



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 12

Modelación

IIC2143 - Ingeniería de Software
Sección 1

Rodrigo Saffie

rasaffie@uc.cl

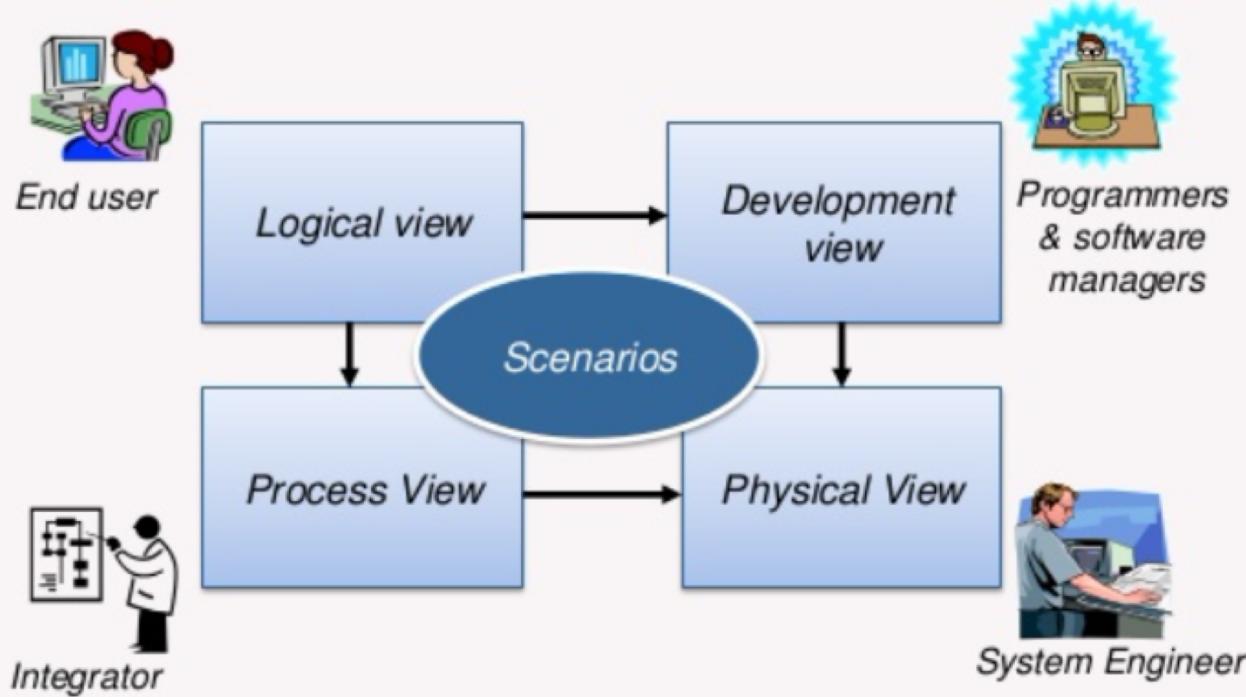
18 de abril de 2018

Modelo 4 + 1

- Propuesto por Philippe Kruchten en 1995
- *Framework* para describir la arquitectura de un *software*, basado en el uso de múltiples vistas concurrentes.

Modelo 4 + 1

The 4+1 View model



- Describes software architecture using five concurrent views.

Vista lógica

- Interesados: Usuarios finales
- Consideraciones: Requisitos funcionales
 - ¿Qué es lo que el sistema debería proveer a sus usuarios?
- Diagramas:
 - Clases
 - Comunicación
 - Secuencia
 - Estado

Vista de procesos

- Interesados: Analistas
- Consideraciones: Requisitos no funcionales (conurrencia, rendimiento, escalabilidad)
 - ¿Qué procesos y reglas tiene el sistema, y cómo interactúan los componentes?
- Diagramas:
 - Actividad

Vista de implementación

- Interesados: Desarrolladores y líderes de proyectos
- Consideraciones: Organización del *software*
 - ¿Cómo se organiza el *software* del sistema?
- Diagramas:
 - Componentes
 - Paquetes

Vista física

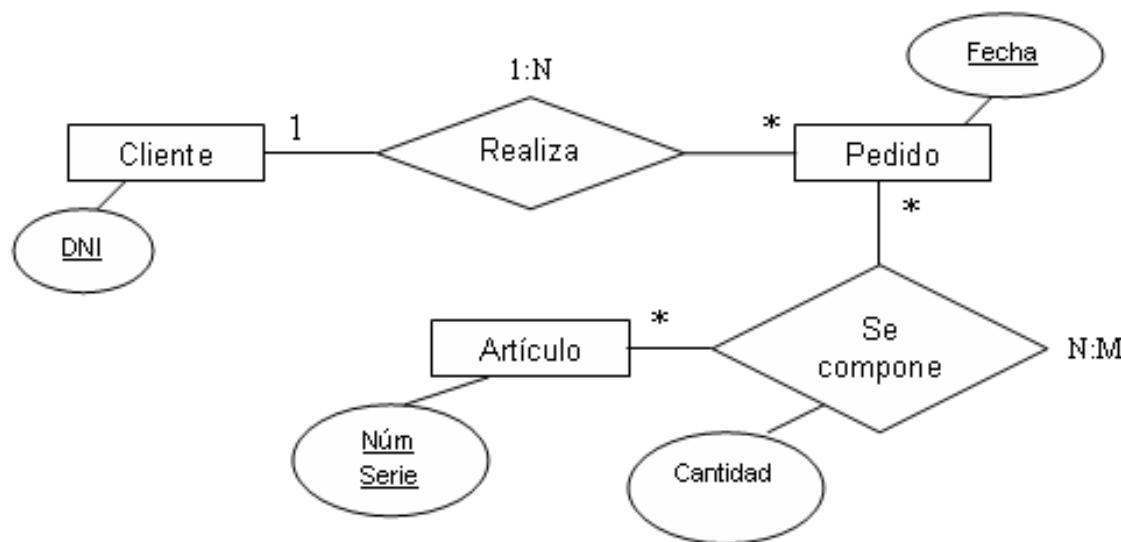
- Interesados: Arquitectos de sistemas
- Consideraciones: Requisitos no funcionales
 - ¿Cómo es el ambiente de ejecución del sistema?
- Diagramas:
 - Despliegue

