DISEÑO DE APLICACIONES 2

```
throw new NotImplementedException();
```

REST API

Definición

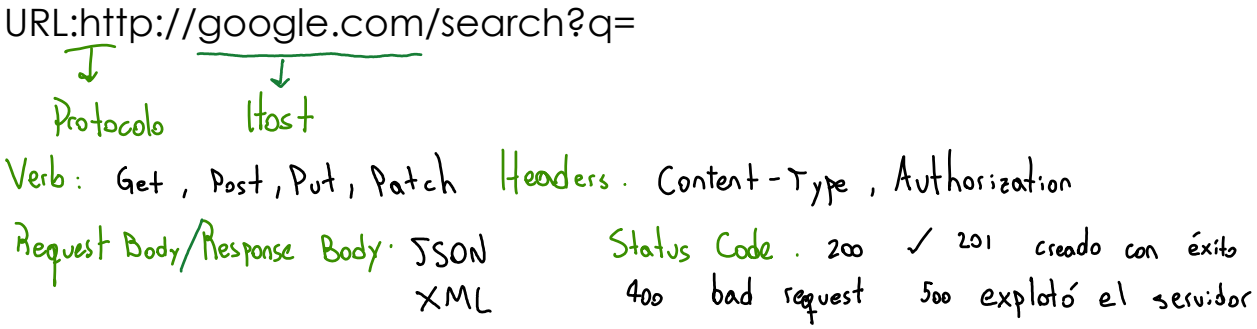
Una REST API (Representational State Transfer) es una forma de definir interfaces entre sistemas que utilizan HTTP para obtener datos o indicar la ejecución de operaciones sobre datos, en cualquier formato (XML, JSON, etc) sin las abstracciones adicionales de los protocoloes basados en patrones de intercambio de mensajes como SOAP.

Características

- Rest permite la escabilidad gracias a estos puntos clave:
  Protocolo cliente/servidor sin estado: Cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.
- Un conjunto de operaciones bien definidas que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones CRUD en bases de datos que se requieren para la persistencia de datos.
- Una sintaxis universal para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.
- El uso de hipermedios, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Recurso

Un concepto importante en REST es la existencia de recursos (elementos de información), que pueden ser accedidos utilizando un identificador global. Para manipular estos recursos, los componentes de la red (clientes y servidores) se comunican a través de una interfaz estándar (HTTP) e intercambian representaciones de estos recursos.



Un recurso puede no ser algo que este en la base. REST define la comunicación a nivel más abstracto, podría pedir un reporte que se genere y no esté persistido.

Diseño de apis

Para esta api es probable que un programador no necesite de documentación para entenderla:

Resource	POST create	GET read	PUT update	DELETE delete
/dogs	Create a new dog	List dogs	Bulk update dogs	Delete all dogs
/dogs/1234	Error	Show Bo	If exists update Bo If not error	Delete Bo

- No usar más de dos niveles para las URLs.
- No usar verbos, usar los de HTTP para operar sobre los recursos.
- Elegir nombres concretos sobre abstractos. Por ejemplo, si tuviera una api que expone blogs, videos, novedades y articulos, y todos estos están implementados por la abstracción items, si expongo items en la api no se entiende que corno es items (depende un poco de cada caso).
- Usar sustantivos en plural para los recursos.
- Usar bien los status codes.
- Hacer versionado de la api.
- Usar "limit" y "offset" para que a los programadores les sea fácil paginar.
- Si se trata de un response que no involucra recursos, usar verbos en lugar de sustantivos.
- Proveer shorcuts en las respuestas de la api.

MÉTRICAS

- **Atributo:** Una propiedad medible, física o abstracta, que comparten todas las entidades de una misma categoría.

- **Medida:** Es una indicación cuantitativa de extensión, cantidad, dimensiones, capacidad de algunos atributos de un proceso o producto.

- **Métrica:** Una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado.

Las métricas sirven para obtener los valores sobres los que se pueden tomar acciones para mejorar el diseño, o estimar futuras fases del desarrollo como asociación con atributos de calidad, refactoring, aplicar patrones de diseño, estimación de esfuerzo de testing.

Se pueden dividir en tres categorías:
✧ Orientadas al paquete (assembly)
✧ Orientadas a la clase
✧ Orientadas a los métodos

La calidad del diseño está muy condicionada por las relaciones existentes entre los paquetes del sistema.

Bajo acoplamiento y alta cohesión entre paquetes son características de calidad deseadas en cualquier aplicación bien diseñada.

Una medición objetiva del grado de reusabilidad y mantenibilidad de un sistema puede ser determinante en la evaluación de la calidad del software.

Las métricas no deben ser el único mecanismo para evaluar la calidad del diseño.

Malas métricas no implican necesariamente un mal diseño y buenas métricas no implican necesariamente un buen diseño. Aunque, es muy difícil encontrar un diseño excelente asociado a métricas características de un mal diseño.

Cohesión relacional

Mide la relación entre clases de un paquete.

★ $H = \frac{R + 1}{N}$

R = Número de relaciones entre clases internas al paquete (relaciones que no conectan con clases fuera del paquete).
N = Número de clases e interfaces dentro del paquete.

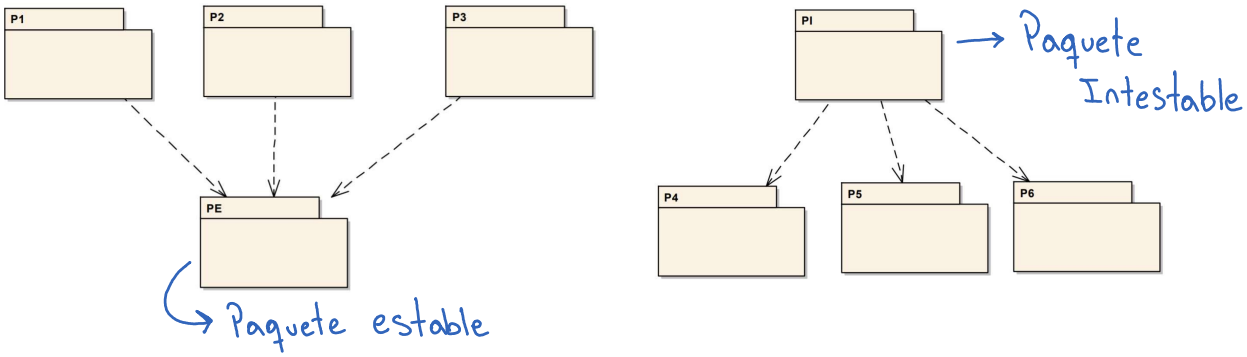
! Es buena si está entre 1.5 y 4.0

Se cuenta como dependencia cuando hay una relación (de las que siguen) entre una clase o interfaz C hacia una clase o interfaz D.

- ✧ Si C tiene un atributo del tipo D.
- ✧ Si C tiene un operación con un parámetro del tipo D.
- ✧ Si C tiene una asociación, agregación, o composición con navegabilidad hacia D.
- ✧ C es hijo de D.
- ✧ C implementa la interfaz D.
- ✧ Las asociaciones bidireccionales se cuentan dos veces.

Estabilidad

La inestabilidad de un paquete impacta en el esfuerzo requerido para realizar un cambio.
Se deben examinar como impactan en la estabilidad las dependencias del paquete.



Principio de Dependencias Estables

- ★ La dirección de las dependencias entre paquetes debe ser acorde con la estabilidad de los mismos.
- ★ Un paquete debe depender solamente de paquetes que son más estables que él.

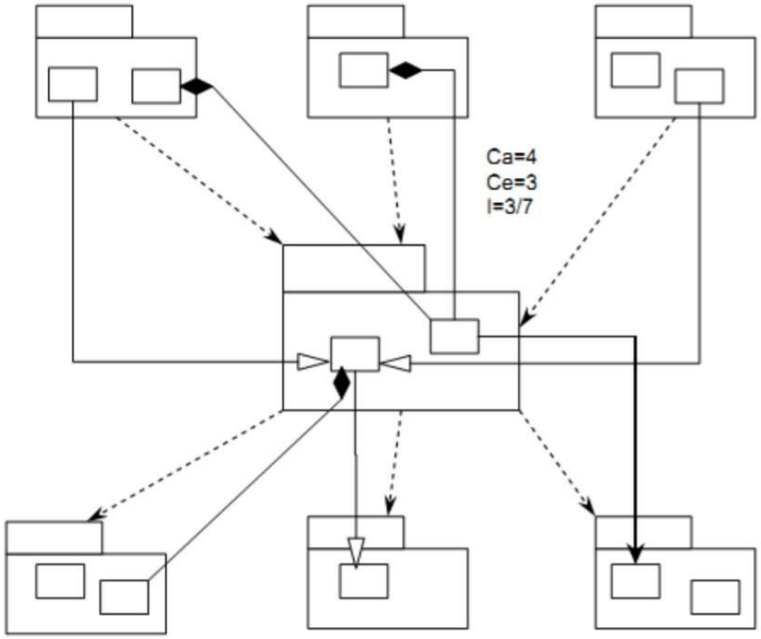
Medición de estabilidad

Los paquetes con más dependencias entrantes deberán exhibir un alto grado de estabilidad.

