Tema 1
Principios Fundamentales del Diseño Software

Pablo Sánchez

Dpto. Ingeniería Informática y Electrónica
Universidad de Cantabria
Santander (Cantabria, España)
p.sanchez@unican.es

Objetivos

Objetivos

1. Repasar el concepto de modularidad software.
2. Conocer y saber aplicar los principios GRASP.
3. Conocer y saber aplicar los principios SOLID.
4. Conocer y saber aplicar las técnicas de Diseño por Contrato.

Bibliografía Básica

Craig Larman.
Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
Prentice Hall, 3 edition, October 2004.

Robert C. Martin and Micah Martin.
Agile Principles, Patterns and Practices in C#.
Prentice Hall, July 2006.

Bertrand Meyer.
Object-Oriented Software Construction.
Prentice Hall, March 2000.

Índice

1. Introducción.
2. Modularidad.
3. Principios GRASP.
4. Principios SOLID.
5. Diseño por Contrato.
6. Sumario.

¿Por Qué Estudiar los Principios del Diseño Software?

- El caso de los monos, los plátanos y la escalera. (ver)
- Papá, ¿por qué somos del Atlético? (ver)

Los principios fundamentales del diseño software permiten conocer por qué las cosas son como son, es decir, por qué somos del Atlético.

Índice

1. Introducción.
2. Modularidad.
3. Principios GRASP.
4. Principios SOLID.
5. Diseño por Contrato.
6. Sumario.

Módulos Software

Módulo Software

Un *módulo software* es un conjunto de datos y operaciones que ofrecen una funcionalidad determinada de forma precisa a través de una cierta *interfaz*.

Principio de Ocultación [Parnas, 1972]

Cada módulo software debe proporcionar los detalles relevantes para su utilización a la vez que oculta aquellos detalles y decisiones que sean irrelevantes para su utilización.

Propiedades Ideales de una Descomposición Modular

Cohesión

Medida del grado de relación existentes entre los diversos elementos pertenecientes a un mismo módulo software.

Acoplamiento

Medida del grado de dependencia de un módulo software respectos a otros módulos software.

Propiedades deseables de un módulo software

1. Está débilmente acoplado a otros módulos (cambios en otros módulos no le afectan).
2. Posee una alta cohesión (sirve a un único propósito).
3. Tiene una *granularidad adecuada*.

Beneficios de las Descomposiciones Modulares

1. Desarrollo paralelo e independiente de cada módulo.
2. Sustitución de módulos (e.g. interfaz de escritorio por web).
3. Posibilidad de pruebas independientes.
4. Desarrollo de aplicaciones por ensamblado de módulos.

Índice

1. Introducción.
2. Modularidad.
3. Principios GRASP.
4. Principios SOLID.
5. Diseño por Contrato.
6. Sumario.

GRASP

GRASP

*General Responsability Assignment Software Patterns**

(*) Prefiero el término *Principle* en lugar de *Pattern*

Concepto de Responsabilidad

Responsabilidad de Acción Hacer algo, invocar comportamientos, controlar y coordinar comportamientos.

Responsabilidad de Conocimiento Datos propios, datos derivados y objetos relacionados.

Lista de Principios GRASP

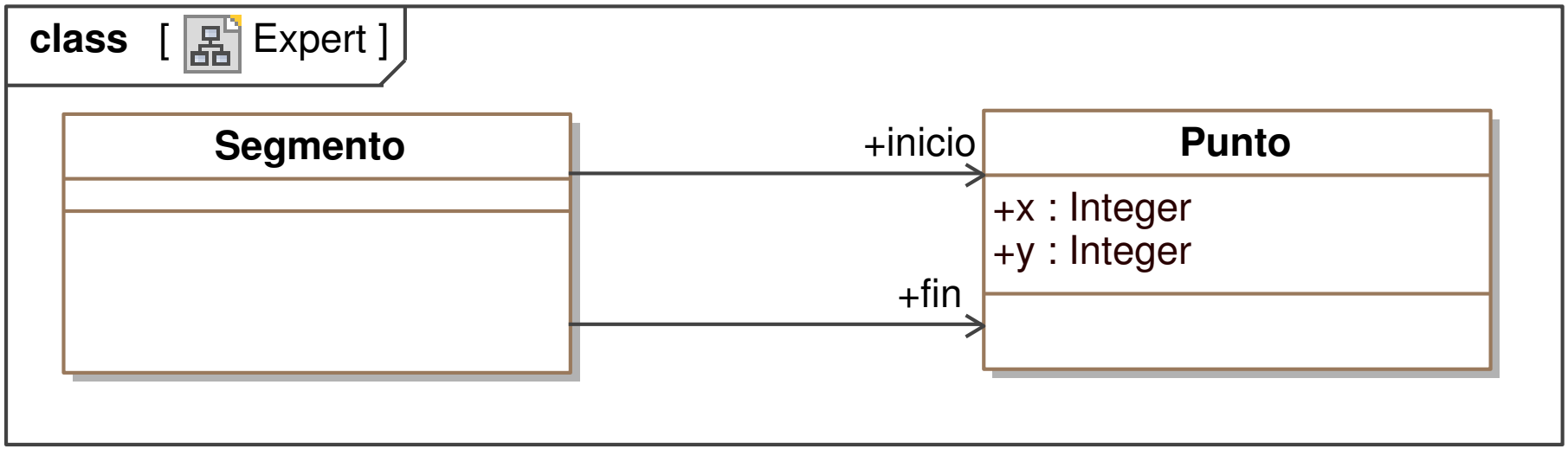
1. Expert
2. Creator
3. High cohesion
4. Low coupling
5. Controller
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations (Don't Talk to Strangers)

GRASP: Experto

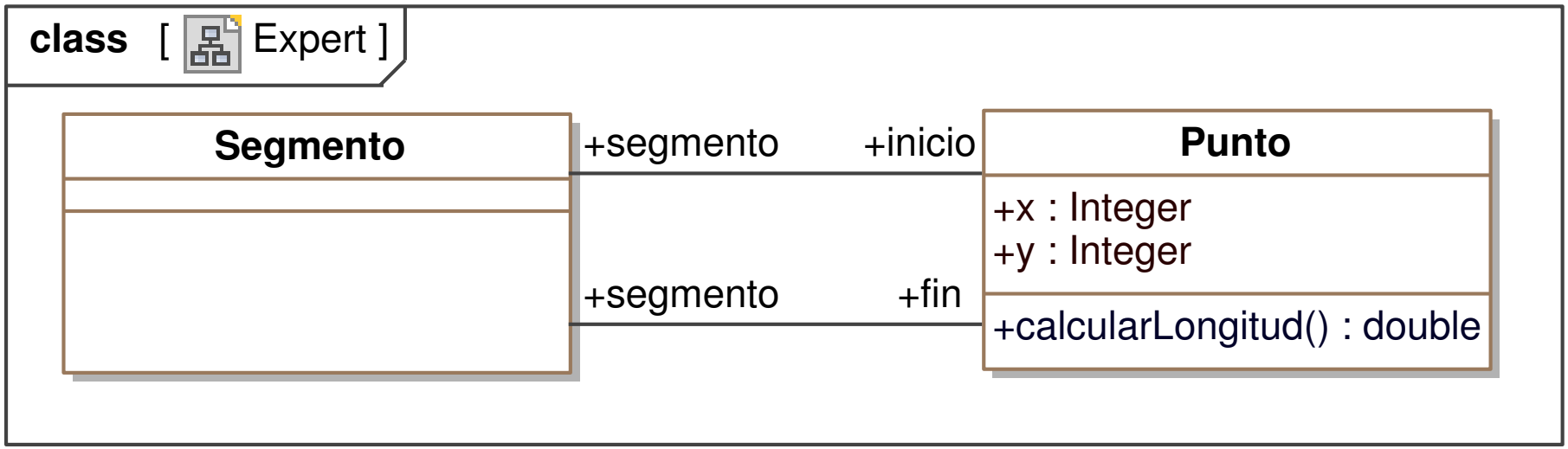
Principio Experto

Asignar cada responsabilidad al objeto que tiene la información necesaria para llevarla a cabo, que será el *experto* en esa información.

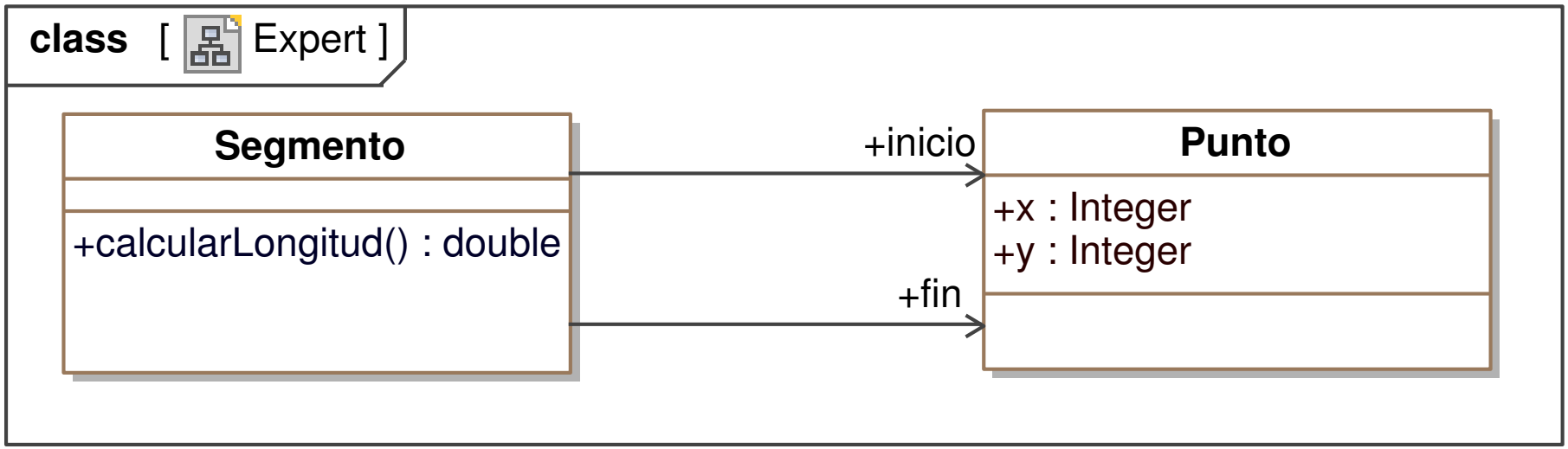
GRASP: Experto



GRASP: Experto



GRASP: Experto



GRASP: Ventajas Experto

1. Aumenta la cohesión, ya que el experto tiene la información para realizar la tarea.
2. Disminuye el acoplamiento, ya que no hay que acoplar las clases responsables de realizar la tarea con el experto.

