Implementación de modelos de dominio, de UML a Java

Repositorios de Información

Por partes…

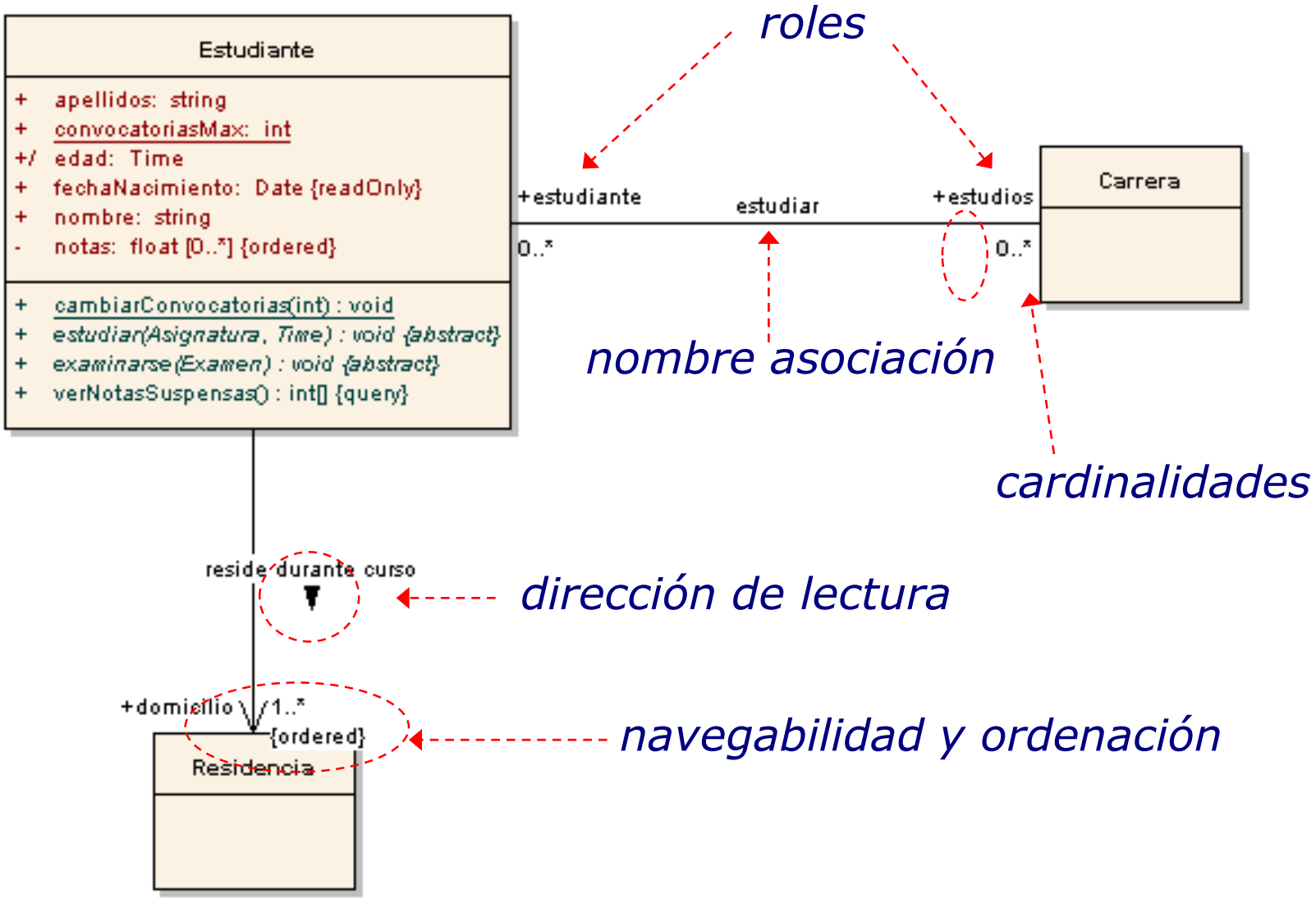
De UML a Java

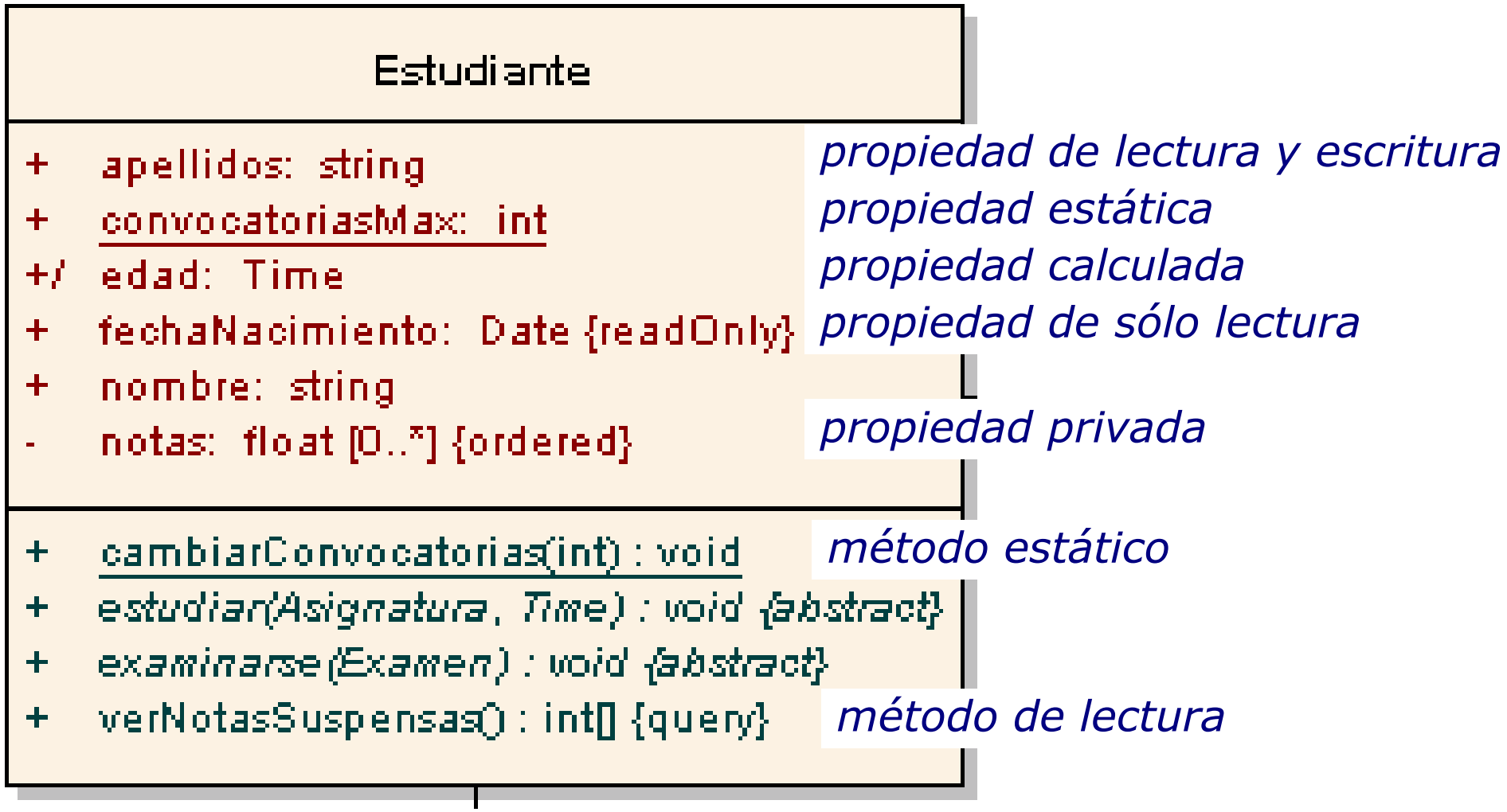
- Cosas básicas…

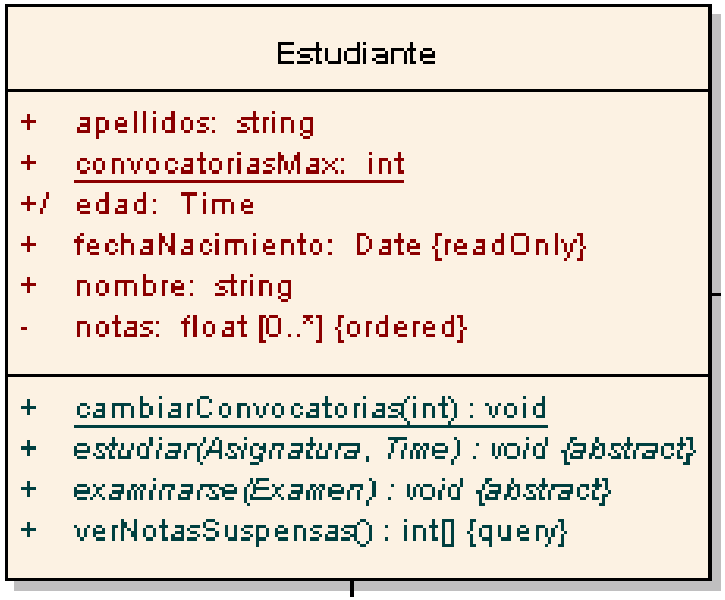
Un modelo de domino a Java

- Sus detalles…

Clases y Asociaciones




Estudiante
+ apellidos: string
+ convocatoriasMax: int
+/ edad: Time
+ fechaNacimiento: Date {readOnly}
+ nombre: string
- notas: float [0..*] {ordered}
+ cambiarConvocatorias(int) : void
+ estudiar(Asignatura, Time) : void {abstract}
+ examinarse(Examen) : void {abstract}
+ verNotasSuspensas() : int[] {query}
propiedad de lectura y escritura
propiedad estática
propiedad calculada
propiedad de sólo lectura
propiedad privada
método estático
método de lectura




```
public class Estudiante {

    private static int convocatoriasMax;
    private String apellidos;
    private String nombre;
    private Date fechaNacimiento;
    private float notas[];
    ...
```

