10. UML

es la formalización de aspectos del software en diagramas. No se usan los diagramas formales pero se usan aproximaciones a estos. Posee diagramas de estructura y comportamiento.

Algunos concepto compartidos entre diagramas

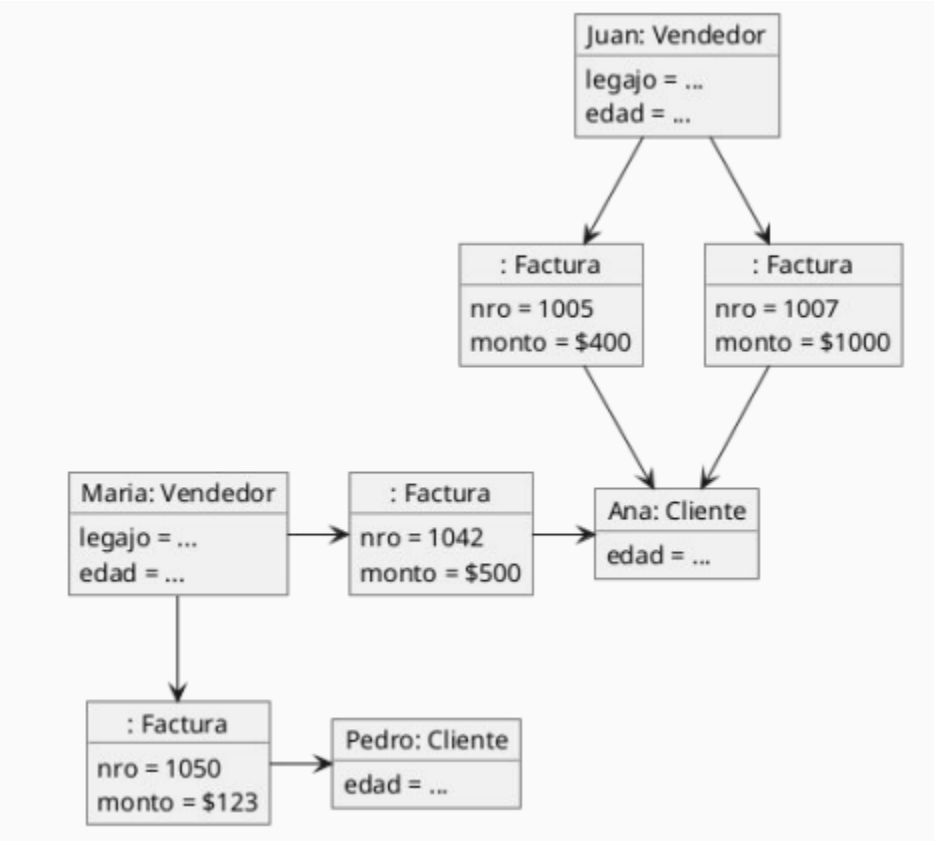
> *estereotipo*:

- mecanismo de extensión, se represnta con <<>>
- modifica el comportaimiento (inyecta propiedades, especifica semánticas para que la denomincacipon sea adecuada para el problema en particualr)
- puede ser útil en diferentes diagramas, extiende un elemento UML:
  - en diagramas de componentes:
    - "*component*": módulo lógico reusable (como librería, contiene clases,interfaces)
    - "*excutable*": la componente se puede ejecutar (una app)
  - en diagramas de despliegue:
    - "*node*":
      - representa un recurso computacional que se puede ejecutar software
      - puede ser lógico o físico
      - una especialización de node es *device* que se asocia a un hardware concreto
    - "*execution enviroment*": entorno lógico de un nodo (browser, docker, mongo db, node.js)
  - en diagramas de clase:
    - "*interface*"
    - "*datatype*": resepresenta un tipo simple
    - "*enumeration*": tipo con conjunto finito de valores posibles

