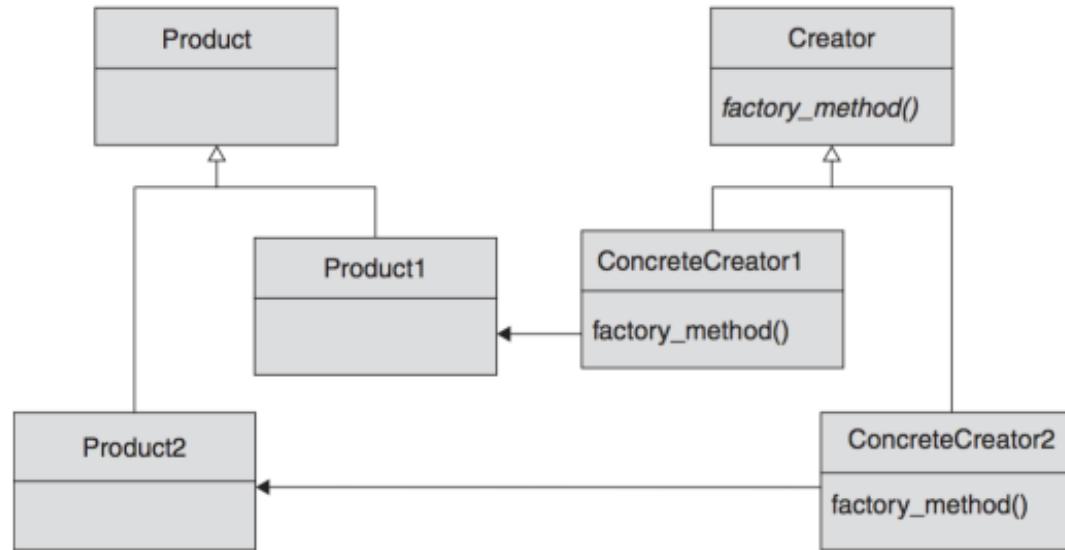


Factory Method



Este patrón es en realidad el patrón template method aplicado al problema de creación de objetos

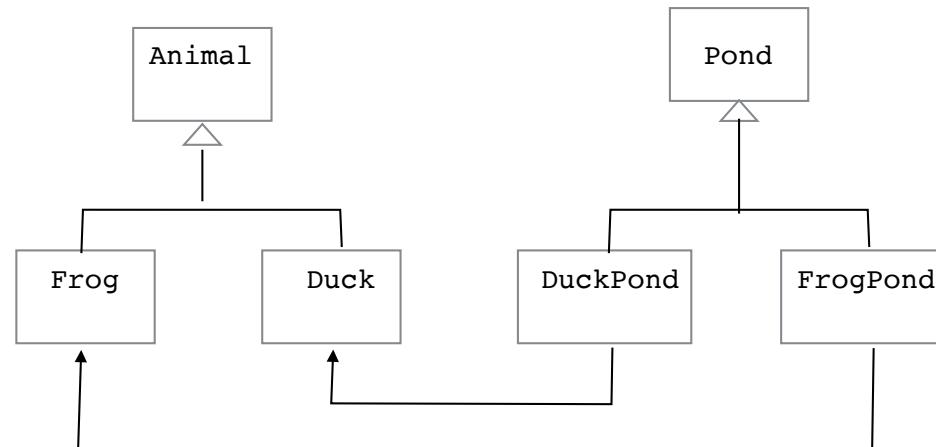
La laguna de patos

```
class Pond
  def initialize(number_animals)
    @animals = []
    number_animals.times do |i|
      animal = new_animal("Animal#{i}")
      @animals << animal
    end
  end
  def simulate_one_day
    @animals.each { |animal| animal.speak}
    @animals.each { |animal| animal.eat}
    @animals.each { |animal| animal.sleep}
  end
end

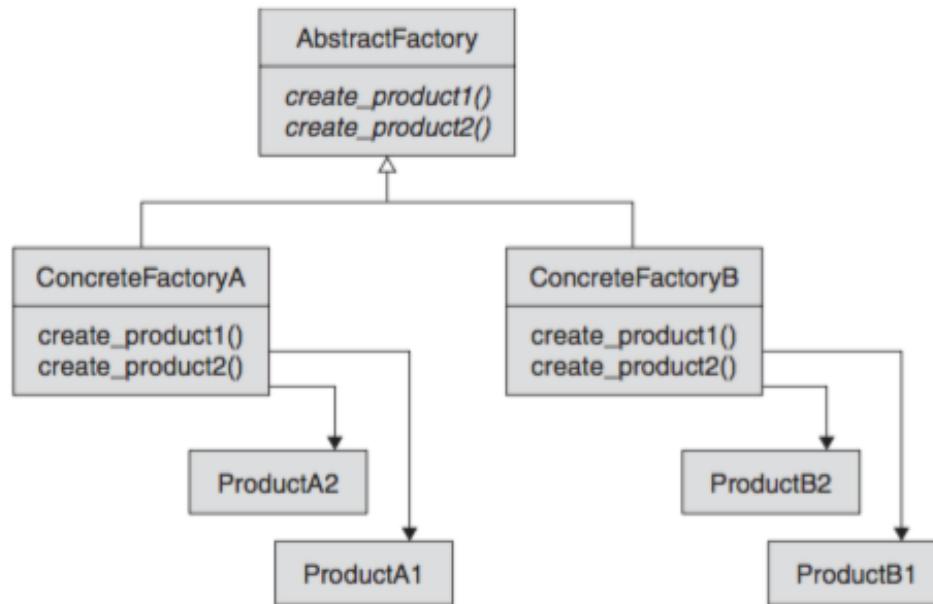
class DuckPond < Pond
  def new_animal(name)
    Duck.new(name)
  end
end

class FrogPond < Pond
  def new_animal(name)
    Frog.new(name)
  end
end

pond = FrogPond.new(3)
pond.simulate_one_day
```