! Son los más difíciles de cambiar.

★ $$I = \frac{C_e}{C_e + C_a}$$

I = Grado de inestabilidad asociado a un paquete.
Ca = Acoplamiento aferente, o dependencias entrantes. Número de clases fuera del paquete que dependen de clases dentro del paquete.
Ce = Acoplamiento eferente, o dependencias salientes. Número de clases fuera del paquete de las cuales dependen clases dentro del paquete.



Interpretación

★ Valor cercano a 0: Máxima estabilidad porque no se depende de otros paquetes. Se debe intentar extender haciéndolo abstracto (Abierto/Cerrado, DIP y Dependencias Estables).0

★ Valor cercano a 1: Máxima inestabilidad porqjue depende de otros paquetes. Este paquete es fácil de cambiar debido a que impacta en pocos paquetes.

La abstracción puede ser usada para incrementar la estabilidad, separando "lo que se hace" de "cómo se hace".

Los paquetes más estables deberían ser más abstractos, mientras que los más inestables deberían ser paquetes más concretos.

Principio de Abstracciones Estables

- Los paquetes más estables deben tender a ser abstractos.
- Los paquetes inestables deben ser concretos.

Medición de la abstracción

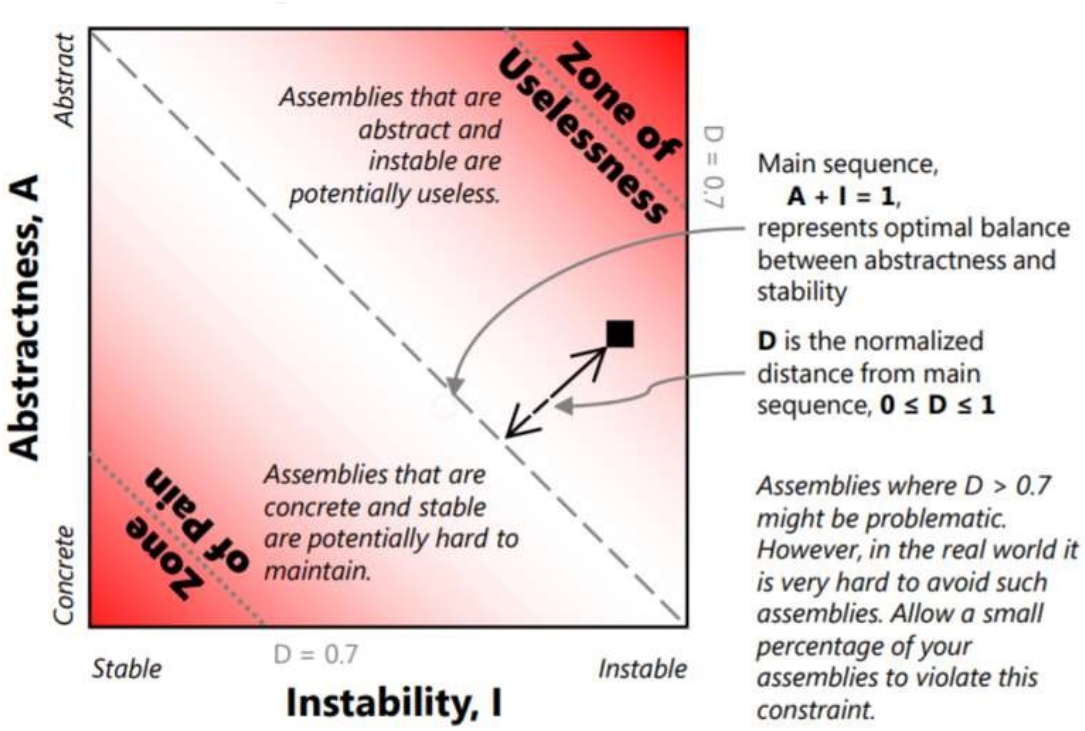
La abstracción de un paquete se mide calculando el ratio entre el número de clases abstractas e interfaces del paquete, y el número total de clases en el paquete.

- $A = \frac{N_a}{N_c}$

A = Abstracción del paquete.
N_a = Cantidad de clases abstractas e interfaces en el paquete.
N_c = Cantidad de clases concretas, abstractas e interfaces del paquete.

A cercano a 0 implica que es un paquete concreto.
A cercano a 1 implica que es un paquete abstracto.

Abstracción VS Estabilidad



PRINCIPIOS DE DISEÑO

The first five principles are principles of *class design*. They are:

SRP	The Single Responsibility Principle	*A class should have one, and only one, reason to change.*
OCP	The Open Closed Principle	*You should be able to extend a classes behavior, without modifying it.*
LSP	The Liskov Substitution Principle	*Derived classes must be substitutable for their base classes.*
ISP	The Interface Segregation Principle	*Make fine grained interfaces that are client specific.*
DIP	The Dependency Inversion Principle	*Depend on abstractions, not on concretions.*

The next six principles are about packages. In this context a package is a binary deliverable like a .jar file, or a dll as opposed to a namespace like a java package or a C++ namespace.

The first three package principles are about package *cohesion*, they tell us what to put inside packages:

REP	The Release Reuse Equivalency Principle	*The granule of reuse is the granule of release.*
CCP	The Common Closure Principle	*Classes that change together are packaged together.*
CRP	The Common Reuse Principle	*Classes that are used together are packaged together.*

The last three principles are about the couplings between packages, and talk about metrics that evaluate the package structure of a system.

ADP	The Acyclic Dependencies Principle	*The dependency graph of packages must have no cycles.*
SDP	The Stable Dependencies Principle	*Depend in the direction of stability.*
SAP	The Stable Abstractions Principle	*Abstractness increases with stability.*

Cohesión

REP - Release Reuse Equivalence Principle

The unit of reuse is the unit of release. Effective reuse requires tracking of releases from a change control system. The package is the effective unit of reuse and release.

El código no tiene que ser reusado copiándolo de una clase a otra ya que se vuelve inmantenible. El código debe ser reusado incluyendo librerías.

CCP - Common Closure Principle

The classes in a package should be closed together against the same kinds of changes. A change that affects a package affectas all the classes in that package.

Más importante que la reusabilidad es la mantenibilidad.
Si el código cambia, es preferible que impacte solo en un paquete a que impacte en todo el código. De esta forma, solo hay que liberar el paquete modificado y no todo el sistema.

CRP - Common Reuse Principle

The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

Este principio nos ayuda a decidir que clases deberían estar en un paquete. Las clases que se reusan juntas, van juntas en un mismo paquete.

Acoplamiento

ADP - Acyclic Dependencies Principle

The dependency structure between packages must be a directed acyclic graph (DAG). That is, there must be no cycles in the dependecy structure.

SDP - Stable Depenedencies Principle

The dependencies between packages in a design should be in the direction of the estability of the packages. A package should only depend upon packages that are more stable that it is.

SAP - Stable Abstraction Principle

Packages that are maximally stable should be maximally abstract. Instable packages should be concrete. The abstracion of a package should be in proportion to its stability.

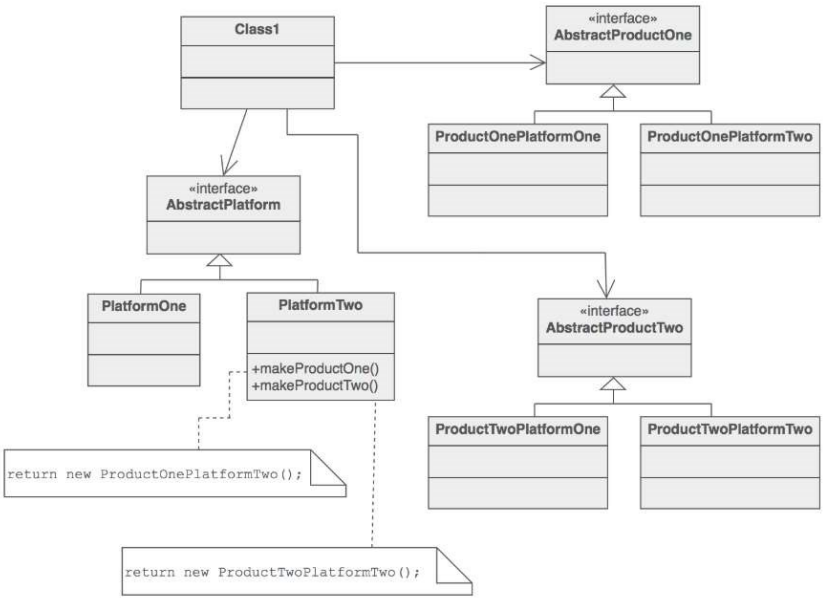
PATRONES DE DISEÑO

Creacionales

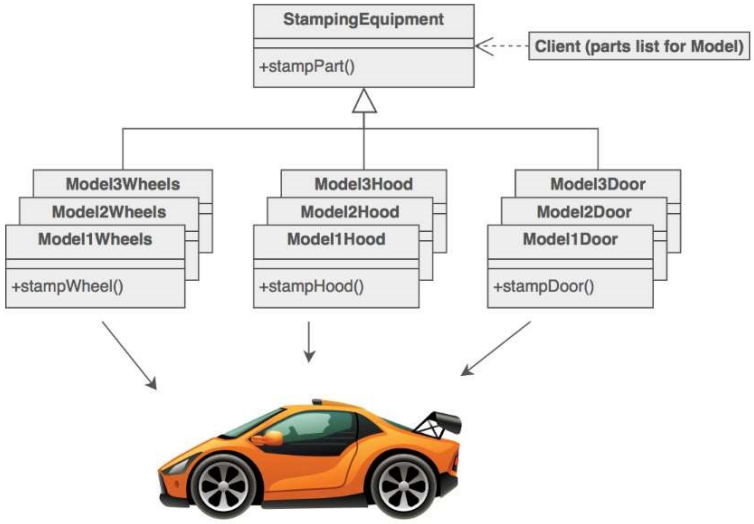
Abstract Factory

Provee una interface para crear familias de objetos relacionados sin especificar sus clases concretas.
Suele implementarse como un singleton.