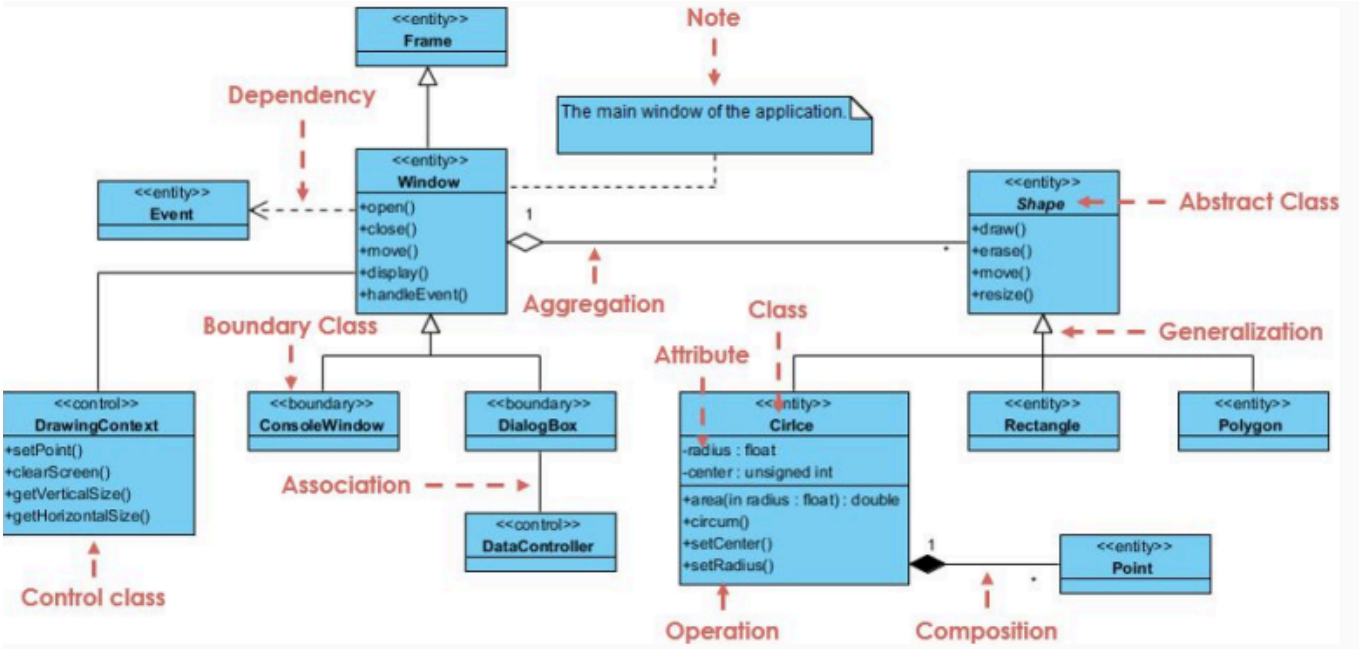
    - "*stereotype*": defines un nuevo estereotipo
- hay etereotipos en frameworks, como por ejmplo, en springboot:
    - @RestController <- que agrega comportamiento adicional en runtime.
    - @JpaEntity <- que generan métodos adicionales en compilación.

