



Principios de diseño GRASP

Elizabeth Suescún Monsalve, PhD
esuescu1@eafit.edu.co

Agenda

✓ ~~S.O.L.I.D~~

✓ GRASP

APPLYING UML AND PATTERNS

An Introduction to Object-Oriented Analysis and Design
and Iterative Development

THIRD EDITION



"People often ask me which is the best book to introduce them to the world of OO design.
Ever since I came across it, *Applying UML and Patterns* has been my unreserved choice."

—Martin Fowler, author of *UML Distilled* and *Refactoring*

CRAIG LARMAN

Foreword by Philippe Kruchten

Inspira Crea Transforma

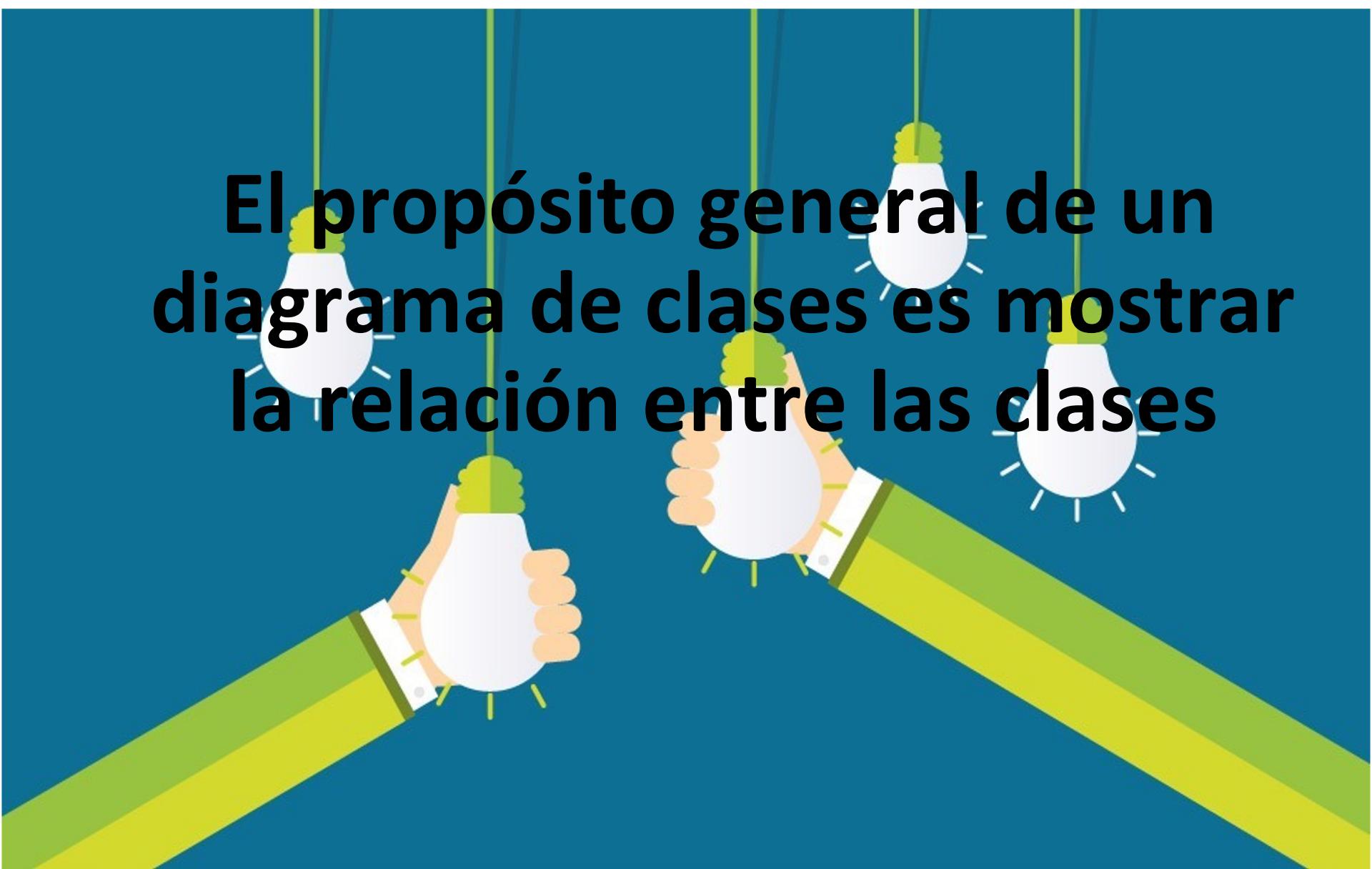
UNIVERSIDAD
EAFIT[®]

¿Qué es una clase?

Definición de las **propiedades** y **comportamiento** de un **tipo de objeto concreto**.

La **instanciación** es la lectura de esas definiciones y la **creación** de un objeto a partir de ella.