propiedad estática

```
public class Estudiante {
    ...
    private static int convocatoriasMax;
    ...
    public static int getConvocatoriasMax() {
        return convocatoriasMax;
    }
    public static void setConvocatoriasMax(int convocatoriasMax) {
        Estudiante.convocatoriasMax = convocatoriasMax;
    }
    ...
```

propiedad de lectura y escritura

```
public class Estudiante {
    ...
    private String apellidos;
    ...
    public String getApellidos() {
        return apellidos;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
    ...
```

propiedad sólo lectura

```
public class Estudiante {
    ...
    private Date fechaNacimiento;
    ...
    public Date getFechaNacimiento() {
        return fechaNacimiento;
    }
    // no tiene setter
    ...
```

propiedad calculada

```
public class Estudiante {
    private Date nacimiento;
    ...

    public Time getEdad() {
        return DateUtil.today().substract( nacimiento ).asTime();
    }
```

propiedad privada

```
public class Estudiante {
    ...
    private float notas[];
    ...
    // sin getters ni setters, manipulada internamente
    ...
```

Cuidado con la encapsulación

Atributos de tipos mutables pueden romper la encapsulación

```
public class Item {
    private String name;
    . . .
    private Date endDate;
    . . .
    private Set<Image> images = new HashSet<Image>();

    public Date getEndDate() {
        return endDate;
    }
    public String getName() {
        return name;
    }
    public Set<Image> getImages() {
        return images;
    }
}
```

Peligro!!!

java.util.Date es mutable

Seguro, String es inmutable

Peligro!!!

El que recibe esta colección le puede añadir o quitar elementos descontroladamente

Atributos mutables pueden romper la encapsulación

```
Item i = ...
print( i.getEndDate() ); // shows 12/12/2012

Date date = i.getEndDate();
date.setDay(25);

print( i.getEndDate() ); // shows 25/12/2012
```

Posibilidades

- *Hacer tipos inmutables* — *Sin setters*, *Valores por constructor*
- *Devolver copias*

```
public Date getEndDate() {
    return endDate.clone();
}
public Date getEndDate() {
    return new Date( this.fecha.getTime() );
}
```

```
public Set<Image> getImages() {
    return new HashSet<>( images );
}
```

```
public Set<Image> getImages() {
    return Collections.unmodifiableSet(images);
}
```

Atributos mutables pueden romper la encapsulación

```
public class WorkOrder {

    private Date date;
    ...

    public WorkOrder(Vehicle v) {
        ...
        this.date = new Date();
    }

    public WorkOrder(Vehicle v, Date date) {
        this( v );
        this.date = new Date( date.getTime() ); // defensive copy
    }

    public Date getFecha() {
        return new Date( this.date.getTime() ); // defensive copy
    }
```

Con los atributos privados que no son inmutables deben hacerse copias defensivas

Tanto en constructores, como setters y getters

Constructores

Declaración + inicialización siempre que sea posible

```
public class Invoice {
    private Long number;
    private LocalDate date;
    private double amount;
    private double vat;

    private InvoiceState state = InvoiceState.NOT_YET_PAID;
    private Set<WorkOrder> workOrders = new HashSet<>();
    private Set<Charge> charges = new HashSet<>();
```

```
public Invoice(Long number) {
    this.state = InvoiceState.NOT_YET_PAID;
    this.workOrders = new HashSet<>();
    this.charges = new HashSet<>();
    ...

}
```

*Inicializa siempre los atributos en la declaración si sus **valores no dependen** de parámetros del constructor*

Constructores

Añade siempre validación de parámetros

```
public Invoice(Long number, LocalDate date, List<WorkOrder> workOrders) {
    ArgumentChecks.isNotNull( number );
    ArgumentChecks.isNotNull( workOrders );
    ArgumentChecks.isNotNull( date );
    ArgumentChecks.isTrue( number >= 0 );

    this.number = number;
    this.date = date;
    for (WorkOrder a : workOrders) {
        addWorkOrder(a);
    }
}
```

Constructores

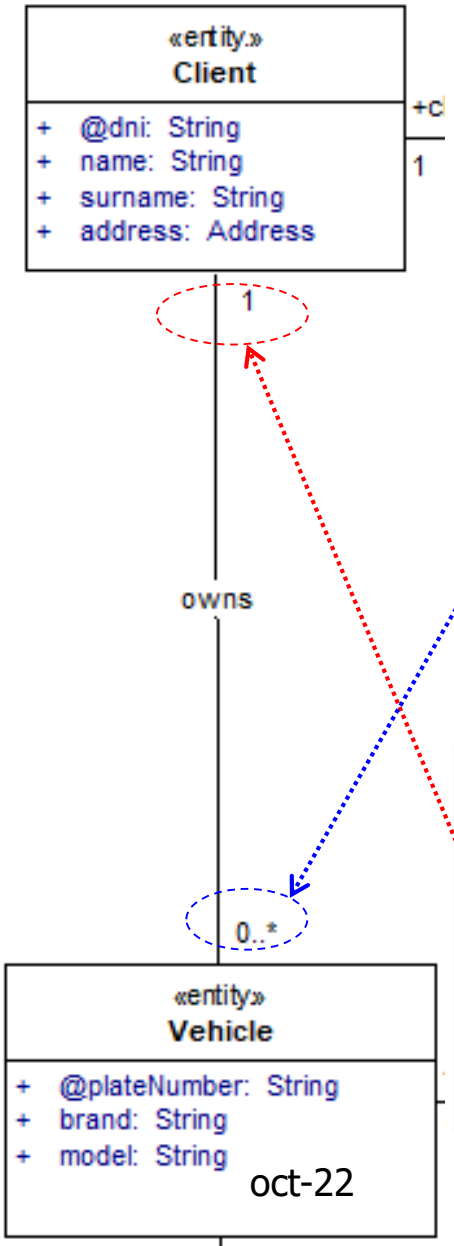
Si hay varios constructores ***Nunca repitas código***
Pon un constructor con todos los parámetros posibles y pasa valores por defecto a la llamada desde los demás

```
public Invoice(Long number) {
    this(number, LocalDate.now(), List.of());
}

public Invoice(Long number, LocalDate date) {
    this(number, date, List.of());
}

public Invoice(Long number, List<WorkOrder> workOrders) {
    this(number, LocalDate.now(), workOrders);
}

public Invoice(Long number, LocalDate date, List<WorkOrder> workOrders) {
    ArgumentChecks.isNotNull( number );
    ArgumentChecks.isNotNull( workOrders );
    ArgumentChecks.isNotNull( date );
    ArgumentChecks.isTrue( number >= 0 );

    this.number = number;
    this.date = date;
    for (WorkOrder a : workOrders) {
        addWorkOrder(a);
    }
}
```

Pasa valores por defecto en la llamada al constructor completo

Constructor completo

Implementación de asociaciones



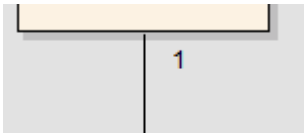

```java
public class Client {
    private String dni;
    private String name;
    private String surname;
    private String email;
    private String phone;
    private Address address;

    private Set<PaymentMean> paymentMeans = new HashSet<PaymentMean>();
    private Set<Vehicle> vehicles = new HashSet<Vehicle>();
```

```java
public class Vehicle {
    private String plateNumber;
    private String make;
    private String model;

    private Client client;
    private VehicleType vehicleType;
    private Set<WorkOrder> workOrders = new HashSet<WorkOrder>();
```

Cardinalidades

- *Uno a uno*
- *Uno a muchos*
- *Muchos a muchos*

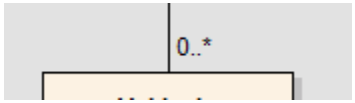
Extremos UNO

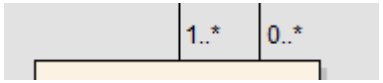


Si el diagrama no especifica cardinalidad se suele interpretar como UNO

```
public class Averia  {
    ...
    private Factura factura;
    ...
```