diagramas de estructura

objetos

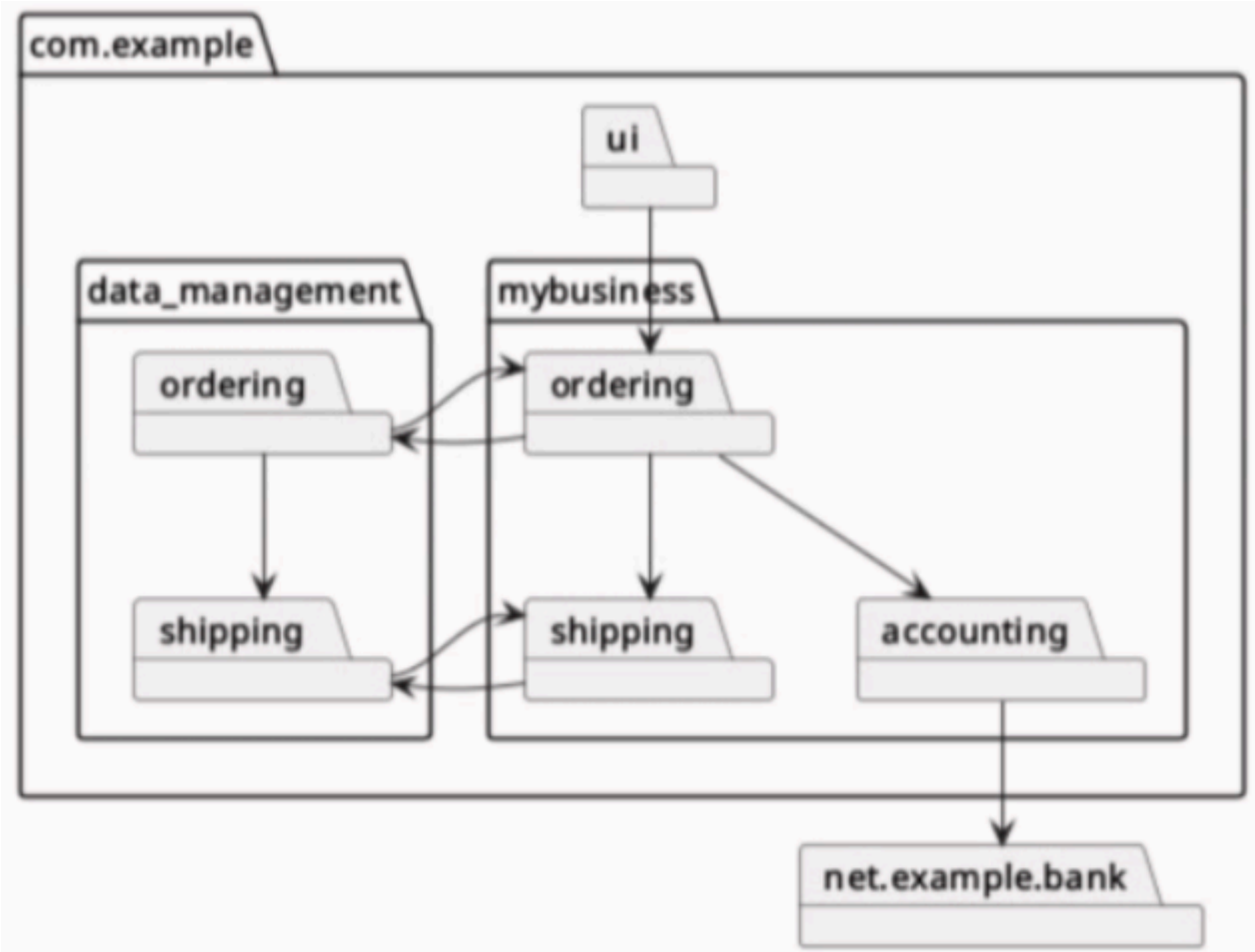


clases

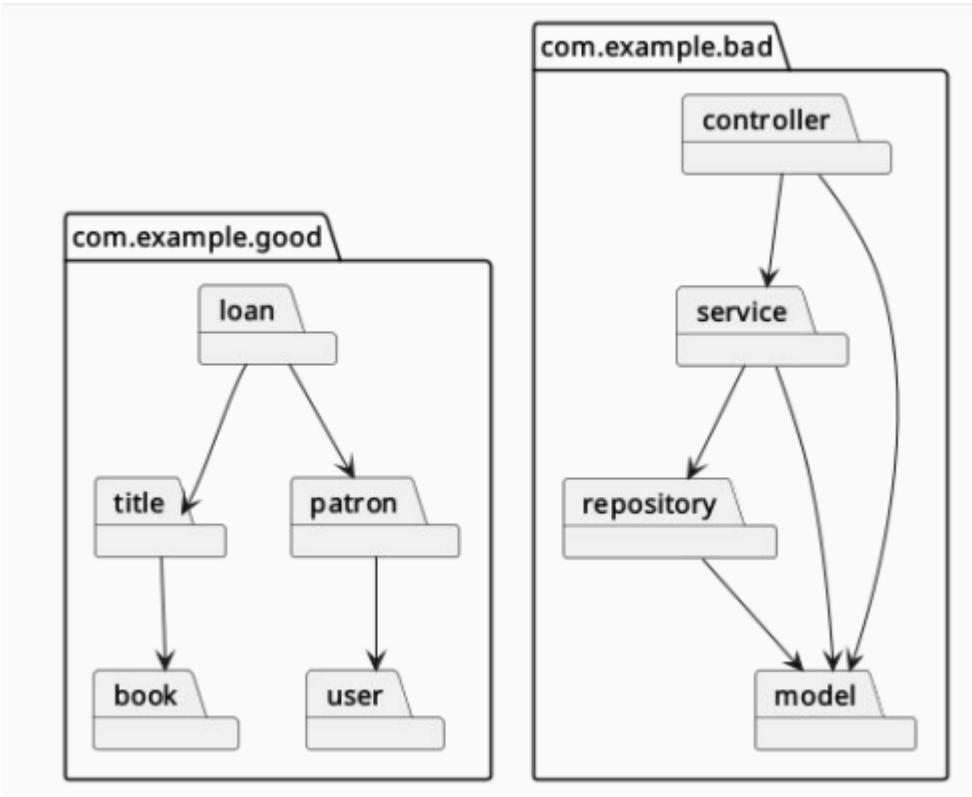


paquetes

- se realiza sobre el diagrama de clases
- su objetivo es organizar y agrupar los elementos del sistema (clases, interfaces, etc.) para mostrar la estructura modular de alto nivel
- muestra la organización y disposición de diversos elementos de un modelo en forma de paquetes.



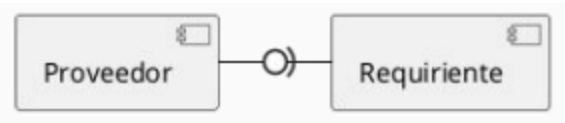
- Agrupando

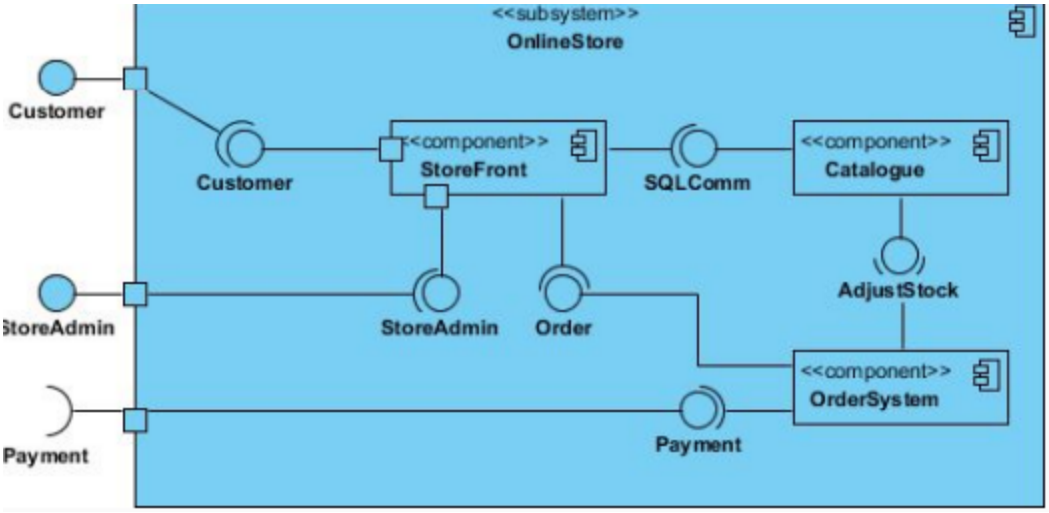


- en example.bad: estructura centrada en el diseño por capas
- el tipo tpecnico de a qué capa pertenece cada clase define su ubicación