**El propósito general de un
diagrama de clases es mostrar
la relación entre las clases**

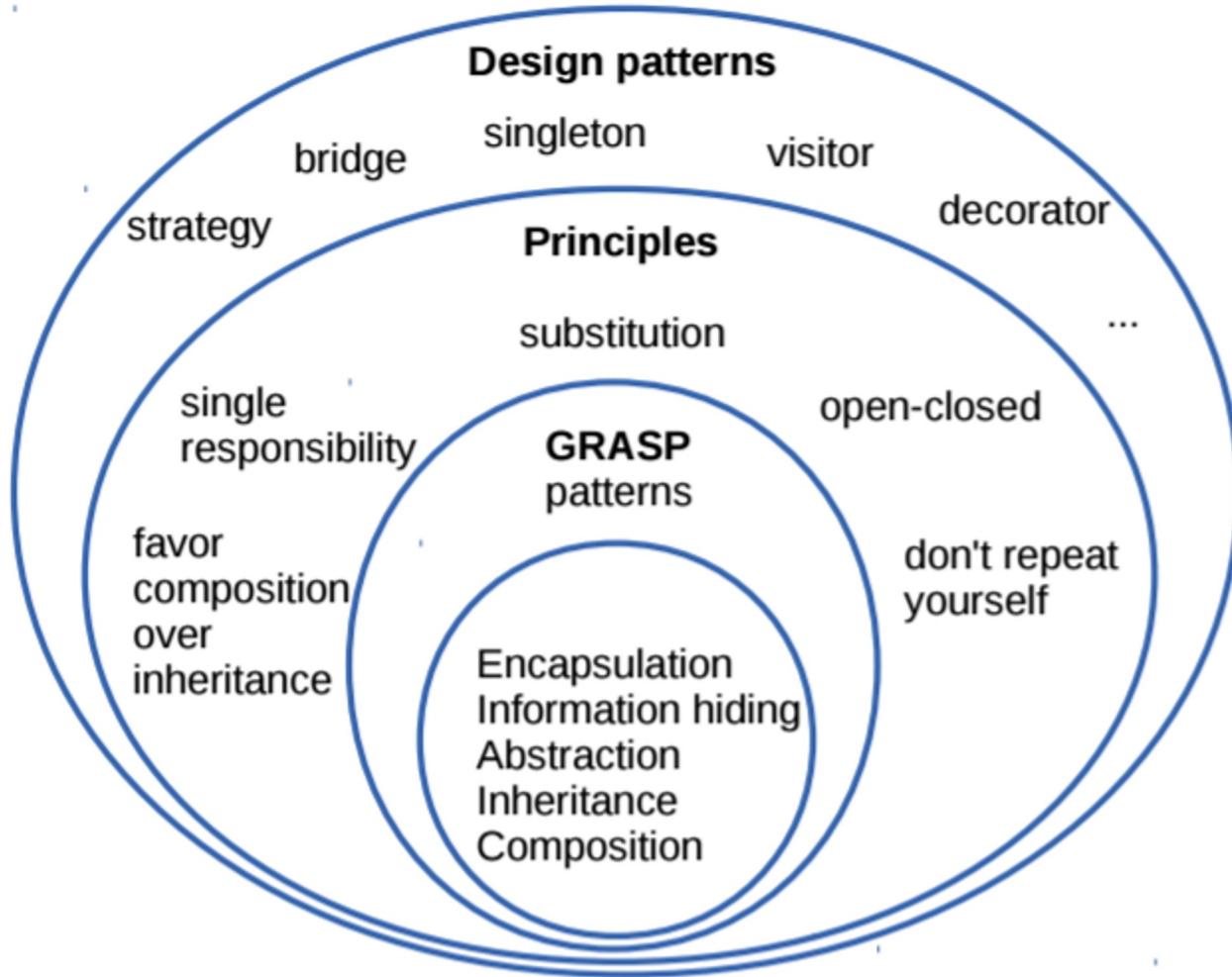


Vala
Magik
Ada
R
Java
Harbour
Clipper
VB.NET
Action Script
Visual Objects
PHP5
Perl
AJAX

x86asm++
JavaScript
Eiffel
Delphi
ABL
ABAP
Python
D
Objective-C
Clarion
Gambas
Visual FoxPro 6

C++
C#
Ruby
Visual Basic 6
Clipper

GRASP



Tomado de :http://www.cvc.uab.es/shared/teach/a21291/temes/object_oriented_design/slides/handouts/GRASP_patterns.pdf

Patrones para asignar responsabilidades



- Un sistema orientado a **objetos** se compone de objetos y el **paso de mensajes** a otros objetos para que lleven a cabo las **operaciones**.
- En los contratos se incluye una conjetura inicial sobre las **responsabilidades** y las **poscondiciones** de las operaciones. por ejemplo: *inicio*, *introducirProducto*, *terminarVenta* y *efectuarPago*.
- Una implementación hábil se fundamenta en los principios **cardinales** que rigen un buen diseño orientado a objetos.

Responsabilidades

Responsabilidad: contrato u obligación de un tipo o clase. **Son las obligaciones que tiene un objeto con respecto a su comportamiento.**

Hay dos categorías:

CONOCER

- Estar enterado de los datos privados encapsulados.
- Estar enterado de la existencia de objetos conexos.
- Estar enterado de cosas que se puede derivar o calcular.

HACER: responsabilidad relacionada:

- Hacer algo en uno mismo.
- Iniciar una acción en otros objetos.
- Controlar y coordinar actividades en otros objetos.

Las responsabilidades se asignan a los objetos en el diseño.

Por ejemplo, puede declararse que "*una Venta es responsable de imprimirse ella misma*" (un hacer) o que "*una Venta tiene la obligación de conocer su fecha*" (un conocer – que puede inferirse del Modelo Conceptual por los atributos y asociaciones).

Responsabilidad no es lo mismo que método: estos se ponen en práctica para cumplir con las responsabilidades. Las responsabilidades se implementan usando métodos que operen solos o en colaboración con otros métodos y objetos.

Definición de Patrón

En la terminología de objetos, el **patrón** es una **descripción de un problema y su solución**, que recibe un nombre y que puede emplearse en **otros contextos**.



Importancia

- En teoría, todos **los patrones poseen nombres muy sugestivos**. El asignar nombre a un patrón, a un método o a un principio ofrece las siguientes ventajas:
 - Apoya el agrupamiento y **la incorporación del concepto a nuestro sistema cognitivo y a la memoria**.
 - Facilita la **comunicación**.

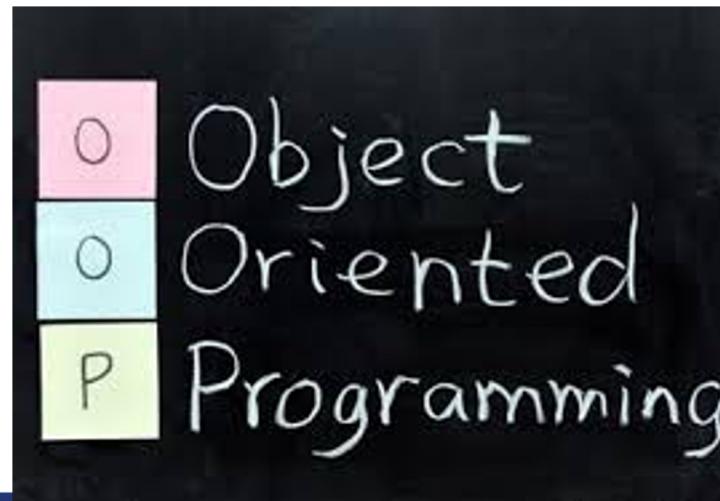
GRASP



- GRASP es un acrónimo que significa *General Responsibility Assignment Software Patterns* (**patrones generales de software para asignar responsabilidades**).
- El nombre se eligió para indicar la importancia de captar (**grasping**) estos principios, si se quiere diseñar eficazmente el software orientado a objetos.