Extremos MUCHO





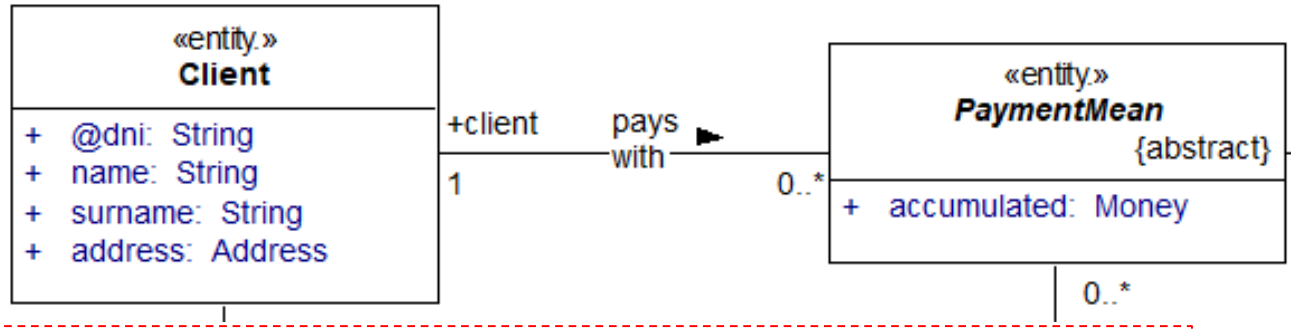
- *Set → 99%*
- *List*
- *Collection*

```
public class Factura {
    ...
    private Set<Averia> averias = new HashSet<Averia>();
    ...
```

Navegabilidad

- *Bidireccional*
- *Unidireccional*

Bidireccional




```
public class Client {
    ...
    private Set<PaymentMean> paymentMeans = new HashSet<>();
    ...
}
```

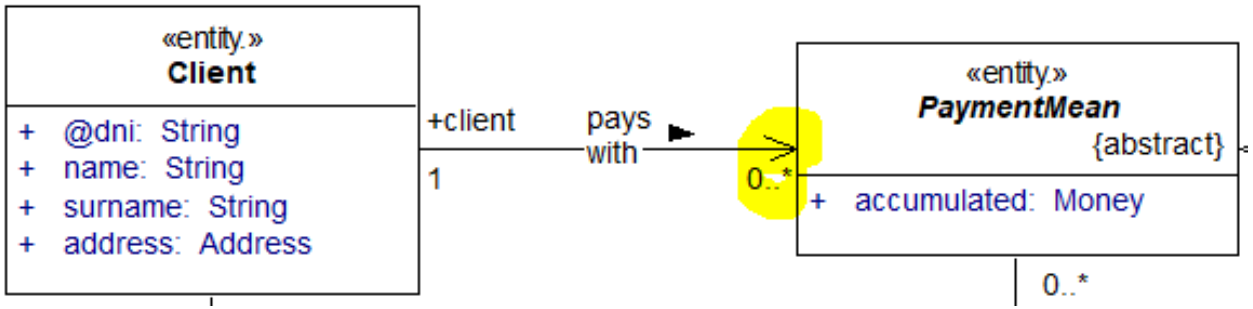
```
public abstract class PaymentMean {
    ...
    private Client client;
    ...
}
```

Referencia cruzadas

Navegabilidad

- *Bidireccional*
- *Unidireccional*

Unidireccional




```
public class Client {
    ...
    private Set<PaymentMean> paymentMeans = new HashSet<>();
    ...
}
```

```
public abstract class PaymentMean {
    ...
    // does not have client reference
    ...
}
```

Navegación de la asociación

- *Extremos UNO*
- *Extremos MUCHO*

Extremos UNO

```
public abstract class PaymentMean {
    ...
    private Client client;
    ...
    public void setClient(Client c) {
        this.client = c;
    }
    public Client getClient() {
        return client;
    }
    ...
}
```

Extremos MUCHO

```
public class Client {
    ...
    private Set<PaymentMean> paymentMeans = new HashSet<>();
    ...
    public Set<PaymentMean> getPaymentMeans() {
        return paymentMeans;
    }
    // no setter for paymentMeans
}
```

Mantenimiento de la asociación

- *Unidireccional: sencillo*
- *Bidireccional: Fundamental mantener las referencias cruzadas*

```
Client c = ...
PaymentMean pm = ...

// link
c.getPaymentMeans().add( pm );
pm.setClient( c );

// unlink
c.getPaymentMeans().remove( pm );
pm.setClient( null );
```

Mantenimiento de la asociación

Fundamental mantener las referencias cruzadas

```
Client c = ...
PaymentMean pm = ...

// link
c.getPaymentMeans().add( pm );
pm.setClient( c );
```

Código repetitivo (propenso a errores)

Rompe encapsulación

```
c.getPaymentMeans().remove( pm );
pm.setClient( null );
```

Práctica NO recomendada

Mantenimiento de la asociación

Alternativa un poco mejor: ***añadir métodos de mantenimiento en uno de los dos extremos***

```
Client c = ...
PaymentMean pm = ...

// link
c.addPaymentMean( pm );

// unlink
c.removePaymentMean( pm );
```

Esta técnica va bien para agregados

Mantenimiento de la asociación

Alternativa un poco mejor: ***añadir métodos de mantenimiento en uno de los dos extremos***

```
Client c = ...
PaymentMean pm = ...

// link
c.addPaymentMean( pm );

// unlink
c.removePaymentMean( pm );
```

```
public class Client {
    ...
    private Set<PaymentMean> paymentMeans = new HashSet<>();
    ...
    public void addPaymentMean(PaymentMean pm) {
        pm.setClient( this );
        paymentMeans.add( pm );
    }

    public void removePaymentMean(PaymentMean pm) {
        paymentMeans.remove( pm );
        pm.setClient( null );
    }

    public Set<PaymentMean> getPaymentMeans() {
        // a copy of the collection
        return new HashSet<>( paymentMeans );
    }
    ...
}
```

¡El orden importa!

Fundamental mantener las referencias cruzadas

Alternativa recomendada: *añadir métodos de mantenimiento estáticos en clase dedicada*

```
Client c = ...
PaymentMean pm = ...

// link
Associations.Pay.link( c, pm );

// unlink
Associations.Pay.unlink( c, pm );
```

Mantenimiento de la asociación

Alternativa recomendada: ***añadir métodos de mantenimiento estáticos en clase dedicada***

```
public class Associations {
    ...
    public static class Pay {
        public static void link(Client c, PaymentMean pm) {
            pm._setClient(c);
            c._getPaymentMeans().add( pm );
        }

        public static void unlink(Client c, PaymentMean pm) {
            c._getPaymentMeans().remove( pm );
            pm._setClient(null);
        }
    }
    ...
}
```

¡El orden importa!

Fundamental mantener las referencias cruzadas

Mantenimiento de la asociación

Alternativa recomendada: ***añadir métodos de mantenimiento estáticos en clase dedicada***

```
public class Associations {
    ...
    public static class Pay {
        public static void link(Client c, PaymentMean pm) {
            pm._setClient(c);
            c._getPaymentMeans().add( pm );
        }

        public static void unlink(Client c, PaymentMean pm) {
            c._getPaymentMeans().remove( pm );
            pm._setClient(null);
        }
    }
    ...
}
```

¡El orden importa!

Fundamental mantener las referencias cruzadas

Mantenimiento de la asociación

Alternativa recomendada: ***añadir métodos de mantenimiento estáticos en clase dedicada***

```
public class Client {
    ...
    private Set<PaymentMean> paymentMeans = new HashSet<>();
    ...

    /*package*/ Set<PaymentMean> _getPaymentMeans() {
        return paymentMeans;
    }
    public Set<PaymentMean> getPaymentMeans() {
        // a copy of the collection
        return new HashSet<>( paymentMeans );
    }
    ...
}
```

¡Dos getters! Para el mismo atributo

Fundamental mantener las referencias cruzadas

Mantenimiento de la asociación

Alternativa recomendada: ***añadir métodos de mantenimiento estáticos en clase dedicada***

```
public abstract class PaymentMean {
    ...
    private Client client;
    ...
    /*package*/ void _setClient(Client c) {
        this.client = c;
    }
    public Client getClient() {
        return client;
    }
    ...
}
```