La Fábrica Abstracta



- Dos factories concretas, cada una capaz de crear su propio set de productos
- Es la misma idea del patrón Estrategy pero aplicado al problema de crear objetos

El Habitat se inicializa con la fábrica de objetos apropiada

```
class PondOrganismFactory
  def new_animal(name)
    Frog.new(name)
  end
  def new_plant(name)
    Algae.new(name)
  end end

class JungleOrganismFactory
  def new_animal(name)
    Tiger.new(name)
  end
  def new_plant(name)
    Tree.new(name)
  end
end

class Habitat
  def initialize(number_animals, number_plants, organism_factory)
    @organism_factory = organism_factory
    @animals = []
    number_animals.times do |i|
      animal = @organism_factory.new_animal("Animal#{i}")
      @animals << animal
    end
    @plants = []
    number_plants.times do |i|
      plant = @organism_factory.new_plant("Plant#{i}")
      @plants << plant
    end
  end
  # Rest of the class...
end

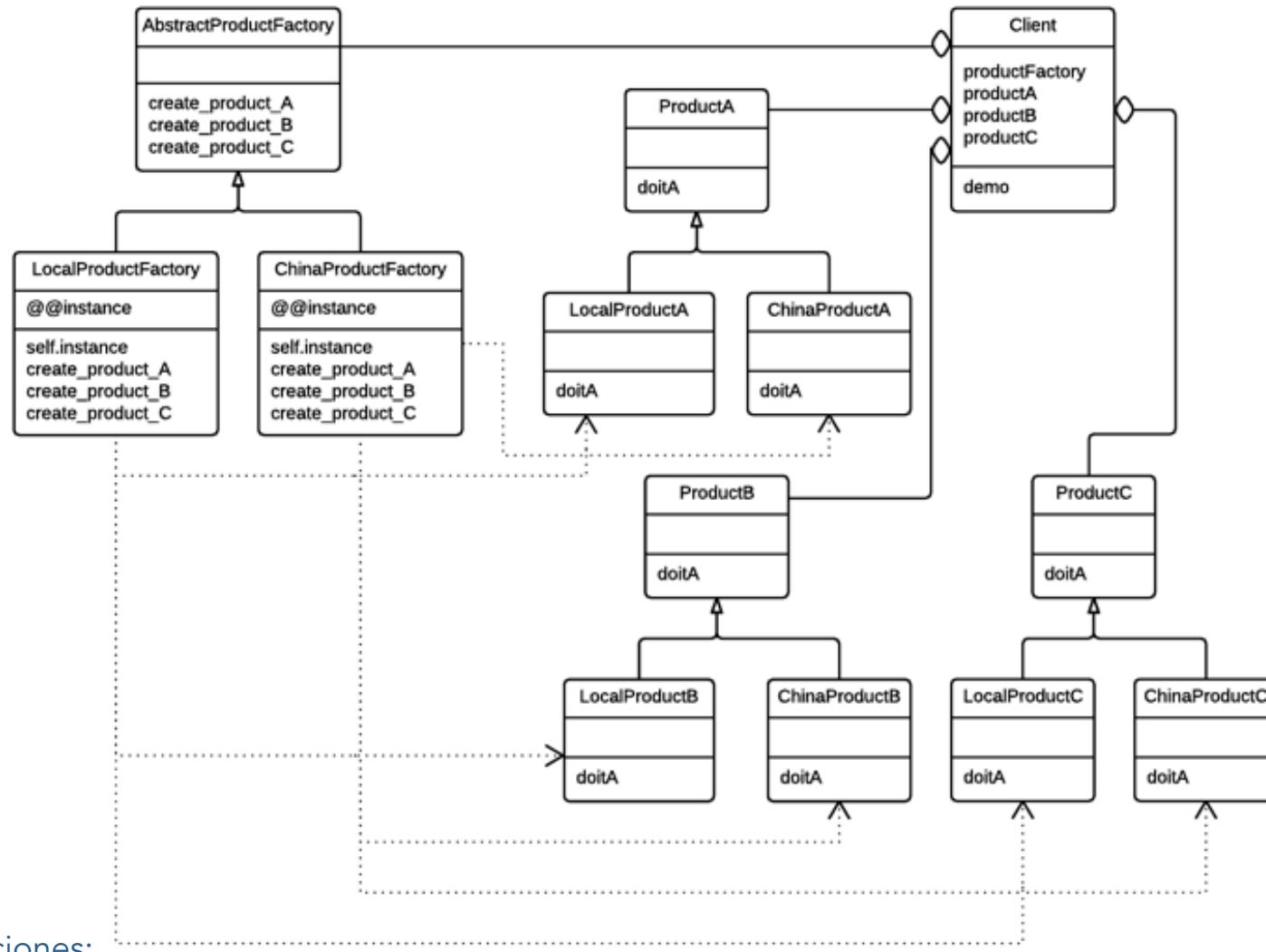
jungle = Habitat.new(1, 4, JungleOrganismFactory.new)
jungle.simulate_one_day
pond = Habitat.new( 2, 4, PondOrganismFactory.new)
pond.simulate_one_day
```

