UNIVERSIDAD DE MÁLAGA

Modelado y Diseño de Software
Grado en Ingeniería del Software

Tema 5

Diseño orientado a objetos

Diseño

Concepto

- Conjunto de planes y decisiones para definir un producto con los suficientes detalles como para permitir su realización física de acuerdo a unos requisitos

Objetivos

- satisfacer la especificación funcional del producto
- respetar requisitos sobre forma, rendimiento, coste, etc.
- ajustarse a las limitaciones del medio de destino

Diseño de software

- primera etapa *técnica* del proceso de desarrollo
  - centrada en el dominio de la solución
- proporciona un modelo técnico del sistema
  - representación técnica del software a desarrollar
- determina la calidad del software
  - correcto: *acorde a la función deseada.*
  - fiable: *incluso en situaciones anormales.*
  - eficiente: *buen uso de los recursos.*
  - económico: *desde un punto de vista industrial.*
  - flexible: ***adaptable al cambio, reutilizable**.*

Diseño orientado a objetos 3

Diseño orientado a objetos con UML

- toma como base los diagramas del modelo conceptual
  - que evoluciona y se completa
  - hasta convertirse en el modelo de diseño
- pero se centra en el dominio de la solución
  - no en el dominio del problema
  - en función de las tecnologías de implementación elegidas
  - en particular el lenguaje de programación
- utiliza los mismos diagramas UML
  - clases, objetos, interacción, etc.
- y algunos nuevos
  - como los diagramas de paquetes

Diseño orientado a objetos 4

Diseño orientado a objetos
Agenda

- revisar, completar y dar coherencia al modelo
  - visibilidad (de atributos y operaciones)
  - completar la lista de operaciones
    - *getters* y *setters*
    - recorrido de relaciones
    - provenientes del modelo dinámico
  - clases abstractas e interfaces; clases genéricas
- diseño de las relaciones
  - reificación de n-arias y con atributos
  - direccionalidad
- implementación del comportamiento
- empaquetamiento

Diseño orientado a objetos 5

Diseño orientado a objetos
Clases

- unidad modular de los LOO
  - encapsulamiento
  - independencia modular
    - alta cohesión
    - bajo acoplamiento
    - facilita el mantenimiento y la reutilización
- una clase por cada entidad del modelo conceptual
  - descomponiendo en clases entidad, control e interfaz
  - Añadiendo clases de diseño
    - Por ejemplo estructuras de datos o reificación de asociaciones

Diseño orientado a objetos 6

Clases
Acoplamiento

- medida de la interrelación entre dos elementos
  - cualquier interacción entre clases causa acoplamiento
    - por asociación, incluyendo agregación y composición
    - por dependencia, incl. creación de objetos y uso en argumentos
    - por herencia y realización de interfaces
      - directa: por herencia de atributos y operaciones
      - indirecta: por herencia de acoplamiento
  - acoplamiento fuerte (vs. débil)
    - una clase conoce detalles de implementación de otra
      - en particular, nombres de atributos

  Conclusión:

  - reducir la interfaz pública de las clases lo más posible

Diseño orientado a objetos 7

Clases

- Nombre
  - utilizar consistentemente una convención de nombres
  - sustantivo o frase sustantiva
    - en singular y sin abreviaturas
    - empezando cada palabra con mayúscula

    Ejemplos:

    - Alumno, AlumnoDeGrado
- Invariantes
  - documentar como estereotipos o restricciones

    Ejemplos:

    - Alumno «permanent»
    - Rectangulo { anchura > altura }

Diseño orientado a objetos 8

Clases

- Constructores
  - Utilizar los constructores para restringir las opciones de los usuarios
  - Sirven para asegurar la correcta creación e inicialización de los objetos
  - También pueden ayudar a garantizar las restricciones de ciertas asociaciones (por ejemplo, las que necesitan a otro objeto siempre asociado, debido a una multiplicidad mayor o igual que 1)
  - Es importante determinar la visibilidad correcta

Diseño orientado a objetos 9

Atributos y operaciones
Visibilidad

Nivel	UML	Java
pública	+feature	public
de paquete	~feature	
protegida	#feature	protected
privada	–feature	private