Setter restringido

Fundamental mantener las referencias cruzadas

Modelos de dominio

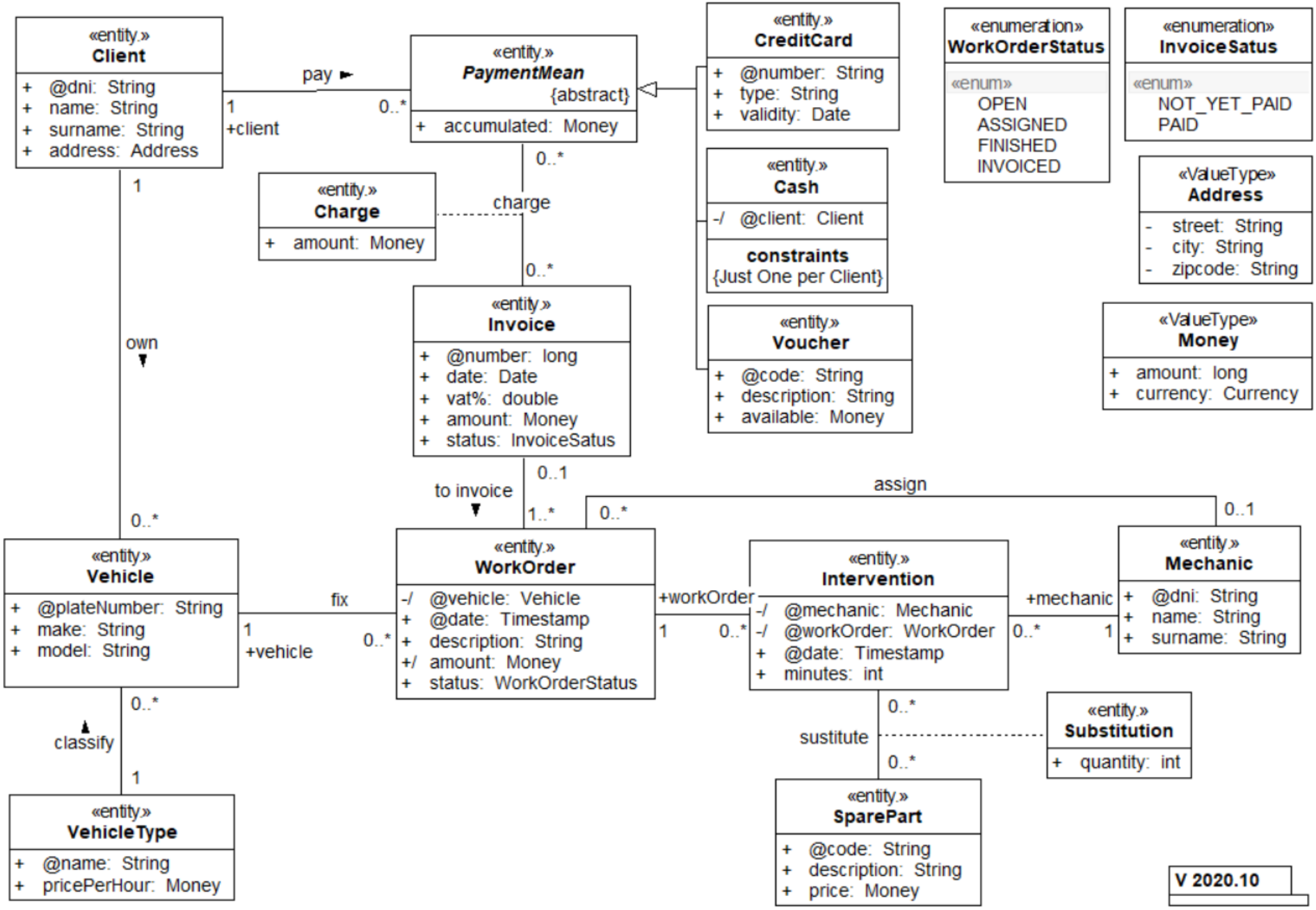
- Representación de conceptos del dominio
  - Son un mapa conceptual
  - Presenta y relaciona los conceptos más importantes del dominio/contexto/problema
- Un diagrama UML de clases se ajusta bastante bien al propósito