Una empresa produce 3 productos distintos: A, B y C. Sin embargo cada uno de ellos los produce en dos versiones, una proveniente de la fábrica local de alta calidad y una proveniente de una fábrica en China que los produce a bajo costo. Sólo hay una fábrica en cada uno de estos sitios. Queremos simular esta fábrica usando uno o mas patrones de diseño de los aprendidos en clase.

Es muy probable que Ud estime conveniente usar la fábrica abstracta para producir las dos fábricas concretas que se describen. En este caso sería necesario asegurar que hay solo una fábrica concreta de cada una de ellas (hay un patrón para ello)

Los productos A incluyen método doitA, los productos B un doitB y los C un doitC.

- a) Haga un diagrama de clases UML que muestre en forma gráfica la solución
- b) Escriba el código de la fábrica abstracta y de las dos fábricas concretas
- c) Escriba una clase Cliente que se inicializa con una fábrica concreta la que utiliza para crear instancias de objetos A, B y C. Cliente contiene además un método demo que invoca doitA, doitB y doitC sobre estos objetos.
- d) Escriba un segmento de código que crea un cliente para luego invocar el método demo, primero con una fábrica local y luego con la fábrica china.



Observaciones:

- Las fábricas concretas son implementadas como singleton y por eso aparece el atributo de clase `instance` y el método de clase `instance`
- Hay una fábrica abstracta de la cual heredan las dos fábricas concretas
- Hay una clase abstracta para cada producto y clases concretas para los productos producidos en las fábricas concretas
- Las líneas punteadas indican dependencia. Así los 3 productos concretos locales todos dependen de la fábrica local y los 3 productos concretos chinos dependen de la fábrica china

```

class AbstractProductFactory
def create_product_A
  puts "You should implement this method in the concrete factory"
end
def create_product_B
  puts "You should implement this method in the concrete factory"
end
def create_product_C
  puts "You should implement this method in the concrete factory"
end
end

class LocalProductFactory < AbstractProductFactory
@@instance = LocalProductFactory.new      #requerido para singleton
private_class_method :new                 #requerido para singleton
def self.instance                         #requerido para singleton
  return @@instance
end
def create_product_A
  LocalProductA.new
end
def create_product_B
  LocalProductB.new
end
def create_product_C
  LocalProductC.new
end
end

class ChinaProductFactory < AbstractProductFactory
@@instance = ChinaProductFactory.new      #requerido para singleton
private_class_method :new                 #requerido para singleton
def self.instance                         #requerido para singleton
  return @@instance
end
def create_product_A
  ChinaProductA.new
end
def create_product_B
  ChinaProductB.new
end
def create_product_C
  ChinaProductC.new
end
end

```

```

class Client
attr_accessor :productFactory
def initialize(productFactory)
  @productFactory = productFactory
  @prod_A = @productFactory.create_product_A
  @prod_B = @productFactory.create_product_B
  @prod_C = @productFactory.create_product_C
end
def demo
  @prod_A.doitA
  @prod_B.doitB
  @prod_C.doitC
end
end

cliente1 = Client.new(LocalProductFactory.instance)
cliente1.demo
#output de doitA de un producto A de la clase LocalProductA
#output de doitB de un producto B de la clase LocalProductB
#output de doitC de un producto C de la clase LocalProductC
cliente2 = Client.new(ChinaProductFactory.instance)
cliente2.demo
#output de doitA de un producto A de la clase ChinaProductA
#output de doitB de un producto B de la clase ChinaProductB
#output de doitC de un producto C de la clase ChinaProductC