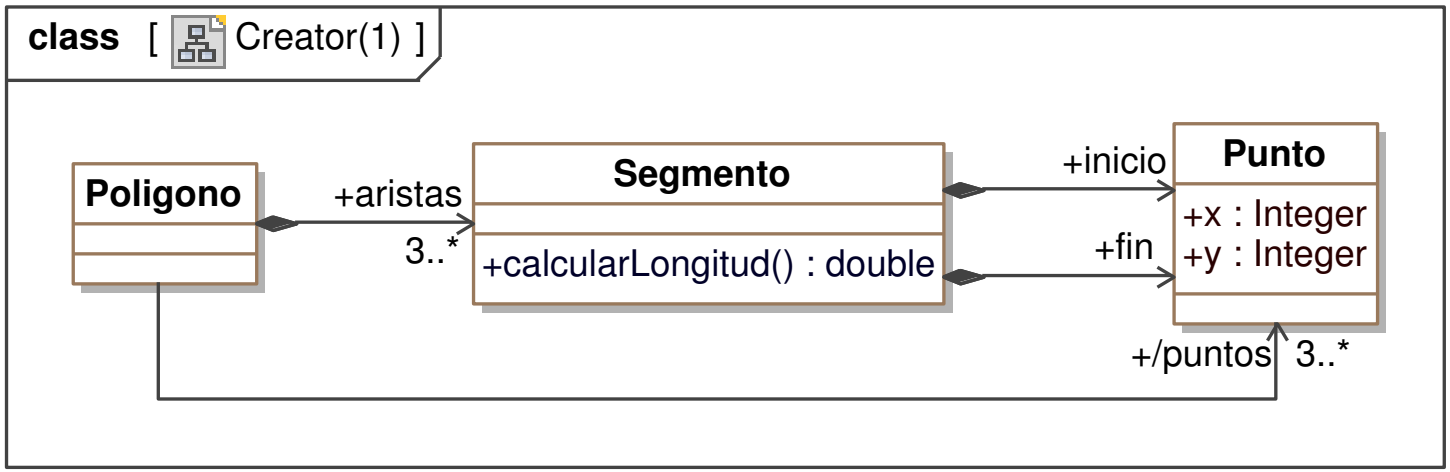
GRASP: Creador

Principio Creador

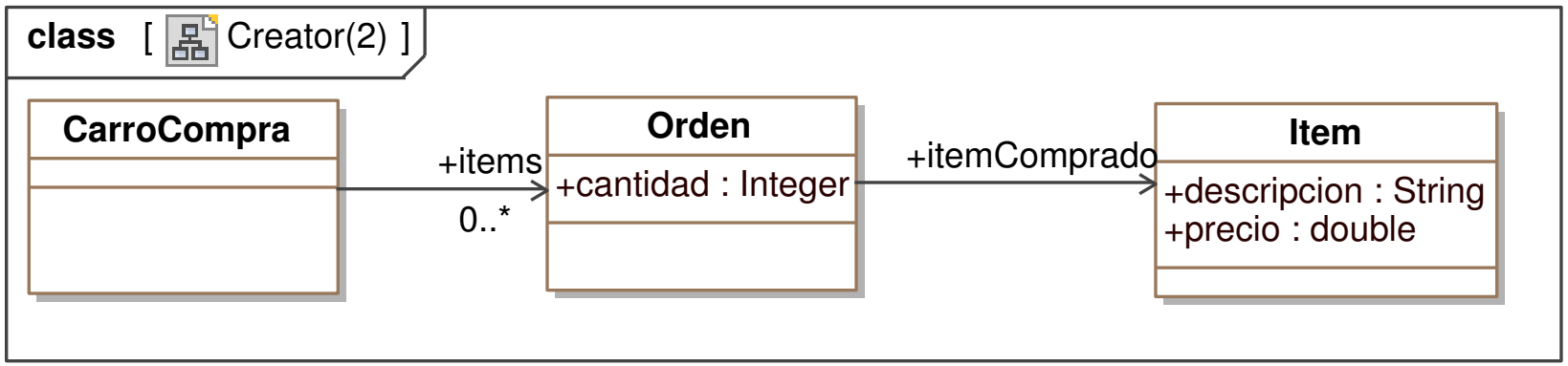
Una clase B debe ser responsable de crear objetos de una clase A si se satisface alguno de los siguientes elementos (por orden de preferencia):

1. Los objetos de la clase B se forman por agregación de objetos de la clase A.
2. Los objetos de la clase B contiene objetos de la clase A.
3. Los objetos de la clase B almacenan a los objetos de la clase A.
4. Los objetos de la clase B usan con bastante frecuencia objetos de la clase A.
5. Los objetos de la clase B tienen los datos necesarios para crear los objetos de la clase A.

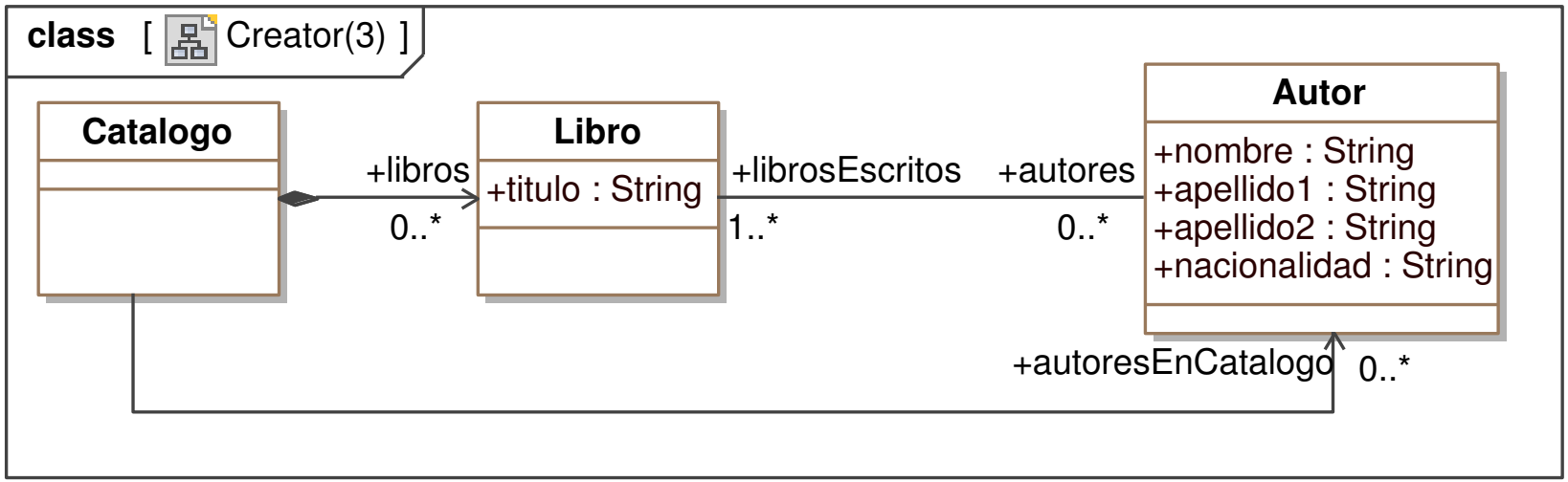
GRASP: Creador



GRASP: Creador



GRASP: Creador



GRASP: Ventajas Creador

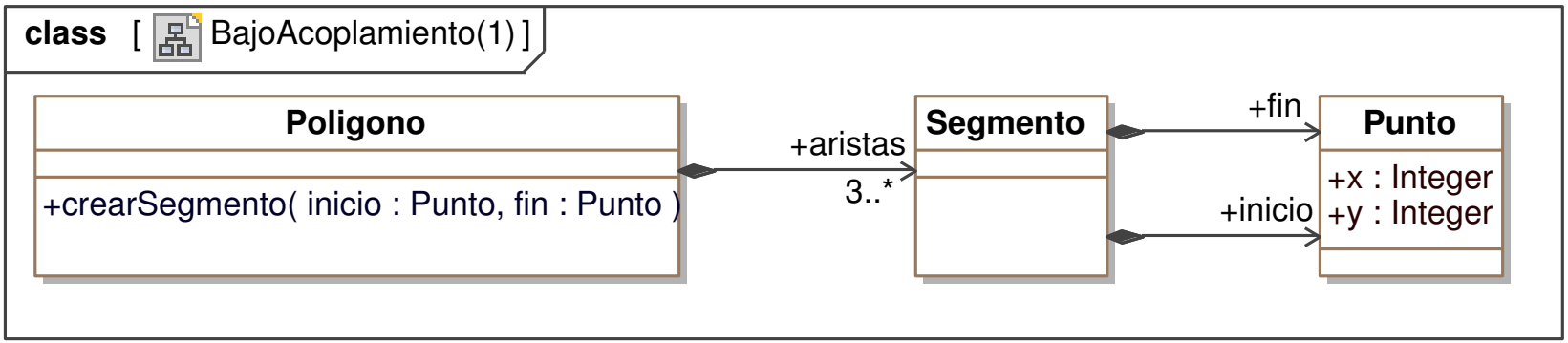
1. Disminuye el acoplamiento ya que la clase que crea el objeto iba a estar acoplada de algún modo u otro a la clase a la cual pertenecen los objetos creados.

GRASP: Bajo Acoplamiento

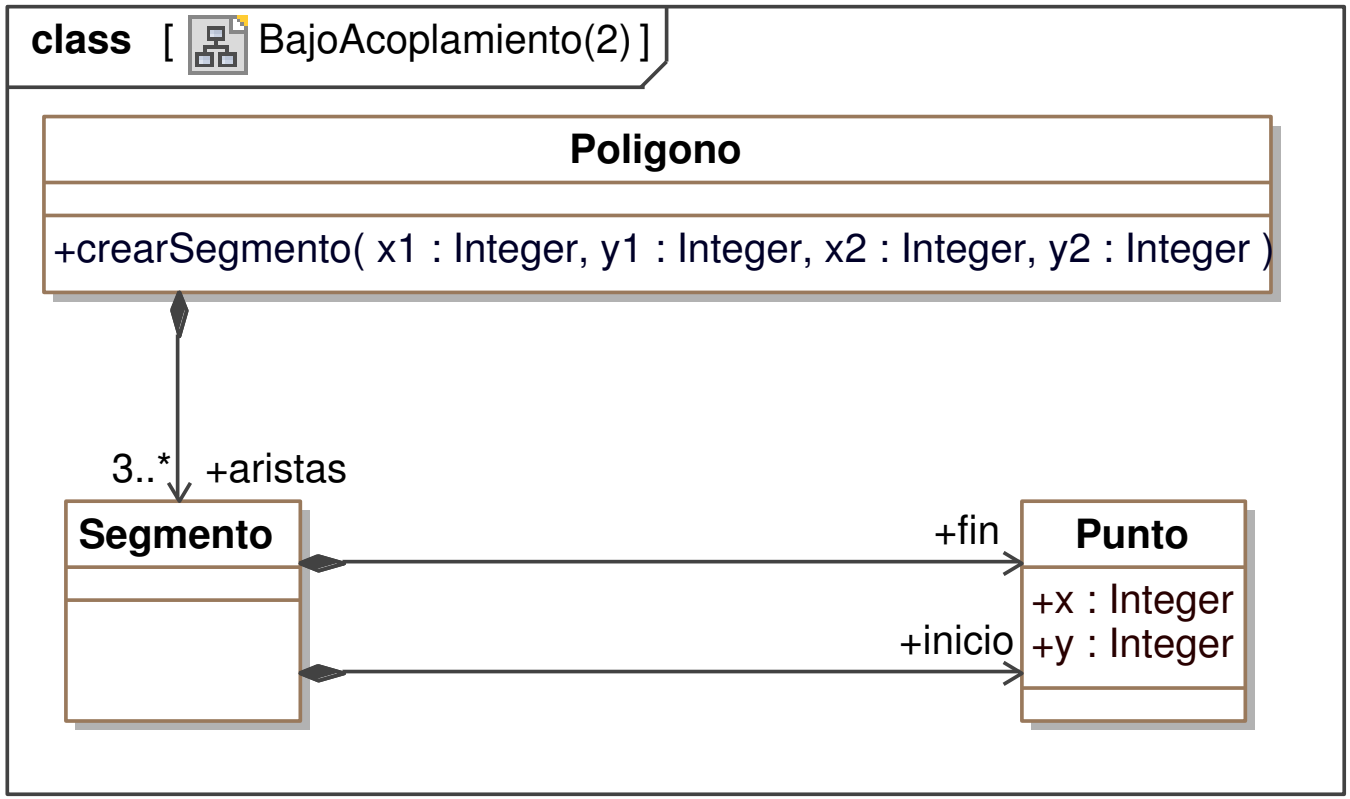
Principio Bajo Acoplamiento

Las responsabilidades se deben asignar de forma que se minimice el acoplamiento.

GRASP: Bajo Acoplamiento



GRASP: Bajo Acoplamiento



GRASP: Bajo Acoplamiento

Formas de Acoplamiento

1. Una clase A tiene una referencia a una clase B.
2. Una clase A tiene un método con objetos de una clase B como parámetros.
3. Una clase A usa como variable local referencias a una clase B.
4. Una clase A es subclase (directa o indirecta) de una clase B.