Vista de escenarios

- Interesados: Todos
- Consideraciones: Consistencia, validez
 - ¿Qué es lo que el sistema debería hacer?
- Diagramas:
 - Casos de uso

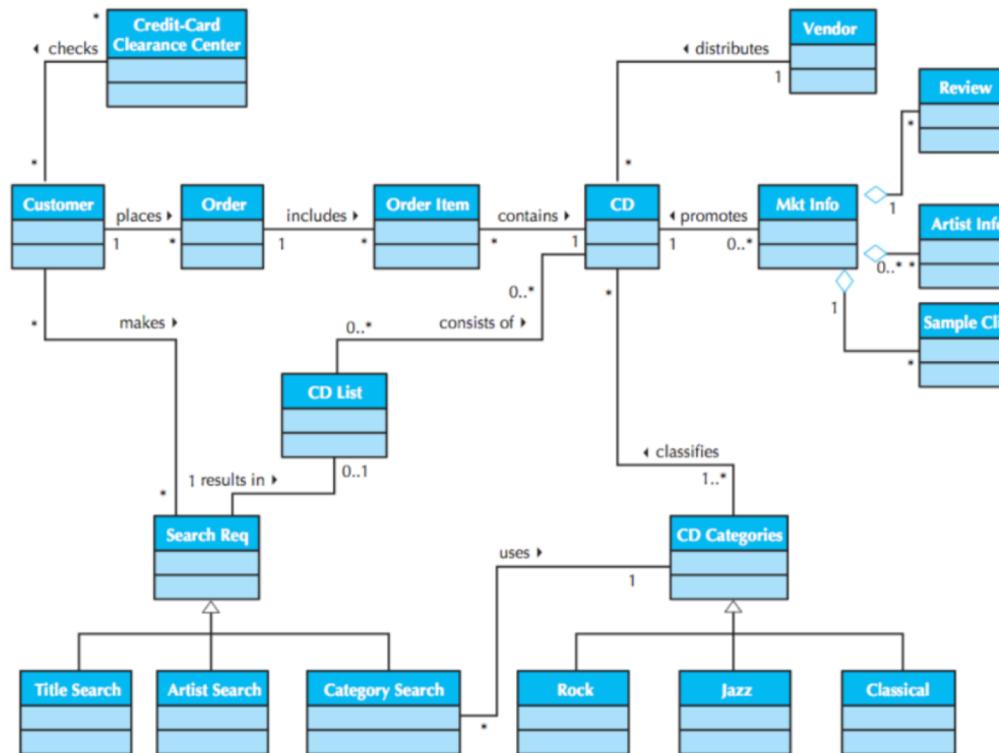
Modelo Entidad-Relación

- Herramienta para modelar datos persistentes de un sistema, junto con sus relaciones



Modelo de Dominio

- Similar a un modelo de datos Entidad-Relación (E-R), pero no necesariamente los objetos son persistentes



Modelo de Dominio

- Buscar candidatos a objetos en requisitos, casos de uso e historias de usuario
- Deben cumplir con:
 - Almacenar algún dato
 - Todas las instancias contienen mismos atributos y métodos
 - ¿más de 1 método o atributo?

Modelo de Dominio - Consejos

- Definir clases para la solución no es simple
- Puede ser un proceso iterativo
- Detectar responsabilidades más que atributos o operaciones
- Si una clase no tiene métodos o atributos probablemente no es una clase

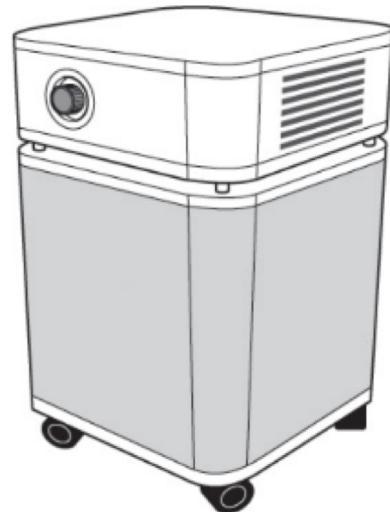
Unified Modeling Language 2.0

- UML: estándar para especificar un sistema
- 1º versión propuesta en 1997
- Ayuda a entender y explicar un sistema de manera consistente
- *The Elements of UML 2.0 Style*

