection → *go away*
- ➢ 4xx: Client Error → *you f* up*
- ➢ 5xx: Server Error → *i f* up*

Principios de diseño de seguridad

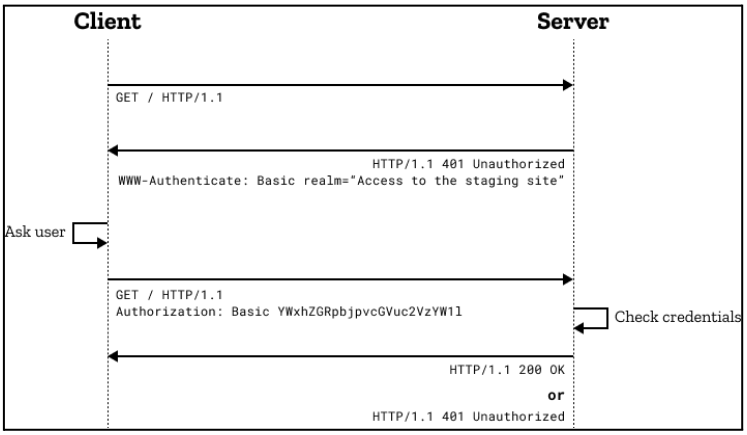
- **Least Privilege** → menor requerido
- **Fail-Safe Defaults** → no tener acceso a recursos por defecto
- **Complete Mediation** → validar permisos de acceso
- **Keep** (the interface) **simple**
- **HTTPS**
- **Password Hashes**
- **Never expose information on URLs** → para evitar ser logueadas
- **Requests w Timestamps** → firmar las peticiones
- **Input authentication**

Autenticación y Autorización

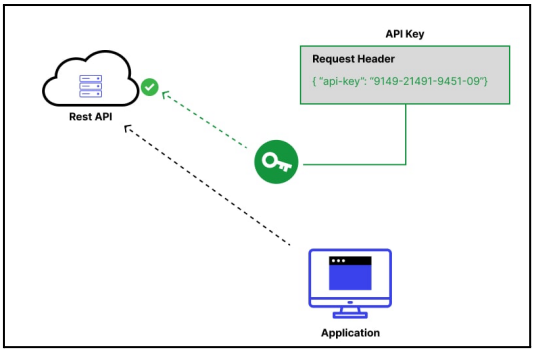
- Autenticar: quién sos
- Autorizar: qué podés hacer