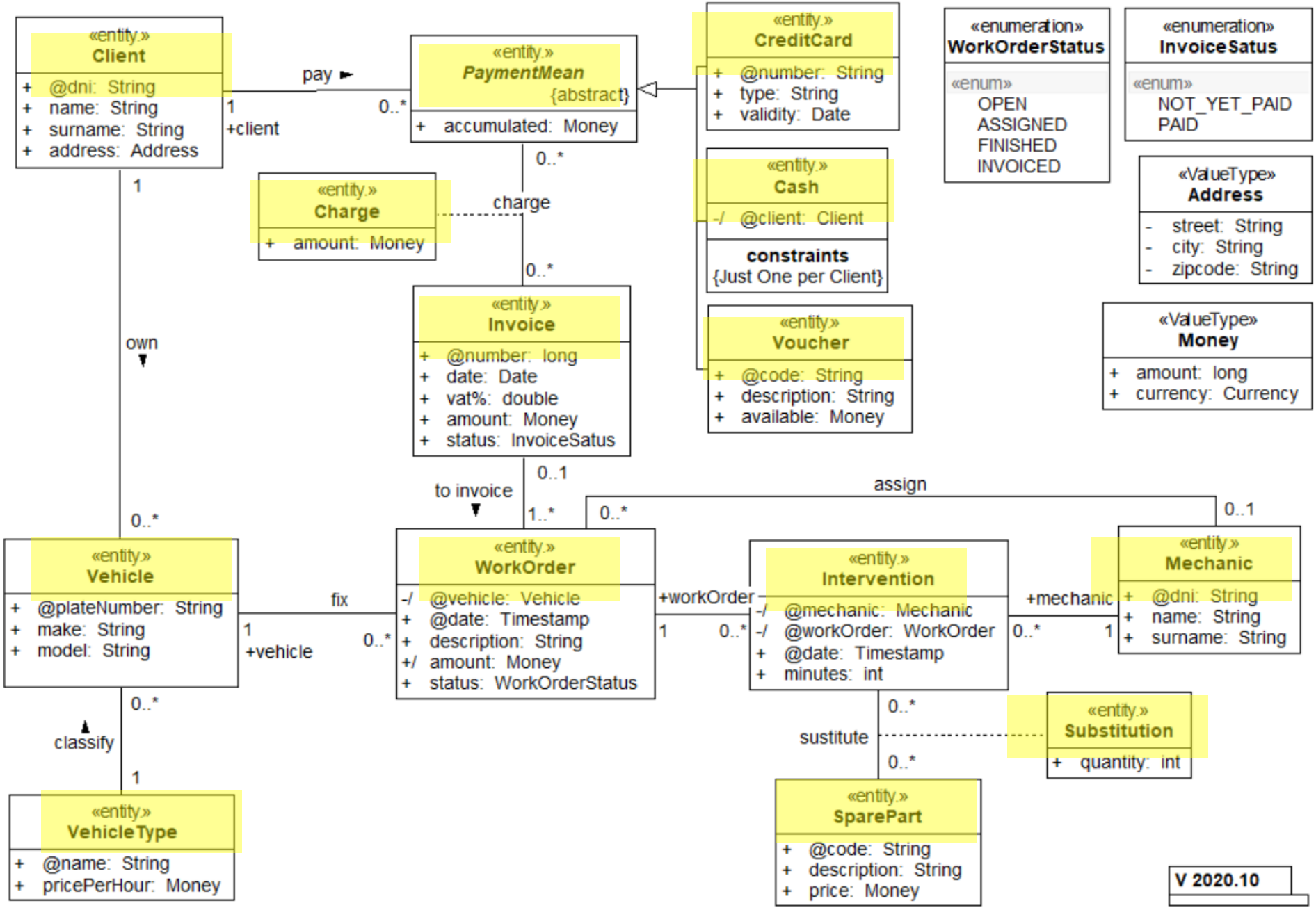
Tipos de clases en modelos de dominio

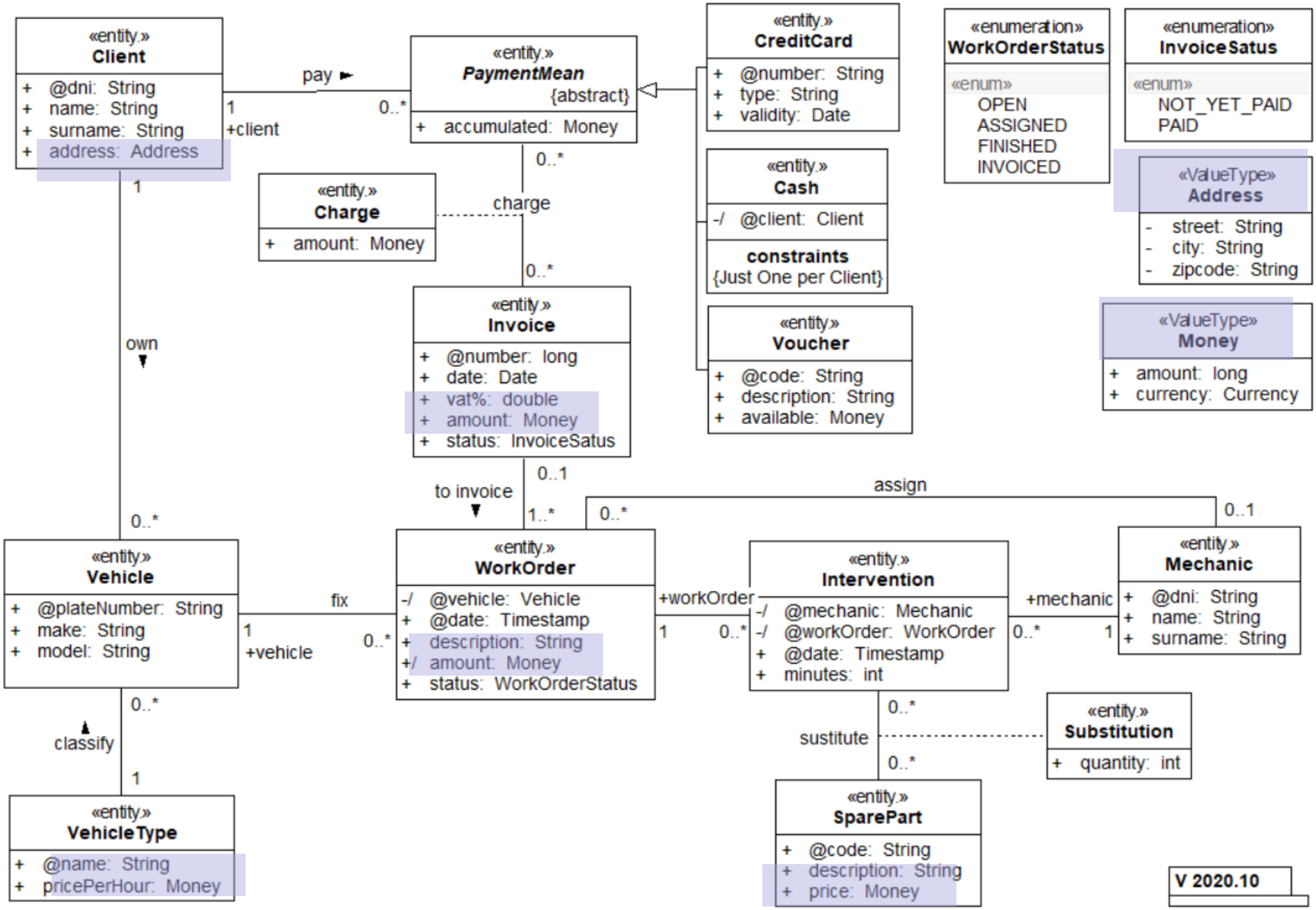
- *Entidades*
- *Tipos valor: ValueTypes*

Inmutables

No se rompe la encapsulación al devolverlos en los getters


«entity»
Client
+ @dni: String
+ name: String
+ surname: String
+ address: Address
pay
1
+client
0..*
«entity»
PaymentMean
{abstract}
+ accumulated: Money
«entity»
CreditCard
+ @number: String
+ type: String
+ validity: Date
«enumeration»
WorkOrderStatus
«enum»
OPEN
ASSIGNED
FINISHED
INVOICED
«enumeration»
InvoiceSatus
«enum»
NOT_YET_PAID
PAID
«entity»
Charge
+ amount: Money
charge
0..*
«entity»
Cash
-/ @client: Client
constraints
{Just One per Client}
«ValueType»
Address
- street: String
- city: String
- zipcode: String
«entity»
Invoice
+ @number: long
+ date: Date
+ vat%: double
+ amount: Money
+ status: InvoiceSatus
«entity»
Voucher
+ @code: String
+ description: String
+ available: Money
«ValueType»
Money
+ amount: long
+ currency: Currency
own
to invoice
0..1
1..*
assign
«entity»
Vehicle
+ @plateNumber: String
+ make: String
+ model: String
fix
+vehicle
«entity»
WorkOrder
-/ @vehicle: Vehicle
+ @date: Timestamp
+ description: String
+/ amount: Money
+ status: WorkOrderStatus
+workOrder
«entity»
Intervention
-/ @mechanic: Mechanic
-/ @workOrder: WorkOrder
+ @date: Timestamp
+ minutes: int
+mechanic
«entity»
Mechanic
+ @dni: String
+ name: String
+ surname: String
classify
«entity»
VehicleType
+ @name: String
+ pricePerHour: Money
sustitute
«entity»
Substitution
+ quantity: int
«entity»
SparePart
+ @code: String
+ description: String
+ price: Money
V 2020.10


«entity»
Client
+ @dni: String
+ name: String
+ surname: String
+ address: Address
pay
1
+client
0..*
«entity»
PaymentMean
{abstract}
+ accumulated: Money
«entity»
CreditCard
+ @number: String
+ type: String
+ validity: Date
«entity»
Cash
-/ @client: Client
constraints
{Just One per Client}
«entity»
Voucher
+ @code: String
+ description: String
+ available: Money
«enumeration»
WorkOrderStatus
«enum»
OPEN
ASSIGNED
FINISHED
INVOICED
«enumeration»
InvoiceSatus
«enum»
NOT_YET_PAID
PAID
«ValueType»
Address
- street: String
- city: String
- zipcode: String
«ValueType»
Money
+ amount: long
+ currency: Currency
«entity»
Charge
+ amount: Money
charge
0..*
1
own
0..*
«entity»
Invoice
+ @number: long
+ date: Date
+ vat%: double
+ amount: Money
+ status: InvoiceSatus
0..1
to invoice
1..*
assign
0..*
0..1
«entity»
Vehicle
+ @plateNumber: String
+ make: String
+ model: String
fix
1
+vehicle
0..*
«entity»
WorkOrder
-/ @vehicle: Vehicle
+ @date: Timestamp
+ description: String
+/ amount: Money
+ status: WorkOrderStatus
+workOrder
1
0..*
«entity»
Intervention
-/ @mechanic: Mechanic
-/ @workOrder: WorkOrder
+ @date: Timestamp
+ minutes: int
+mechanic
0..*
1
«entity»
Mechanic
+ @dni: String
+ name: String
+ surname: String
0..*
classify
1
«entity»
VehicleType
+ @name: String
+ pricePerHour: Money
0..*
sustitute
0..*
«entity»
Substitution
+ quantity: int
«entity»
SparePart
+ @code: String
+ description: String
+ price: Money
V 2020.10


«entity»
Client
+ @dni: String
+ name: String
+ surname: String
+ address: Address
pay
1
+client
0..*
«entity»
PaymentMean
{abstract}
+ accumulated: Money
«entity»
CreditCard
+ @number: String
+ type: String
+ validity: Date
«entity»
Cash
-/ @client: Client
constraints
{Just One per Client}
«entity»
Voucher
+ @code: String
+ description: String
+ available: Money
«enumeration»
WorkOrderStatus
«enum»
OPEN
ASSIGNED
FINISHED
INVOICED
«enumeration»
InvoiceSatus
«enum»
NOT_YET_PAID
PAID
«ValueType»
Address
- street: String
- city: String
- zipcode: String
«ValueType»
Money
+ amount: long
+ currency: Currency
«entity»
Charge
+ amount: Money
charge
0..*
0..*
1
own
0..*
«entity»
Invoice
+ @number: long
+ date: Date
+ vat%: double
+ amount: Money
+ status: InvoiceSatus
0..1
1..*
to invoice
assign
0..*
0..1
«entity»
Vehicle
+ @plateNumber: String
+ make: String
+ model: String
fix
1
+vehicle
0..*
«entity»
WorkOrder
-/ @vehicle: Vehicle
+ @date: Timestamp
+ description: String
+/ amount: Money
+ status: WorkOrderStatus
+workOrder
1
0..*
«entity»
Intervention
-/ @mechanic: Mechanic
-/ @workOrder: WorkOrder
+ @date: Timestamp
+ minutes: int
+mechanic
0..*
1
«entity»
Mechanic
+ @dni: String
+ name: String
+ surname: String
0..*
classify
1
«entity»
VehicleType
+ @name: String
+ pricePerHour: Money
0..*
sustitute
0..*
«entity»
Substitution
+ quantity: int
«entity»
SparePart
+ @code: String
+ description: String
+ price: Money
V 2020.10

Entidades

- Representa la existencia de "algo" en el dominio (en la realidad) que es de interés y que tiene identidad propia
  - Sus propiedades pueden cambiar a lo largo del tiempo pero sigue siendo "ella"
  - Orden de trabajo, Factura, Vehículo, Cliente…
- Puede estar asociada con otras entidades
- Su ciclo de vida es independiente de otras entidades

Representación de la identidad

Identidad: forma de distinguir una cosa de otra…

… basándonos en algún rasgo o característica

Rasgo o característica --> atributo(s)