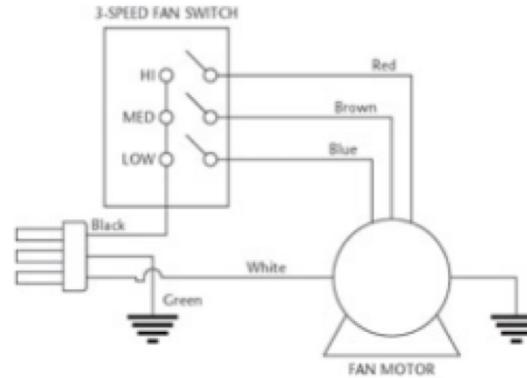
Unified Modeling Language 2.0

Lenguaje visual para describir artefactos de software

artefacto



lenguaje visual



Unified Modeling Language 2.0

```
public class Payment {  
    private float amount;  
    public Payment (float cashTendered){this.amount = cashTendered;}  
    public float getAmount() { return amount; }  
}  
  
public class ProductCatalog{  
    private HashTable productSpecifications = new Hashtable();  
    public ProductCatalog(){  
        ProductSpecification ps = new ProductSpecification(100, 1, "product 1");  
        productSpecifications.put (new Integer(100), ps);  
        ps = new ProductSpecification(200, 1, "product 2");  
        productSpecifications.put (new Integer(200), ps);  
    }  
  
    public ProductSpecification getSpecification (int upc){  
        return (ProductSpecification)productSpecifications.get(new Integer(upc));  
    }  
}  
  
class POST {  
    private ProductCatalog productCatalog;  
    private Sale sale;  
    public POST(ProductCatalog catalog){productCatalog = catalog;}  
    public void endSale() {sale.becomeComplete();}  
    public void enterItem(int upc, int quantity){  
        if (isNewSale() ) sale = new Sale();  
        ProductSpecification spec = productCatalog.specification(upc);  
        sale.makeLineItem(spec, quantity);  
    }  
    public void makePayment(float cashTendered){  
        sale.makePayment(cashTendered);  
    }  
    private boolean isNewSale(){return (sale == null) ||(sale.isComplete() );}  
}  
  
public class ProductSpecification{  
    private int upc = 0;  
    private float price = 0;  
    private String description = "";  
    public ProductSpecification(int upc, float price, String description){  
        this.upc = upc;  
        this.price = price;  
        this.description = description;  
    }  
    public int getUPC {return upc; }  
    public float getPrice() {return price; }  
    public String getDescription() {return description; }  
}  
  
class Sale {  
    private Vector lineItems = new Vector();  
    private Date date = new Date();  
    private boolean isComplete = false;  
    private Payment payment;  
    public float getBalance () {return payment.getAmount() - total(); }  
    public void becomeComplete() {isComplete = true; }  
    public boolean isComplete() {return isComplete; }  
    public void makeLineItem(ProductSpecification spec, int quantity) {  
        lineItems.addElement(new SaleLineItem(spec, quantity));  
    }  
    public float total(){  
        float total = 0;  
        Enumeration e = lineItems.elements();  
        while(e.hasMoreElements() ){  
            total += ( (SaleItem) e.nextElement() ).subtotal();  
        }  
        return total;  
    }  
    public void makePayment (float cashTendered) {  
        payment = new Payment (cashTendered);  
    }  
}  
  
class SaleLineItem{  
    private int quantity;  
    private ProductSpecification productSpec;  
    public SaleLineItem (ProductSpecification spec, int quantity) {  
        this.productSpec = spec;  
        this.quantity = quantity;  
    }  
    public float subtotal() { return quantity* productSpec.getPrice(); }  
}  
  
class Store {  
    private ProductCatalog productCatalog = new ProductCatalog();  
    private POST post = new POST (productCatalog);  
    public POST getPOST() {return post; }  
}
```

Artefacto

Unified Modeling Language 2.0

Representación Visual del Artefacto

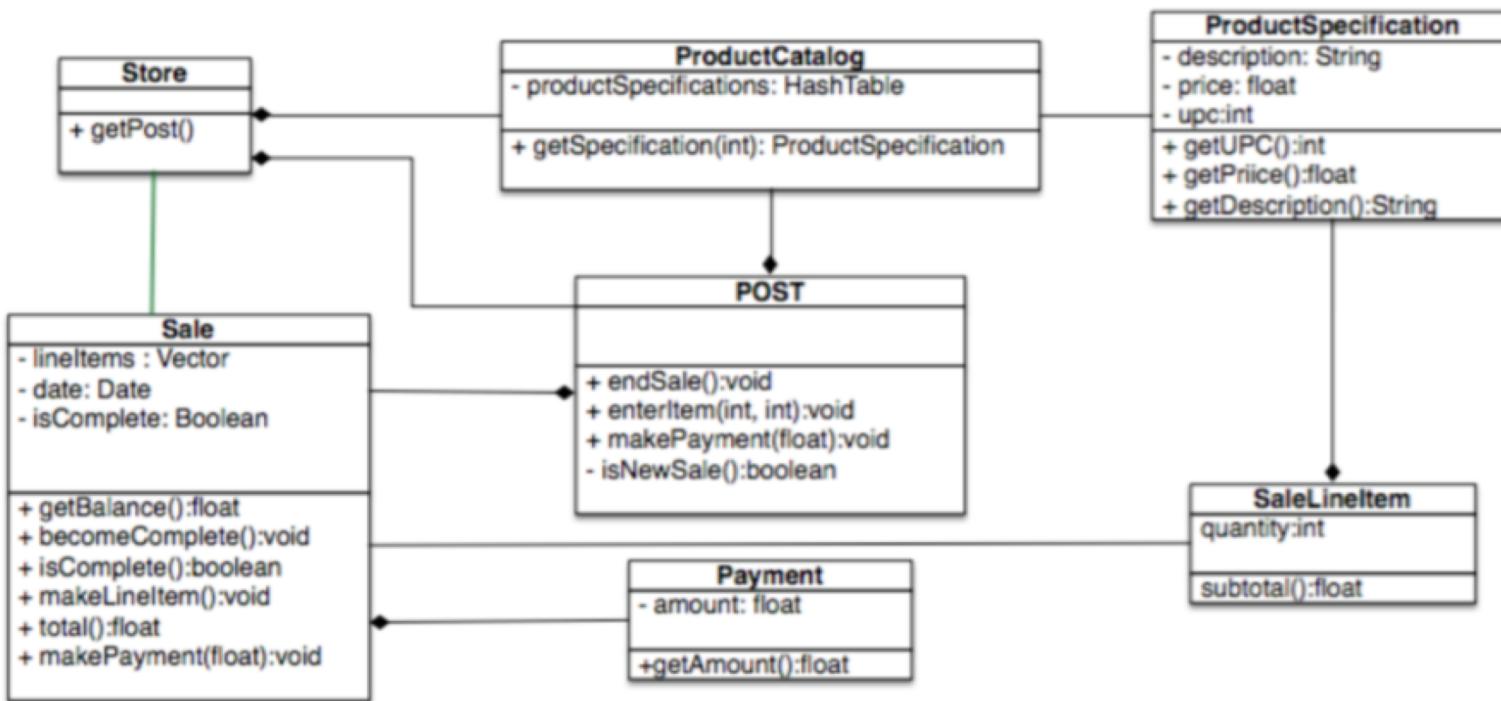
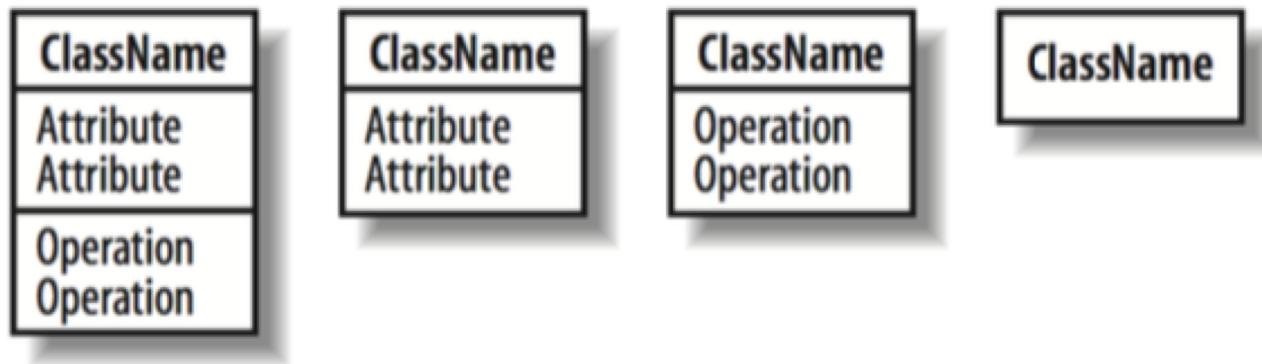