```

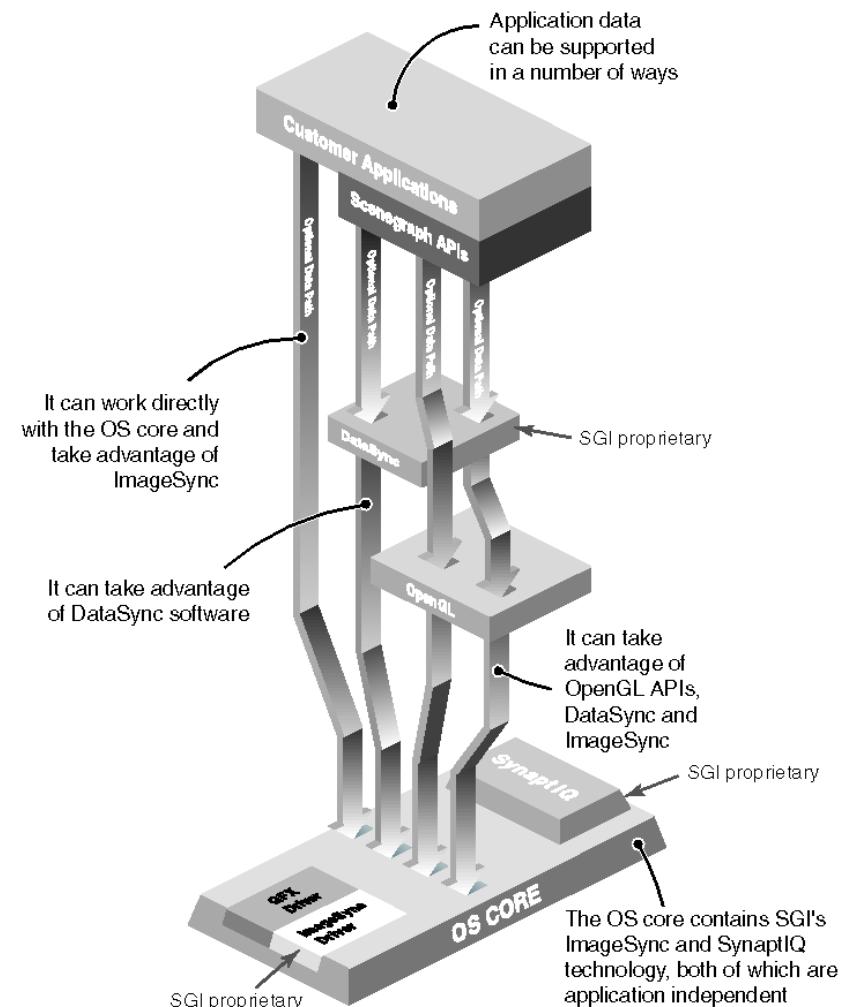
Observaciones:

- nótese que cuando se crea el cliente se le pasa como parámetro una fábrica
- la fábrica no puede ser creada con new porque ese método está deshabilitado, sino con el método de clase instance (singleton)

Arquitectura de Software

¿De qué estamos hablando ?

- estructura: descomposición en componentes y sus interacciones
- grandes decisiones de diseño basadas en requerimientos (incluyendo los no funcionales)



Factores que influencian las decisiones de arquitectura

- performance - minimizar el número de comunicaciones
- security - organización en capas
- availability - redundancia controlada
- maintainability - componentes autocontenidoas fácilmente intercambiables

Arquitectura vs Diseño

- La separación no es tan nítida pero hay aspectos que son claramente de arquitectura
 - la "forma" del sistema: client-server, web-based, native mobile client, etc
 - estructura del software (componentes, capas)
 - tecnologías (lenguaje, plataforma de despliegue)
 - frameworks
 - enfoque de logro de performance, escalabilidad, etc

Arquitectura y Agilidad

- Una buena arquitectura habilita la agilidad
 - por ejemplo arquitectura de microservicios (mas adelante) facilita desarrollo incremental
 - separación clara de componentes facilita los tests
 - se puede agregar funcionalidades con mayor facilidad sobre arquitectura sólida

Architectural Drivers

- requerimientos funcionales
- atributos de calidad (performance, seguridad, etc)
- restricciones
- principios (consistencia y claridad del código)

Atributos de Calidad

- performance (tiempo de respuesta, latencia, etc)
- escalabilidad
- disponibilidad
- seguridad
- continuidad operacional (recuperación)
- accesibilidad (usuarios con necesidades)
- monitoreo

Algunas Restricciones

- Tiempo y presupuesto (las obvias)
- Tecnológicas
 - lista de tecnologías aprobadas
 - sistemas existentes e interoperabilidad
 - plataforma destino
 - open source (yes/no)
 - madurez de tecnologías a incluir
 - relaciones con los proveedores
 - fallas anteriores
- Relativas a personas
 - tamaño del equipo
 - habilidades del equipo
 - se pueden agregar especialistas
 - la mantención la hará el mismo equipo ?