Métodos

- **Basic Auth**: credenciales en texto plano

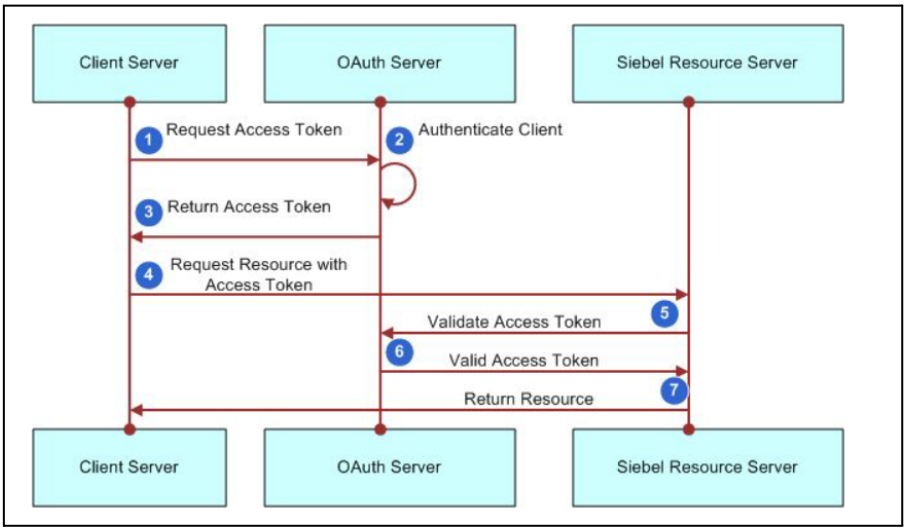


- **Api Keys**: keys secretas para comunicación entre sistemas

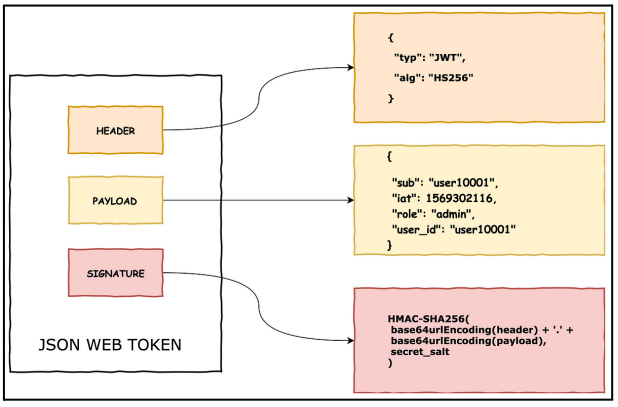


- **Bearer Authentication**: token de seguridad que habilita a hacer algo/acceder a algo
- **OAuth**: delegar autenticación a un tercero que gestione cuentas; diferentes flujos especificados según lo que se necesite (implicit, explicit, etc.)

- Access Tokens → firman y verifican a qué recursos un usuario puede acceder
- ID Token (OpenID) → firman y verifican que un usuario es quién dice ser
- Refresh Token → permite renovar el access token (e ID Token), evitando re-utilizar las credenciales del usuario.



- **JWT**: forma de firmar un token, que puede o no contener información, a fin de evitar enviar las credenciales de forma repetida en cada petición; se puede verificar que la firma es válida evitando el acceso repetitivo a una base de datos (firmado, encriptado, o ambos)



Refresh

Los tokens de acceso deberían tener un tiempo limitado de vida → el refresh token se utiliza una vez para obtener un nuevo token de acceso.
Credencial que permite obtener nuevos tokens sin utilizar credenciales.

Versionado

No se dicta una manera standard, puede ser mediante la URI o un Header HTTP.

HATEOAS

Una aplicación REST debe permitir que un cliente interactúe con la aplicación sin conocimiento previo de

```
GET /accounts/12345 HTTP/1.1
Host: bank.example.com
```

The response is:

```
HTTP/1.1 200 OK

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
```

la misma, mediante ***hypermedia*** (links a qué se puede hacer con un recurso).

Paginados,filtros y ordenamientos

Se debe implementar de manera que si es un contenido cacheable, tiene que tener la posibilidad de serlo. Por lo tanto, la mejor manera de realizarlos es mediante query strings. También se puede realizar mediante body, pero es más difícil de cachear.

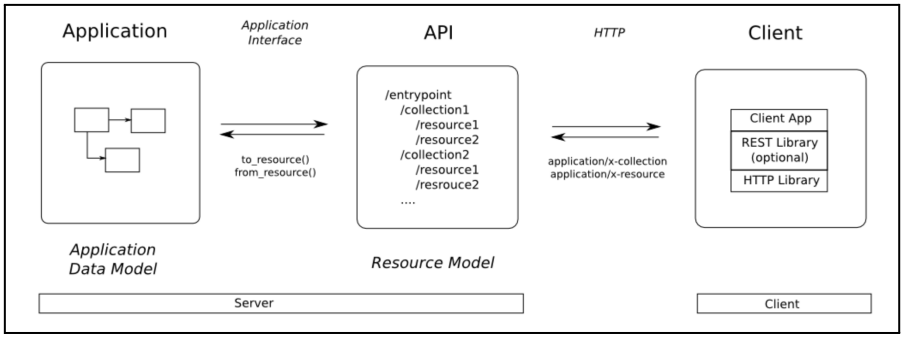
Interfaz uniforme

Mantener una interfaz uniforme a través de todos los recursos, incluso con los mensajes de error.

Diseño

1. entender los detalles importantes para los cuales se crea la API
2. modelar la funcionalidad del programa tal que todos los casos de uso se encuentren plasmados en la API siguiendo los principios REST

Componentes de Diseño → Aplicación - Código - Cliente



Aplicación

"Business Logic"
Existe independientemente de la API REST. Tiene un estado dinámico que varía según las distintas operaciones que se ejecuten. Asumimos que el estado está descrito en un diagrama de entidad-relación.

Código

"Bridge between business logic and REST style"
Accede al estado de la aplicación mediante la interfaz que esta provee y la representa en la API REST. Se adapta el modelo de la aplicación a la arquitectura de la API. Los componentes resultantes son recursos de API REST que van a ser utilizados luego. Las relaciones entre ellos son expresadas mediante hipervínculos.
Se definen 2 funciones de utilidad: *to_resource()*, *from_resource()*.