Debe haber algún atributo (o combinación) que sirva de identificación

- Los atributos que forman identidad son inmutables

Representación de la identidad

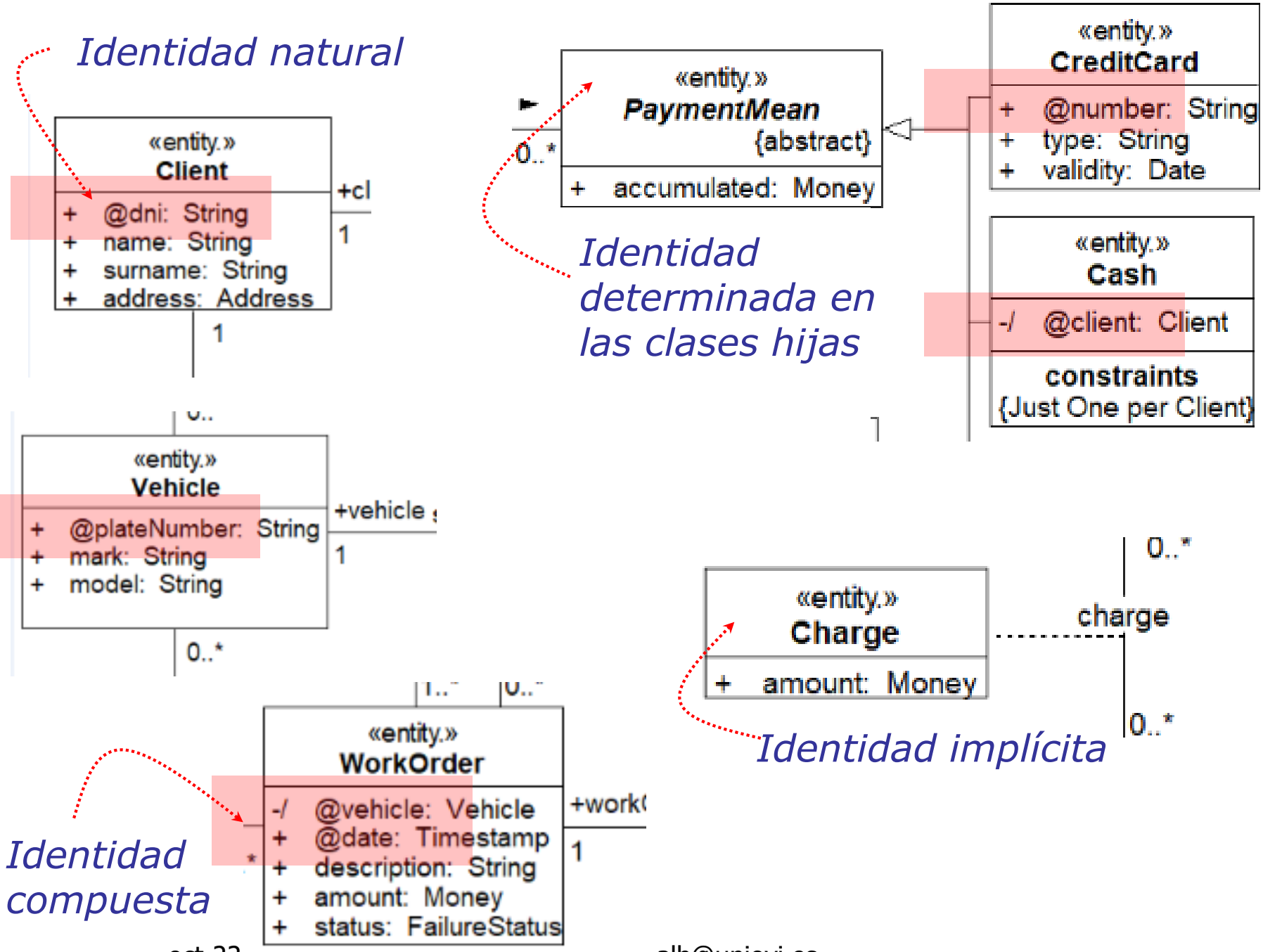
Identidad natural

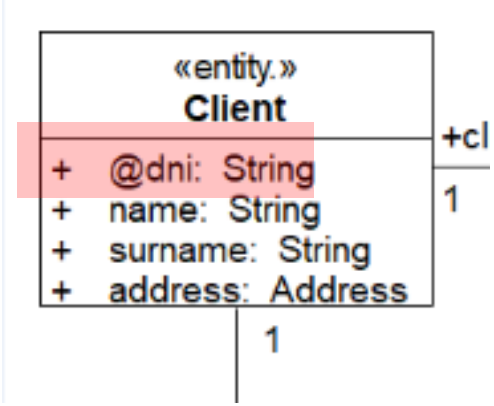
- Basada en atributos naturales del domino
- Siempre debe existir atributo o combinación que formen identidad natural

Identidad artificial

- Basada en atributo extra (artificial) con valor generado sin repetición posible

```
public class Client {

    private String id = UUID.randomUUID().toString();
    private String dni;
    ...
```


Identidad natural
«entity.»
Client
+ @dni: String
+ name: String
+ surname: String
+ address: Address
+cl
1
1
«entity.»
PaymentMean
{abstract}
+ accumulated: Money
0..*
Identidad determinada en las clases hijas
«entity.»
CreditCard
+ @number: String
+ type: String
+ validity: Date
«entity.»
Cash
-/ @client: Client
constraints
{Just One per Client}
«entity.»
Vehicle
+ @plateNumber: String
+ mark: String
+ model: String
+vehicle
1
0..*
«entity.»
Charge
+ amount: Money
0..*
charge
0..*
Identidad implícita
«entity.»
WorkOrder
-/ @vehicle: Vehicle
+ @date: Timestamp
+ description: String
+ amount: Money
+ status: FailureStatus
+work
1
*
Identidad compuesta



La identidad natural siempre se recibe en el constructor, aunque se use identidad artificial

No puede haber constructor con menos parámetros que la identidad natural

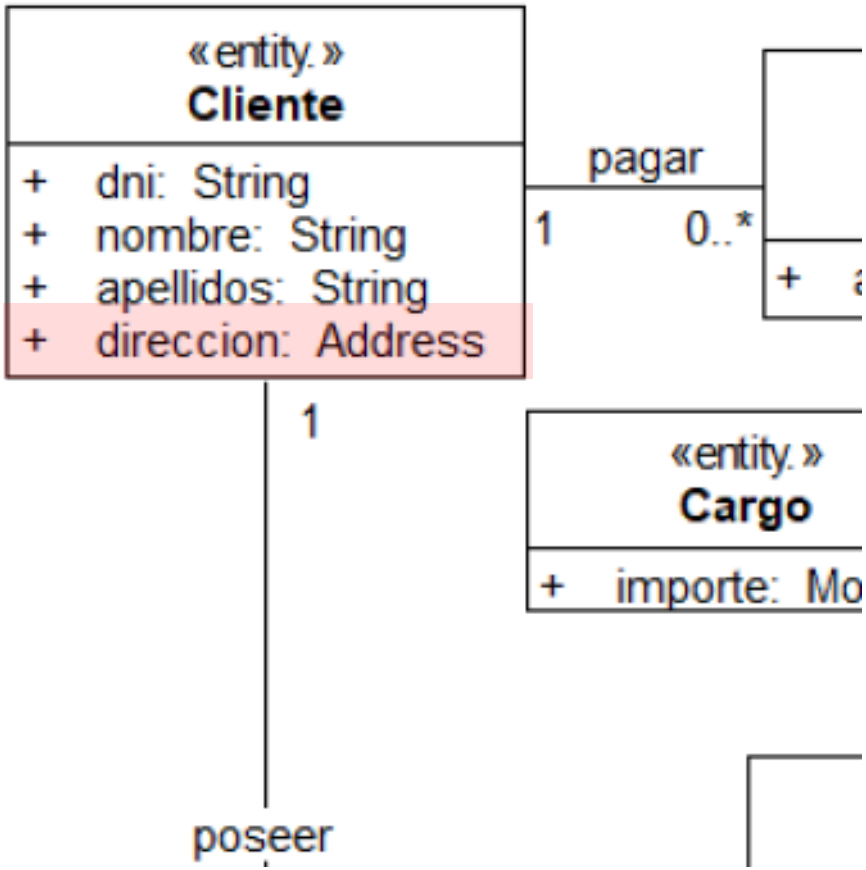
```
public Client(String dni) {
    Argument.isTrue( dni != null && dni.length() > 0 );
    this.dni = dni;
}

public Client(String dni, String name, String surname) {
    this( dni );
    this.name = name;
    this.surname = surname;
}
```

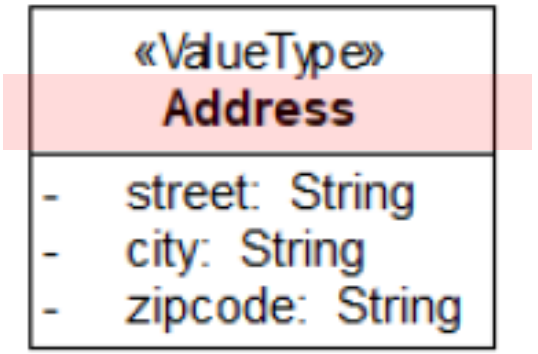
Los constructores se reúsan

Tipos Valor (Value Type)