Diagrama de clases

- Representa las clases de un sistema, junto con sus atributos, operaciones y relaciones.
- Sirve para:
 - Traducir el modelo de dominio en una implementación de *software*
 - Representar el diseño de un *software* orientado a objetos

Diagrama de clases - Elementos

- Rectángulos para representar clases



- Líneas para representar asociaciones entre clases

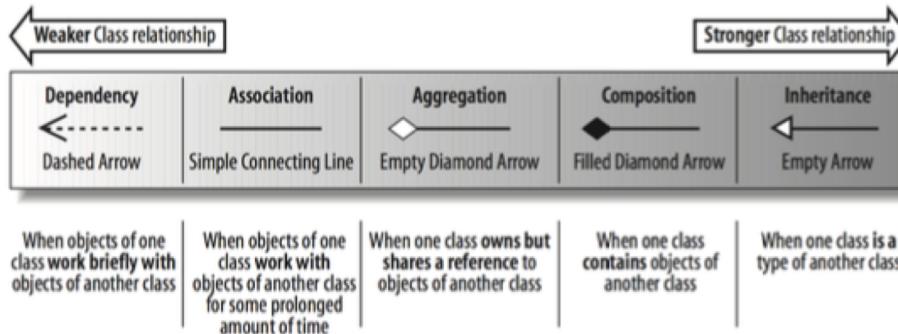


Diagrama de clases - Visibilidad

- Qué elementos de una clase son visibles para instancias de otra clases
- Existen 4 tipos de visibilidad:
 - público (+): accesible por todos
 - protegido (#): accesible solamente a descendientes
 - privado (-): solo para uso interno
 - *package* (~): accesible a clases del mismo *package*