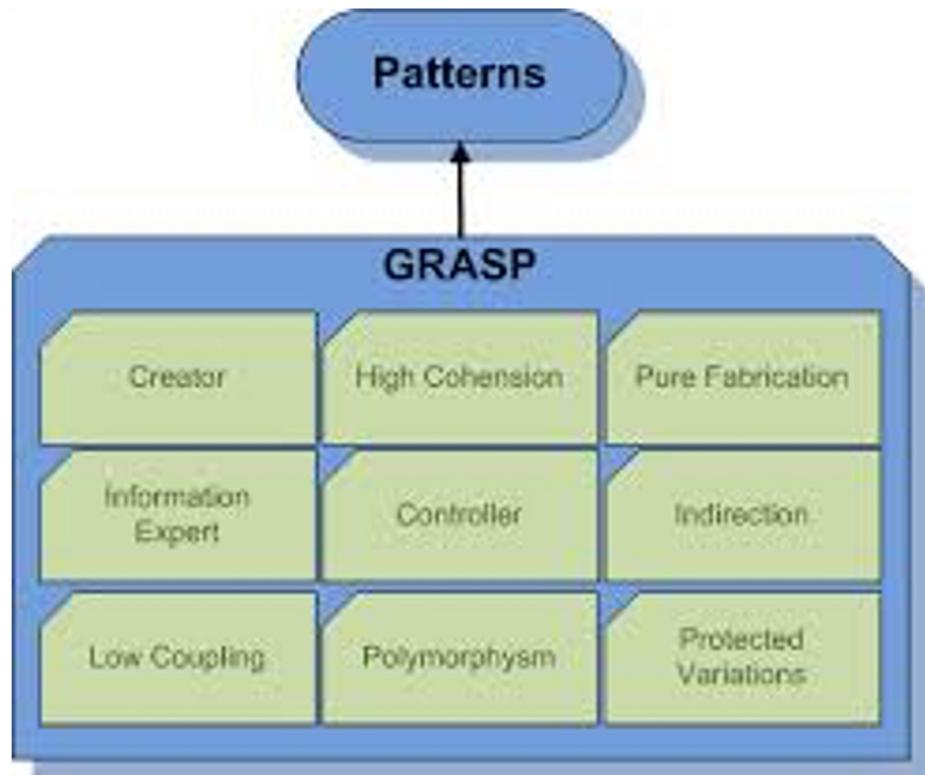
GRASP

Los patrones GRASP son parejas de **problema-solución** con un nombre, que codifican **buenos principios** y **sugerencias** relacionados frecuentemente con la **asignación de responsabilidades**.



GRASP

1. Experto
2. Creador
3. Controlador
4. Alta Cohesión
5. Bajo Acoplamiento
6. Polimorfismo
7. Fabricación Pura
8. Indirección
9. Variaciones Protegidas



Punto de Venta, ejemplo para el tema:

Aplicación para una **tienda o restaurante** que **registra las ventas**.



Cada venta contiene **uno o más elementos de uno o más tipos de productos** y sucede en una **fecha**.

Un producto tiene una especificación que incluye descripción, precio unitario e identificador.

La aplicación también **registra pagos** en efectivo **asociados a las ventas**. Un pago es por cierta cantidad igual o más grande que el total de la venta.



Experto

Inspira Crea Transforma

JUEGOS

UNIVERSIDAD
EAFIT[®]

Experto

Problema

¿Cuál es el principio general para **asignar responsabilidades** a los objetos?



Experto

Sobre el Problema...

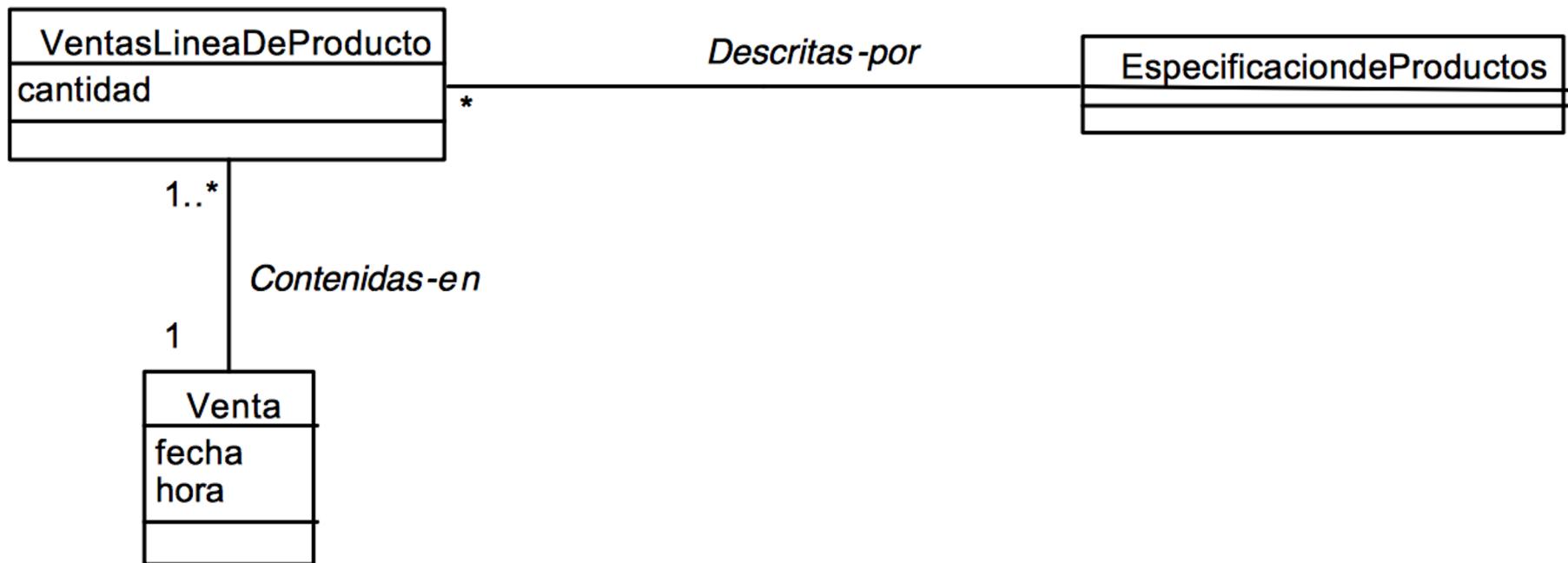
- Un **modelo de clase** puede definir docenas y hasta cientos de clases de software, y una aplicación tal vez requiera el cumplimiento de cientos o miles de **responsabilidades**.
- Si estas se asignan en forma adecuada, los sistemas tienden a ser más fáciles de **entender, mantener y ampliar**, y se nos presenta la oportunidad de **reutilizar los componentes** en futuras aplicaciones.