- utilizar el nivel más restringido posible
  - permite reducir el acoplamiento entre clases
  - recordar la Ley de Demeter

Diseño orientado a objetos 10

Operaciones

- Nombre
  - utilizar consistentemente una convención
    - verbo de acción o frase verbal
    - sin abreviaturas
    - notación Camel empezando con minúscula

    Ejemplos:
    - matricular(Asignatura), calcularNotaMedia()
- documentar precondiciones y poscondiciones
  - en OCL, preferentemente
- separar métodos *query* y modificadores
  - las *queries* no deben tener efectos secundarios
  - Plantearse si los modificadores deben devolver valor (*)

Diseño orientado a objetos 11

Operaciones Ejemplos

Mal	**Bien**
openAcc()	openAccount()
mailingLabelPrint()	printMailingLabel()
purchaseparkingpass()	purchaseParkingPass()
purchase_ticket()	purchaseTicket()
saveTheObject()	save()
GetFirstName()	getFirstName()
fetchLastName()	getLastName()
changeLastName()	setLastName()

Diseño orientado a objetos 12

Atributos

- Nombre
  - utilizar consistentemente una convención
    - nombre o frase sustantiva, sin abreviaturas
    - en plural para atributos múltiples
    - notación Camel empezando con minúscula

    Ejemplos:
    - edad, telefonoDeContacto
  - también para los argumentos de las operaciones
    - nombre y orden
- indicar invariantes con estereotipos, restricciones o valores por defecto o iniciales

  Ejemplos: {readOnly} {notNull} {id} {100..999}

Diseño orientado a objetos 13

Atributos

- darles visibilidad privada
  - documentar por qué no la tienen
- utilizar métodos get y set para el acceso
  - incluso dentro de la propia clase!!!!!
  - implementar la lógica de validación en setAtributo()
  - salvo que involucre varios atributos a la vez

  Ejemplo: validación de fechas
- utilizar inicialización perezosa en getAtributo()
  - para atributos costosos y usados rara vez

  Ejemplo: patrón Singular

Diseño orientado a objetos 14

Atributos
Ejemplos

Mal	**Bien**
fName	firstName
firstname	firstName
studentFirstName	firstName
nameLast	lastName
firstNameString	firstName
hTTPConnection	httpConnection
subjectList	subjects

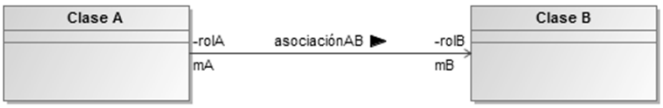
Diseño orientado a objetos 15

Constantes y valores

- Las constantes en Mayúsculas y usando el separador "_" entre palabras
  - Ejemplos: PI, MAX_STUDENTS, LONGITUD
- Los valores de los tipos enumerados deben usar la notación Camel, empezando en minúscula
  - Ejemplos: male, female

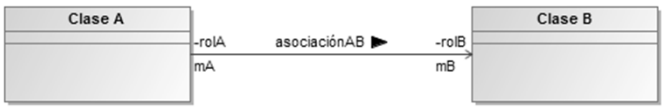
Ingeniería del Software. Diseño 16

Diseño orientado a objetos
Asociaciones



- se debe diseñar una estrategia de implementación
  - que puede ser general, o bien
  - particular para cada una de ellas.
- habitualmente las herramientas CASE para UML
  - establecen de forma predeterminada una estrategia
  - que se aplica para la generación automática de código

Asociaciones



- se implementan mediante código de *andamiaje*
  - añadiendo atributos y operaciones a las clases
- aspectos que influyen en el diseño:
  - tipo de asociación: simple, n-aria, con atributos, …
  - direccionalidad: unidireccional, bidireccional
  - multiplicidad: 1:1, 1:M, M:N
  - restricciones: ordered, …

Asociaciones
Andamiaje

- atributos
  - estructura básica para implementar la asociación
  - en principio, tendrán visibilidad privada
- operaciones
  - definen la interfaz de la asociación: get(), set(), add(), …
  - restringiremos la visibilidad al máximo
- como base para sus nombres utilizaremos:
  - nombre de los roles del extremo opuesto
    - para el nombre del atributo
  - nombre de la clase del extremo opuesto
    - para el tipo del atributo
  - Multiplicidad del rol del extremo opuesto
    - para la multiplicidad del atributo