Diagrama de clases - Atributos

- Nombre y tipo
- Pueden ponerse como parte de la asociación

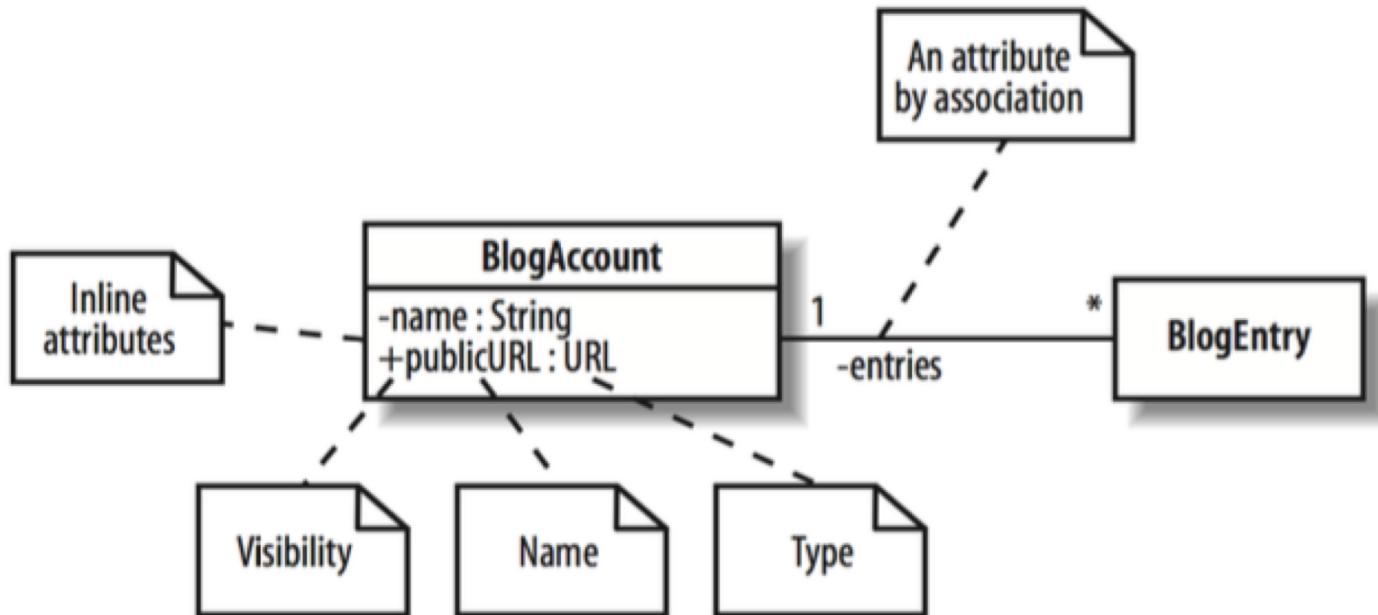


Diagrama de clases - Operaciones

- Nombre, parámetros y retorno
-

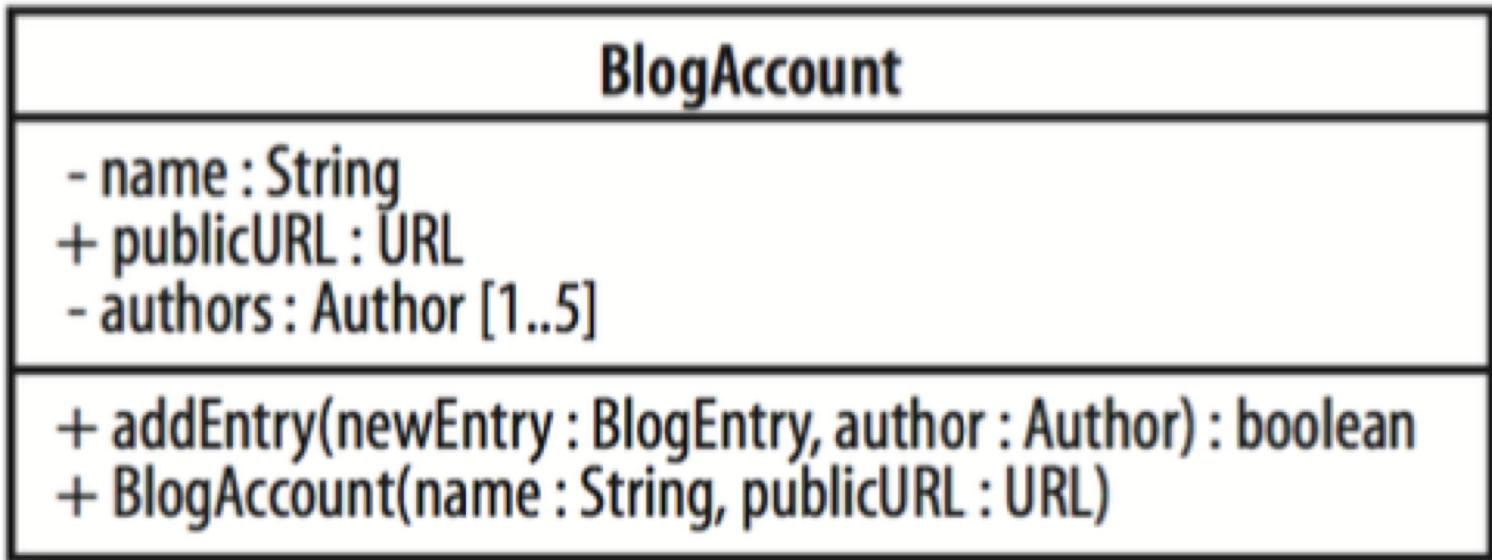


Diagrama de clases - Relaciones

- Dependencia: una clase necesita usar objetos de otra clase

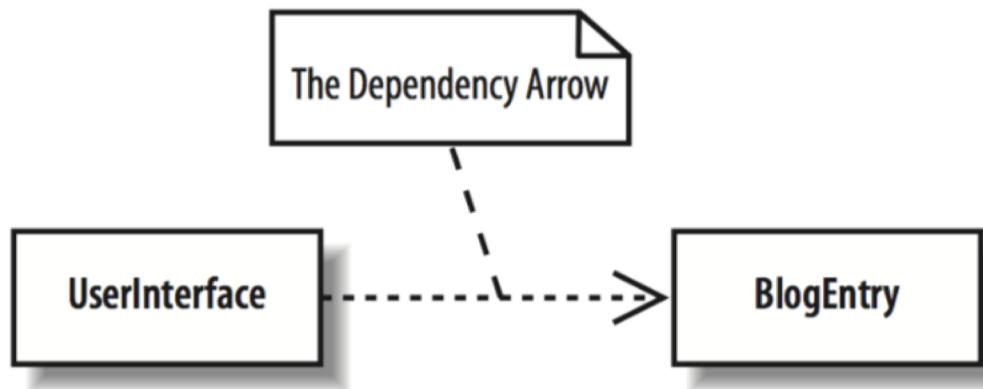


Diagrama de clases - Relaciones

- Asociación: una clase contiene una referencia a un objeto de otra clase
- Puede agregarse flecha para indicar dirección de navegabilidad