Experto

Solución

Asignar una responsabilidad al experto en información: **la clase que cuenta con la información necesaria para cumplir la responsabilidad.**

Ejemplo



Experto

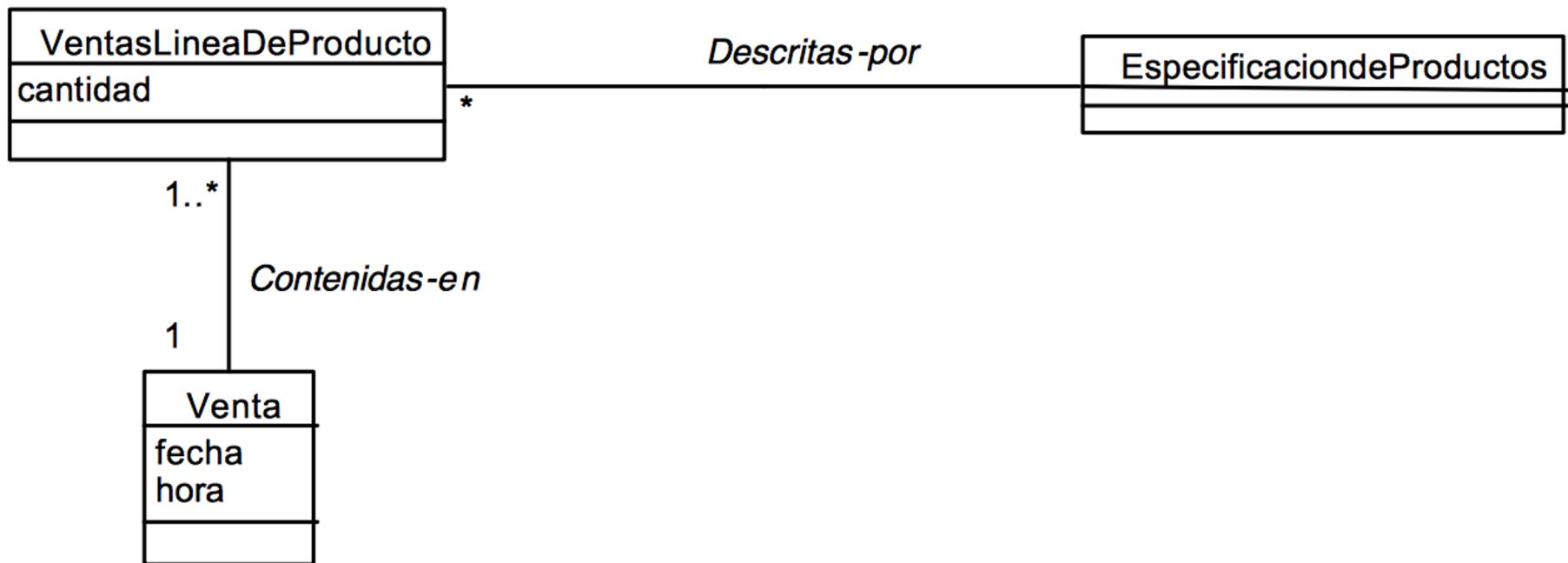
Ejemplo (1/3)

En la aplicación de **punto de venta**, alguna clase necesita conocer **el gran total de la venta**. Comienza asignando las responsabilidades con una definición clara de ellas. **A partir de esta recomendación se plantea la pregunta:**

¿Quién es el responsable de conocer el gran total de la venta?
Desde el punto de vista del patrón Experto, deberíamos buscar la clase de objetos que posee **la información** necesaria para **calcular el total**.

Experto

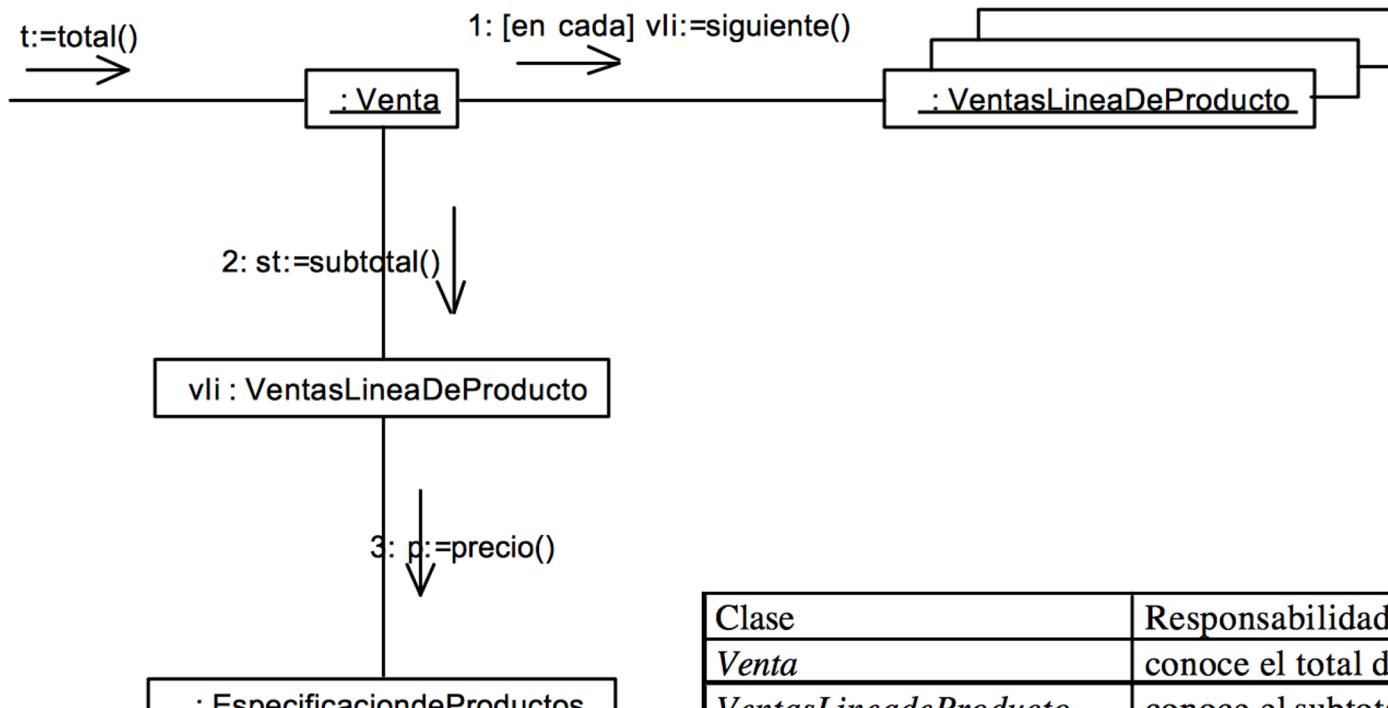
Ejemplo (2/3)



Experto

Ejemplo (3/3)

para cumplir con la responsabilidad de conocer y dar el **total de la venta**, se asignaron tres responsabilidades a las tres clases de objeto así:



Clase	Responsabilidad
<code>Venta</code>	conoce el total de la venta
<code>VentasLineadeProducto</code>	conoce el subtotal de la línea de producto
<code>EspecificaciondeProductos</code>	conoce el precio del producto