Diseño orientado a objetos 19

Asociaciones
Direccionalidad

- en el modelo conceptual
  - las asociaciones son generalmente bidireccionales
  - pero en muchos casos solo necesitan recorrerse en una dirección
- para determinar la direccionalidad en el diseño
  - revisar las operaciones en las clases relacionadas
  - y el envío de mensajes en los diagramas de interacción
- evitar las asociaciones bidireccionales
  - siempre que sea necesario
  - incluso rediseñando las interacciones

Diseño orientado a objetos 20

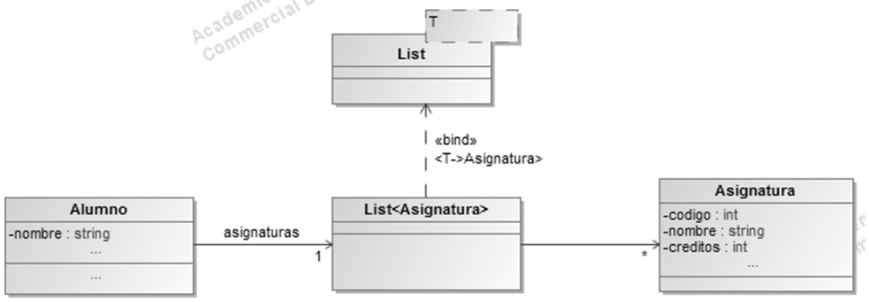
Asociaciones unidireccionales con extremo 1


```
public class Alumno {
  private String nombre;
  private Direccion domicilio;
  public Direccion getDomicilio()
    { return domicilio; }
  public void setDomicilio(Direccion domicilio)
    { this.domicilio = domicilio; }
  }
```

- el andamiaje no se incluye en los diagramas de clases

Asociaciones unidireccionales con extremo M > 1

- se pueden implementar mediante una lista:



Asociaciones unidireccionales con extremo M > 1


```
public class Alumno {
  private String nombre;
  private List<Asignatura> asignaturas
                = new LinkedList<Asignatura>();
  public Enumeration<Asignatura> getAsignaturas() {...}
  public void addAsignatura(Asignatura asig) {...}
  public void removeAsignatura(Asignatura asig) {...}
  ...
  }
```

Diseño orientado a objetos 23

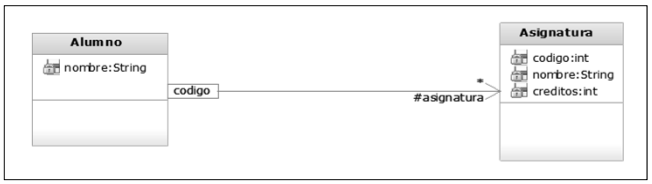
Asociaciones unidireccionales con extremo M > 1


```
public class Alumno {
  private String nombre;
  private List<Asignatura> asignaturas
                = new LinkedList<Asignatura>();
  public Enumeration<Asignatura> getAsignaturas() {...}
  public Asignatura getAsignatura(int i) {...}
  public void addAsignatura(Asignatura asig) {...}
  public void removeAsignatura(Asignatura asig) {...}
  ...
  }
```