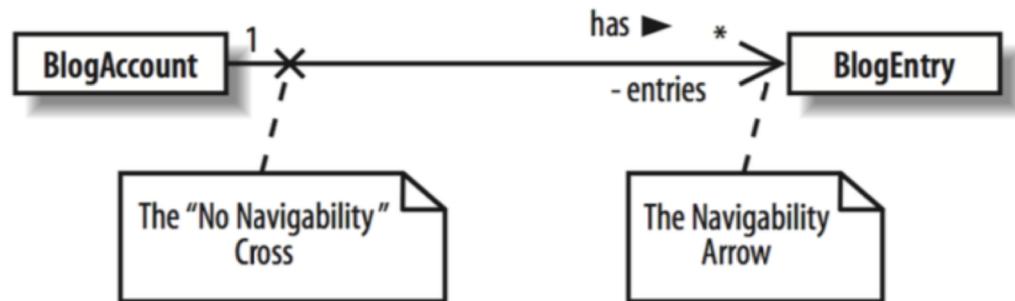
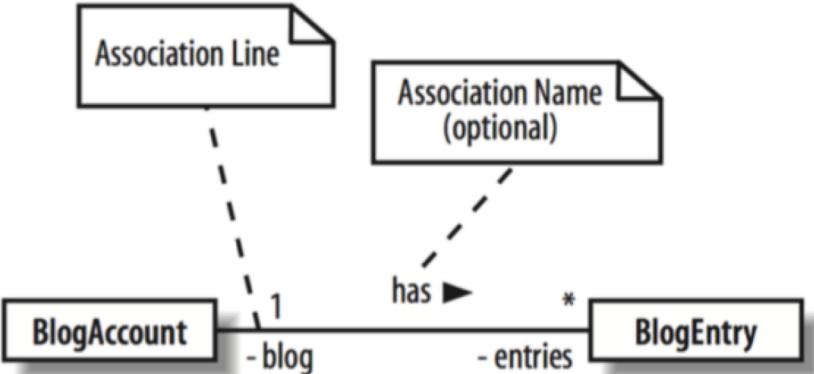