Cliente

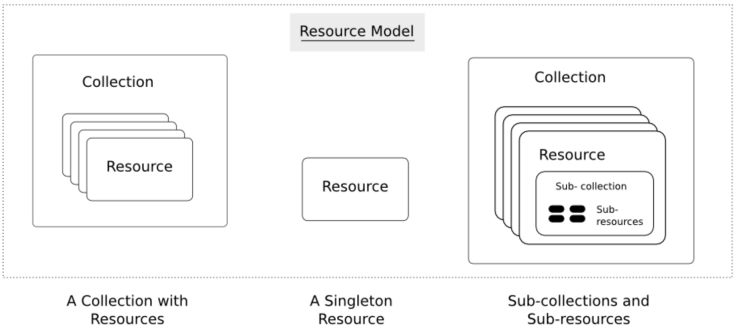
Consume la API REST mediante mensajes de tipo HTTP.

Recursos

Objeto con

- ➢ tipo
  - scalar (number, string, boolean, null)
  - array
  - object
- ➢ información relacionada
- ➢ relaciones con otros recursos
  - collection/{name} → link a la colección relacionada
  - resource/{name} → link al recurso relacionado
  - form/{name} → link al form relacionado
- ➢ set de métodos que operan sobre él
  - GET (collection/resource) → recursos de colección
  - HEAD (collection/resource) → cabecera de recursos de colección
  - POST (collection) → nuevo recurso en colección
  - PUT / PATCH (resource) → actualiza recurso
  - DELETE (resource) → elimina recurso
  - OPTIONS (any) → métodos HTTP disponibles & otras opciones(headers - importante por seguridad)

Agrupados en colecciones homogéneas (que también son recursos).



Informativos → Mapeo de Información

- "*_type*" como sufijo del recurso
- claves no pueden empezar con "_"
- colecciones modeladas como arreglo de objetos

Metadata

- id (string) → ID del recurso
- href (string) → URL del recurso
- link (object) → relación entre un recurso y otro

URLs

Entry Point

La URL del entry point es comunicado a los usuarios para que puedan acceder a la API REST.

Structure

Cada colección/recurso tiene su propio URL. Se recomienda que sea absoluto:

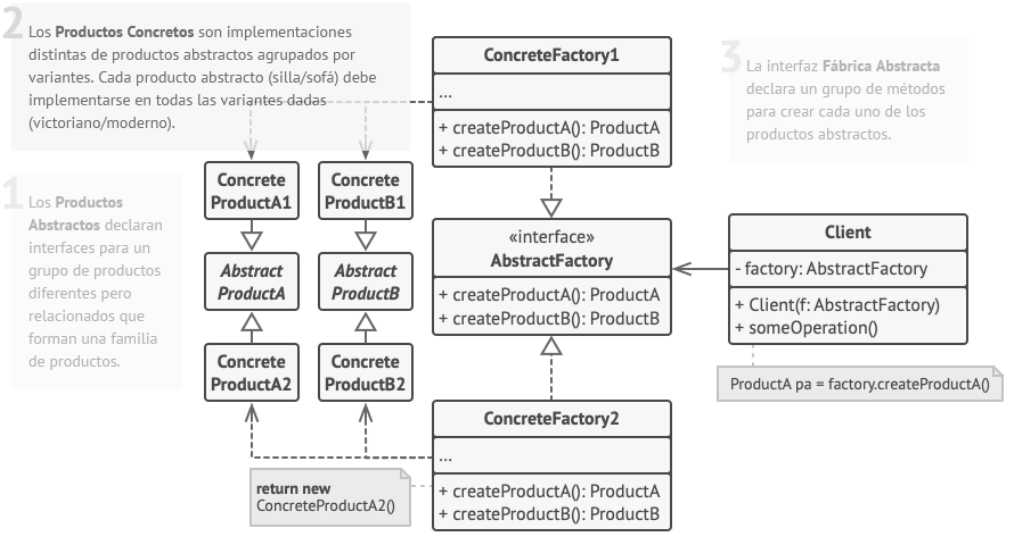
URL	Description
/api	The API entry point
/api/:coll	A top-level collection named “coll”
/api/:coll/:id	The resource “id” inside collection “coll”
/api/:coll/:id/:subcoll	Sub-collection “subcoll” under resource “id”
/api/:coll/:id/:subcoll/:subid	The resource “subid” inside “subcoll”