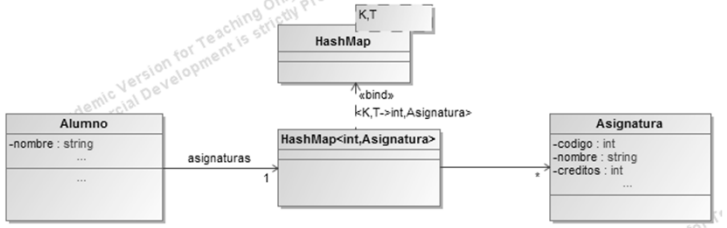
Diseño orientado a objetos 24

Asociaciones calificadas

- una clave o *calificador* identifica el objeto asociado

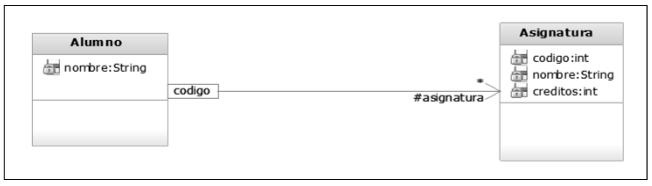


- se implementan mediante un diccionario o tabla hash



Diseño orientado a objetos 25

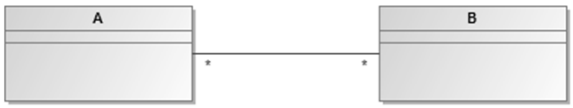
Asociaciones calificadas




```
public class Alumno {
  private String nombre;
  private HashMap<int,Asignatura> asignaturas
                  = new HashMap<int,Asignatura>();
  public Enumeration<Asignatura> getAsignaturas() {...}
  public Asignatura getAsignatura(int codigo)
  public void addAsignatura(int codigo, Asignatura a) {...}
  public void removeAsignatura(int codigo) {...}
  ...
  }
```

Diseño orientado a objetos 26

Asociaciones bidireccionales



- se implementan como dos asoc. unidireccionales
  - mediante pares de referencias
  - hay que asegurar la consistencia!!!!
  - especialmente complicado en asociaciones M:N

Conclusiones:

  - reconsiderar si es imprescindible la bidireccionalidad
  - reificar e implementar mediante una clase asociación

Diseño orientado a objetos 27

Asociaciones mediante clase asociación

- reificando la asociación:

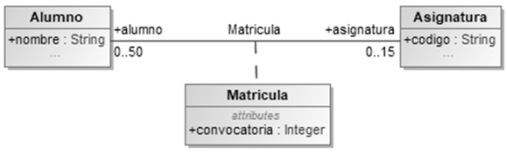
- la clase asociación se encarga de la consistencia!!!!

```
AsocAB(A a, B b) {
  this.a = a;
  this.b = b;
  a.addAsocAB(this);
  b.addAsocAB(this);
  }
```

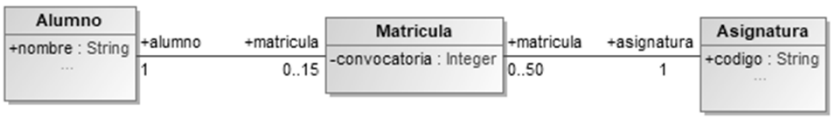
```
remove() {
  a.removeAsocAB(this);
  b.removeAsocAB(this);
  }
```

Diseño orientado a objetos 28

Asociaciones con atributos y n-arias



- aplicaremos de nuevo reificación:



- y la clase Matrícula se encarga de mantener la consistencia

Diseño orientado a objetos 29

Código de andamiaje

- Debe garantizar la consistencia
- En las clases relacionadas:
  - No deben permitirse setters públicos en las estructuras que implementan las asociaciones
    - Las operaciones que gestionan las asociaciones solo deben ser invocadas por la clase que implementa la asociación (restringir la visibilidad)
  - Los getters no deben permitir la actualización, sobre todo en las listas
    - En Java, usar java.util.Enumerator o bien usar Collections.unmodifiableCollection()
    - Usar java.util.Iterator puede servir, aunque permite modificar la colección, y eso es algo que no queremos que suceda!

Ingeniería del Software. Diseño 30

Supongamos el modelo conceptual mostrado a continuación, que representa a empresas y empleados, y los contratos de trabajo en curso que pueden tener en u

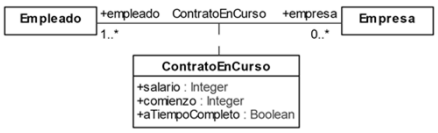
Ejemplo (Septiembre 2020)

Supongamos ese modelo conceptual con las siguientes restricciones:

- Los atributos “salario” y “comienzo” de la clase ContratoEnCurso tienen que ser estrictamente positivos.
- Un empleado no puede tener más de un contrato a tiempo completo, aunque puede tener otros muchos a tiempo parcial. También puede ocurrir que un empleado no tenga contratos a tiempo completo, solo a tiempo parcial.
- Una empresa puede tener a lo más un 40 % de sus empleados a tiempo parcial.
- Los años de comienzo de los contratos en curso de un empleado no pueden coincidir, pues se supone que un empleado no puede firmar dos contratos en una misma fecha

Supongamos el modelo conceptual mostrado a continuación, que representa a empresas y empleados, y los contratos de trabajo en curso que pueden tener en u

Ejemplo (Septiembre 2020)



Se pide:

- Convertir el modelo conceptual en un modelo de diseño, utilizando la herramienta MagicDraw, que permita la implementación, en un lenguaje orientado a objetos como Java, de las entidades, asociaciones y restricciones correspondientes. Además, debe respetarse la restricción del modelo conceptual de que un empleado no puede tener más de un contrato en curso con la misma empresa.
- Desarrolle una implementación en Java correspondiente a dicho modelo de diseño, que incorpore el código de andamiaje para crear objetos y relaciones entre ellos garantizando en todo momento la coherencia de los objetos y las relaciones de la implementación (incluir las restricciones del modelo es opcional, no obligatorio).

Solución

Con las restricciones:

context ContratoEnCurso inv ContratoOk:
self.salario>0 and self.comienzo>0

context Empleado inv ALoMasUnoCompleto:
self.contratoEnCurso->select(aTiempoCompleto)->size()<=1

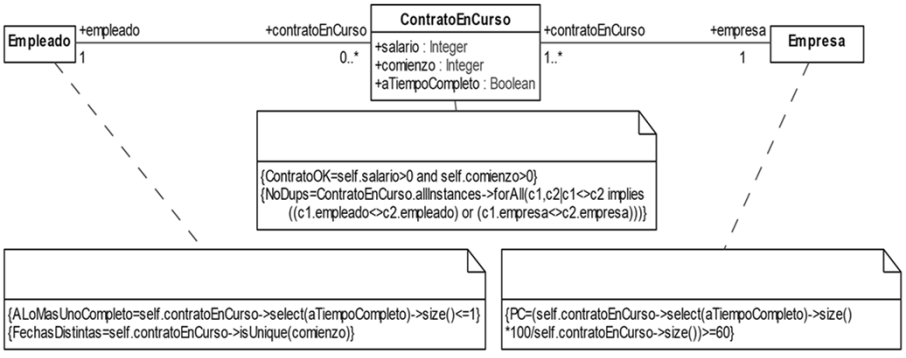
context Empresa inv PC:
(self.contratoEnCurso->select(aTiempoCompleto)->size()*100/
self.contratoEnCurso->size())>=60

context Empleado inv FechasDistintas:
self.contratoEnCurso->isUnique(comienzo)

Además de la que no permite que haya mas de un contrato en curso entre un mismo empleado y una misma empresa:

context ContratoEnCurso inv NoDups:
ContratoEnCurso.allInstances->forAll(c1,c2|c1<>c2 implies
((c1.empleado<>c2.empleado) or (c1.empresa<>c2.empresa)))

Solución



Clase Asoc. ContratoEnCurso

```
public class ContratoEnCurso {
    int salario;
    int comienzo;
    boolean aTiempoCompleto;
    Empleado empleado;
    Empresa empresa;

    public ContratoEnCurso(Empleado e, Empresa a, int salario, int comienzo,
                           boolean aTiempoCompleto) {
        this.salario=salario;
        this.comienzo=comienzo;
        this.aTiempoCompleto=aTiempoCompleto;
        this.empleado = e;
        this.empresa = a;
        e.addContrato(this);
        a.addContrato(this);
    }
    public int getSalario() { return this.salario; }

    public int getComienzo() { return this.comienzo; }

    public boolean getATiempoCompleto() { return this.aTiempoCompleto; }
}
```