Principios

- De Desarrollo
 - estandares de codificación
 - full unit testing
 - revisión de código
- Arquitectónicos
 - separación de lógica de negocios
 - componentes sin estado
 - consistencia eventual
 - procedimientos almacenados
 - gestión del sistema
 - auditoría flexibilidad
 - extensibilidad
 - mantenibilidad
 - aspectos legales y regulatorios
 - localización e internacionalización

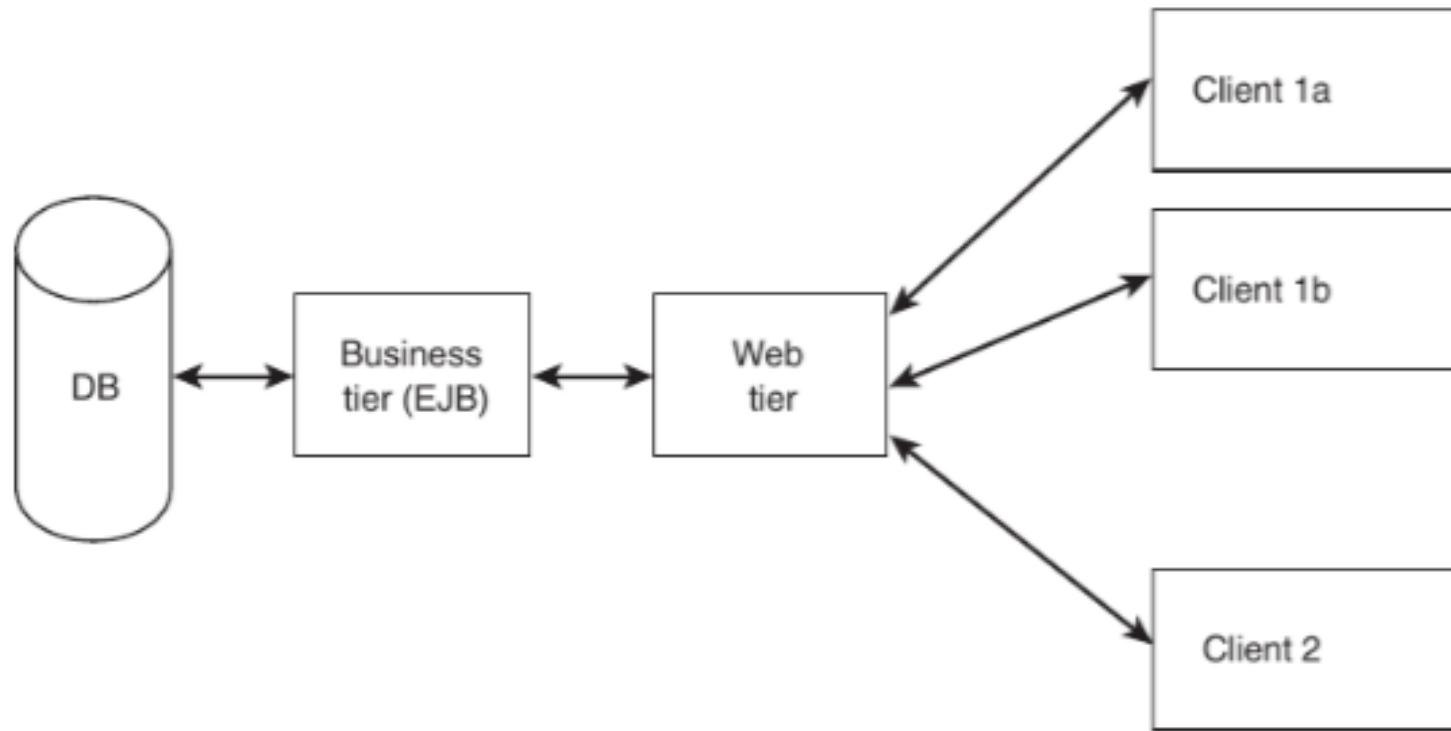
Patrones Arquitectónicos

- Similar a patrones de diseño
- Soluciones que se encuentran a menudo en el mundo real
- Permiten no tener que partir de cero