Patrones de diseño

Creacionales

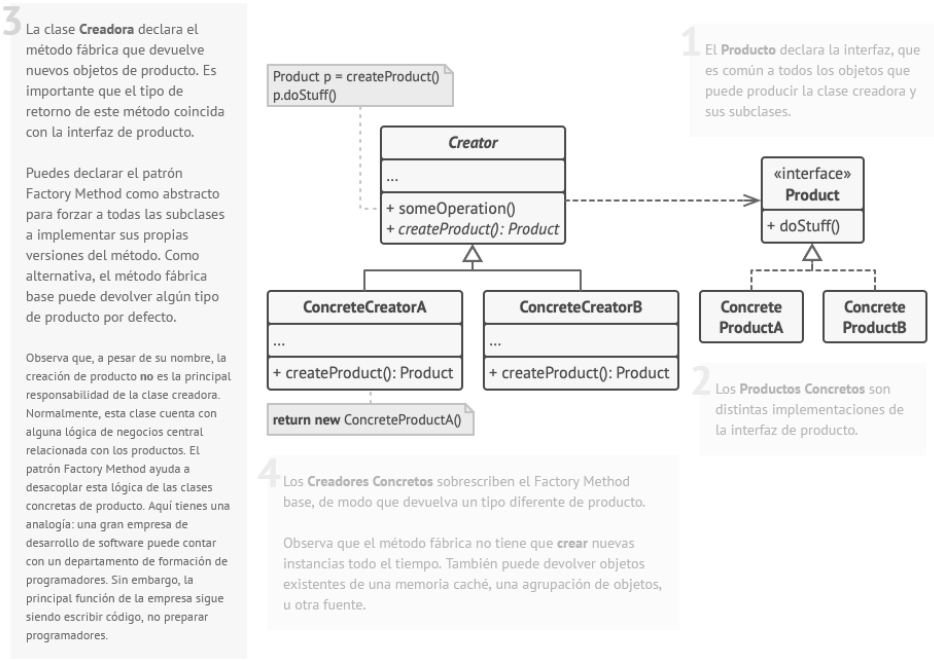
Abstract Factory

Crea una familia de objetos relacionados entre sí, sin necesidad de saber las clases concretas que se van a crear.



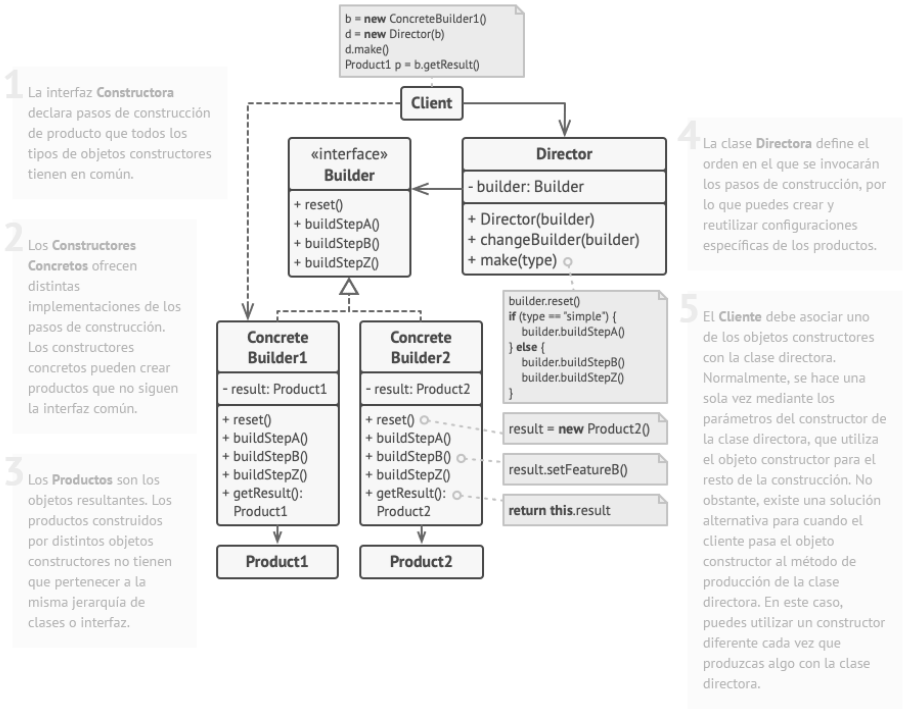
Factory Method

Crea un objeto, en base a alguna configuración, sin la necesidad de saber la clase concreta que se va a crear.