Clase Asoc. ContratoEnCurso

```
public class ContratoEnCurso {
    int salario;
    int comienzo;
    boolean aTiempoCompleto;
    Empleado empleado;
    Empresa empresa;

    public ContratoEnCurso(Empleado e, Empresa a, int salario, int comienzo,
                           boolean aTiempoCompleto) throws AssertionError {

        assert (salario>0); //restriccion (1)
        assert (comienzo>0); //restriccion (2)
        assert (!aTiempoCompleto || !e.tieneContratoAtiempoCompleto()); //restriccion (2)
        assert (!e.trabajaEn(a)); // duplicidad de contratos

        this.salario=salario;
        this.comienzo=comienzo;
        this.aTiempoCompleto=aTiempoCompleto;
        this.empleado = e;
        this.empresa = a;
        e.addContrato(this);
        a.addContrato(this);

        assert (e.sinSolapamientos()); //restriccion (4)
        assert(a.checkPC()); //restriccion (3) comprobamos tras añadir el contrato
    }
    public int getSalario() { return this.salario; }

    public int getComienzo() { return this.comienzo; }

    public boolean getATiempoCompleto() { return this.aTiempoCompleto; }
}
```

Clase Empleado

```
public class Empleado {
    private List<ContratoEnCurso> contratoEnCurso;
    public Empleado() {
        contratoEnCurso = new ArrayList<ContratoEnCurso>();
    }
    protected void addContrato(ContratoEnCurso c) {
        contratoEnCurso.add(c);
    }
    protected void rmContrato(ContratoEnCurso c) {
        contratoEnCurso.remove(c);
    }
    public Enumeration<ContratoEnCurso> getContratoEnCurso() {
        return java.util.Collections.enumeration(contratoEnCurso);
    }
}
```


Clase Empleado (cnt'd)

```
// Métodos adicionales para comprobar restricciones
protected boolean tieneContratoAtiempoCompleto() {
    Enumeration<ContratoEnCurso> it = this.getContratoEnCurso();
    while (it.hasMoreElements()) {if (it.nextElement().aTiempoCompleto) return true;}
    return false;
}

protected boolean trabajaEn(Empresa a) {/
    Enumeration<ContratoEnCurso> it = this.getContratoEnCurso();
    while (it.hasMoreElements ()) {if (it.nextElement().empresa == a) return true;}
    return false;
}

protected boolean sinSolapamientos() {
    Enumeration<ContratoEnCurso> it1 = this.getContratoEnCurso();
    Enumeration<ContratoEnCurso> it2 = this.getContratoEnCurso();
    while (it1.hasMoreElements ()) {
        while (it2.hasMoreElements ()) {
            if (it1.nextElement()!=it2.nextElement() &&
                it1.nextElement().comienzo == it2.nextElement().comienzo)
                return false;
        }
    }
    return true;
}
```

Clase Empresa

```
public class Empresa {
    private List<ContratoEnCurso> contratoEnCurso;
    public Empresa(Empleado e, int salario, int comienzo) {
        // necesita al menos un empleado y por tanto crea su contrato.
        // ha de ser a tiempo completo para garanzatizar el 60 % minimo
        contratoEnCurso = new ArrayList<ContratoEnCurso>();
        ContratoEnCurso c = new ContratoEnCurso(e,this,salario,comienzo,true);
        // no hace falta añadir c a la lista, se hace directamente al crear el contrato
    }
    protected void addContrato(ContratoEnCurso c) {
        contratoEnCurso.add(c);
    }
    protected void rmContrato(ContratoEnCurso c) {
        contratoEnCurso.remove(c);
    }
    public Enumeration<ContratoEnCurso> getContratoEnCurso() {
        return java.util.Collections.enumeration(contratoEnCurso);
    }
}
```


Clase Empresa (cnt'd)