La Abstract Factory dfine un Factory Method por cada producto. Cada Factory Method encapsula el new y las clases concretas.

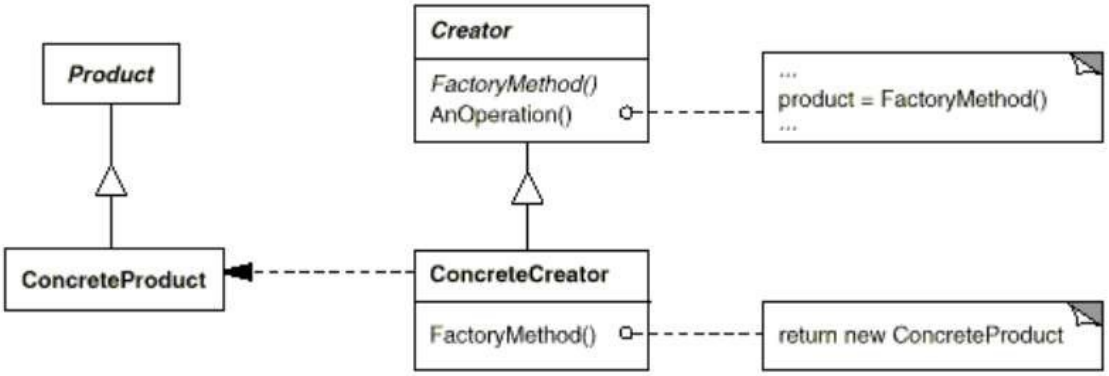


Ejemplo

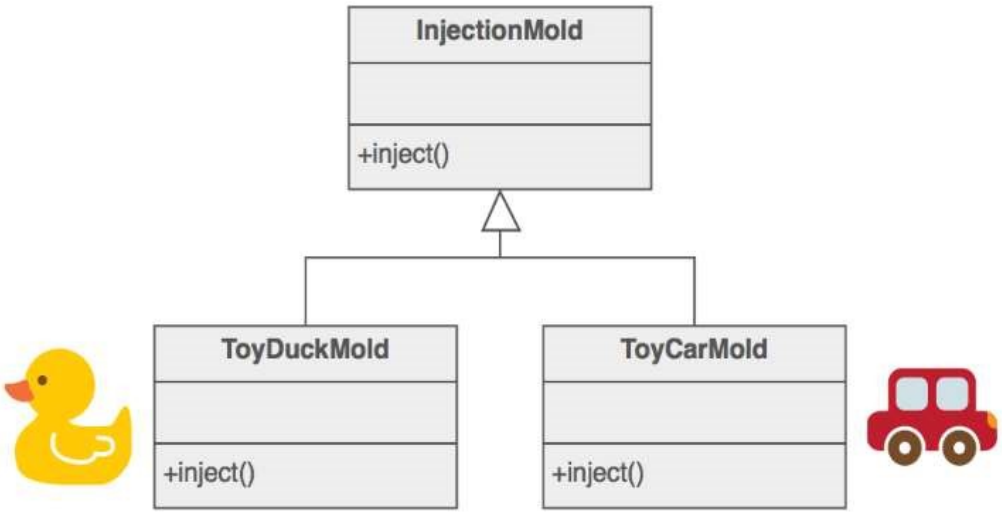


Factory Method

Define una interface para la creación de objetos, pero permite que las subclases decidan que clases instancian.
Factory Method crea objetos de la misma forma que Template Method implementa algoritmos.
Una super clase define todo el comportamiento genérico (usando virutals) y luego delga los detalles de la creación a las subclases que son entregadas al cliente.

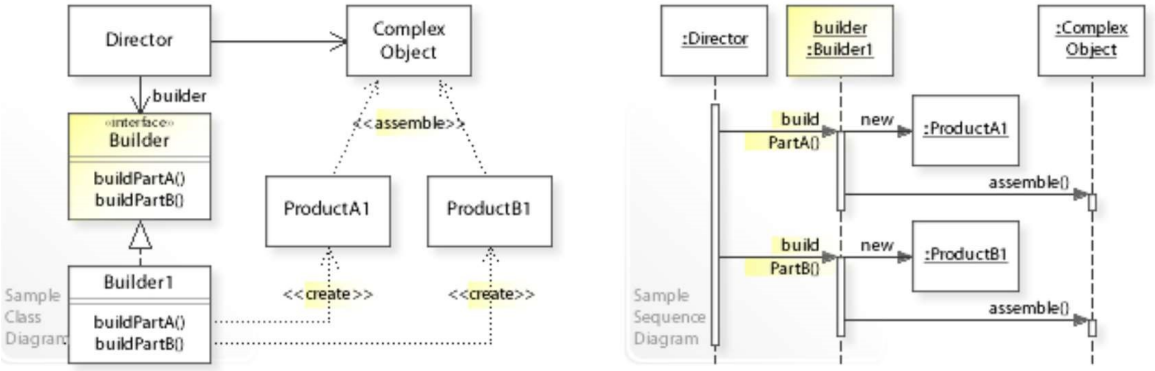


Ejemplo
The Factory Method defines an intefeace for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc) is determined by the mold.



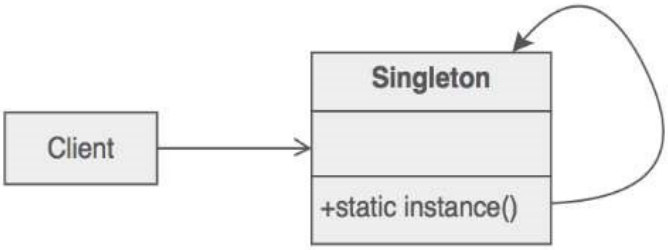
Builder

El patrón Builder sirve para proveer una solución flexible a la creación de objetos complejos. La intención d Builder es separar la construcción de objetos complejos de su representación. De esta forma, el mismo proceso de creación puede crear diferentes representaciones.



Singleton

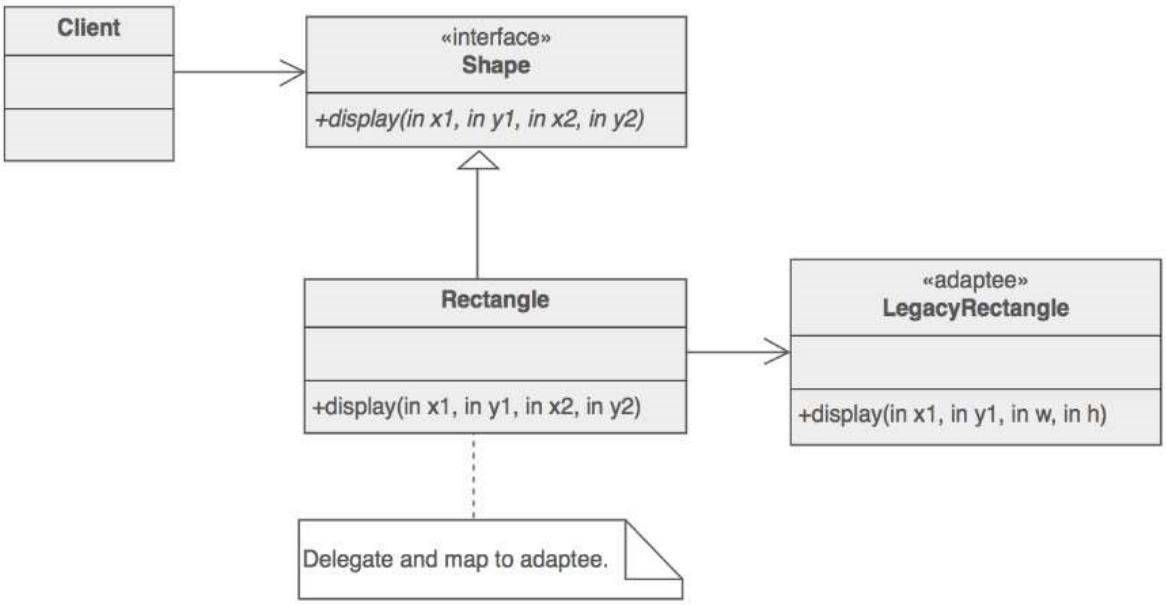
Asegura que una clase tendrá solamente una instancia y provee un punto global por el cual acceder a ella.
Se usa cuando la aplicación requiere de una y solo una instancia de ese objeto.