Diagrama de clases - Relaciones

- Agregación: una clase es dueña de objetos de otras clases

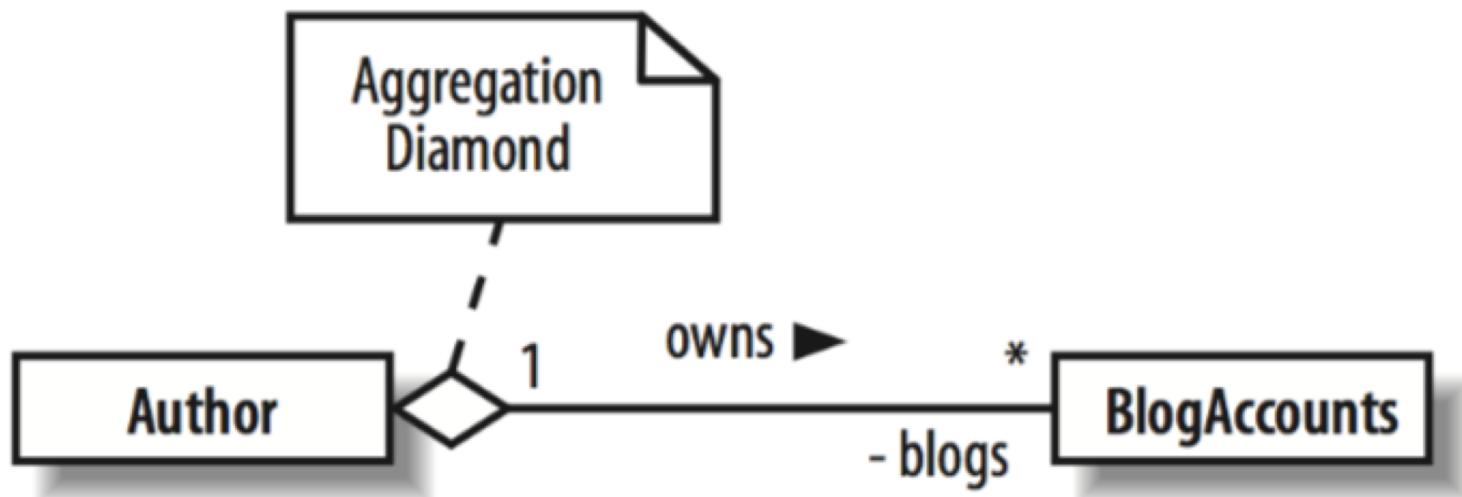


Diagrama de clases - Relaciones

- Composición: una clase es definida por objetos de otras clases

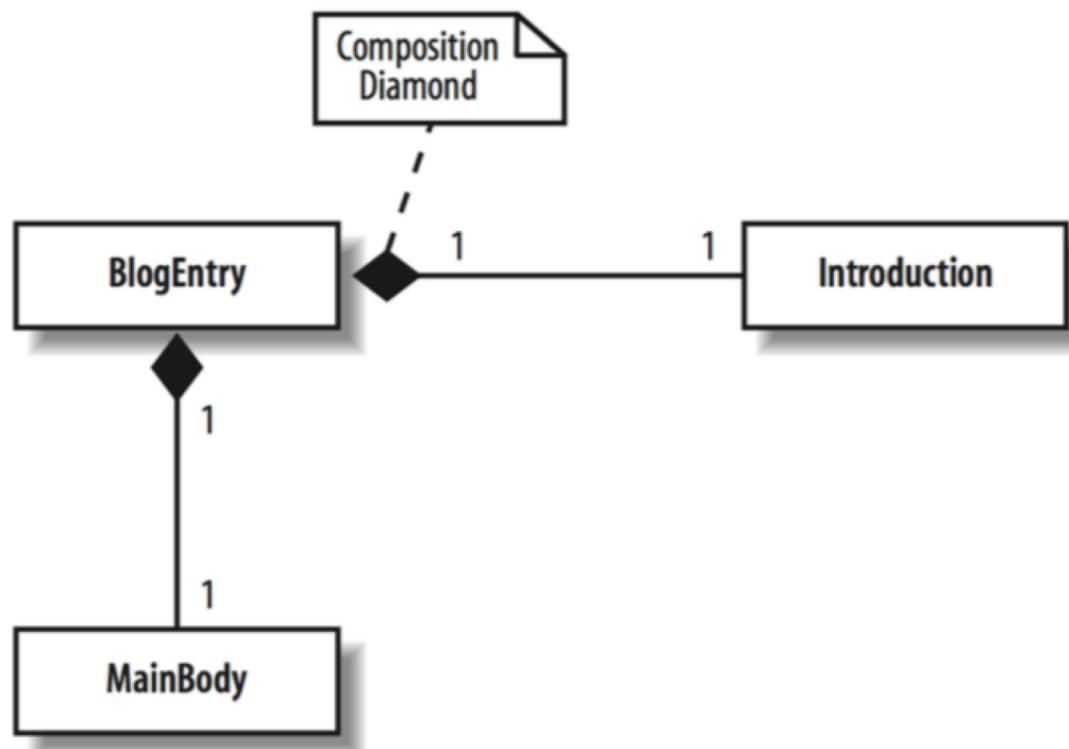


Diagrama de clases - Relaciones

- Generalización: una clase es una especificación de otra
- También se conoce como “herencia”

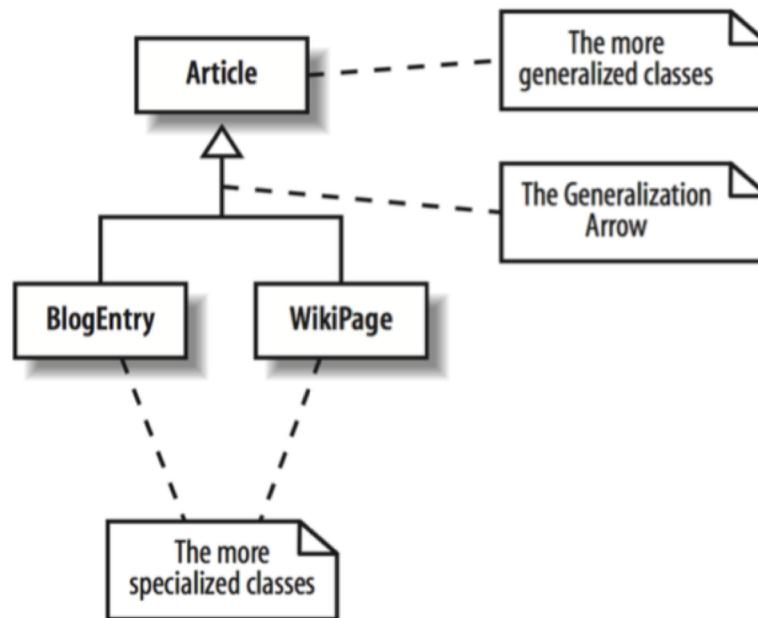
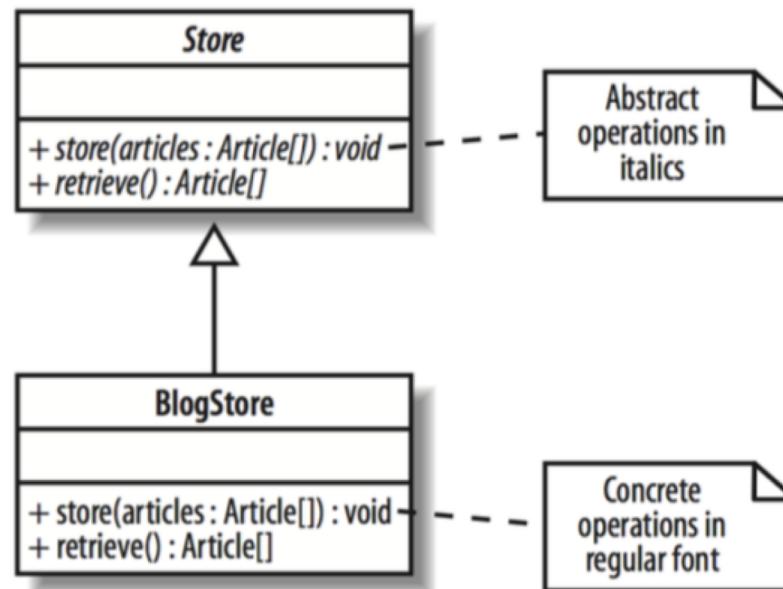


Diagrama de clases - Relaciones

- Generalización: una clase es una especificación de otra
- Pueden ser clases abstractas



Interfaces

- Colección de operaciones (métodos) que las clases concretas deben implementar
- Se pueden interpretar como un contrato
- Facilitan la extensibilidad del código



Stereotype Notation

Or



"Ball" Notation

Clases abstractas e interfaces en Ruby

```
class FunctionalInterface
  def do_thing
    raise "This is not implemented!"
  end
end

class Herp < FunctionalInterface
  def do_thing
    puts "herp"
  end
end

class Derp < FunctionalInterface
  def do_thing
    puts "derp"
  end
end

class Blorp < FunctionalInterface
  def do_some_other_thing
    puts "whoops"
  end
end
```