GRASP: Ventajas Bajo Acoplamiento

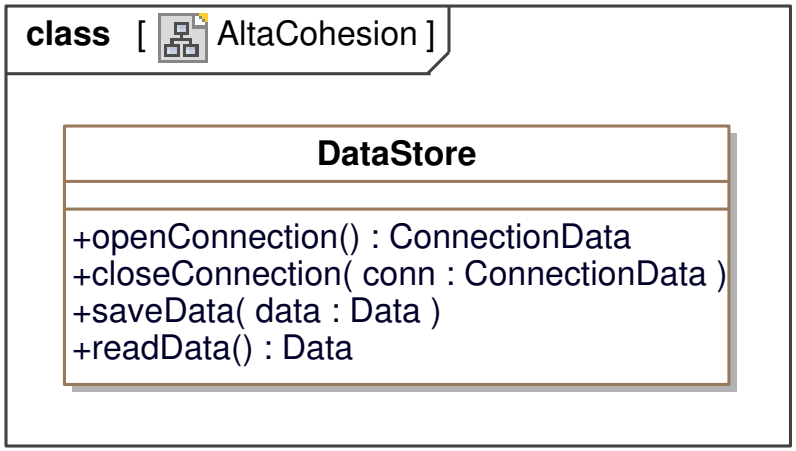
1. Un bajo acoplamiento mejora la estabilidad de los diseños.
2. Mejora la reutilización al disminuir las dependencias.
3. Mejora la facilidad de comprensión de cada módulo.

GRASP: Alta Cohesión

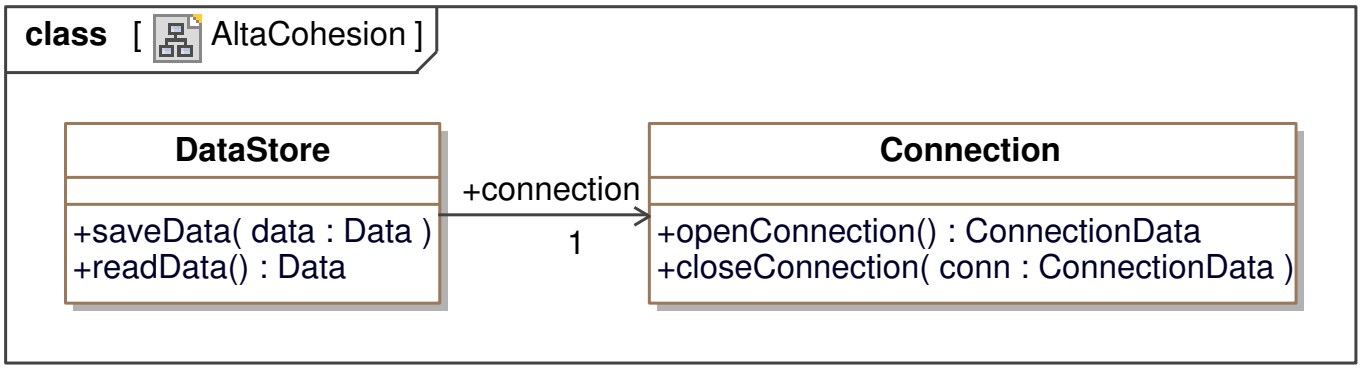
Principio Alta Cohesión

Las responsabilidades se deben asignar de forma que se maximice la cohesión.

GRASP: Alta Cohesión



GRASP: Alta Cohesión



GRASP: Alta Cohesión

Grados de Cohesión

Muy baja cohesión Una clase monolítica realiza muchas responsabilidades y con diferentes propósitos.

Baja cohesión Una clase realiza demasiadas responsabilidades de propósito similar.

Cohesión moderada Una clase tiene un número bajo de responsabilidades, las cuales corresponden a un mismo área o propósito, pero están débilmente relacionadas entre ellas.

Alta Cohesión Una clase tiene pocas responsabilidades, las cuales corresponden a un mismo área o propósito.

GRASP: Ventajas Alta Cohesión

1. Mejora la reutilización de piezas individuales.
2. Ayuda a disminuir el acoplamiento medio de los módulos.
3. Mejora la facilidad de comprensión de cada módulo.

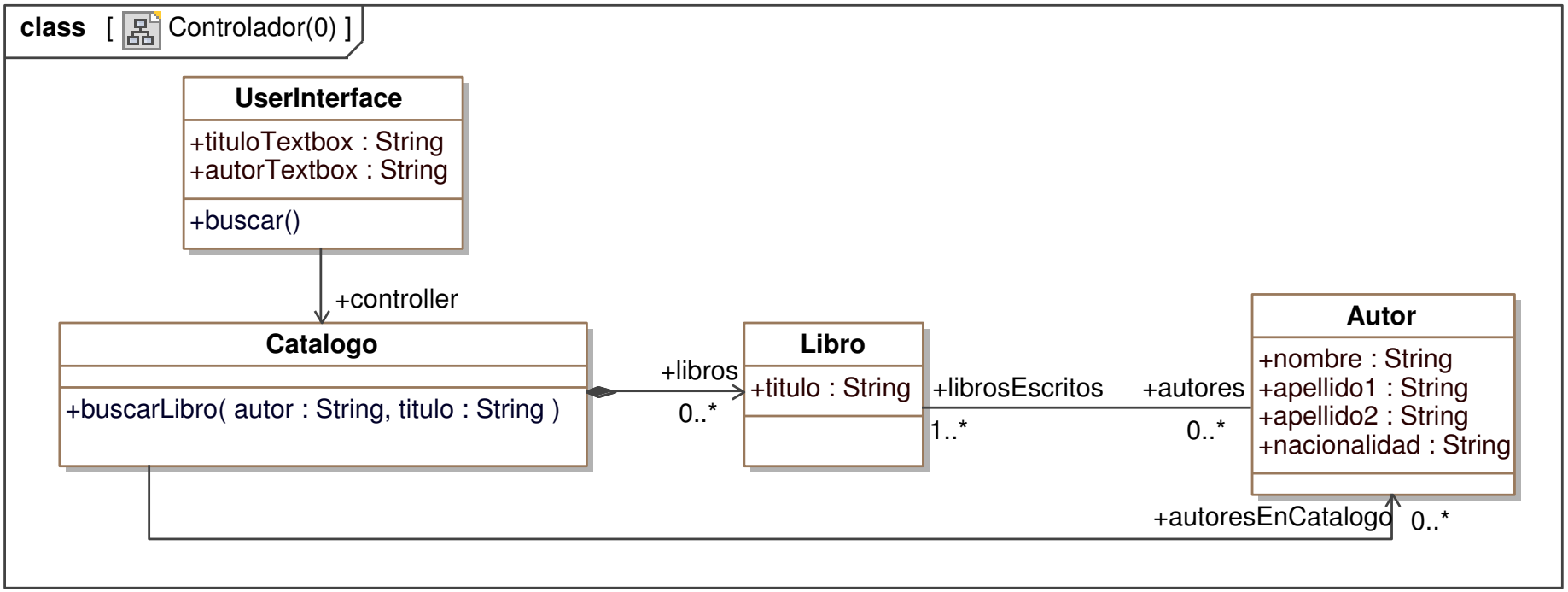
GRASP: Controlador

Principio Controlador

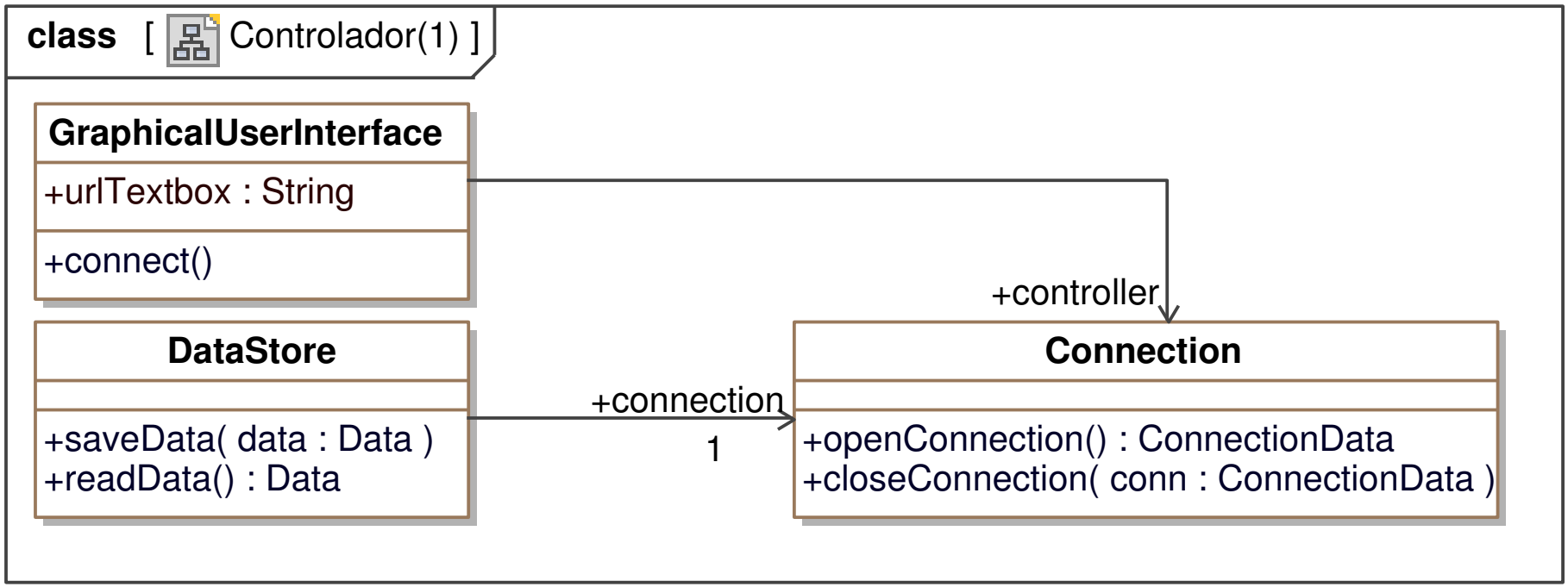
La responsabilidad de gestionar un evento del sistema debe corresponder a una clase *A* que satisfaga alguna de las siguientes condiciones:

1. La clase *A* representa el sistema u organización como un todo (*controlador fachada*).
2. La clase *A* representa un elemento del mundo real que está involucrado en la resolución de dicha responsabilidad (*controlador de rol*).
3. La clase *A* representa un gestor artificial para todos los eventos de un determinado caso de uso (*controlador de caso de uso*).

GRASP: Controlador



GRASP: Controlador



GRASP: Ventajas Controlador

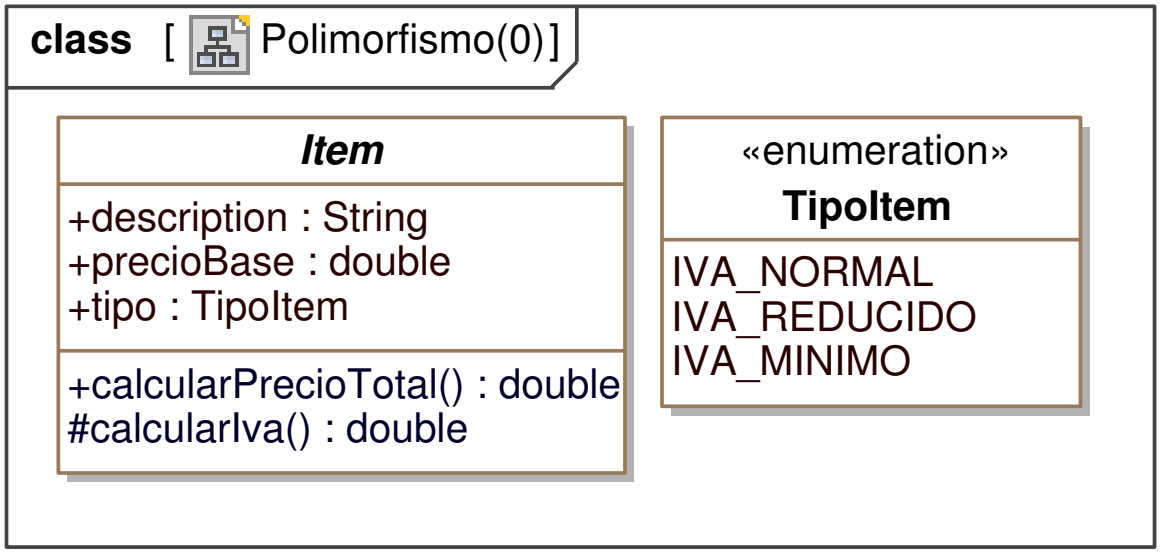
1. Favorece la reutilización.
2. Mejora el control del estado de la aplicación.