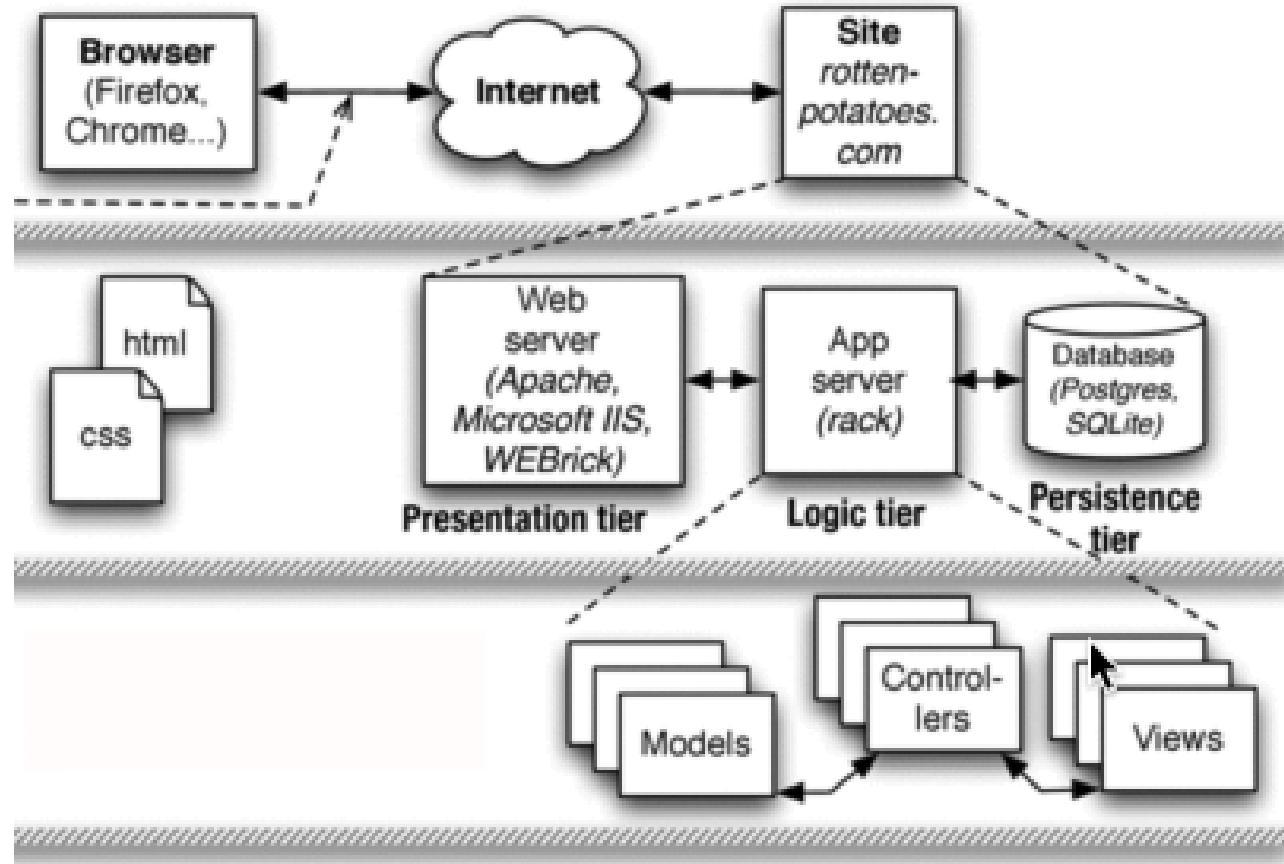
Client-Server y Tiered

- Dos grandes roles: cliente y servidor
- Servidor recibe y procesa solicitudes de cliente
- Tremendamente popular en los 80s y 90s dio origen a una variación conocida como “3 capas” o “thin client”
- Thin client
 - tier 1: server
 - tier 2: application server
 - tier 3: client

Arquitectura de referencia J2EE

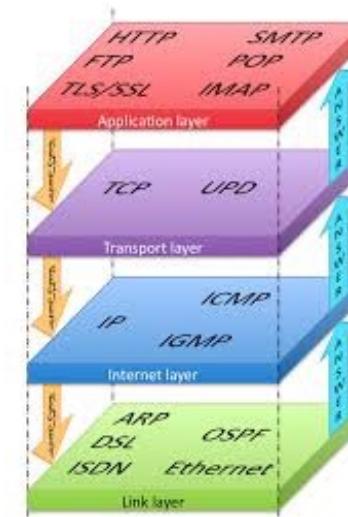
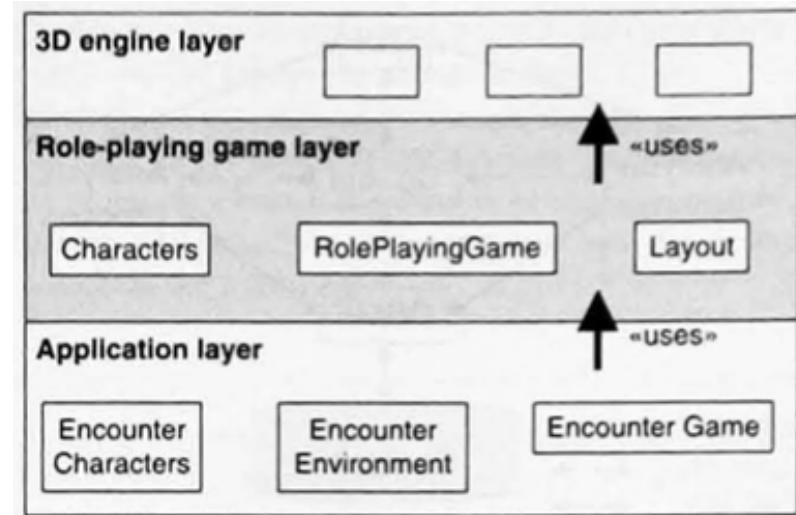