Experto

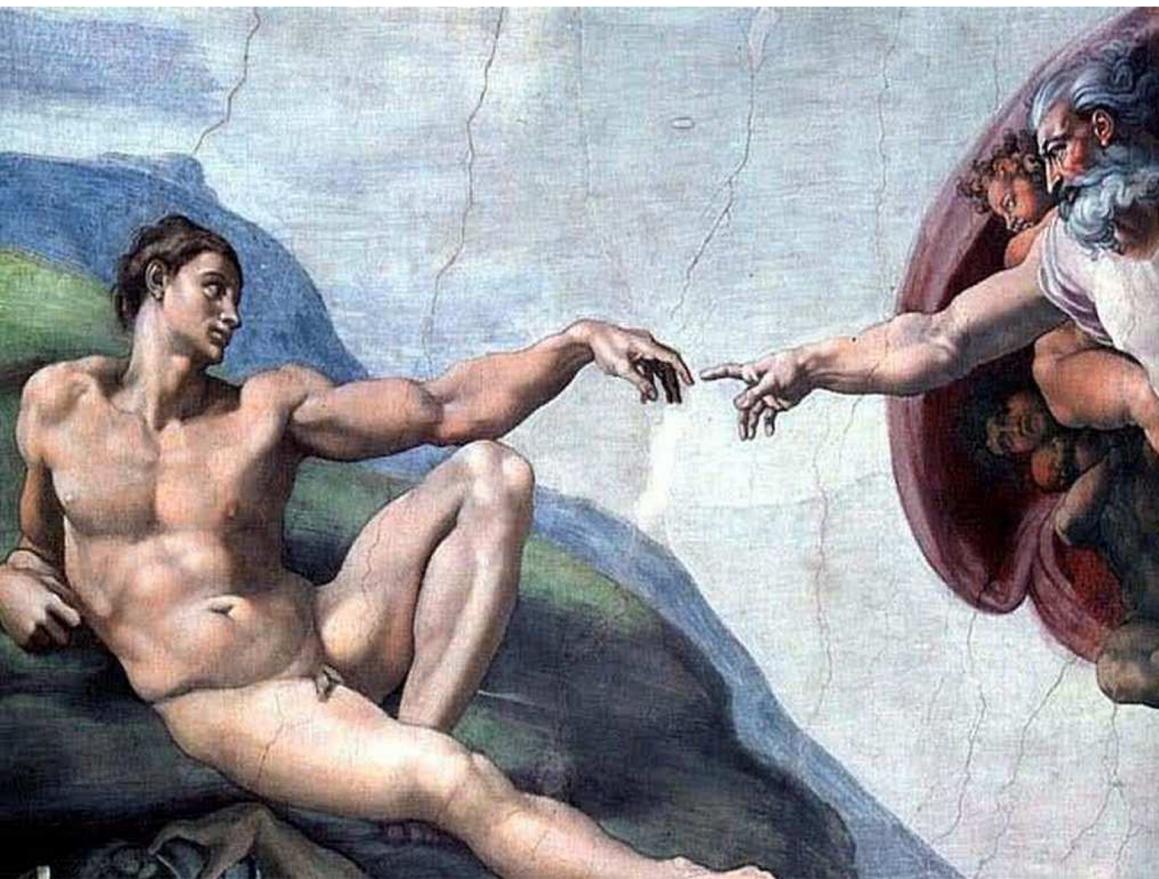
Importancia:

- Experto es un patrón que **se usa más que cualquier otro** al asignar responsabilidades; es un principio básico que suele ser útil en el diseño orientado a objetos.
- Nótese, que *el cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos.*

Experto

Beneficios:

- Se conserva el **encapsulamiento**, ya que los objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un **bajo acoplamiento**, lo que favorece al hecho de tener sistemas más robustos y de fácil mantenimiento.
- El comportamiento se distribuye entre las clases que cuentan con la información requerida, alejando con ello definiciones de **clases “sencillas”** y más **cohesivas** que son más fáciles de comprender y de mantener. Así se brinda soporte a una **alta cohesión**.



Creador

Inspira Crea Transforma

UNIVERSIDAD
EAFIT[®]

Creador

Problema:



¿Quién debería ser responsable de crear una nueva **instancia** de alguna clase?

Creador

Sobre el Problema:

La **creación de objetos** es una de las actividades más frecuentes en un sistema orientado a objetos.

En consecuencia, conviene contar con un principio general para asignar las responsabilidades concernientes a ella.

El diseño, bien asignado, puede soportar un **bajo acoplamiento**, una mayor **claridad**, el **encapsulamiento** y el **reuso**.

Creador

Sobre el Patrón:

- El propósito fundamental de este patrón es **encontrar un creador que debemos conectar con el objeto producido** en cualquier evento.
- Al escogerlo como creador, se da soporte al **bajo acoplamiento**.

Creador

Solución:

Asignarle a la clase B la responsabilidad de crear una instancia de la clase A en uno de los siguientes casos:

- B **agrega** los objetos A.
- B **contiene** los objetos A.
- B **registra** las instancias de los objetos A o
- B **utiliza** especialmente los objetos A.
- B **tiene los datos** de inicialización que serán transmitidos a A cuando este objeto sea creado (así que B es un Experto respecto a la creación de A). B es un **creador de los objetos A**.

Si existe más de una opción, prefiera la clase B que agregue o contenga la clase A.