- en este caso hay muchas dependencias cruzadas: controller depende de model directa e indirectamente
- acoplamiento cruzada, dificil de escalar por rigidez del diseño, no hay agrupamiento por funcionalidad del negocio y demuestra la mala separación de responsabilidades (controller no debe acceder a model si no es a través de un service).
- en example.good: se basa en funcionalidades del sistema, no tipos técnicos.
- cada módulo contiene lo que necesita (controllers, etc).
- los paquetes interactuan entre sí a través de interfaces claras.

componentes





- muestra cómo el software se estructura en componentes independientes y cómo estas dependen de otras componentes, cómo se relacionan.
- representa las componentes (con rectangulos), las interfaces (como semicirculos entre las aristas que conectan componentes), y las dependencias (como una flecha que indica que uno depende de otro para funcionar.)

despliegue

- muestra cómo se distribuyen los componentes del sistema en hardware.
- cómo y dónde se ejecutan los componentes, su *disposición física en nodos* (componentes de hardware), ilustra también las **configuraciones físicas de software y hardware escenciales para la ejecución**, y operación del sistema.

Cada componente es "enchufable", la encapsulación general es de la forma:

Device

detalles físicos: por ejemplo laptop/pc

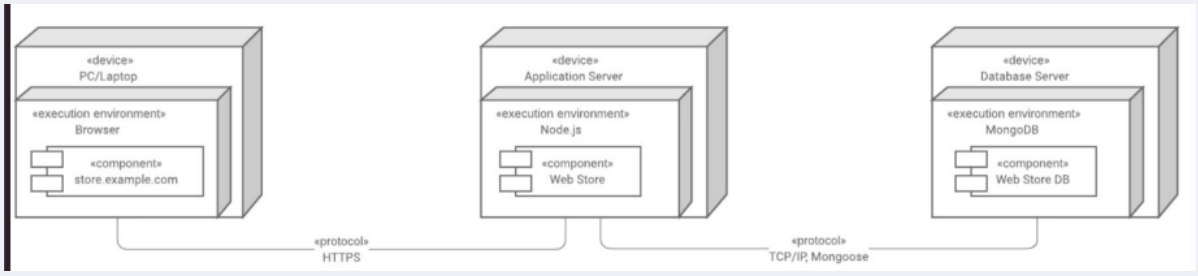
Enviroment

ambiente de despliegue dentro del device. por ejmp: browser, node.js, mongo.db.

Component

componente o servicio que se despliega en dicho ambiente. Por ejemplo: example.com (en el browser), o web store (en node.js, levantando una tienda), o web store DB (en mongo db)

- muestra también la comunicación entre los ambientes que interactúan



en el ejemplo:

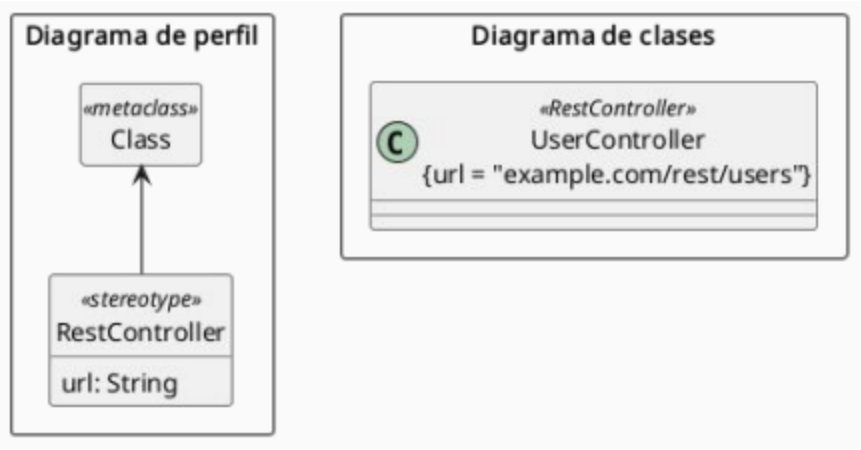
- <*device*>:
  - cubos que representan los elementos físicos de un sistema
  - por ejemplo: pc, laptop, database server, application server.
- los ambientes de ejecución:
  - representan un *entorno dentro del dispositivo*.