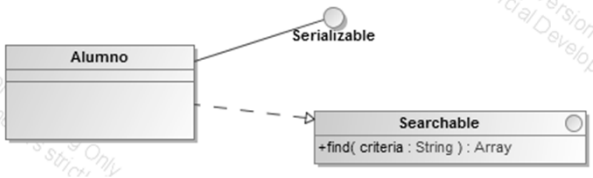
```
public class Empresa {
    private List<ContratoEnCurso> contratoEnCurso;
    public Empresa(Empleado e, int salario, int comienzo) {
        // necesita al menos un empleado y por tanto crea su contrato.
        // ha de ser a tiempo completo para garanzatizar el 60 % minimo
        contratoEnCurso = new ArrayList<ContratoEnCurso>();
        ContratoEnCurso c = new ContratoEnCurso(e,this,salario,comienzo,true);
        // no hace falta añadir c a la lista, se hace directamente al crear el contrato
    }
    protected void addContrato(ContratoEnCurso c) {
        contratoEnCurso.add(c);
    }
    protected void rmContrato(ContratoEnCurso c) {
        contratoEnCurso.remove(c);
    }
    public Enumeration<ContratoEnCurso> getContratoEnCurso() {
        return java.util.Collections.enumeration(contratoEnCurso);
    }
    // Métodos adicionales para comprobar restricciones
    protected boolean checkPC() {
        Enumeration<ContratoEnCurso> it = this.getContratoEnCurso();
        int tc = 0, total=0;
        while (it.hasMoreElements()) {total++;if (it.nextElement().aTiempoCompleto) tc++;}
        return (tc*100/total)>=60;
    }
}
```

Herencia
Clases abstractas

- Es conveniente factorizar los atributos y métodos comunes a varias clases
  - incluso añadiendo una nueva clase al diseño
  - aunque no sea posible implementar algún método
- Clase abstracta
  - define una interfaz que no implementa totalmente
  - puede incluir atributos y métodos concretos
  - se representan en *cursiva*

Herencia
Interfaces

- conjunto de signaturas de operaciones públicas
  - comportamiento común a varias clases disimilares
    - que no comparten implementación (atributos, métodos)
  - en UML permite definir atributos
    - en Java deben implementarse como operaciones
    - excepto si son constantes de clase (*final static*)



  - su uso potencia la flexibilidad y extensibilidad
  - permite sacar partido al polimorfismo

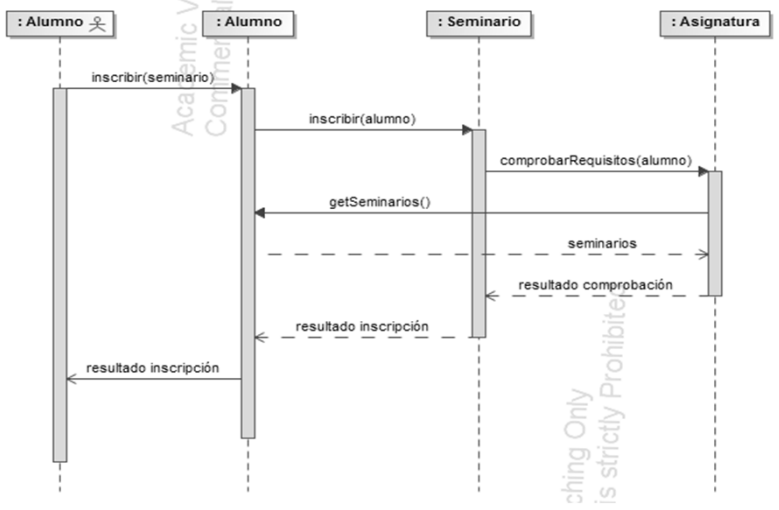
Diseño orientado a objetos
Comportamiento

- el modelo dinámico contiene información relevante para el diseño y la implementación
  - permite determinar la dirección de las asociaciones
  - completa la lista de operaciones de una clase
  - ayuda a definir operaciones auxiliares
  - determina argumentos, valores de retorno y excepciones
- hay que comprobar la consistencia
  - entre los distintos modelos y diagramas
  - con el modelo llamada/retorno del lenguaje de implementación elegido

Diseño del comportamiento
Diagramas de interacción

- si un objeto recibe un evento o mensaje
  - su clase debe definir una operación para responder a él
  - los eventos que emite a continuación determinan el esquema de implementación de la operación
- si un objeto emite un evento o mensaje
  - comprobaremos que tiene acceso al objeto receptor
    - recorriendo una asociación
    - por haberlo recibido previamente como argumento
    - por ser un objeto global (*static*)
  - algunos mensajes emitidos son respuesta a invocaciones anteriores
    - emparejaremos invocaciones y respuestas