GRASP: Polimorfismo

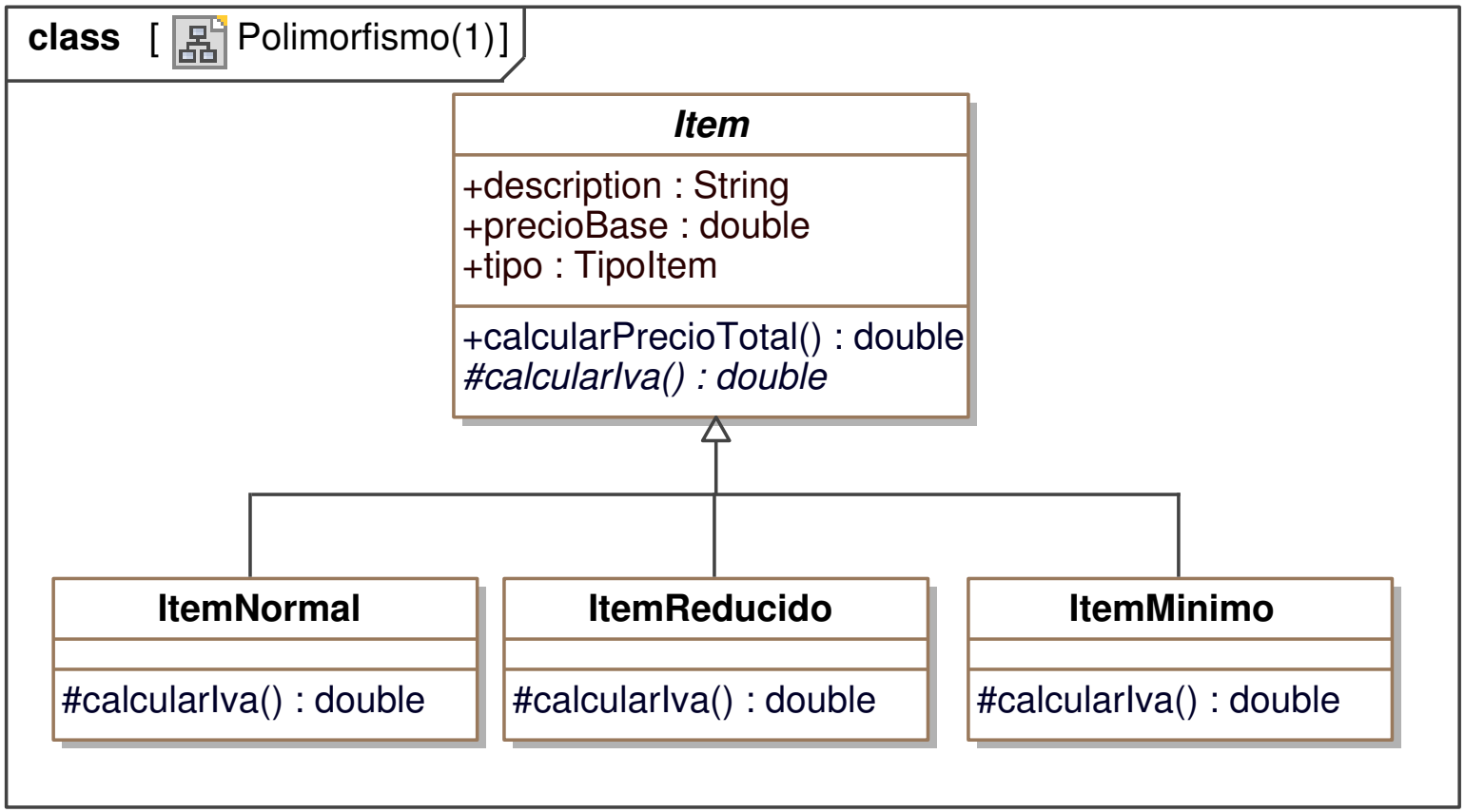
Principio Polimorfismo

Si una responsabilidad varia dependiendo del tipo que debe realizarla, asignar una responsabilidad por cada tipo para el cual la responsabilidad varíe.

GRASP: Polimorfismo



GRASP: Polimorfismo



GRASP: Ventajas Polimorfismo

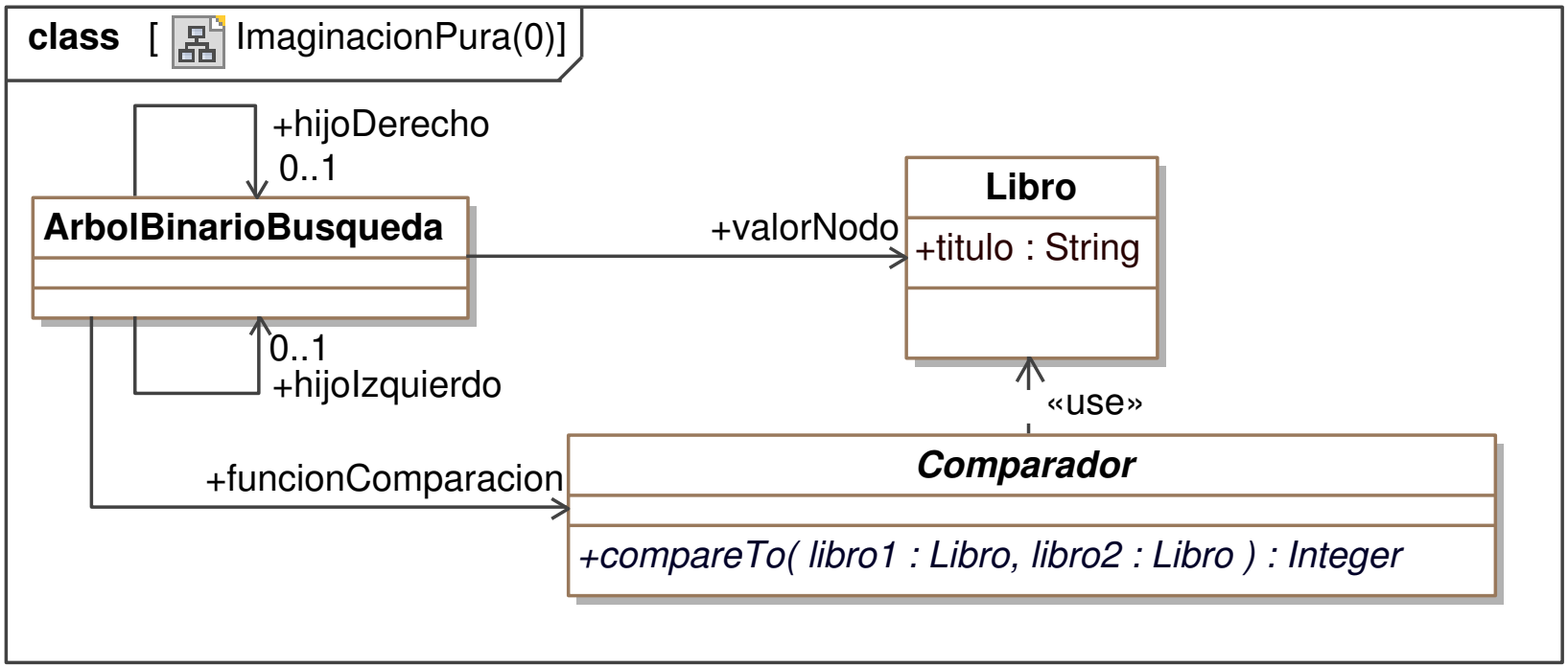
1. Mejora la cohesión.
2. Mejora la estabilidad del diseño.
3. Mejora la escalabilidad.

GRASP: Imaginación Pura

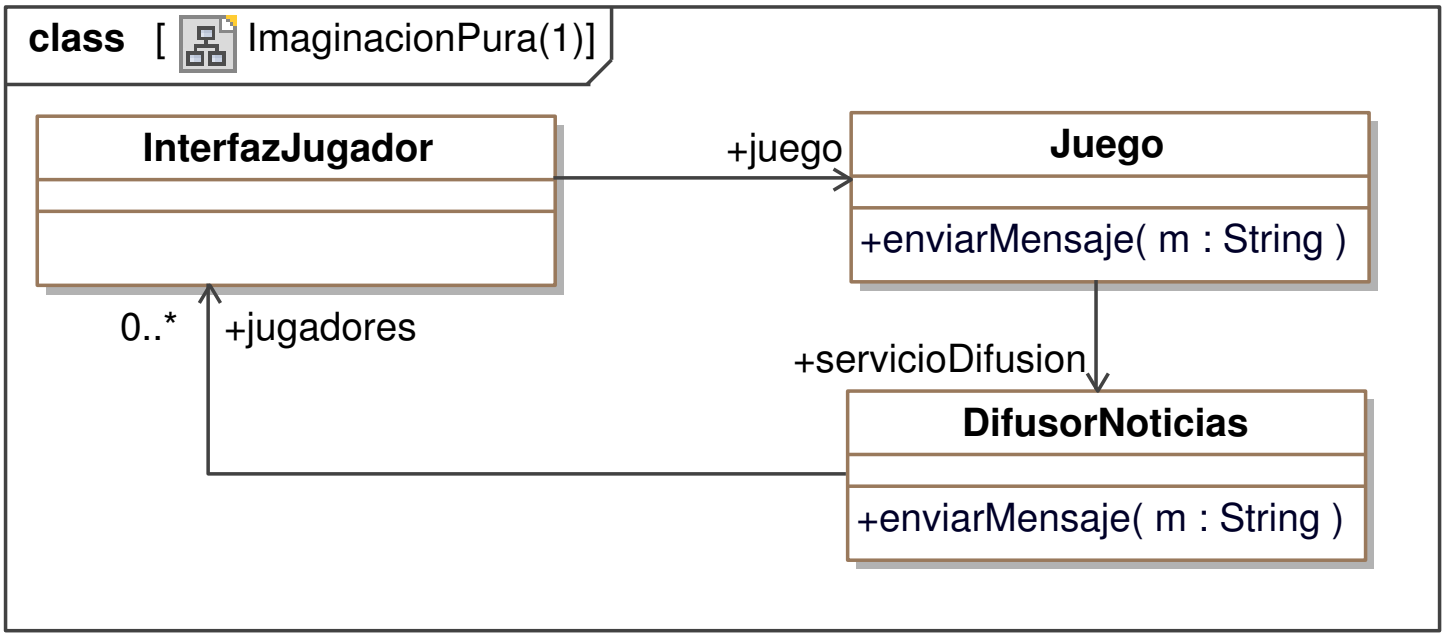
Principio Imaginación Pura

Asignar un conjunto fuertemente cohesionado de responsabilidades a una clase que no tiene correspondencia alguna con ningún elemento real del dominio del problema con objeto de aumentar la cohesión, disminuir el acoplamiento y/o incrementar la reutilización.