        - Puede ser un motor de ejecución de código o una base de datos.
        - Por ejemplo:
            - en una pc/laptop: browser (app de cliente, de front)
            - en un applcation server: node.js (app que levanta back)
            - en el database server: mongoDB (app de la base de datos)
    - los componentes:
        - representan un módulo que se debe desplegar en un entorno en particular.
        - es lo que se *ejecuta dentro del entono* .
        - por ejemplo:
            - en el browser, la componente `store.example.com` (representa la UI levantada en el browser, tiene una abstracción para enviar peticiones)
            - en node.js: `web store` (en este caso, un único servicio lógico que atiende peticiones de front por HTTPS, procesa la lógica de negocio, consulta y actualiza la bd y responde al cliente.)
            - en mongo db: `web store DB` (representa la aplicación web store DB con las que interactua el back y realiza el procesamiento para responder, es decir algoritmos de motores de bases de datos)
    - los protocolos de comunicación usando diferentes protocolos como:
        - HTTPS <-cliente con serv
        - TCP, Mongoose <-serv con mongo

perfil

El perfil uml representa la extensión del *metamodelo UMl* para definir estereotipos personalizados. estos estereotipos personalizados se pueden aplicar luego en modelo concretos (por ejemplo en un diagrama de clase).
Par definir el perfil UML referimos a los estereotipos y a las metaclase sobre las

cuales definiremos que nuestros estereotipos se puedan aplicar. Por ejemplo:



1. Definición del estereotipo (en el perfil uml):

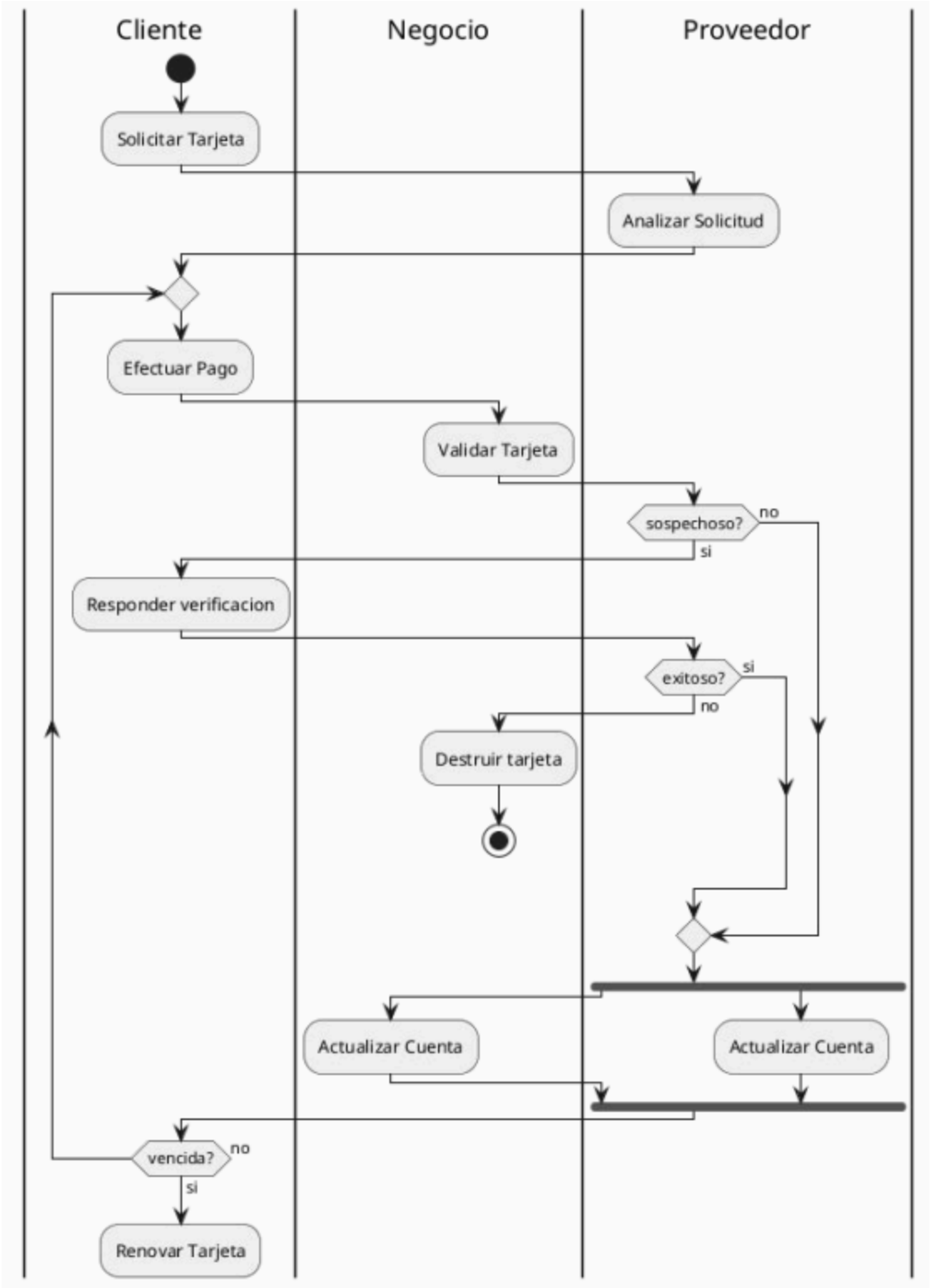
- se define un estereotipo que represente la extensión semántica de elementos UML. en este caso *RestController*, este contiene un atributo *url* que se tendrá que definir cuando se use el estereotipo
- se define sobre qué metaclase se puede aplicar. en este caso *class*. esto se representa con una flecha que apunta a la *metaclase* sobre la cual se puede aplicar el estereotipo.
- se puede usar como estereotipo en el diagrama de clases, definiendo su url en este ejemplo.