Diagramas de interacción
Ejemplo



Diagramas de interacción
Implementación

En la clase Alumno:

```
public void inscribir(Seminario seminario)
  throws ExcepcionInscripcion
    {
    seminario.inscribir(this);
    }

public Enumeration<Seminario> getSeminarios()
    {
    return this.seminarios;
    }
```

Diagramas de interacción
Implementación

En la clase Seminario:

```
public void inscribir(Alumno alumno)
  throws ExcepcionInscripcion
    {
    if (!(getAsignatura().comprobarRequisitos(alumno)))
      throw new ExcepcionInscripcion(this,alumno);
    else
      addAlumno(alumno);
    }
```

Diseño orientado a objetos 47

Diagramas de interacción
Implementación

En la clase Asignatura:

```
public boolean comprobarRequisitos(Alumno alumno)
  {
  List<Seminario> seminarios = alumno.getSeminarios();
    if (...)
        return true;
     else
        return false;
    }
  return (...):
```

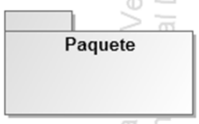
Diseño orientado a objetos 48

Diseño del comportamiento
Diagramas de estados

- indican las distintas formas en que un objeto responde a un evento
  - cada evento corresponde a una operación de la clase
    - las guardas y efectos indican la implementación
  - un mismo evento aparece varias veces en el diagrama
    - típicamente una vez por cada estado
    - todos los efectos se implementan *en la misma operación*
- estrategias de implementación
  - mediante comportamiento condicional
  - mediante tablas de estados
  - mediante el patrón de diseño Estado

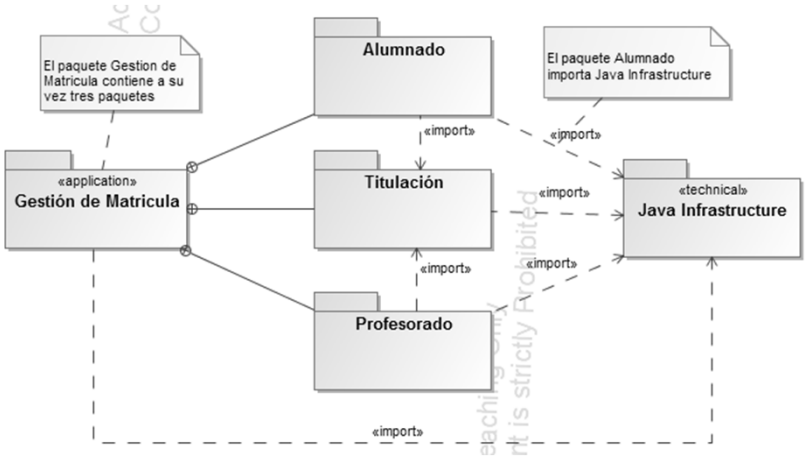
Diseño orientado a objetos
Empaquetamiento

- permite organizar elementos en grupos o paquetes
  - hace los diagramas más fáciles de entender
  - cualquier elemento puede ser empaquetado
    - modelos y diagramas
    - casos de uso
    - clases
      - reflejan la estructura física del código fuente
      - organizan en proceso de desarrollo
  - en UML los paquetes se representan mediante carpetas



Empaquetamiento
Diagramas de paquetes

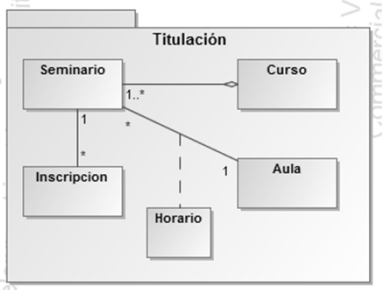
- muestran paquetes y dependencias entre paquetes



Diseño orientado a objetos 51

Diagramas de paquetes
Empaquetamiento de clases

- muestra la asignación de clases a paquetes
- suelen agruparse en el mismo paquete:
  - clases relacionadas mediante herencia
  - clases relacionadas mediante composición/agregación
  - clases que colaboran en los diagramas de interacción



Diseño orientado a objetos 52