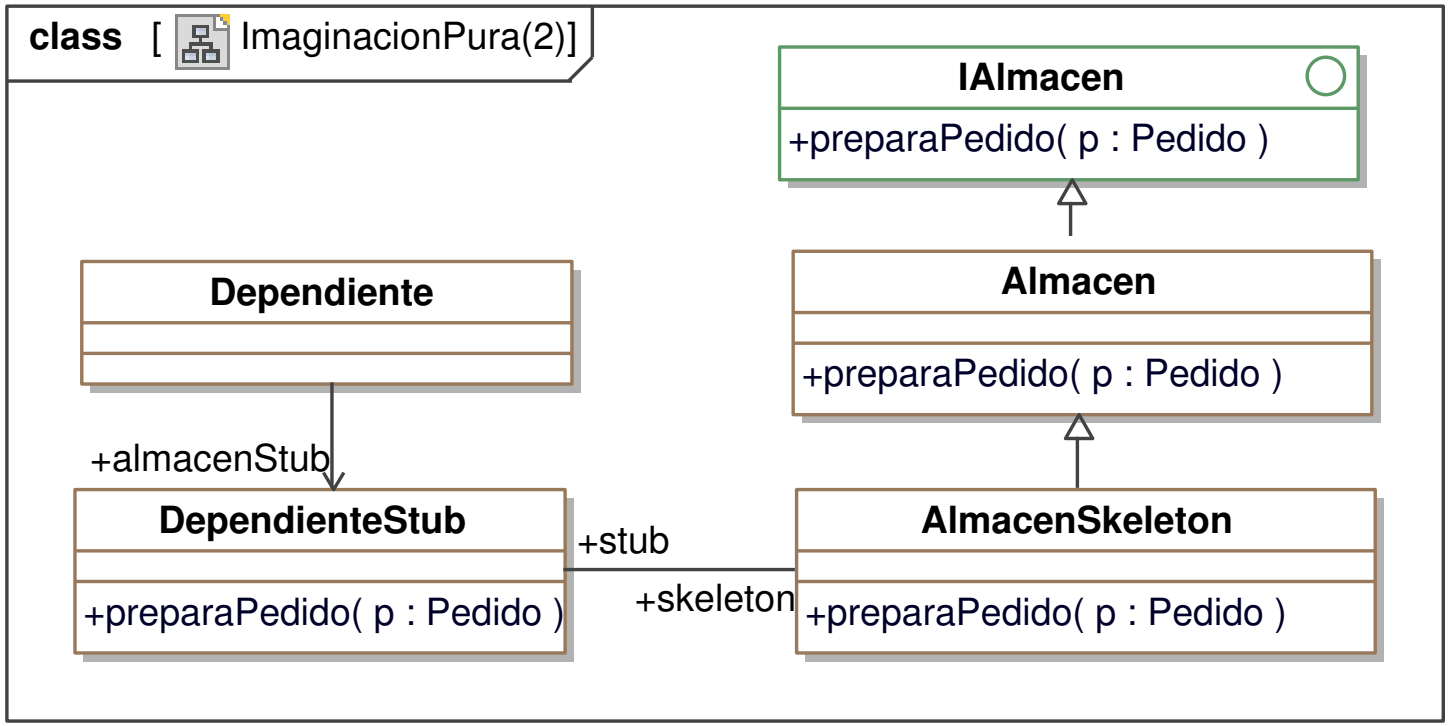
GRASP: Imaginación Pura



GRASP: Imaginación Pura



GRASP: Imaginación Pura



GRASP: Ventajas Imaginación Pura

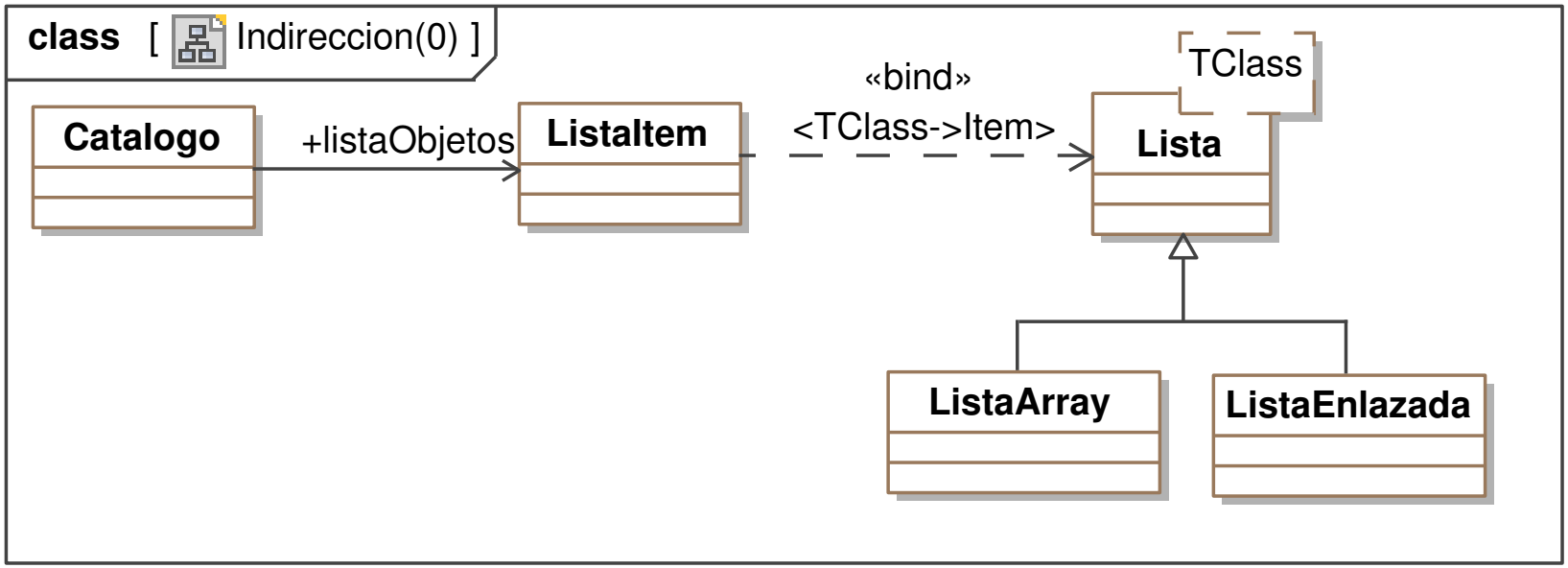
1. Mejora la cohesión.
2. Mejora la estabilidad del diseño.
3. Favorece la reutilización.

GRASP: Indirección

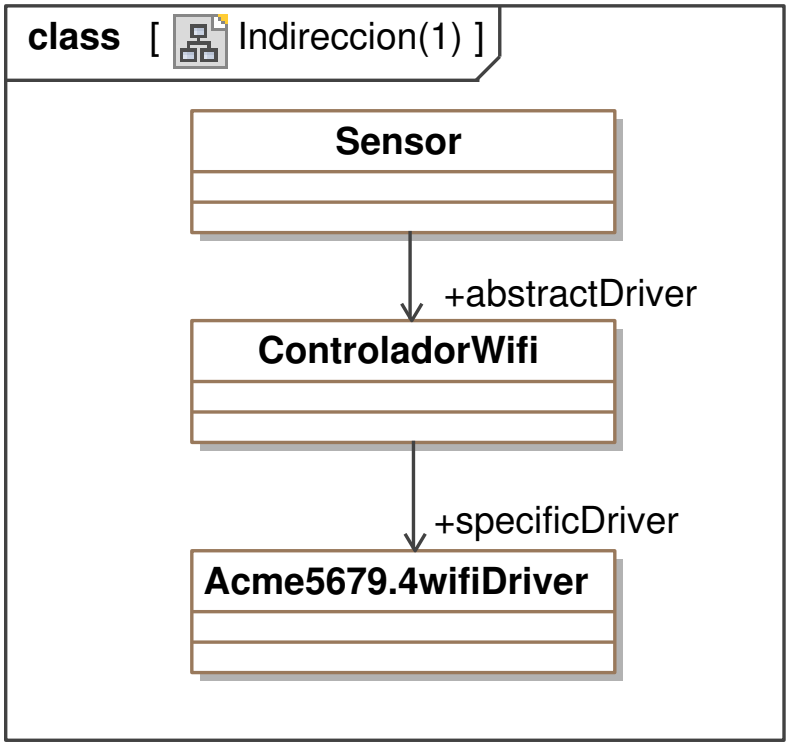
Principio Indirección

Asignar una responsabilidad a un módulo intermedio de forma que medie entre otros módulos, evitando que estos últimos queden acoplados.

GRASP: Indirección



GRASP: Indirección



GRASP: Ventajas Indirección

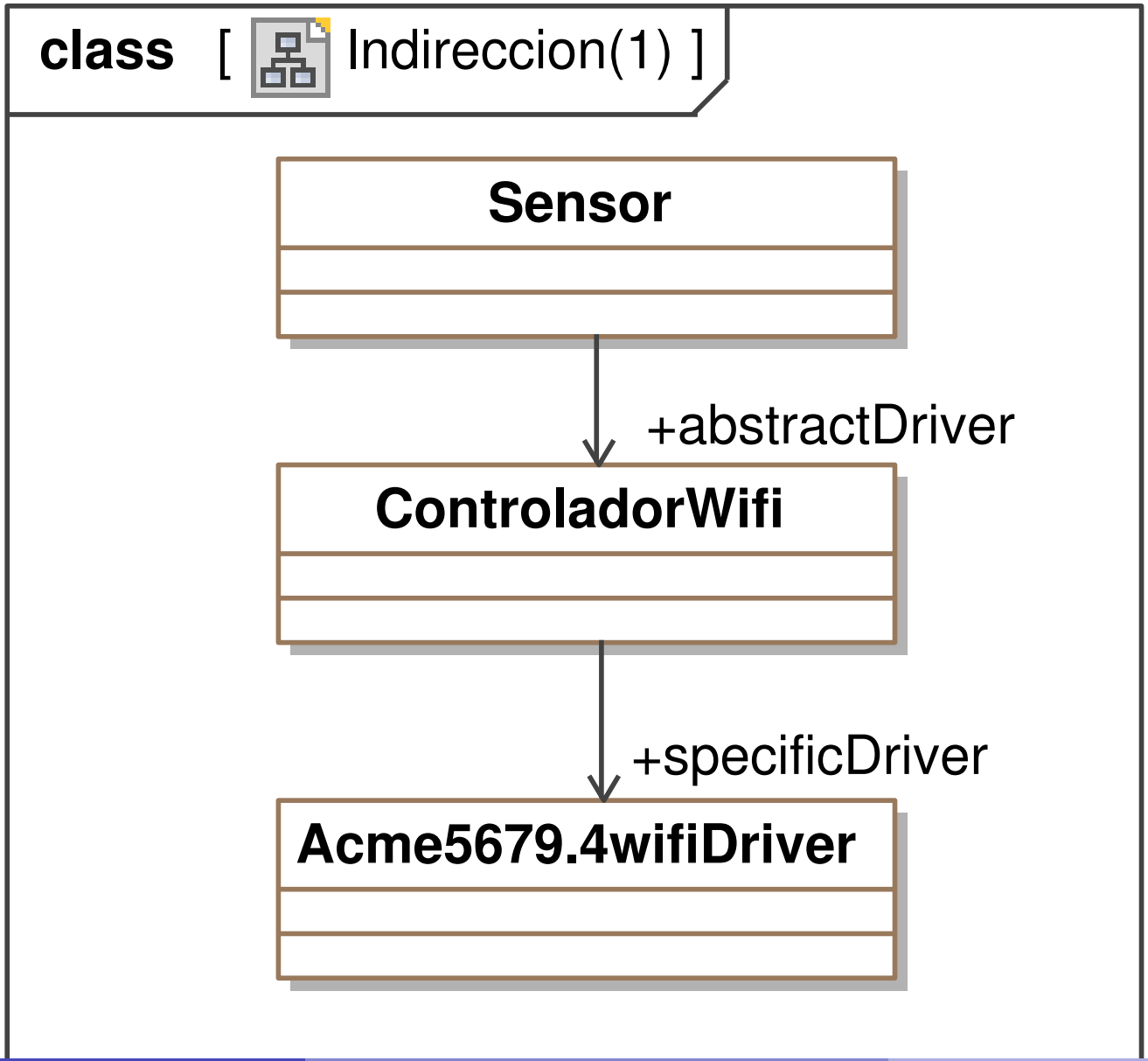
1. Reduce el acoplamiento.
2. Mejora la estabilidad del diseño.