Estructurales

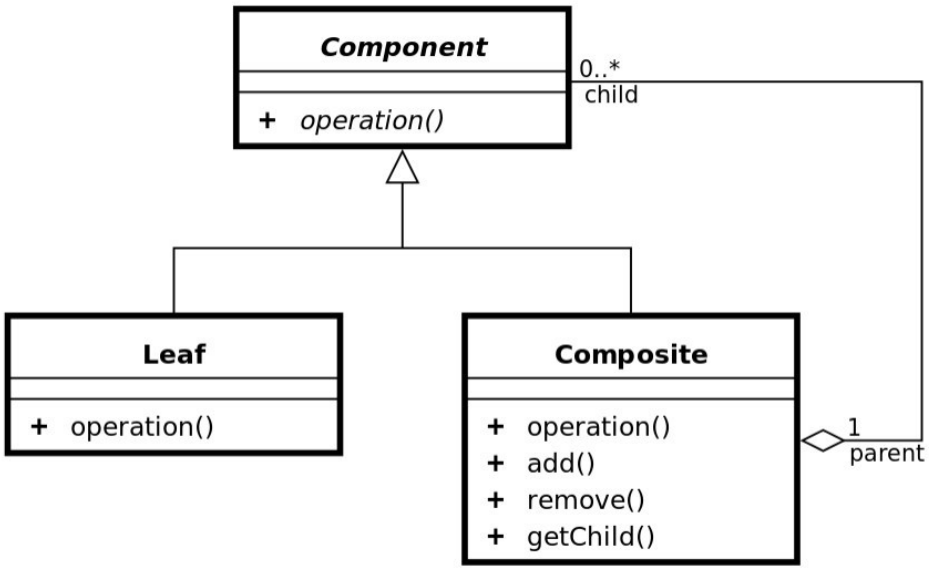
Adapter

Convierte la interface de una clase en la que el cliente espera.



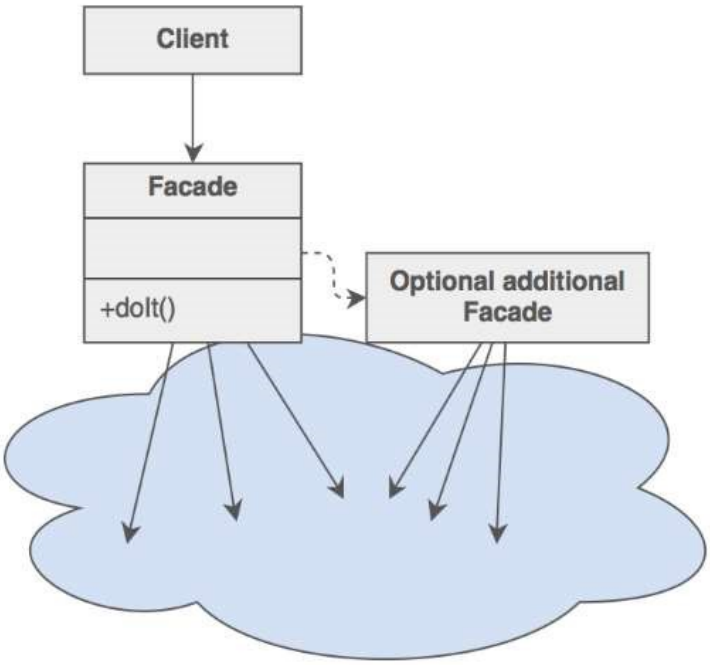
Composite

Compone objetos en estructuras de árbol para representar jerarquías.
Hace uso de la recursión para esto.



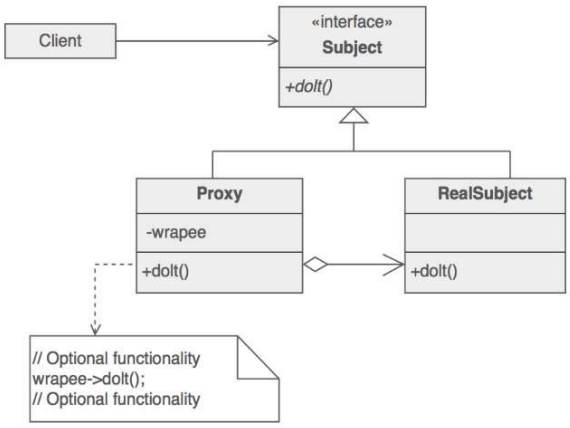
Facade

Provee una interfaz unificada para un set de interfaces en un subsistema.

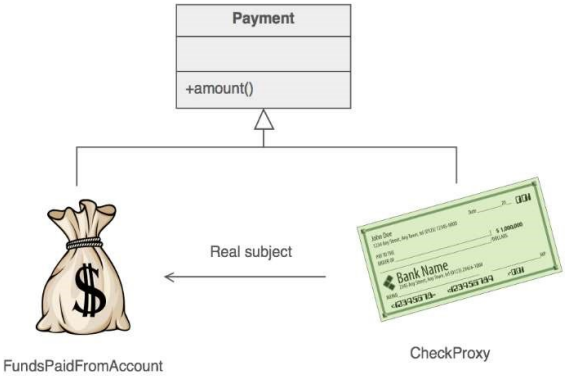


Proxy

Funciona de representante de otro objeto. Usa un nivel extra de indirección para soportar un acceso distribuido, controlado e inteligente.



Ejemplo

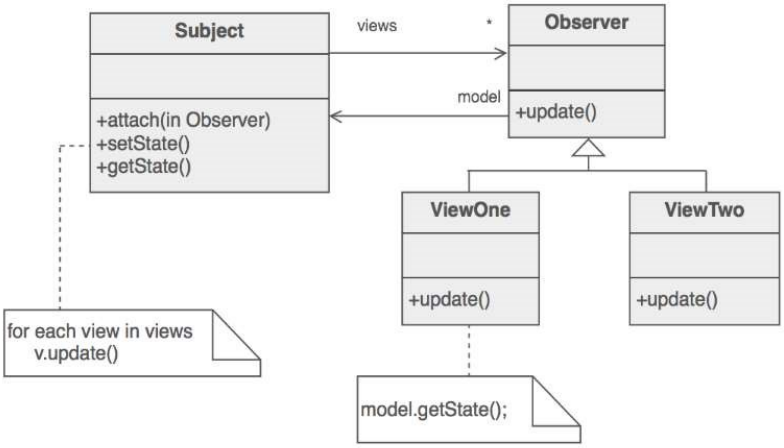


- **Remoto**: Oculta el hecho de que un objeto reside en otro espacio de direcciones.
- **Virtual**: Puede realizar optimizaciones, como la creación de objetos bajo demanda.
- **Protección**: Permite realizar diversas tareas de mantenimiento adicionales al acceder a un objeto (control de permisos por ejemplo).

Comportamiento

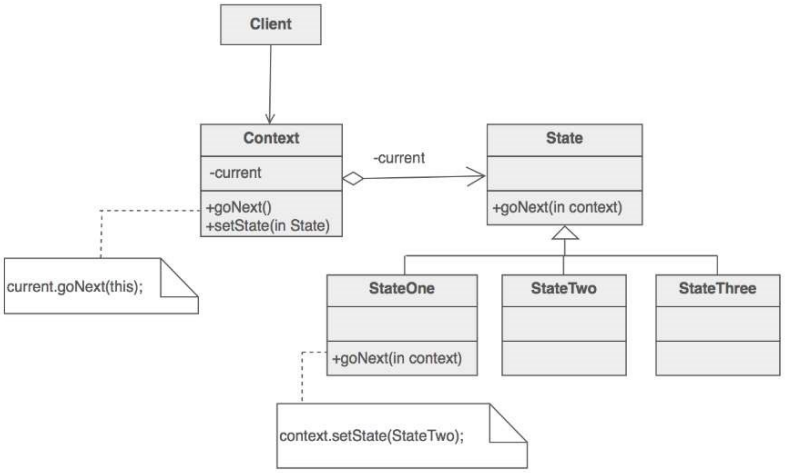
Observer

Sirve para definir dependencias one-to-many entre objetos de forma tal que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.