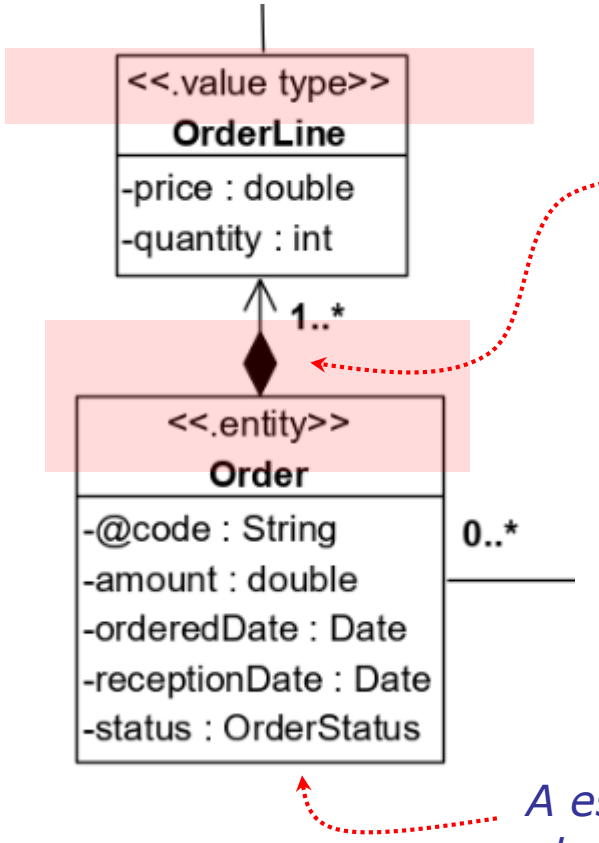
- También son conceptos del dominio
  - Nombre, dirección, apellidos, email, etc.
- Representan un valor, no tienen identidad
  - Una moneda de 2€ no importa si es ésta o aquella, sólo importa que es de 2€
  - Su valor es inalterable
- Son atributos de una entidad
  - Su ciclo de vida depende enteramente de la entidad que las posee
- Son inmutables → no tienen setters
  - Así son los tipos básicos Java: Integer, Double, String, etc., menos Date (?)

Tipos Valor en UML





Tipos valor en UML

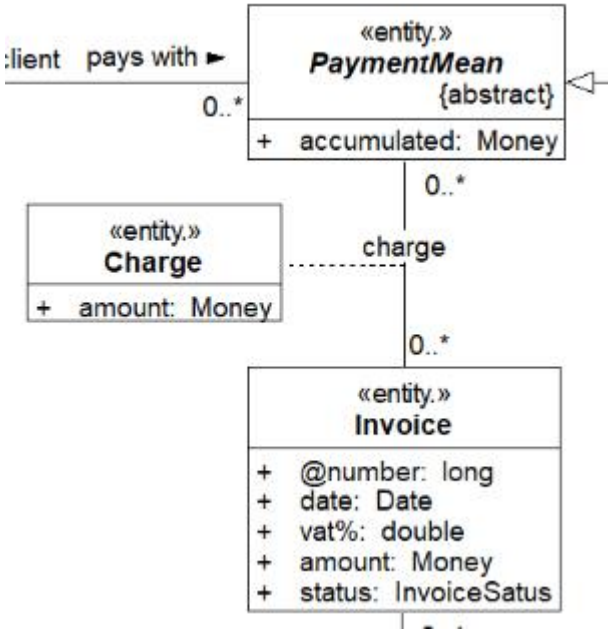


Si es una colección de value types se representan con una composición

(el ciclo de vida del value type depende de la entidad)

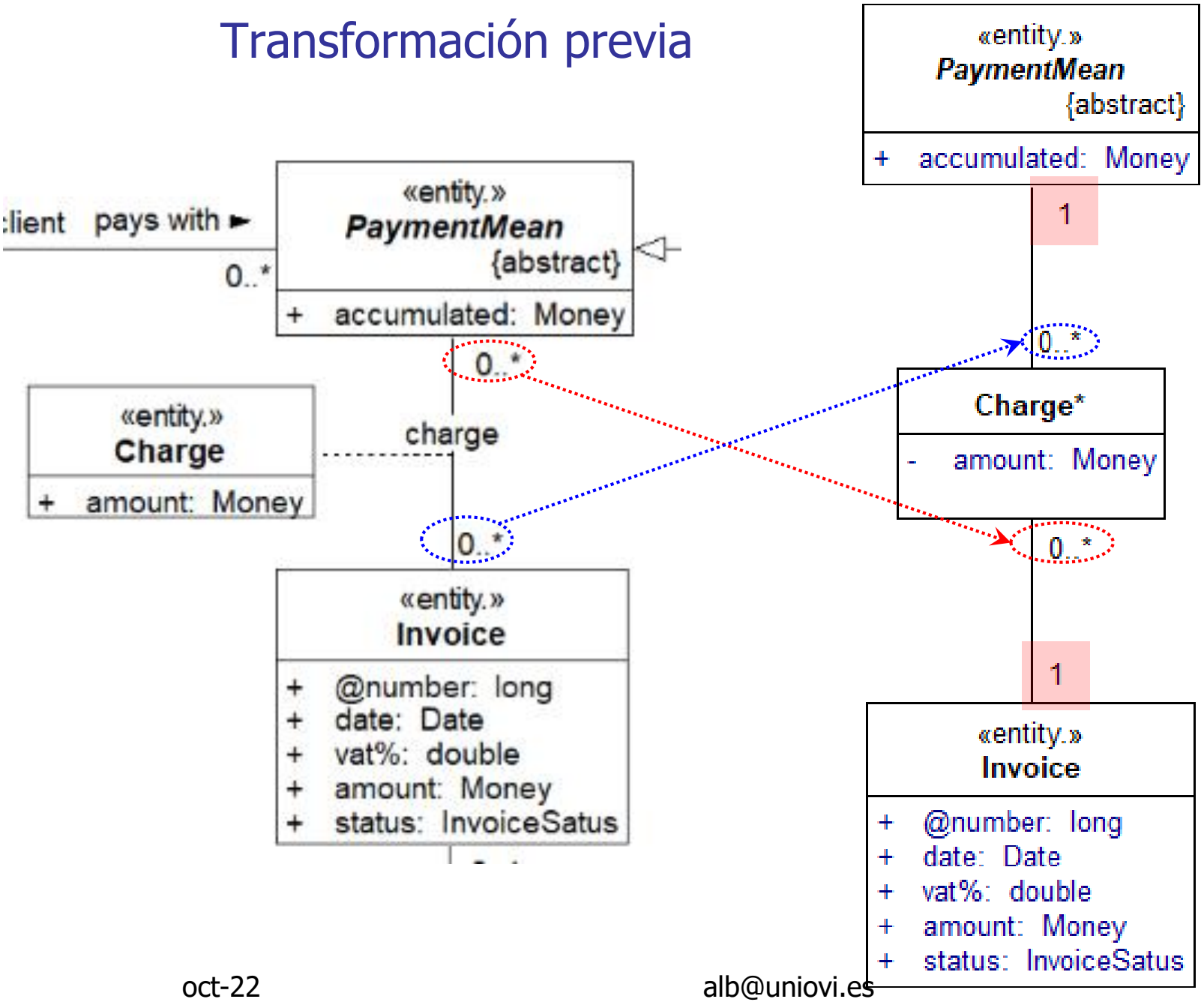
*A esta clase se le llama **agregado** en otros contextos (NoSQL, DDD, etc.)*