GRASP: Acotación de Variaciones

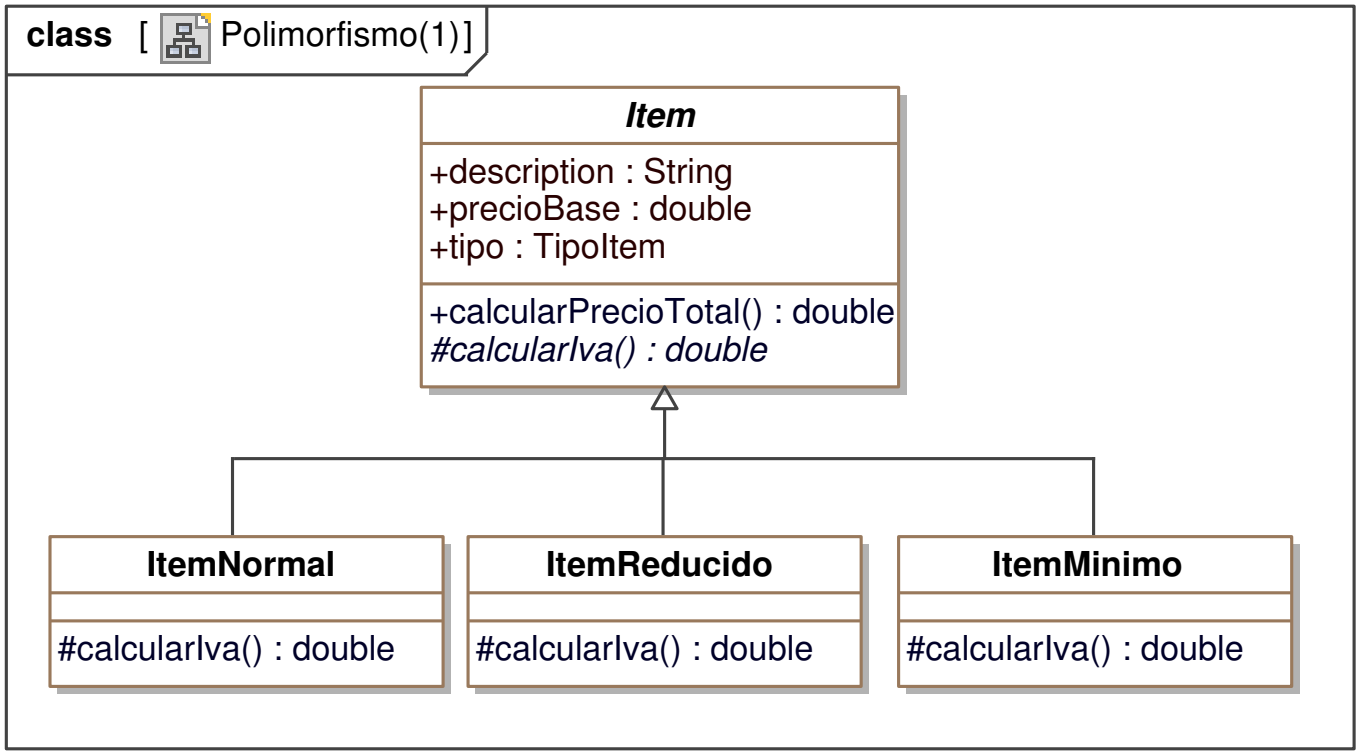
Principio de Acotación de Variaciones

Identificar los puntos de variación de un diseño. A continuación, crear interfaces estables que permitan acomodar futuras variaciones sin afectar al diseño existente.

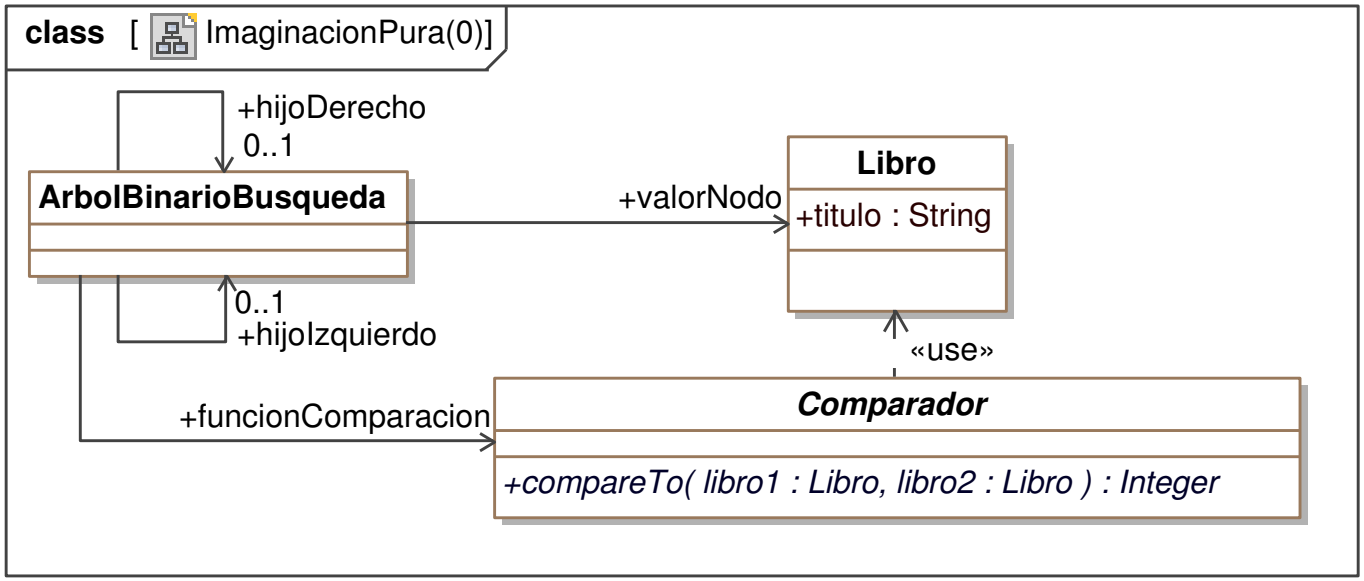
GRASP: Acotación de Variaciones



GRASP: Acotación de Variaciones



GRASP: Acotación de Variaciones



GRASP: Acotación de Variaciones

1. Mejora la estabilidad del diseño.
2. Favorece la reutilización y evolución.
3. Sólo introducir puntos de variación acotado cuando su coste esté justificado.

GRASP: No hables con extraños

Principio No hables con extraños

Asignar una responsabilidad a una clase B que esté naturalmente acoplada a una clase A, cuando la clase A necesite comunicarse con una clase indirecta C que esté naturalmente acoplada a B, con el objetivo de evitar que A y C se tengan que comunicar de forma directa.

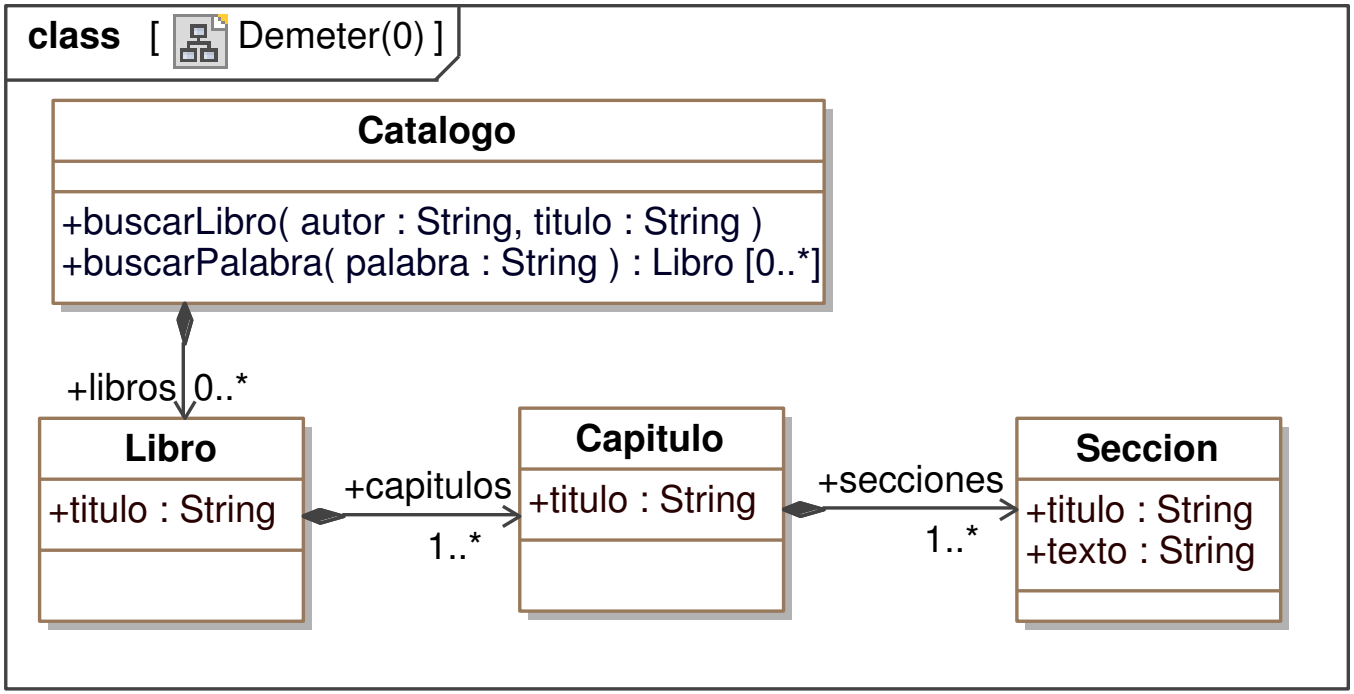
GRASP: No hables con extraños

Ley de Demeter [Lieberherr and Holland, 1989]

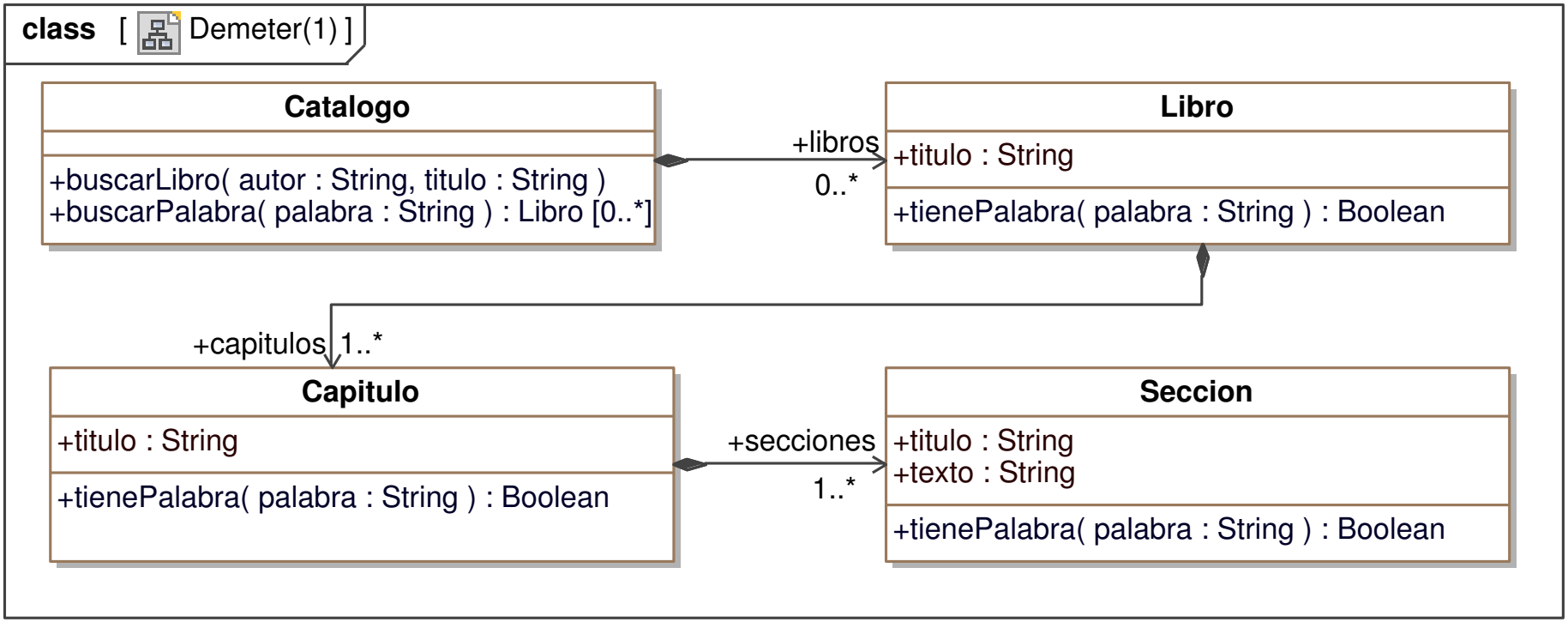
Una método de un objeto X de una clase A sólo debería invocar métodos de objetos que satisfagan las siguientes restricciones:

1. Es el propio objeto X (*this* o *self*).
2. Es un parámetro del método.
3. Es un atributo del objeto X.
4. Un elemento de una colección que es un atributo de X.
5. Un objeto creado dentro del método.