State

Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Es una máquina de estados orientada a objetos



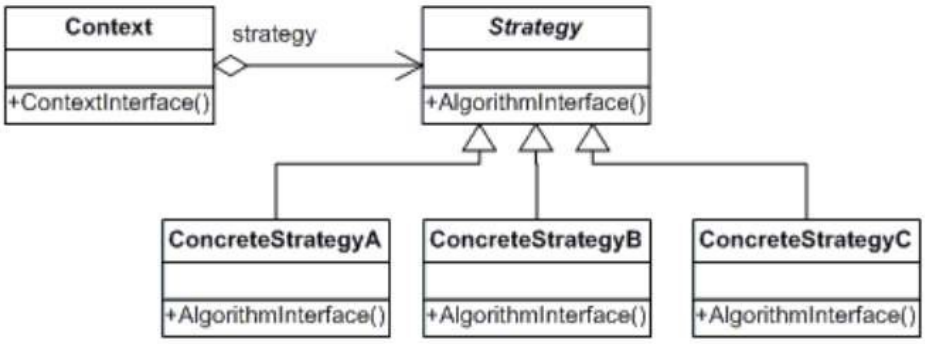
Ejemplo

This pattern can be observed in a vending machine. Vending machines hace states based on the inventory, amaount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.

Strategy

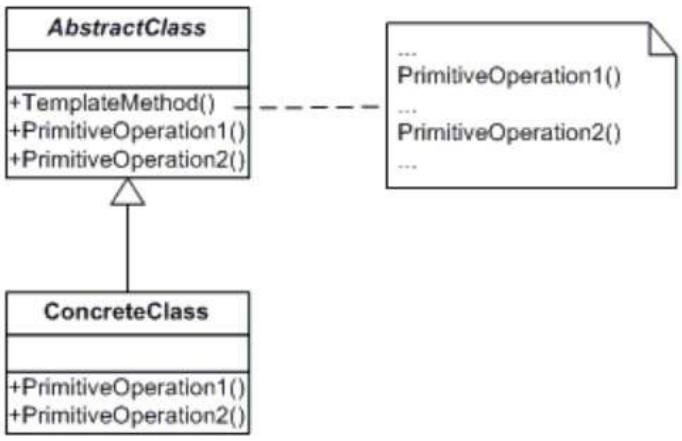
Define a una familia de algoritmos, encapsula a cada uno y los hace intercambiables. Strategy permite que el algoritmo varíe independientemente de los clientes que la usen.

.

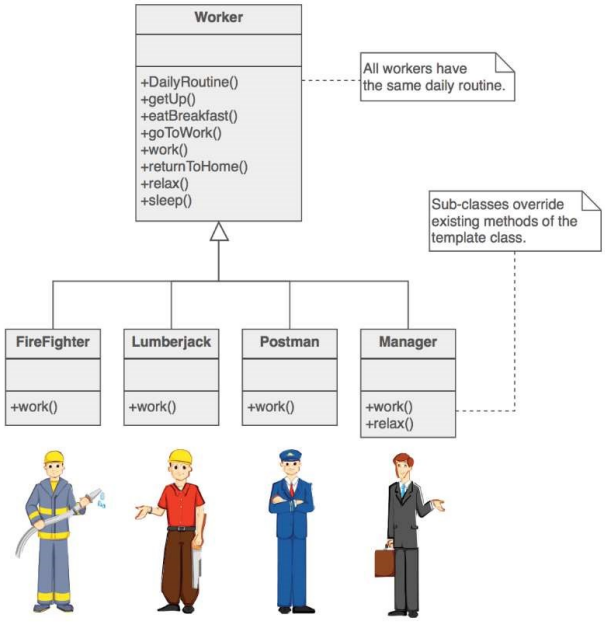


Template Method

Define el esqueleto de un algoritmo en una operación, delegando algunos pasos a subclases.

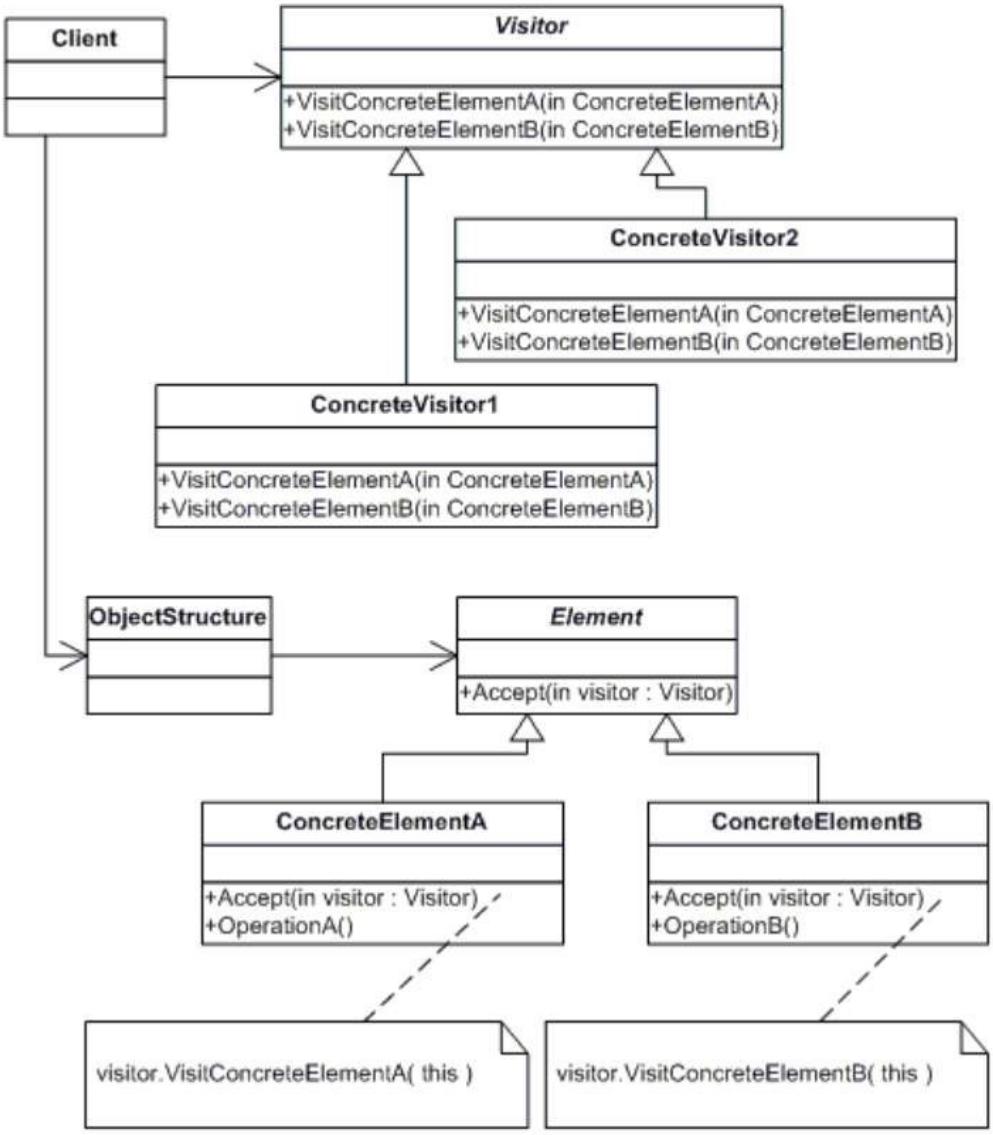


Ejemplo

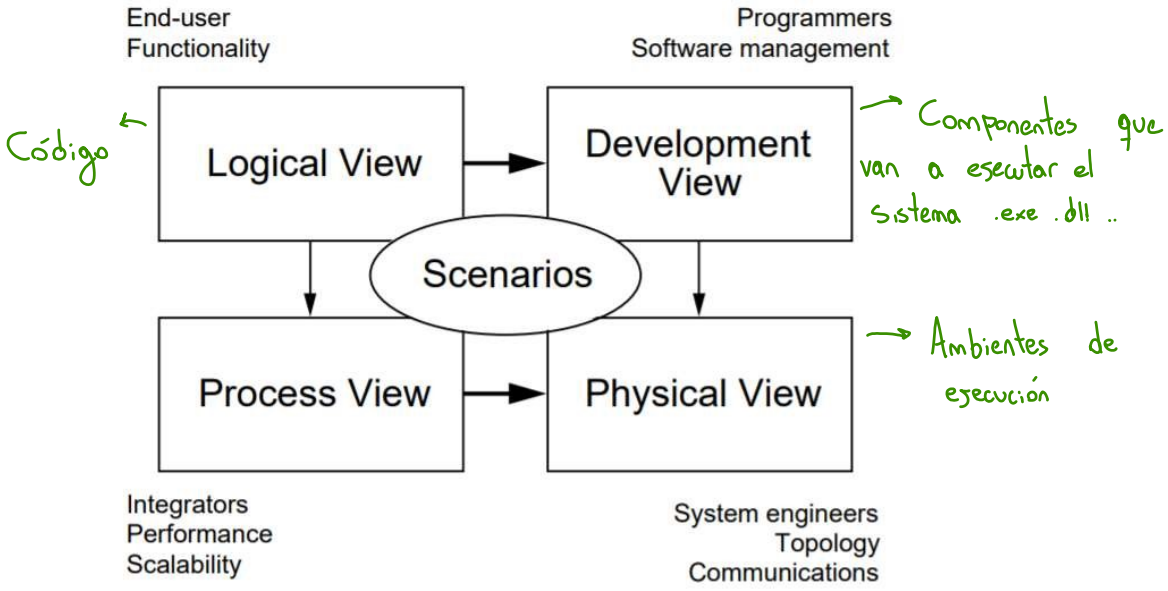


Visitor

Representa una operación a ser realizada en los elementos de una estructura de objetos (como un composite). Visitor permite definir una nueva operación sin cambiar las clases de los elementos con los que opera.