Creador

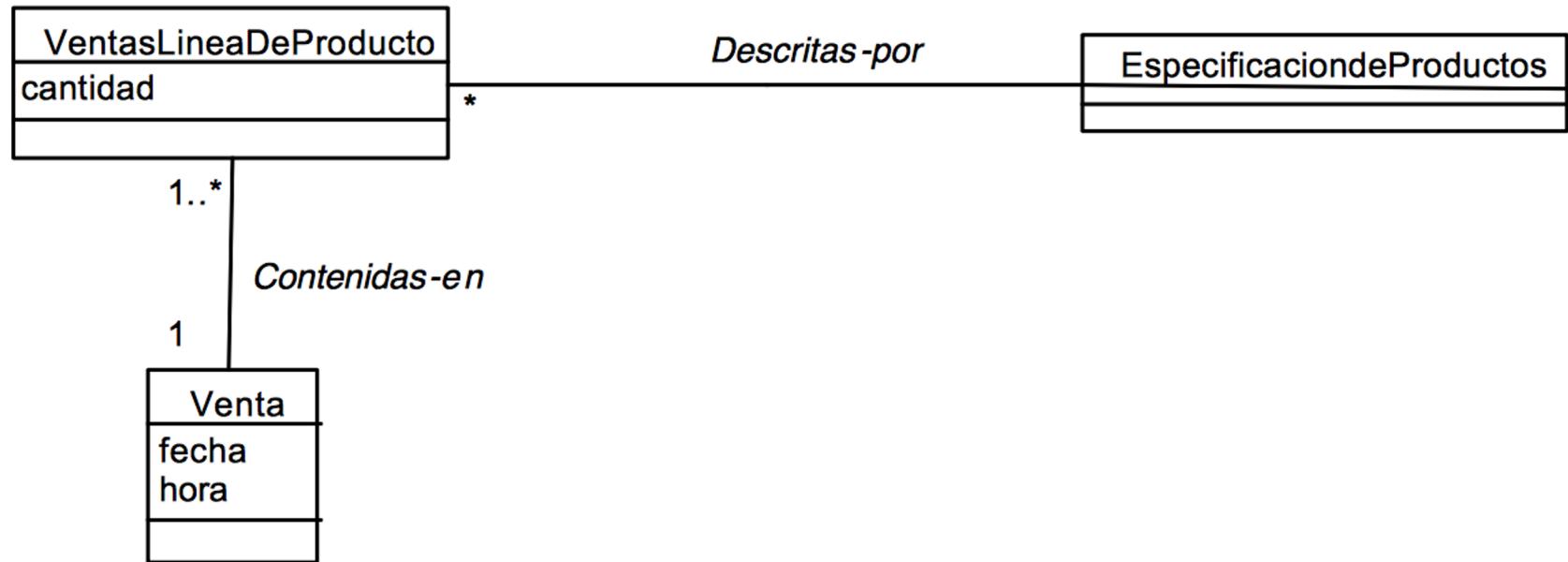
Ejemplo (1/5)

En la aplicación del punto de venta, ¿quién debería encargarse de crear una *instancia VentasLineadeProducto*?

Desde el punto de vista del patrón Creador, deberíamos buscar una clase que **agregue, contenga y realice** otras operaciones sobre este tipo de instancias

Creador

Ejemplo (2/5)



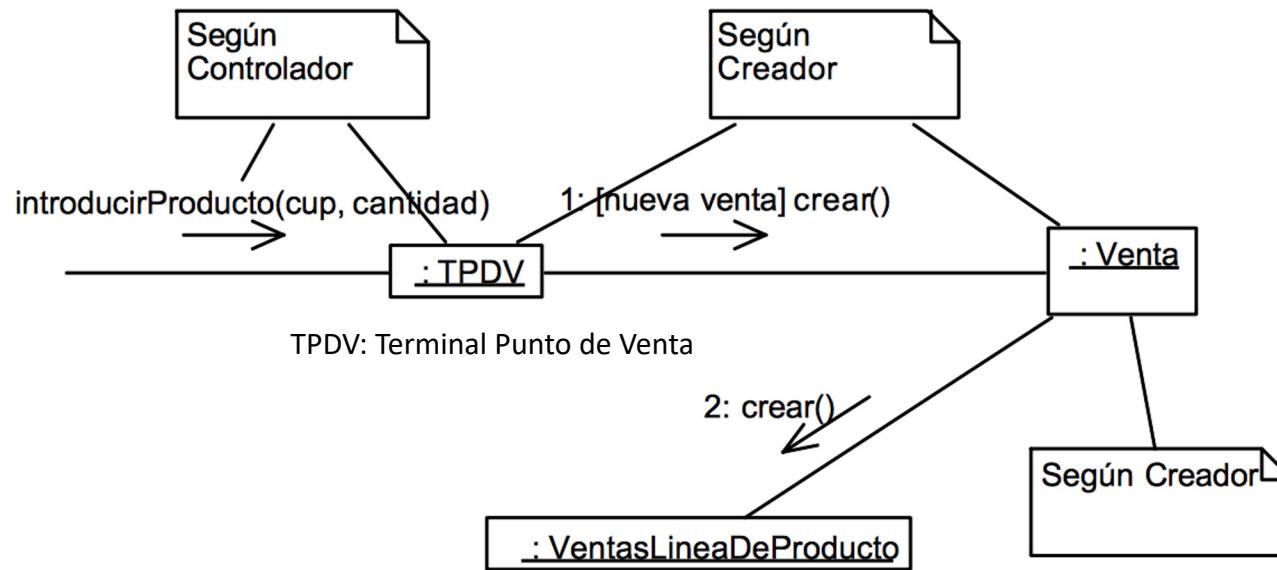
Creador

Ejemplo (3/5)

- Una **Venta** contiene (en realidad, **agrega**) muchos objetos **VentasLineadeProducto**;
- Por ello, el patrón Creador sugiere que **Venta** es idónea para asumir la responsabilidad de crear las instancias **VentasLineadeProducto**.
- Esta asignación de responsabilidades requiere definir en **Venta** un **método de hacer** **LineadeProducto**.

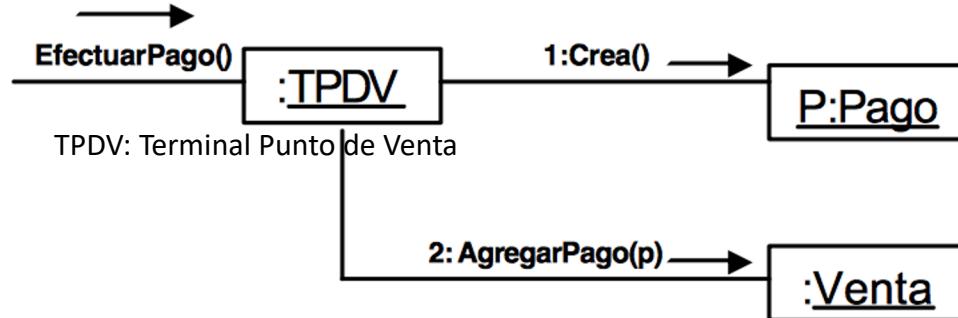
Creador

Ejemplo (4/5)



Creador

Ejemplo (5/5)