Builder

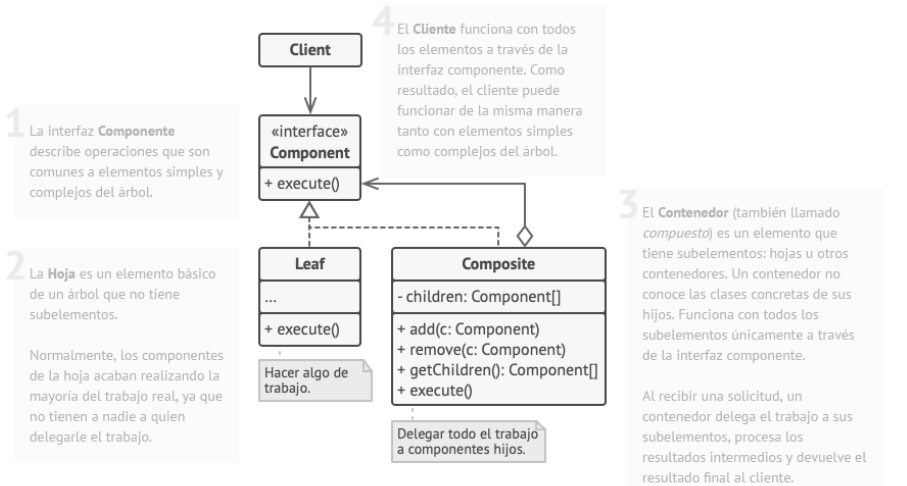
Permite construir un objeto complejo (o cualquier objeto que cumpla esa interfaz), paso a paso, dando la posibilidad de delegar la construcción del mismo a diferentes partes del programa.



Estructurales

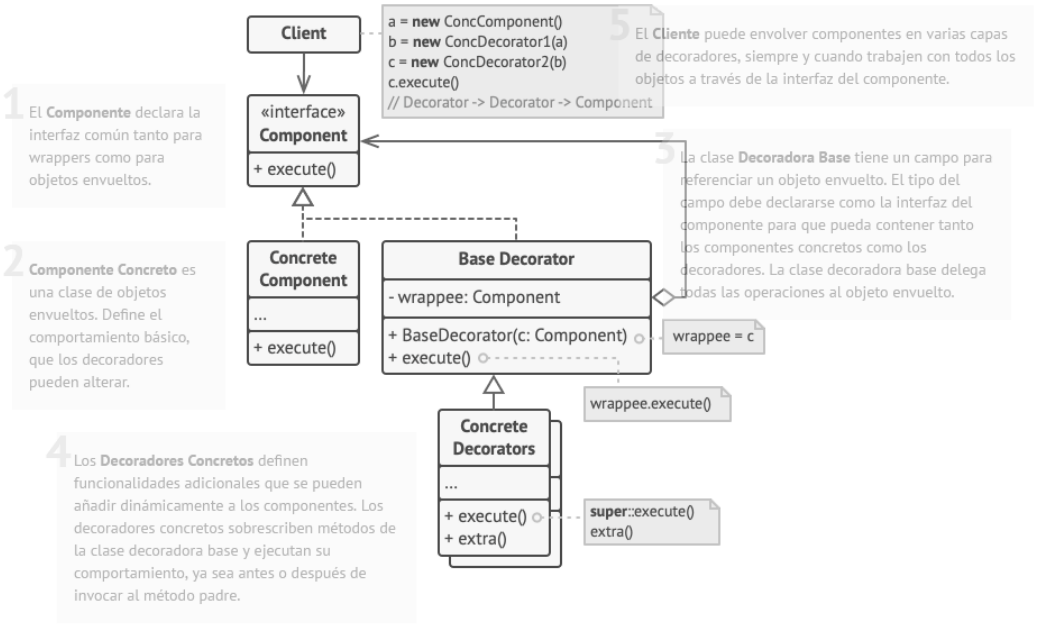
Composite

Permite componer objetos en forma de árbol, y trabajar con esa composición como si fuera un objeto solo. Los nodos intermedios delegan trabajo a sus hijos. Los nodos hojas son los que realizan el trabajo.