MODELO 4 + 1



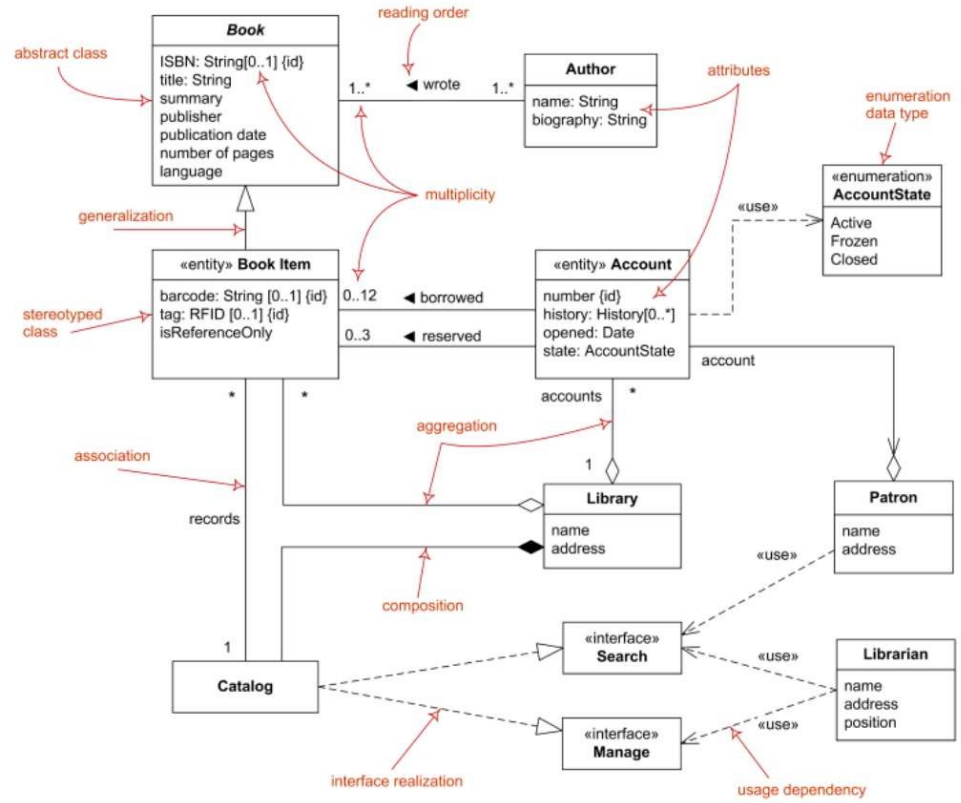
Software architecture deals with abstraction, with decomposition and composition, with style and esthetics. To describe a software architecture, we use a model composed of multiple views or perspectives. In order to eventually address large and challenging architectures, the model we propose is made up of five main views.

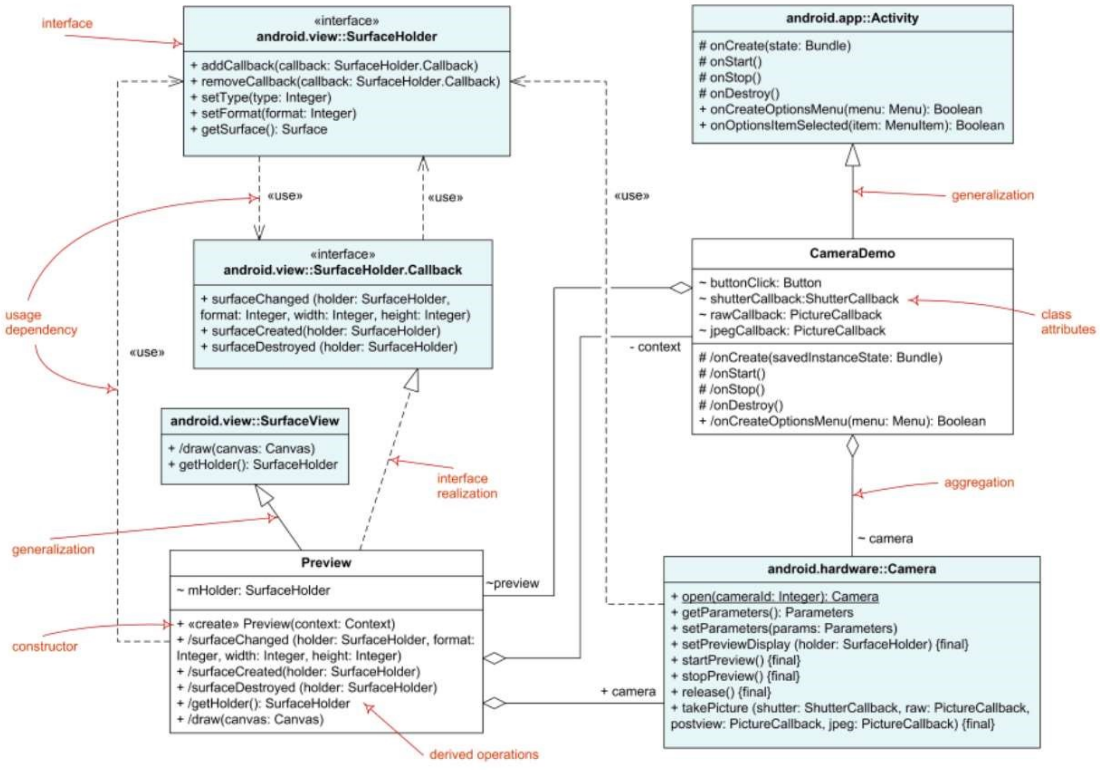
- The logical view, which is the object model of the design (when an object-oriented design method is used).
- The process view, which captures the concurrency and synchronization aspects of the design.
- The physical view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect.
- The development view, which describes the static organization of the software in its development environment.
- The description of an architecture, the decisions made, can be organized around these four views, and then illustrated by a few selected use cases, or scenarios which become a fifth view.

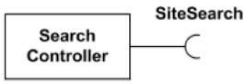
Vista Lógica

The logical architecture primarily supports the functional requirements, what the system should provide in terms of services to its users.
The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of objects or object classes. They exploit the principles of abstraction, encapsulation, and inheritance.