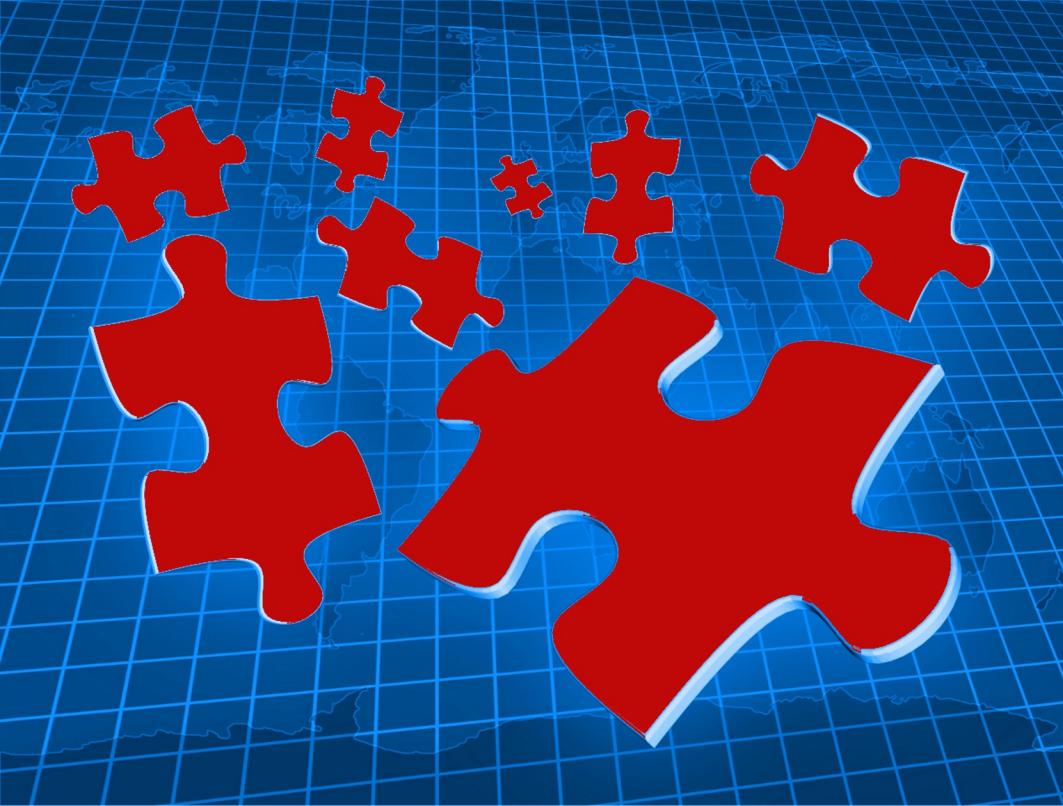


Ojo con este ejemplo que lo vamos a usar en un momento...

Creador

Beneficios:

- En ocasiones encontramos un patrón creador buscando la clase con los datos de inicialización que serán transferidos durante la creación.
- Éste es en realidad un ejemplo del patrón Experto.
- Los datos de inicialización se transmiten durante la creación a través de algún método de inicialización, como un constructor en java que cuenta con parámetros.



Patrón Bajo Acoplamiento

Inspira Crea Transforma

Patrón Bajo Acoplamiento

Problema

¿Cómo dar soporte a una **dependencia escasa** y a un aumento de la reutilización?



Patrón Bajo Acoplamiento

Sobre el problema:

- **El acoplamiento** es una **medida de la fuerza** con que una clase está conectada, conoce, confía o recurre a los objetos de otras clases.
- **Acoplamiento bajo** significa que una clase **no depende** de muchas clases.
- **Acoplamiento alto** significa que una clase recurre a muchas otras clases. Esto presenta los siguientes problemas:
 - Los cambios de las clases afines ocasionan cambios locales.
 - Difíciles de entender cuando están aisladas.
 - Difíciles de reutilizar puesto que dependen de otras clase

Patrón Bajo Acoplamiento

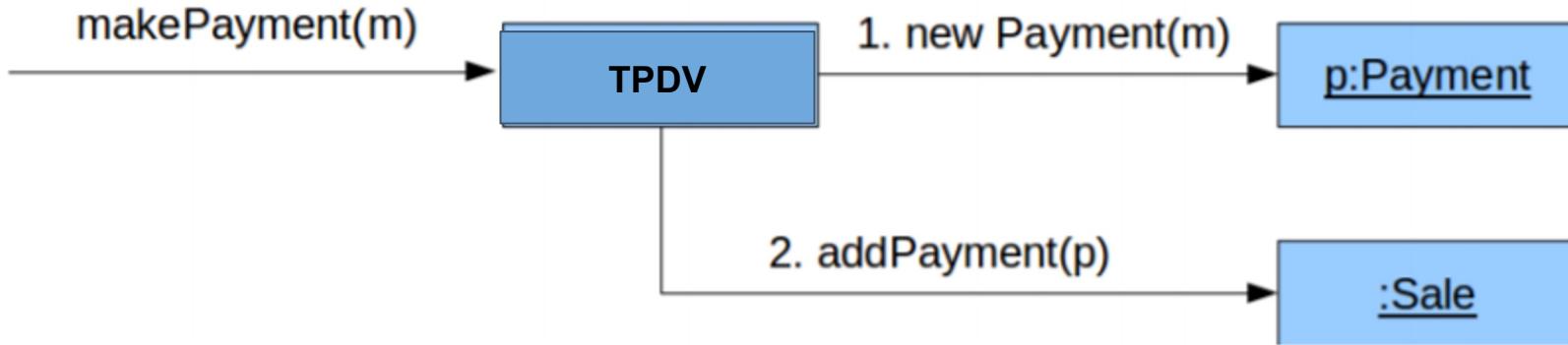
Solución:

Asignar una responsabilidad para mantener bajo acoplamiento.

Ejemplo: En el caso del punto de venta se tienen tres clases Pago, TPDV (Terminal Punto de Venta) y Venta y se quiere **crear una instancia de Pago y asociarla a Venta**. ¿Que clase es la responsable de realizarlo?

Según Experto

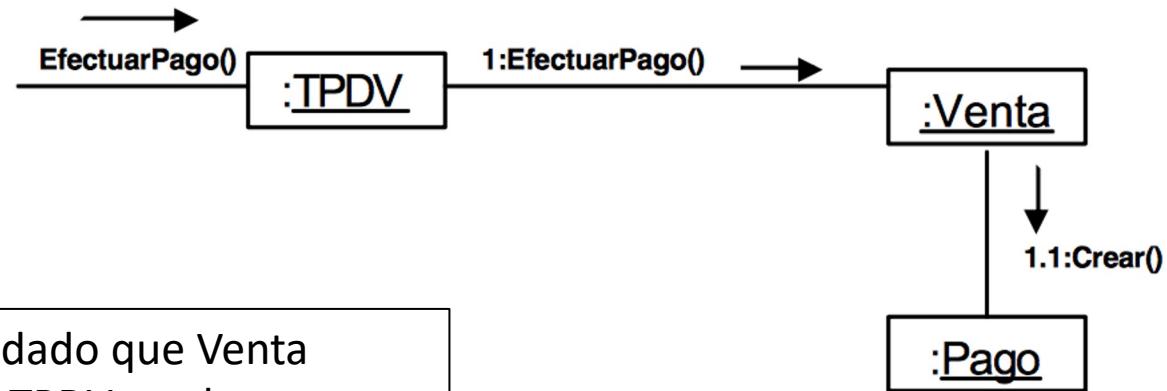
Según el patrón experto la Clase TPDV deberá hacerlo, ver Ejemplo (5/5)



Patrón Bajo Acoplamiento

Ejemplo

Según el patrón de Bajo Acoplamiento la relación debería ser de la siguiente manera:



Esta última asociación es mejor dado que Venta realiza la creación del Pago y no TPDV por lo tanto se reduce la dependencia de este último con el resto de las clases.