diagrama de comportamiento

actividad

- modela proceso de negocio o algoritmos.
- representa *casos de uso de forma detallada*
- se compone de:
  - circulo negro: donde inicia
  - circulo negro con borde blanco: donde termina
  - rectangulo redondeado: acción o tarea
  - hexagono: bifurca segun condiciones
  - línea gruesa horizontal o vertical (fork-join): varias actividades comienzan o

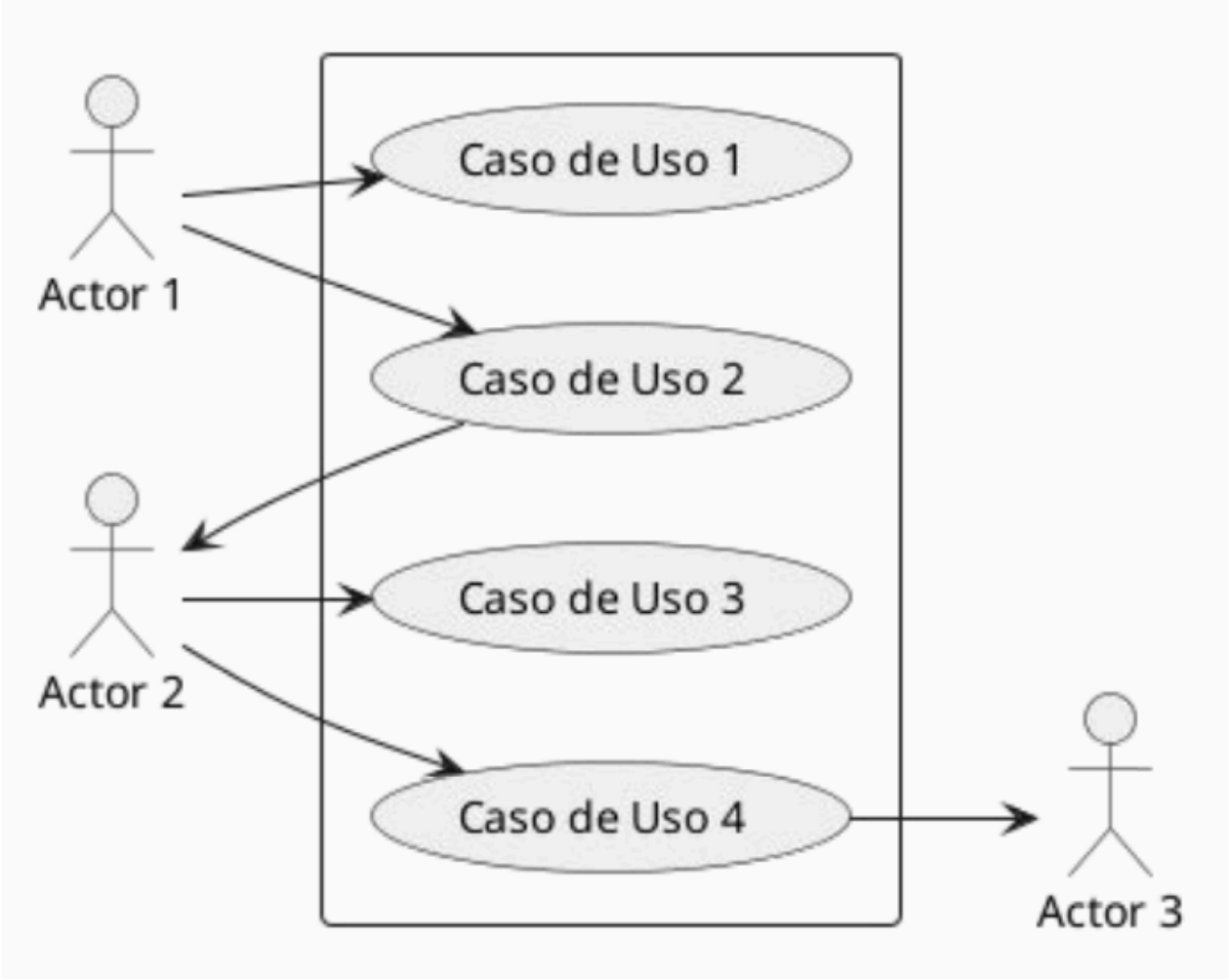
terminan en paralelo. tanto el fork y el join son barras horizontales gruesas.



casos de uso

- representa las *funcionalidades principales de un sistema* desde el punto de vista del usuario.