Diagrama de clases







***Interface** SiteSearch is **used** (required) by SearchController.*

***Interface** SiteSearch is **realized** (implemented) by SearchService.*

Diagrama de paquetes

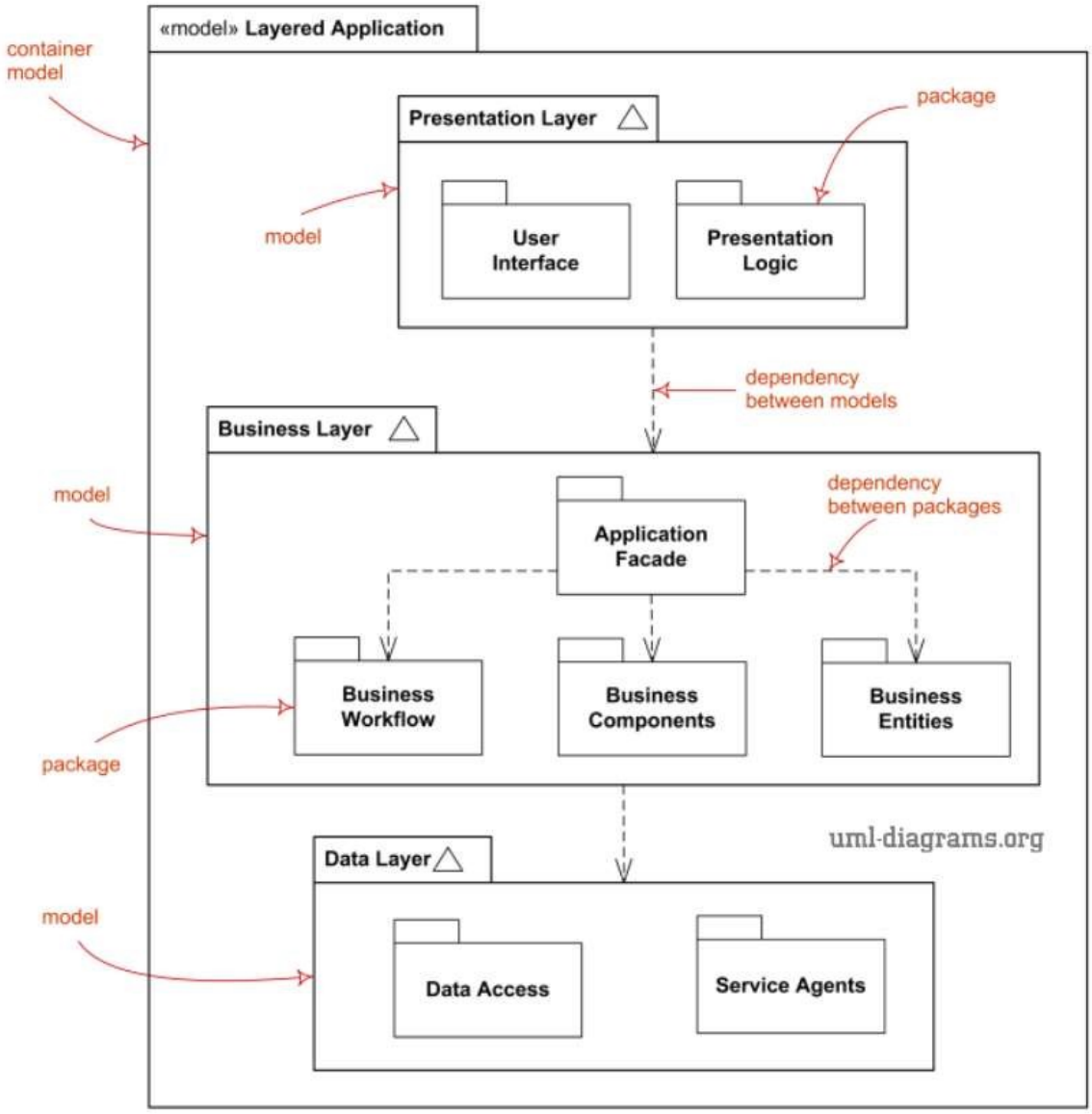


Diagrama de estructura compuesta

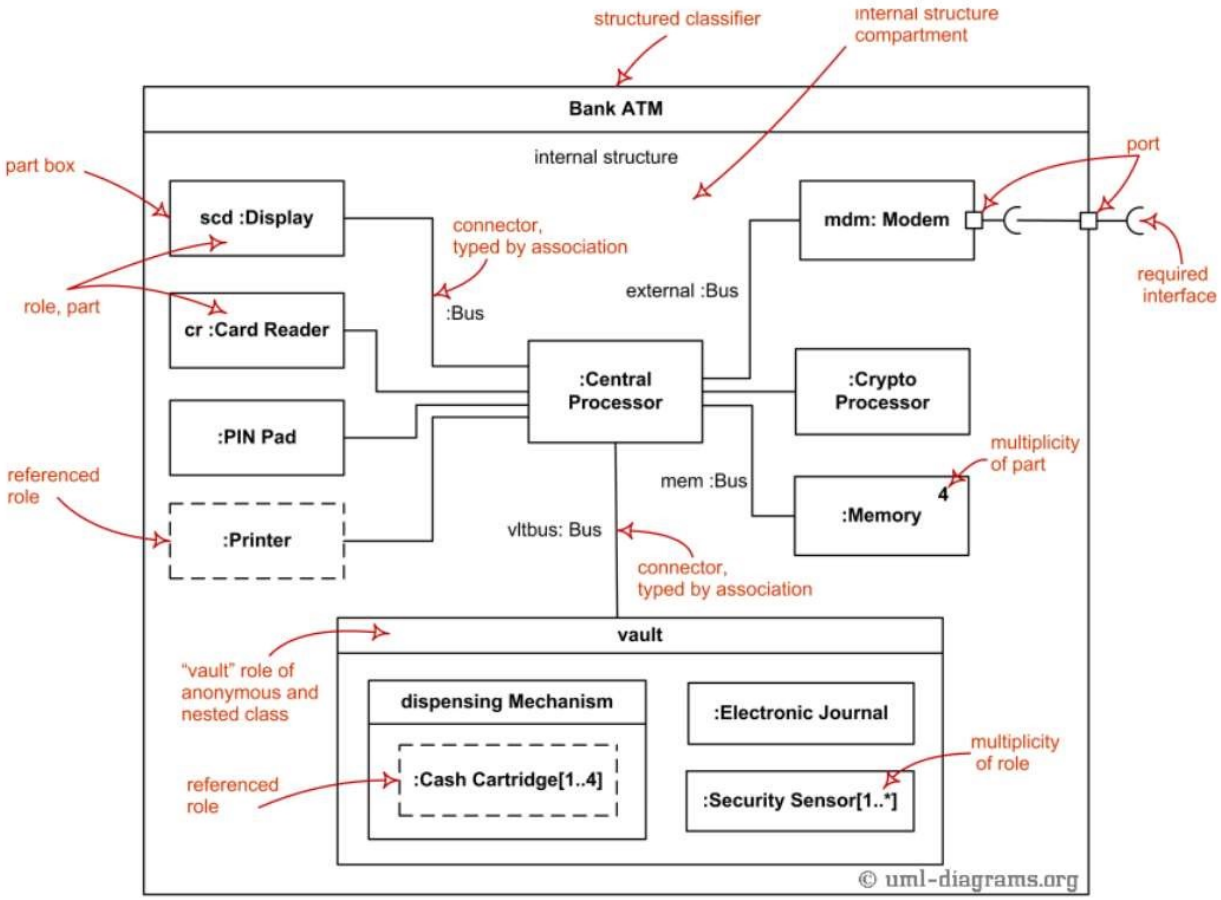
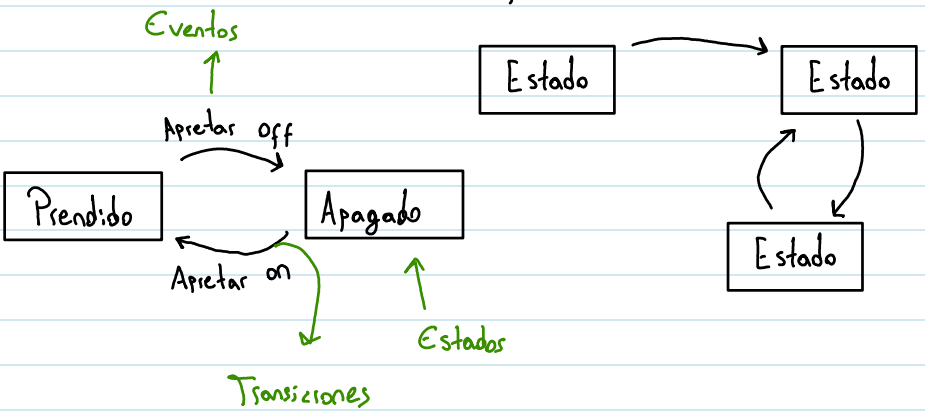


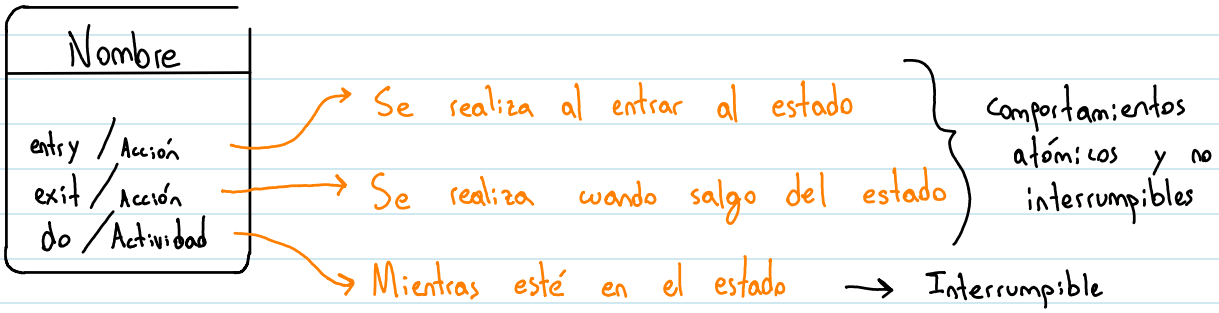
Diagrama de Estados → Se utilizan para especificar comportamiento del sistema o partes de sistemas que son determinísticos

Se representa:



Los estados y las transiciones tienen el comportamiento

Estado

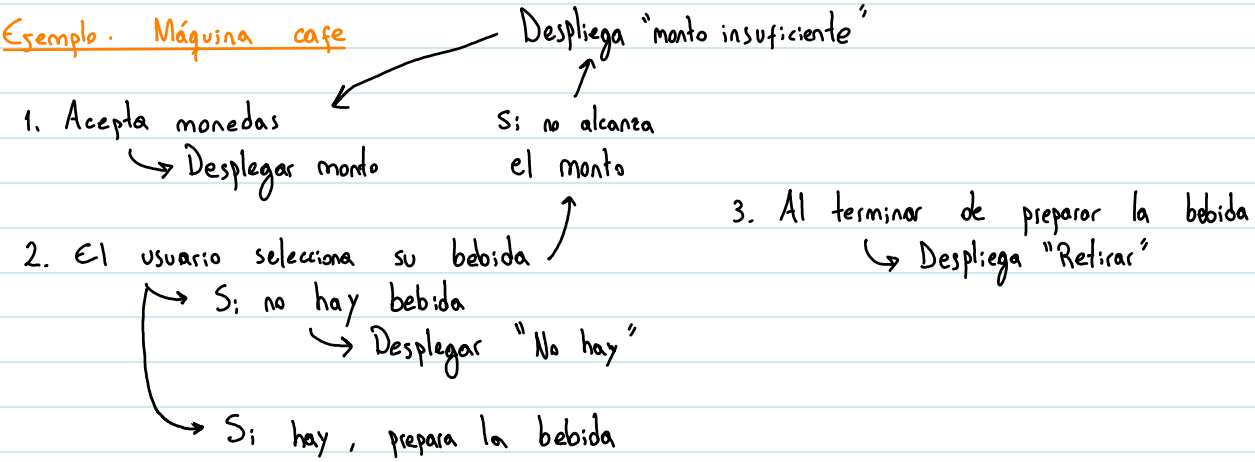


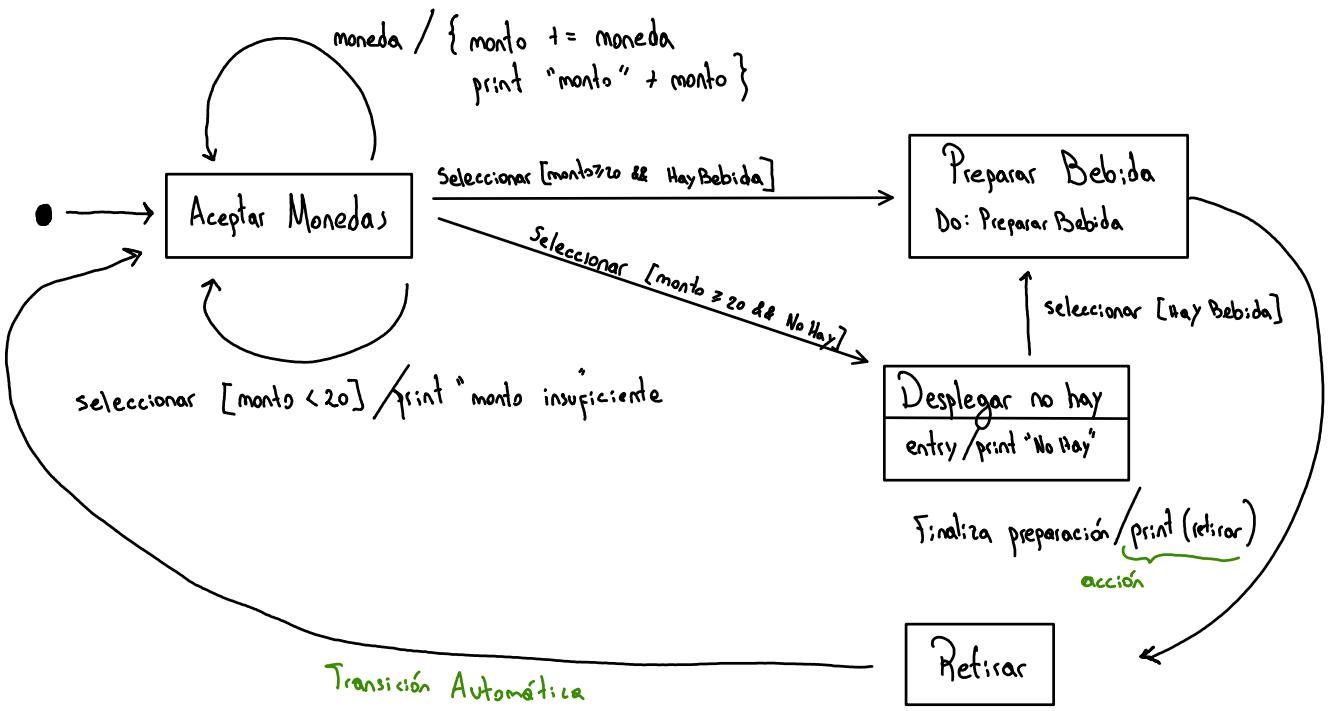
Pseudoestado

- ● Inicial → Puede haber uno por máquina o estado compuesto
- ⊙ Final → Puede haber varios

Evento [condición] / Acción
Transición

Ejemplo. Máquina cafe





Vista de procesos

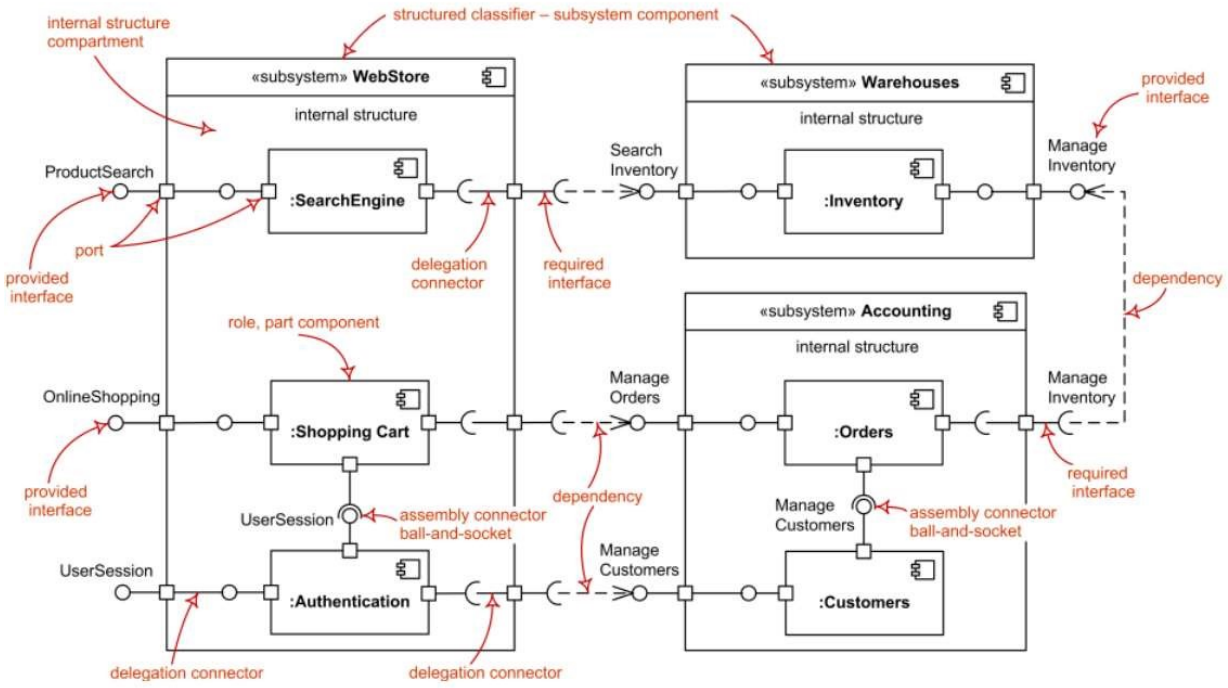
The process architecture takes into account some non-functional requirements, such as performance and availability. It addresses issues of concurrency and distribution, of system's integrity, of fault-tolerance, and how the main abstractions from the logical view fit within the process architecture—on which thread of control is an operation for an object actually executed.

The process architecture can be described at several levels of abstraction, each level addressing different concerns. At the highest level, the process architecture can be viewed as a set of independently executing logical networks of communicating programs (called "processes"), distributed across a set of hardware resources connected by a LAN or a WAN. Multiple logical networks may exist simultaneously, sharing the same physical resources. For example, independent logical networks may be used to support separation of the on-line operational system from the off-line system, as well as supporting the coexistence of simulation or test versions of the software. A process is a grouping of tasks that form an executable unit. Processes represent the level at which the process architecture can be tactically controlled (i.e., started, recovered, reconfigured, and shut down). In 5 addition, processes can be replicated for increased distribution of the processing load, or for improved availability. The software is partitioned into a set of independent tasks. A task is a separate thread of control, that can be scheduled individually on one processing node

Vista de Implementación

The development architecture focuses on the actual software module organization on the software development environment. The software is packaged in small chunks—program libraries, or subsystems— that can be developed by one or a small number of developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it. The development architecture of the system is represented by module and subsystem diagrams, showing the 'export' and 'import' relationships. The complete development architecture can only be described when all the elements of the software have been identified. It is, however, possible to list the rules that govern the development architecture: partitioning, grouping, visibility. For the most part, the development architecture takes into account internal requirements related to the ease of development, software management, reuse or commonality, and to the constraints imposed by the toolset, or the programming language. The development view serves as the basis for requirement allocation, for allocation of work to teams (or even for team organization), for cost evaluation and planning, for monitoring the progress of the project, for reasoning about software reuse, portability and security. It is the basis for establishing a line-of-product.

Diagrama de componentes



Vista de Despliegue

The physical architecture takes into account primarily the non-functional requirements of the system such as availability, reliability (fault-tolerance), performance (throughput), and scalability. The software executes on a network of computers, or processing nodes (or just nodes for short). The various elements identified— networks, processes, tasks, and objects—need to be mapped onto the various nodes. We expect that several different physical configurations will be used: some for development and testing, others for the deployment of the system for various sites or for different customers. The mapping of the software to the nodes therefore needs to be highly flexible and have a minimal impact on the source code itself.

Diagrama de engrega