Aplicación Web (Ruby)



Arquitectura de Capas

- una capa es una colección coherente de componentes
- usa y es usada por a lo más una capa

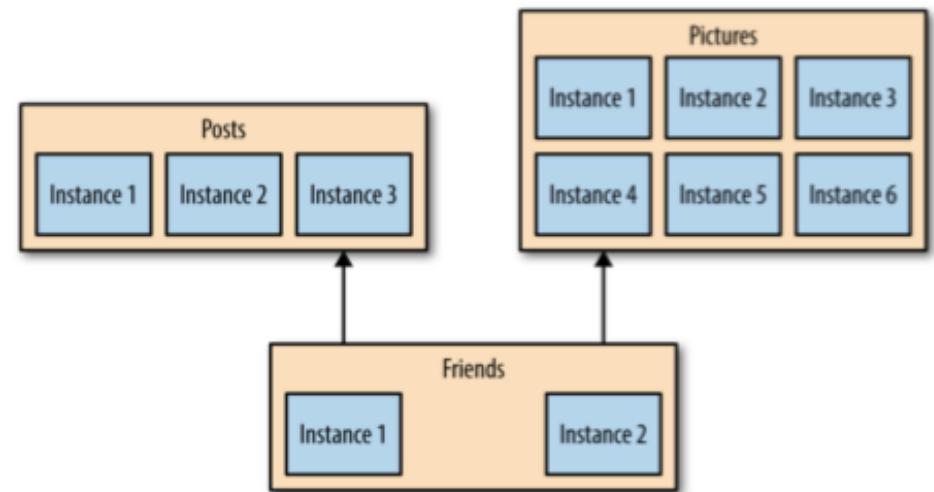
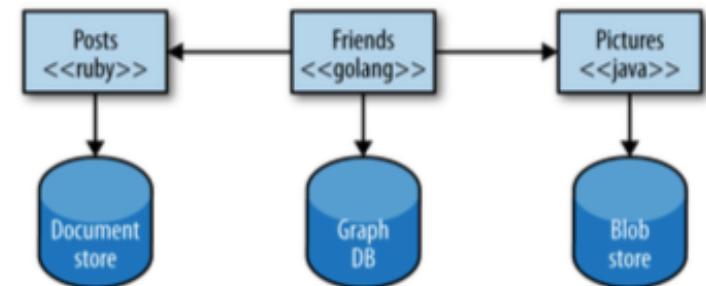


Arquitectura de Servicios: componentes débilmente acopladas

- una componente es una unidad de software que puede ser reemplazada o mejorada (upgrade) en forma independiente
- tradicionalmente las componentes se presentan en forma de módulos o librerías compartidas
- un servicio es una componente que no corre en el mismo proceso y se comunica con el resto mediante un protocolo como http o rpc