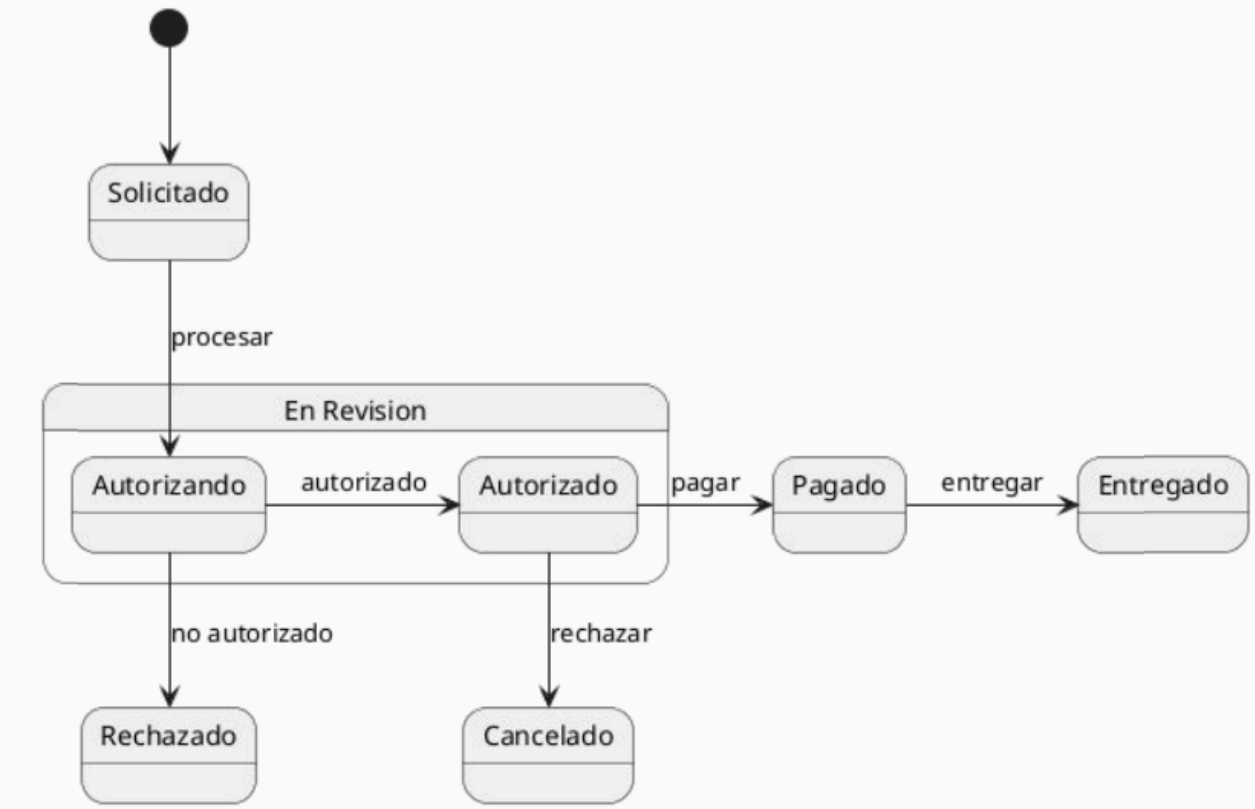
- No entra en detalles de técnicos



maquina de estados

- representa el *comportamiento de un objeto o sistema* en función de sus estados y cómo *cambia de estado* ante ciertos *eventos*.
- Modela el comportamiento dinámicos de un sistema, representa los cambios de estado ante eventos.
- sus componentes:
  - el estado
  - el punto de partida
  - el punto fin del proceso