GRASP: No hables con extraños



GRASP: No hables con extraños



GRASP: No hables con extraños

1. Reduce el acoplamiento.
2. Mejora la estabilidad del diseño.

Índice

1. Introducción.
2. Modularidad.
3. Principios GRASP.
4. Principios SOLID.
5. Diseño por Contrato.
6. Sumario.

Principios SOLID

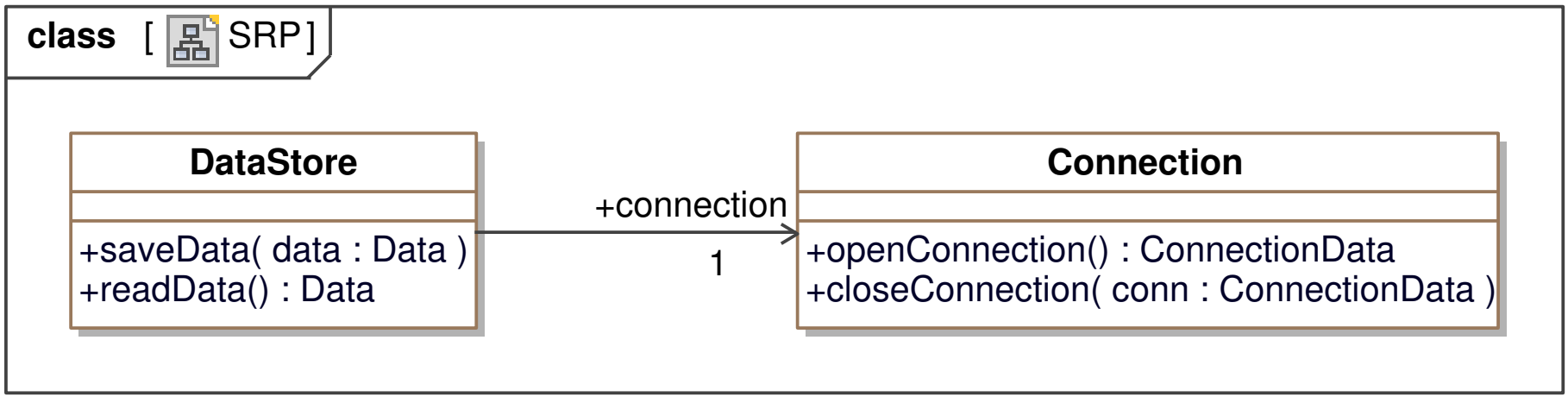
Prinicipios SOLID

1. Single-Responsability Principle.
2. Open-Closed Principle.
3. Liskov-Substitution Principle.
4. Interface-Segregation Principle.
5. Dependency-Inversion Principle.

SOLID: Responsabilidad Única

Principio de Responsabilidad Única

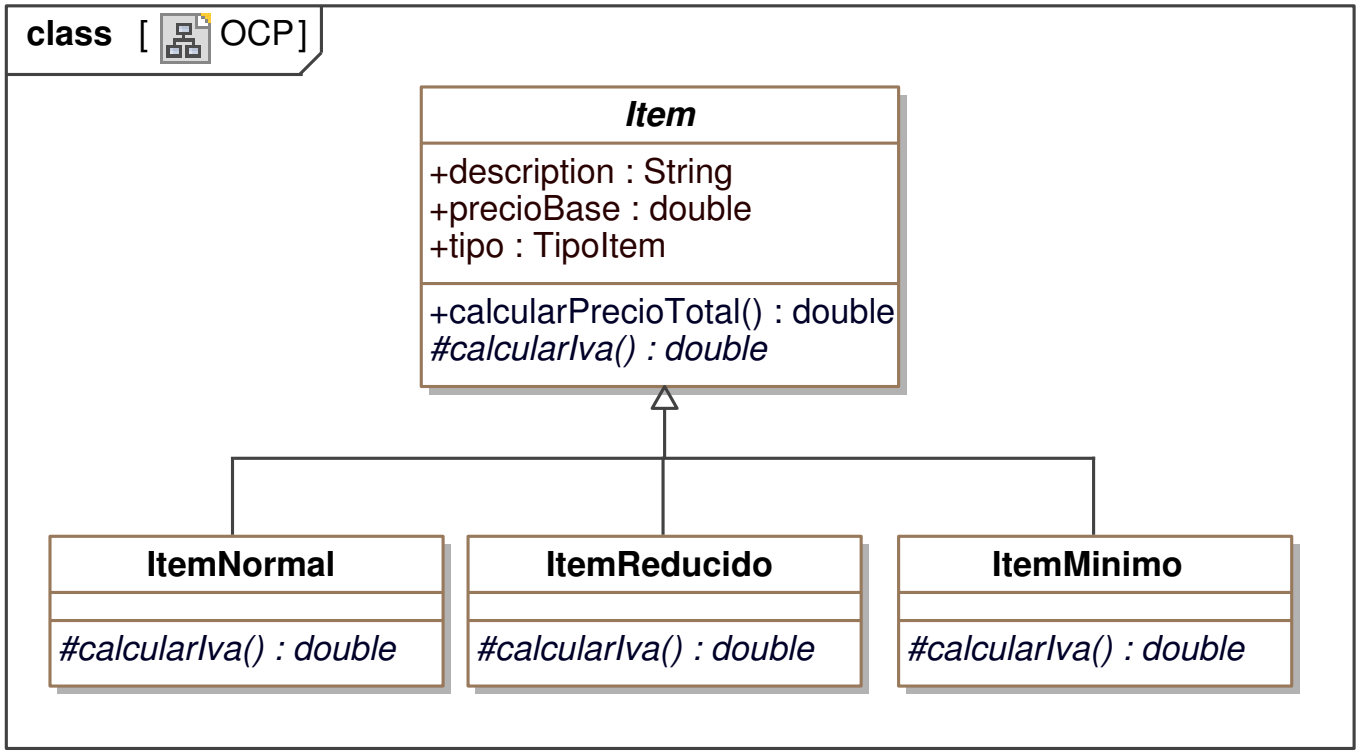
Una clase *A* debe servir a una única responsabilidad.



SOLID: Principio Abierto y Cerrado

Principio Abierto y Cerrado

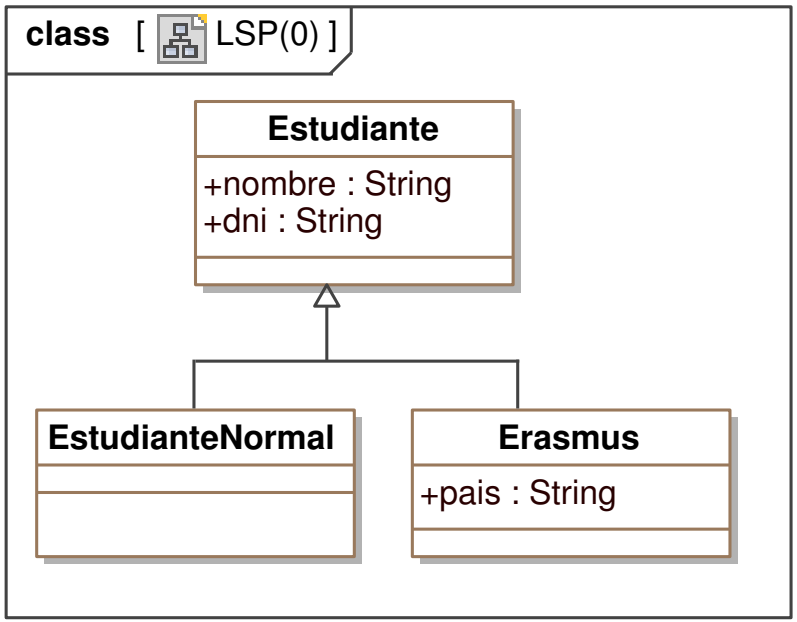
Una clase *A* debe estar abierta para ser extendida pero cerrada a posibles modificaciones.

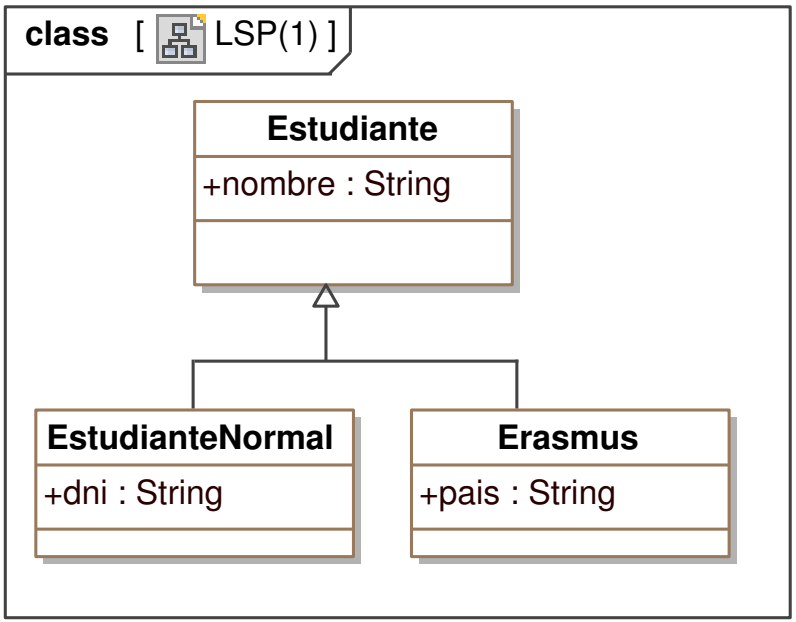


SOLID: Principio de Sustitución de Liskov

Principio de Sustitución de Liskov

Un objeto de una clase *A* debe poder aparecer en cualquier lugar donde se requiera un objeto de cualquier superclase de *A*.