Ventajas

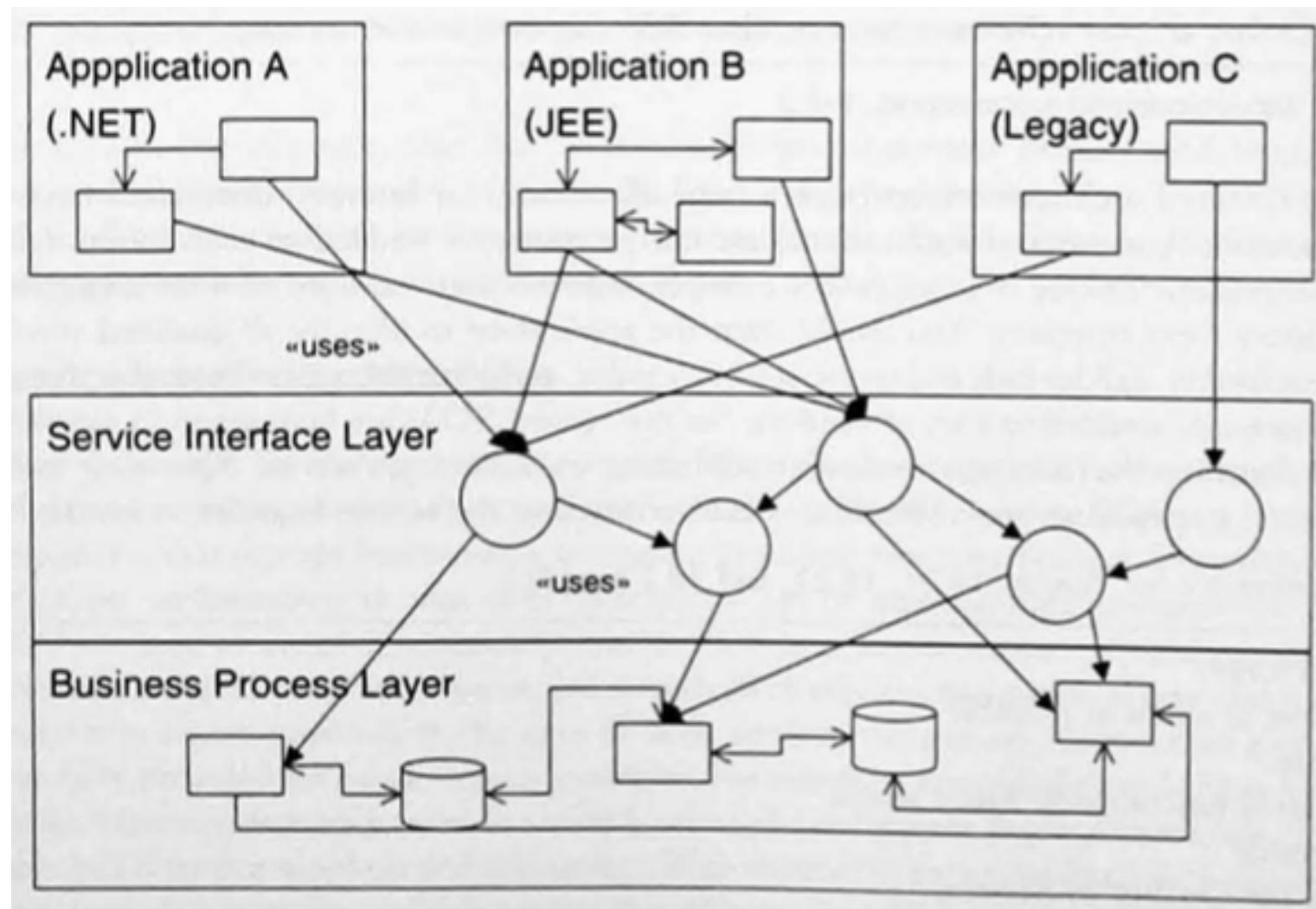
- deployable en forma independiente
 - si se hace un cambio de una componente no es necesario redeployar la aplicación
- interfaz explícita
 - hace más difícil violar acuerdos
- heterogeneidad tecnológica
- resiliencia
- escalamiento flexible



Service Oriented Architecture (SOA)

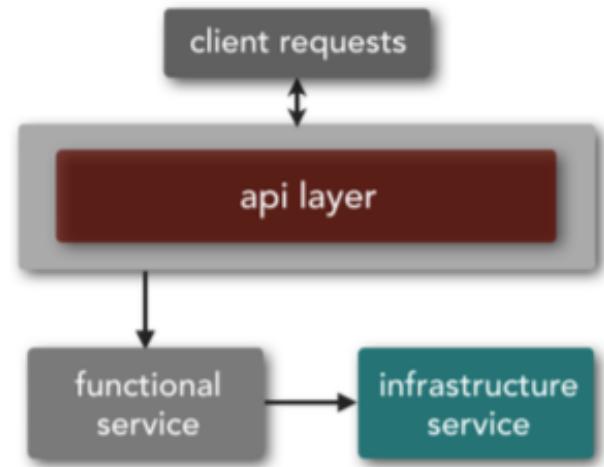
- mediados de los 2000
- Sistema se concibe como una combinación de servicios
- Servicios proveen funcionalidades de acuerdo a una especificación de interfaz
- Pueden ser combinados en forma dinámica
- Intimamente ligado a la popularidad de los Web Services y a la formalización de los procesos de negocio

SOA



La llegada de los microservicios

- 2011 - 2012
- algunos se refieren a "SOA hecho bien"
- primera generación de SOA comenzó a hacerse demasiado compleja
 - SOAP -> middleware
- representa una maduración de las ideas de SOA a la luz de la experiencia práctica
- no requiere capa de middleware
 - cada servicio expone una API (contrato)
- la mayor parte son servicios funcionales con algunos servicios de infraestructura (no se exponen al mundo externo)



Microservicios

- Una aplicación se construye en base a un conjunto de pequeños servicios (HTTP, Rest APIs)
- Servicios pueden escribirse en lenguajes distintos y usar distintos almacenamientos
- Rompe en pequeños servicios la componente que usualmente corresponde al server
- En lugar de tener que replicar la aplicación para que escale se replican los servicios que lo requieran

Características de Arquitectura de Microservicios

- Componentes son Servicios
- Altamente cohesivas y desacopladas (smart endpoints, dumb pipes)
- Usan principios y protocolos de la Web (Http, Rest)
- Organización en base a capacidades de negocio y no de especializaciones tecnológicas (UI, DBA, middleware)
- Gobierno descentralizado (un servicio es totalmente independiente)
- Manejo de datos descentralizado