Implementación de clases asociativas

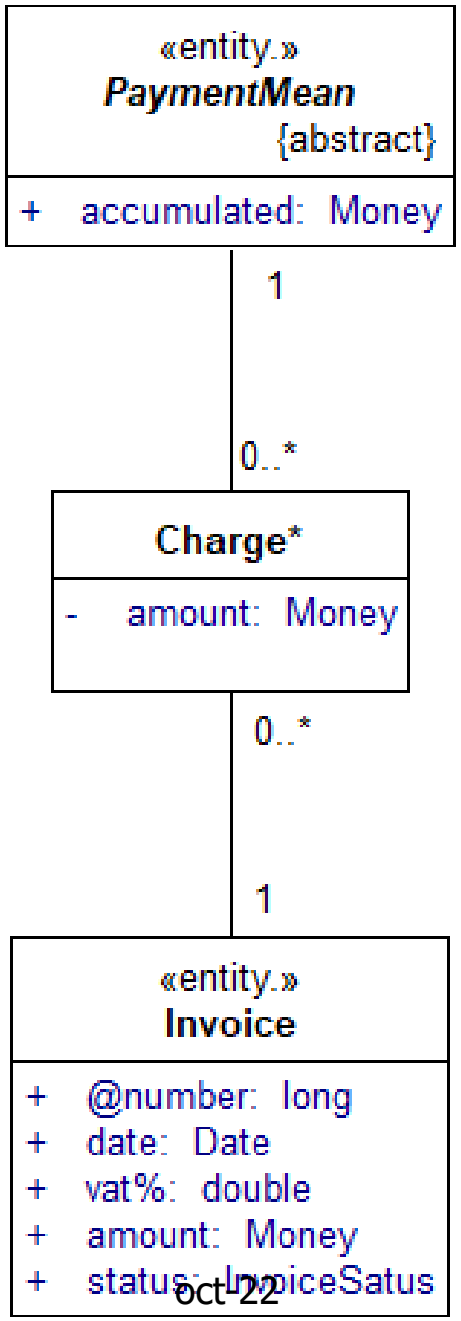


- Representan a la vez clase y asociación
- Permiten añadir atributos y funcionalidad a una asociación
- En java se implementan con una clase
- Cada instancia representa un enlace
- Mismas consideraciones de cardinalidad y navegabilidad
- Identidad compuesta por los dos extremos →dos objetos sólo pueden estar enlazados una vez

Transformación previa



Paso a Java




```java
public class Charge {
    private Invoice invoice;
    private PaymentMean paymentMean;
    ...

    public Charge(Invoice i, PaymentMean pm, ...) {
        ...
        Associations.Charges.link(i, this, pm);
    }

    /*package*/ void _setInvoice(Invoice i {
        this.invoice = i;
    }

    /*package*/ void _setPaymentMean(PaymentMean pm) {
        this.paymentMean = pm;
    }
```

Los dos ramales de la asociación deben estar sincronizados

Paso a Java

```
public class Charge {
    private Invoice invoice;
    private PaymentMean paymentMean;
    ...

    public Charge(Invoice i, PaymentMean pm, ...) {
        ...
        Associations.Charges.link(i, this, pm);
    }

    /*package*/ void _setInvoice(Invoice i {
        this.invoice = i;
    }

    /*package*/ void _setPaymentMean(PaymentMean pm) {
        this.paymentMean = pm;
    }
```

Identidad

Siempre en el constructor

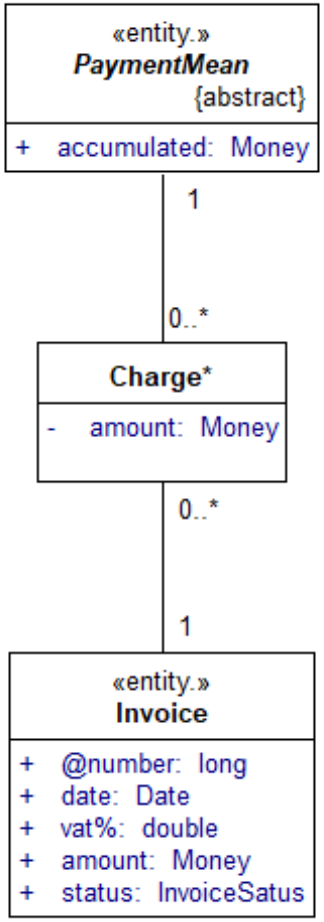
Última línea del constructor

Setters restringidos

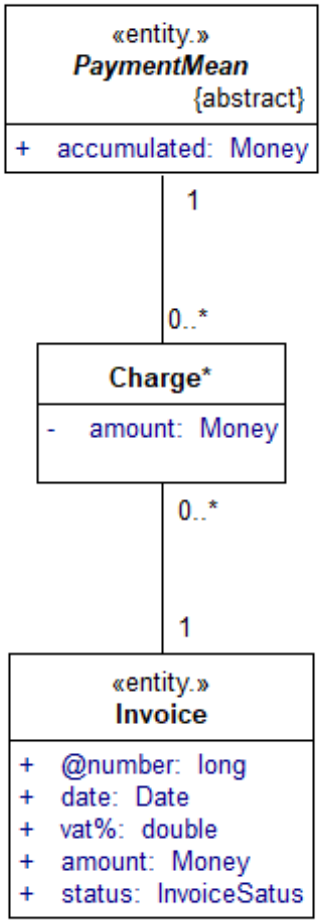
Paso a Java

```
public class Associations {
    ...
    public static class Charges {
        public static void link(Invoice i, Charge c, PaymentMean pm) {
            c._setInvoice(i);
            c._setPaymentMean(pm);

            i._getCharges().add( c );
            pm._getCharges().add( c );
        }

        public static void unlink(Charge c) {
            i._getCharges().remove( c );
            pm._getCharges().remove( c );

            c._setInvoice(null);
            c._setPaymentMean(null);
        }
    }
    ...
}
```

Los dos ramales de la asociación deben estar sincronizados



Paso a Java




```
public class Invoice {
    ...
    private Set<Charge> charges = new HashSet<>();
    /*package*/ Set<Charge> _getCharges() {
        return charges;
    }
    public Set<Charge> getCharges() {
        return new HashSet<>( charges );
    }
    ...
```

_getCharges() Acceso paquete
getCharges() Acceso public

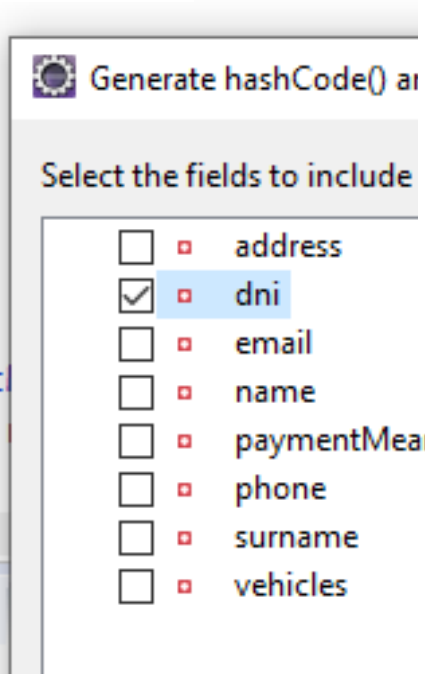
```
 ublic abstract class PaymentMean {
    ...
    private Set<Charge> charges = new HashSet<>();
    /*package*/ Set<Charge> _getCharges() {
        return charges;
    }
    public Set<Charge> getCharges() {
        return new HashSet<>( charges );
    }
    ...
}
```

equals() y hashCode()

- Entidades:
  - hashCode() y equals() redefinido SÓLO sobre los atributos que determinan identidad

```
public class Client {
    private String dni;
    private String name;
    private String surname;
    private String email;
    private String phone;
    private Address address;

    private Set<PaymentMean> payment
    private Set<Vehicle> vehicles =
```

lems @ Javadoc Declaration Search Console



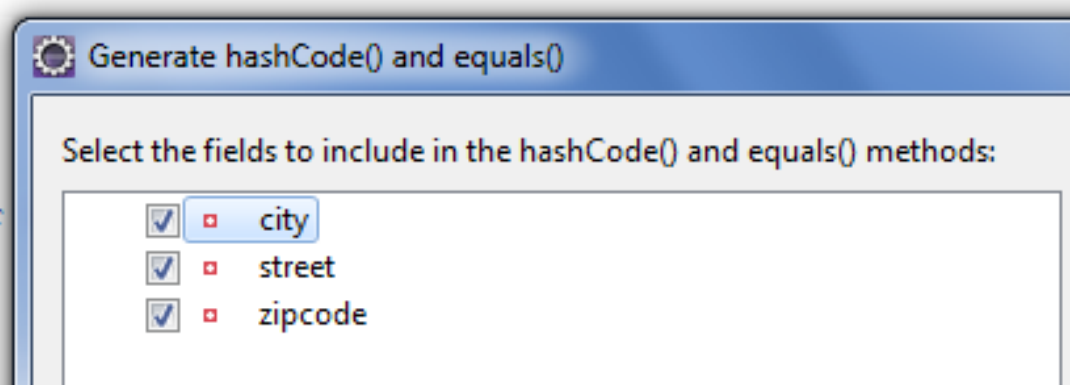
Si se opta por identidad artificial va sobre ese atributo

equals() y hashCode()

Tipos Valor:

- HashCode y equals redefinido sobre TODOS los atributos

```
public class Address implements Serializable {
    private String street;
    private String zipcode;
    private String city;

    /**
     * No-arg constructor for
     */
    public Address() {
    }
```



toString()

Redefinir toString()

- Sin interés funcional pero útil para depuración
- Generar automático con el IDE
- No incluir referencias a otras entidades

```
public class Averia {
    private String descripcion;
    private Date fecha;
    private Double importe;
    private AveriaStatus status
    private Factura factura;
    private Mecanico mecanico;
    private Vehiculo vehiculo;

    protected Set<Intervencion>
```