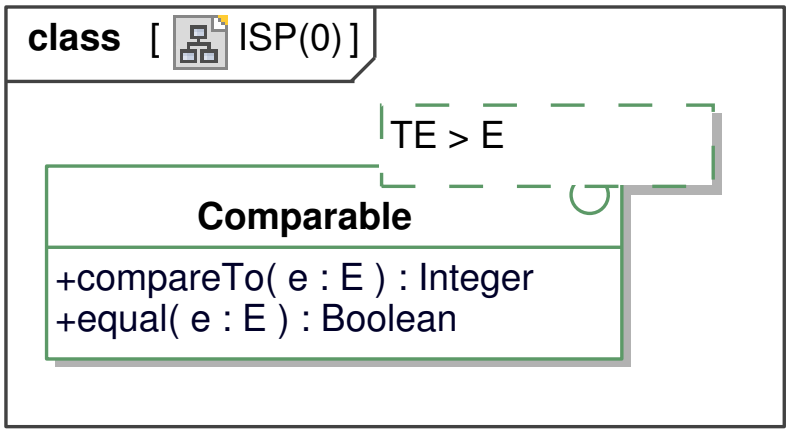


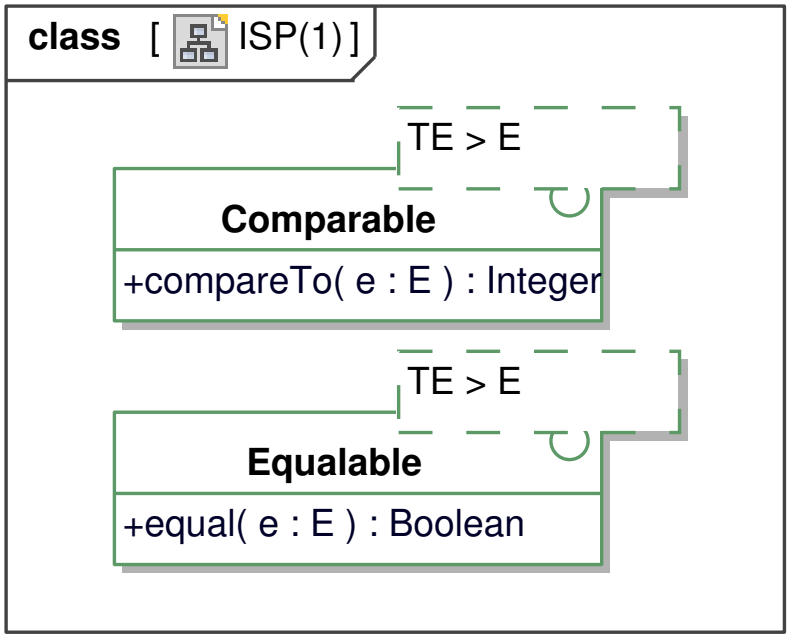


SOLID: Principio de Separación de Interfaces

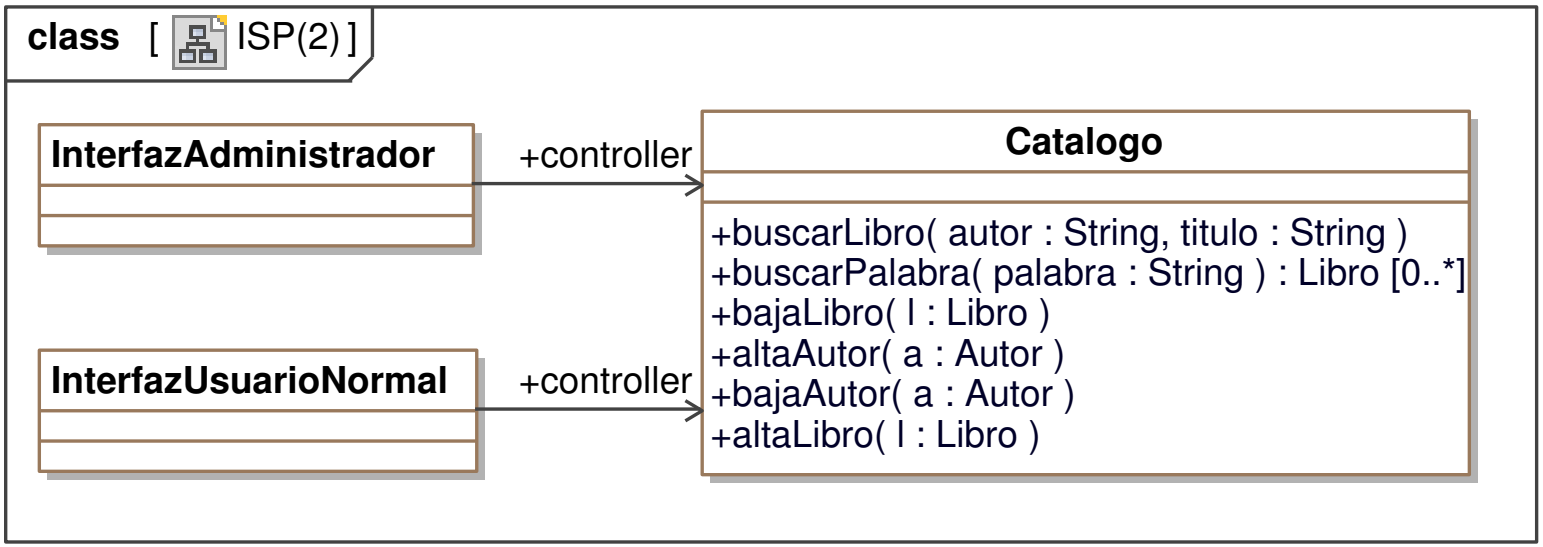
Principio de Separación de Interfaces

Tratar de crear interfaces específicas y concretas mejor que interfaces de propósito general.

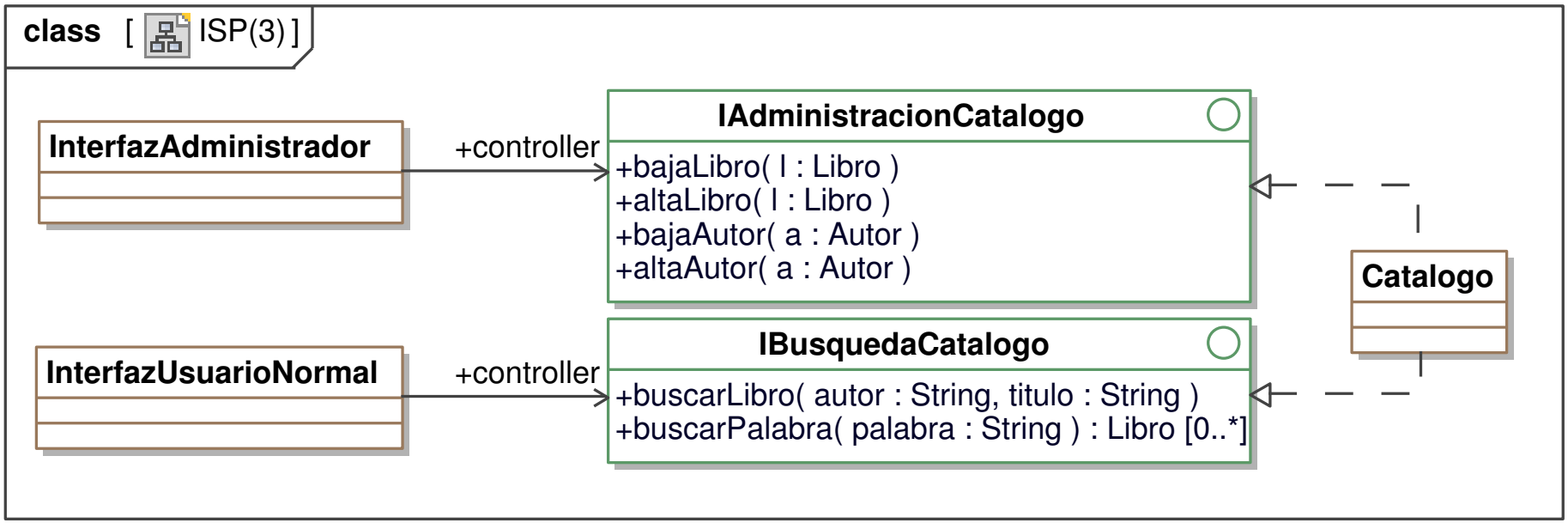




SOLID: Principio de Separación de Interfaces



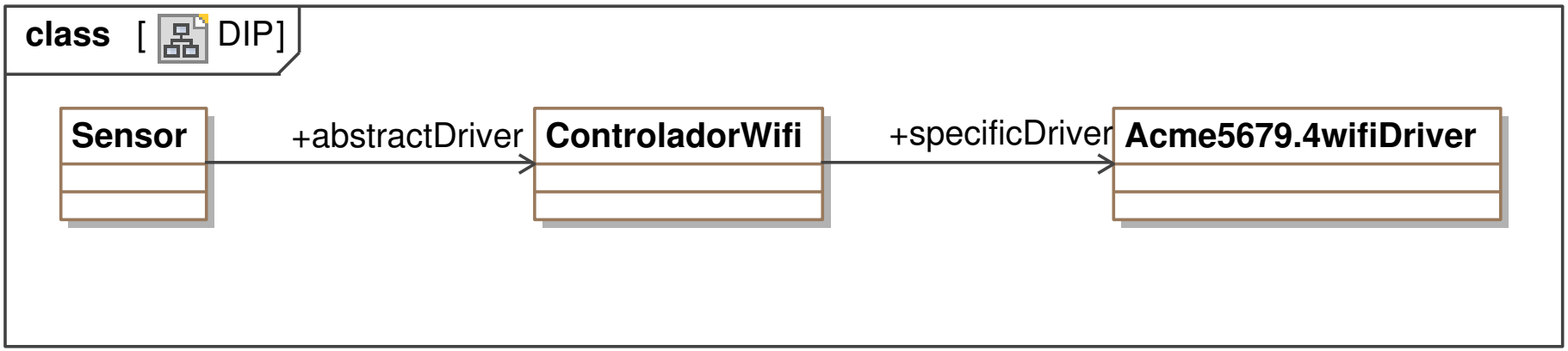
SOLID: Principio de Separación de Interfaces



SOLID: Principio de Inversión de Dependencias

Principio de Inversión de Dependencias

Las clases de alto nivel no deberían depender de las clases de bajo nivel.



Índice

1. Introducción.
2. Modularidad.
3. Principios GRASP.
4. Principios SOLID.
5. Diseño por Contrato.
6. Sumario.

Técnicas de Control de Errores

```
public class MatrizCuadrada {

  int [][] matriz = null;

  public MatrizCuadrada(int dimension) {
        matriz = new int[dimension][dimension];
  } // Matriz

  public void set(int i, int j, int valor) {
        this.matriz[i][j] = valor;
  } // set

  public int get(int i, int j) {
        return matriz[i][j];
  } //get

  public int getDimension() {
        return this.matriz.length;
  } // dimension

} // Matriz
```

Técnica del Avestruz

```
1  public void set(int i, int j, int valor) {
2    if ((0 <= i) && (i < matriz.length) &&
3            (0 <= j) && (j < matriz[0].length)) {
4            matriz[i][j] = valor;
5    } // if
6  } // set
```

Lanzamiento de Excepciones

```
1  public void set(int i, int j, int valor) {
2    if ((0 <= i) && (i < matriz.length) &&
3            (0 <= j) && (j < matriz[0].length)) {
4                  matriz[i][j] = valor;
5    } else {
6          throw new IndexOutOfBoundsException();
7    } // if
8  } // set
```

Lanzamiento de Excepciones

```
public void set(int i, int j, int valor) {
  if ((0 <= i) && (i < matriz.length) &&
          (0 <= j) && (j < matriz[0].length)) {
                matriz[i][j] = valor;
  } else {
          System.out.println("ERROR: MatrizCuadrada.set : " +
                "Índices (" + i + ", " + j + ") fuera de rango ");
  } // if
} // set
```

Contrato

Contrato de un método

Desde un punto de vista formal, un contrato de un método M se define como una tripleta $\{P\}M\{Q\}$, donde P y Q son predicados lógicos que reciben el nombre de *precondición* y *postcondición* respectivamente.

El significado de un contrato es "siempre que se invoque al método M de forma que la precondición se satisfaga, es decir, que P sea verdadero, el método M termina y a su finalización la postcondición Q es verdadera".

Ejemplo de Contrato

```
1  // Pre: ((0 <= i) AND (i < this.getDimension())) AND
2  //      ((0 <= j) AND (j < this.getDimension()))$
3  public void set(int i, int j, int valor) {
4      ..
5  } // set
6  // Post: valor == this.get(i,j)
```

Ejemplo de Código Libre de Errores

```
public static void init(MatrizCuadrada m) {
  for(int i=0;i<m.getDimension();i++) {
        for(int j=0;j<m.getDimension();j++) {
              m.set(i,j,0);
        } // for
  } // for
} // init
```

Principio de No Redundancia

Bajo ningún concepto debe el cuerpo de un método verificar el cumplimiento de la precondición de la rutina.

Sumario

1. Técnica que permite decidir a quién corresponde la responsabilidad de controlar los errores.
2. Permite crear código muy *limpio* y eficiente, con mínimas redundancias.
3. Si siempre se satisfacen las precondiciones podemos asegurar que la aplicación no fallará por errores internos del software.

Índice

1. Introducción.
2. Modularidad.
3. Principios GRASP.
4. Principios SOLID.
5. Diseño por Contrato.
6. Sumario.

¿Qué tengo que saber de todo esto?

1. Saber y entender qué es la modularidad.
2. Comprender y saber aplicar los principios GRASP.
3. Comprender y saber aplicar los principios SOLID.
4. Comprender y saber aplicar el Diseño por Contrato.

Referencias

Lieberherr, K. J. and Holland, I. M. (1989).
Assuring Good Style for Object-Oriented Programs.
IEEE Software, 6(5):38–48.

Parnas, D. L. (1972).
On the Criteria to Be Used in Decomposing Systems into Modules.
Communications of the ACM, 15(12):1053–1058.