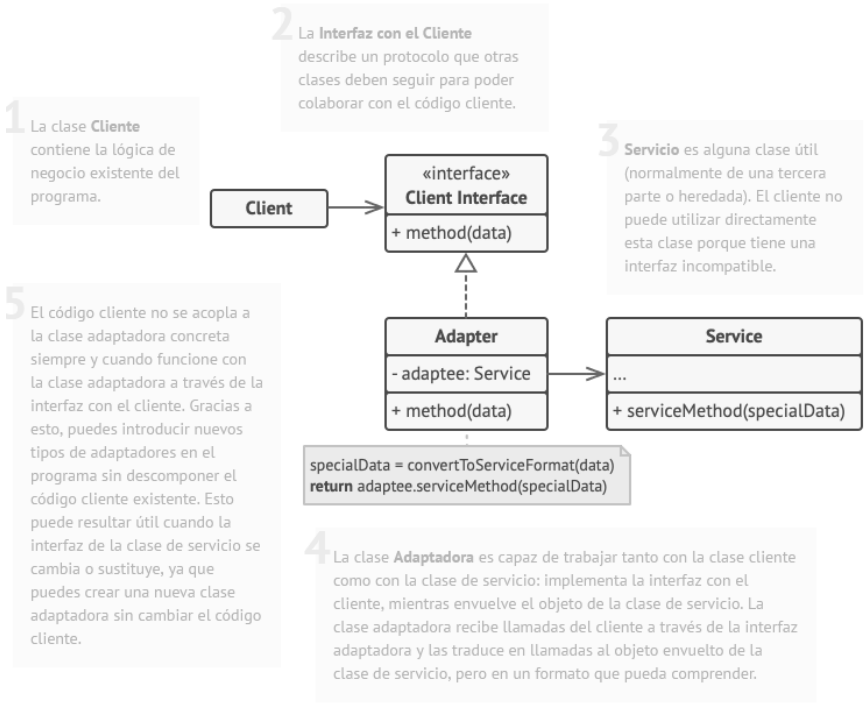
Decorator

Permite añadir funcionalidad a un objeto, conservando la interfaz del objeto base.



Adapter

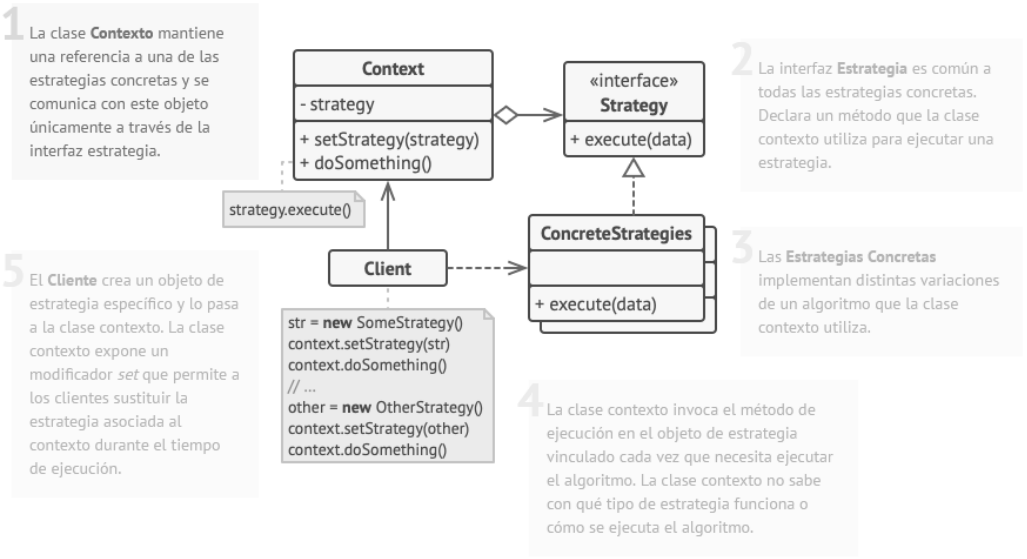
Permite la colaboración entre objetos que normalmente no podrían hacerlo, adaptando la interfaz de uno para que el otro la entienda sin modificar el objeto original.