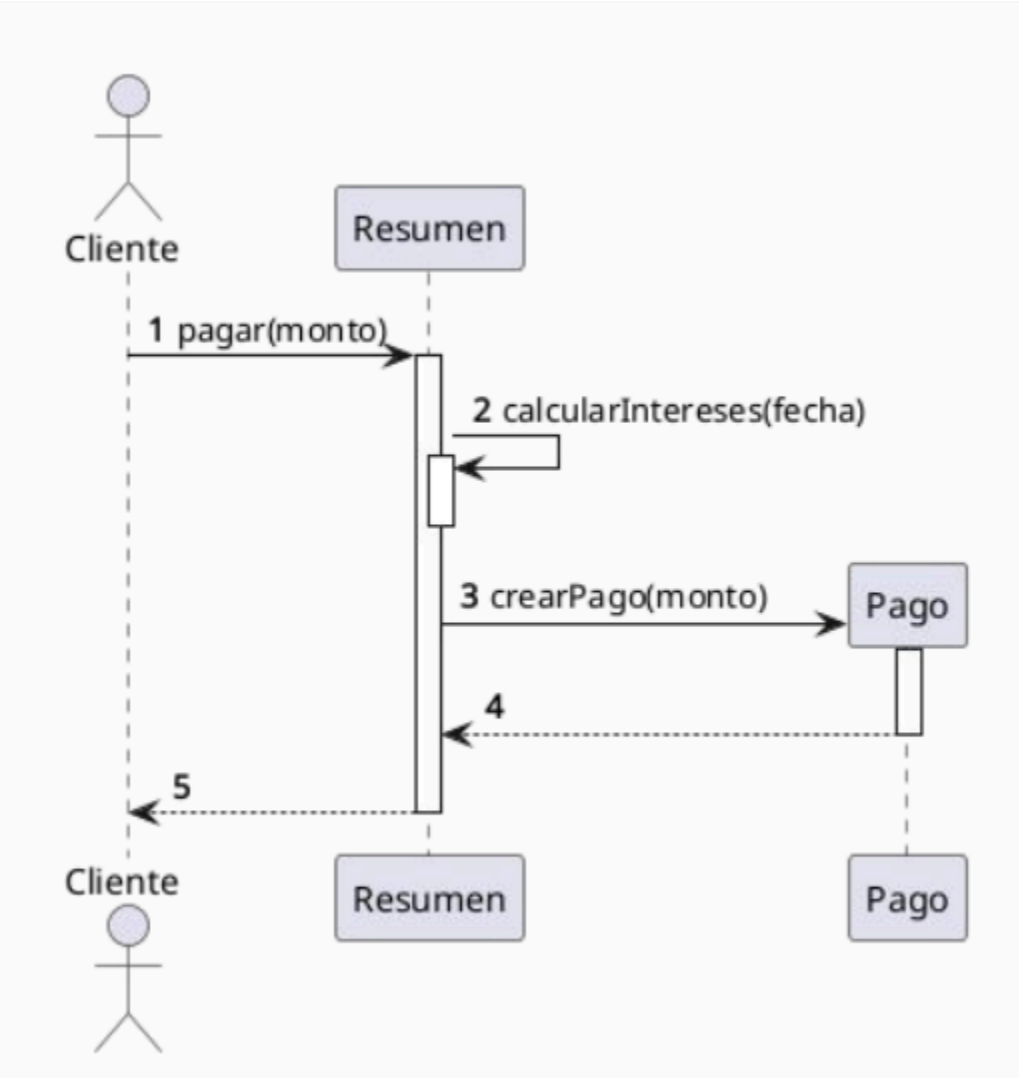
- transiciones orientadas



interacción

secuencia

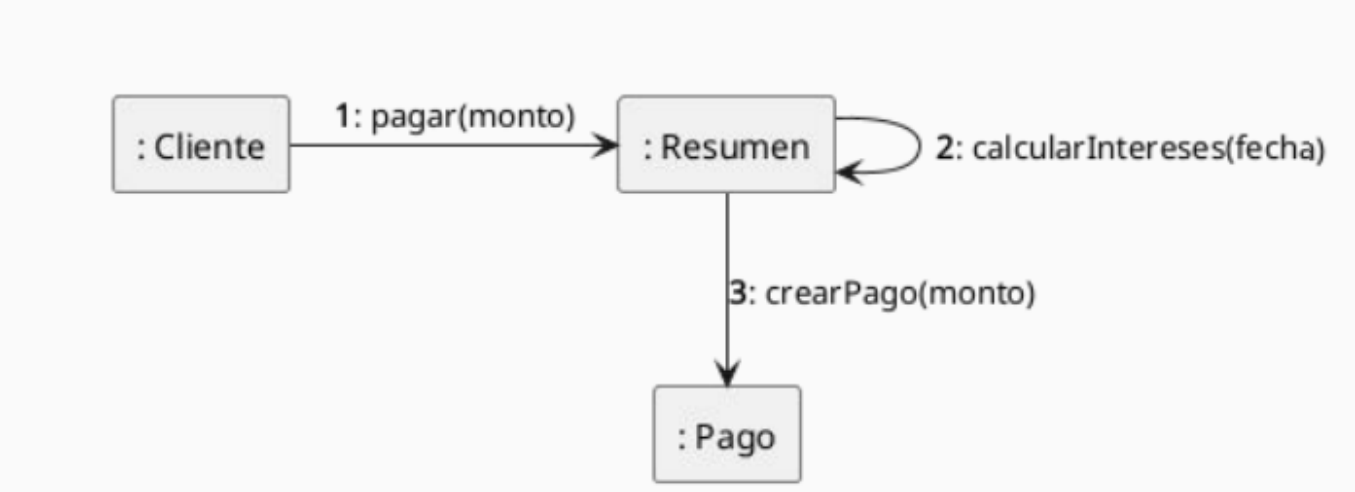
Muestra quién habla con quien en un orden a lo largo del tiempo(entre objetos o componentes)



comunicación

objetivo similar al diagrama de secuencia pero se especifica en la relación entre los objetos (quién se comunica con quién) y no se enfoca en el momento en el que ocurre cada mensaje.
A partir del diagrama de comunicación, solo hay que quitarle los detalles de la comunicación entre objetos y obtenemos el diagrama de objetos (las flechas

permanecen).



tiempos

Modela el desarrollo general de los diferentes procesos de un sistema antes una acción. No especifica detalles internos de implementación, solo sobre la coordinación entre los procesos.
Muestra la evolución de un evento a lo largo del tiempo.