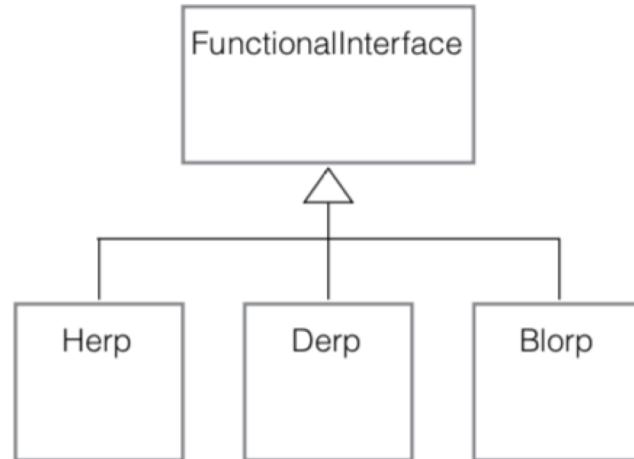


Diagrama de clases - Consejos

- Utilizar terminología basada en el dominio del negocio
- No utilizar abreviaturas: ser descriptivo con los nombres
- Representar características propias del problema, no código autogenerado

Actividad: Reserva aviones

Considera un sistema de reserva de vuelos con las siguientes características:

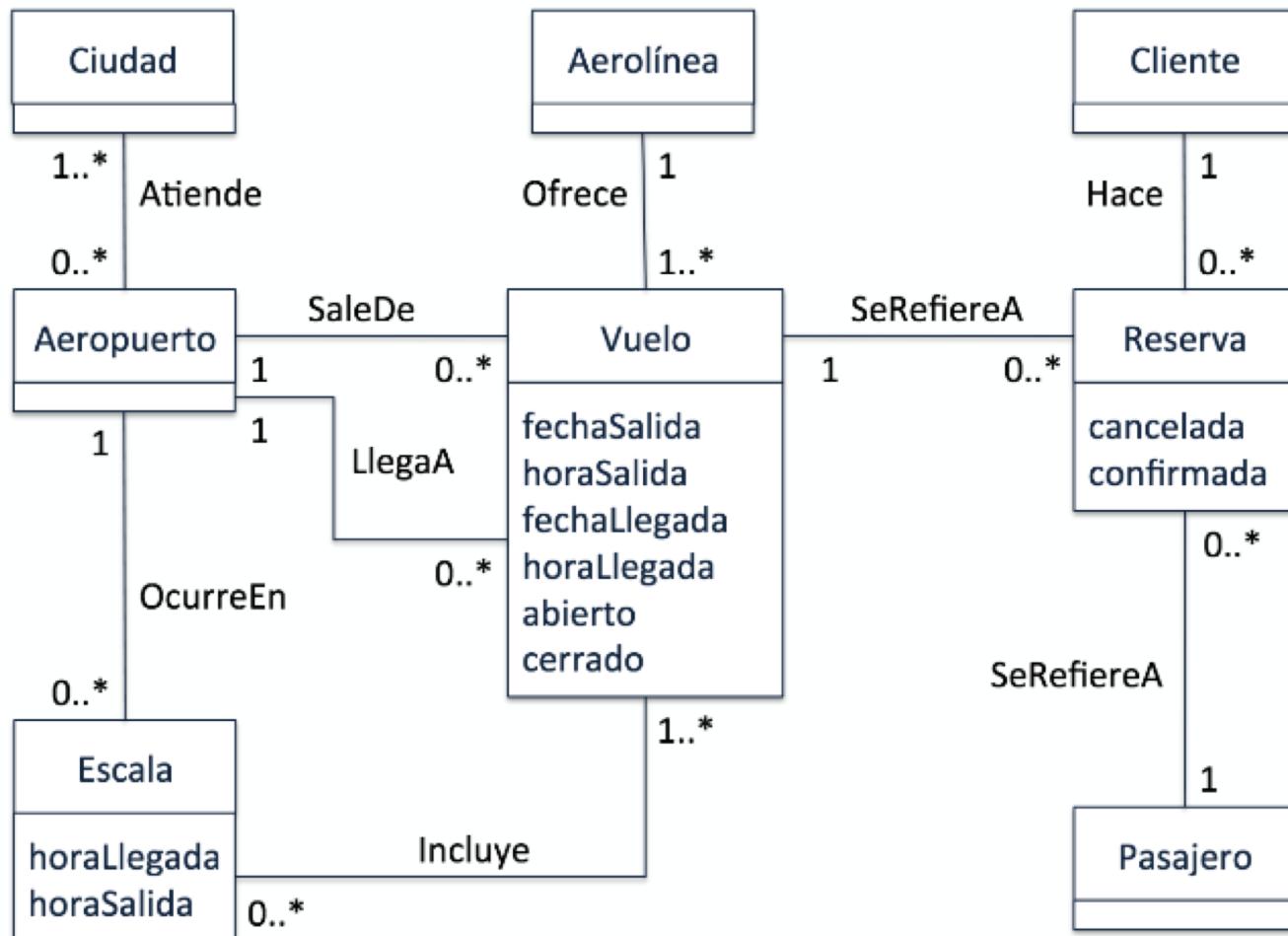
Las líneas aéreas ofrecen varios vuelos. Un vuelo es abierto para hacer reservas y nuevamente cerrado por orden de la compañía. Un cliente puede reservar uno o más vuelos y para diferentes pasajeros. Una reserva se refiere a un único vuelo y a un único pasajero, y puede ser cancelada o confirmada. Los vuelos tienen aeropuerto, fecha y hora de salida y de llegada, y pueden incluir escalas en aeropuertos. Cada escala tiene una hora de llegada y una hora de salida. Cada aeropuerto atiende a una o más ciudades.

Actividad: Reserva aviones

Entidades:

- Aerolíneas
- Vuelos
- Reservas
- Aeropuertos
- Pasajeros
- Clientes
- Ciudades

Actividad: Reserva aviones





Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 12

Modelación

IIC2143 - Ingeniería de Software
Sección 1

Rodrigo Saffie

rasaffie@uc.cl

18 de abril de 2018