Comportamiento

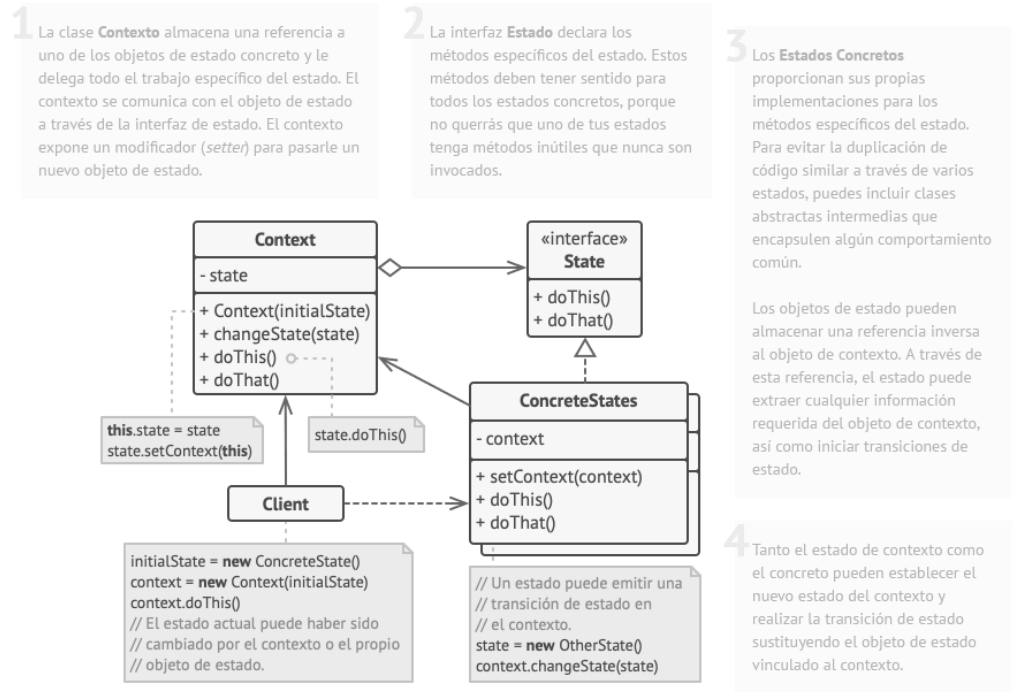
Strategy

Permite modificar el comportamiento de una clase dinámicamente, **desde fuera**. A partir de cierta interfaz, cada estrategia tendrá un comportamiento diferente.



State

Permite modificar el comportamiento de una clase dinámicamente, **desde dentro**, cambiando frente a ciertos estímulos.

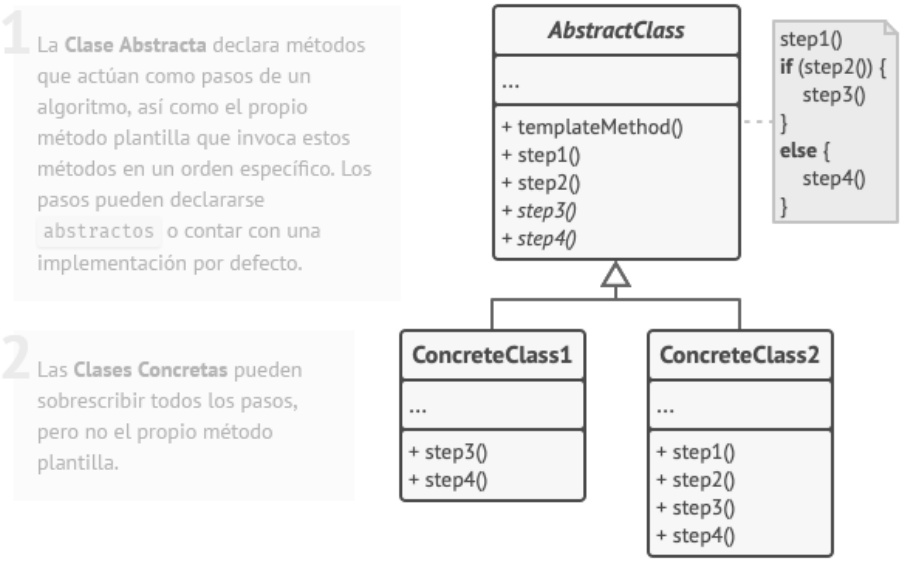


Null Object

Permite modelar el comportamiento de de un estado inexistente/no válido polimórficamente. Generalmente usado con State o Strategy.

Template Method

Permite crear el esqueleto de un algoritmo que utiliza ciertos pasos, que pueden ser modificables, tanto dinámicamente (mediante composición) o estáticamente (mediante herencia).



Command

Permite modelar una petición como un objeto, permitiendo realizar operaciones (parametrizar, retrasar, etc) que de otra forma seria mas dificil.