Patrón Bajo Acoplamiento

*El grado de acoplamiento no puede considerarse aisladamente de otros principios como **Experto y Alta Cohesión**. Sin embargo, es un factor a considerar cuando se intente mejorar el diseño.*



Patrón Bajo Acoplamiento

Por qué usar este patrón:

- No se afecta por cambios de otros componentes
- Fáciles de entender por separado
- Fáciles de reutilizar



Patrón Alta Cohesión

Inspira Crea Transforma

UNIVERSIDAD
EAFIT[®]

Patrón Alta Cohesión

Problema

¿Cómo mantener la **complejidad** dentro de límites manejables?



Patrón Alta Cohesión

Sobre el problema (1/2):

La **cohesión** es una medida de **cuán relacionadas** y enfocadas están las responsabilidades de una clase.

Una alta cohesión caracteriza a las clases con **responsabilidades estrechamente relacionadas** que no realicen un trabajo enorme.

Patrón Alta Cohesión

Sobre el problema (2/2):

Una baja cohesión hace muchas cosas no afines o realiza trabajo excesivo. Esto presenta los siguientes problemas:

- Son difíciles de comprender
- Difíciles de reutilizar
- Difíciles de conservar
- Las afectan constantemente los cambios.



Patrón Alta Cohesión

Solución

Asignar una responsabilidad de modo que la cohesión siga siendo alta.

Patrón Alta Cohesión

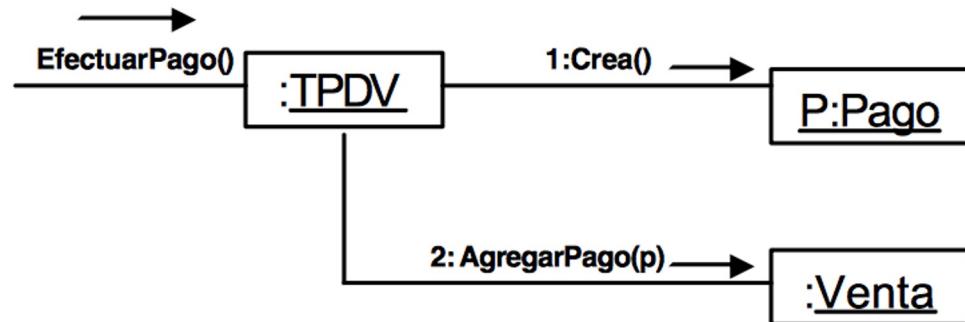
Ejemplo (1/3)

Ejemplo: En el caso del punto de ventas se tienen tres clases **Pago**, **TPDV** y **Venta** y se quiere **crear una instancia de Pago y asociarla a Venta**. Según el principio del **patrón Creador** la clase **TPDV** debe ser la encargada de realizar el pago.

¿Qué pasa si el sistema tiene 50 operaciones, todas recibidas por la clase **TPDV** ?

La clase se iría saturando con tareas y terminaría perdiendo la cohesión

Volvimos a la imagen 5/5

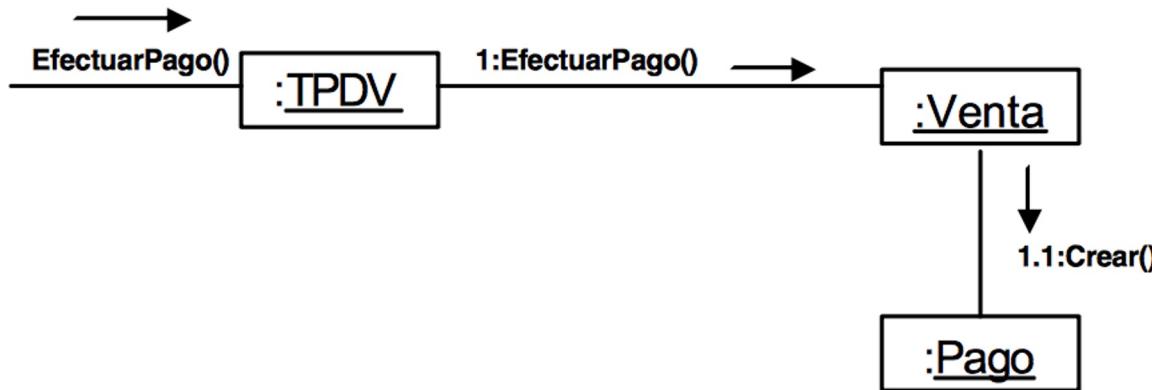


Patrón Alta Cohesión

Ejemplo (2/3)

Un mejor diseño sería así:

- Este diseño **delega** a Venta la responsabilidad de efectuar el pago.
- Este diseño es conveniente ya que da soporte a una alta cohesión y a un bajo acoplamiento.



Según el patrón bajo acoplamiento

Patrón Alta Cohesión

Beneficios:

En la práctica, el nivel de cohesión no puede ser considerado independiente de los otros patrones y principios (e.g. Patrones “Experto” y “Bajo Acoplamiento”).

- Mejoran la **claridad** y facilidad con que se entiende el diseño.
- Se simplifica el **mantenimiento** y las **mejoras de funcionalidad**
- A menudo se genera **un bajo acoplamiento**
- Soporta mayor capacidad de **reutilización**.

Patrón Alta Cohesión

Algunos escenarios:

- **Muy baja cohesión:** Una clase es la única responsable de muchas cosas en áreas funcionales heterogéneas.
- **Baja cohesión:** Una clase tiene la responsabilidad exclusiva de una tarea compleja dentro de un área funcional.
- **Alta cohesión:** Una clase tiene responsabilidades moderadas en un área funcional y colabora con las otras para llevar a cabo las tareas.
- **Cohesión moderada:** Una clase tiene peso ligero y responsabilidades exclusivas en unas cuántas áreas que están relacionadas lógicamente con el concepto de clase pero no entre ellas.



Controlador

Inspira Crea Transforma

UNIVERSIDAD
EAFIT[®]

Patrón Controlador

Problema

¿Quién debería encargarse de **atender un evento** del sistema?



Patrón Controlador

Sobre el Problema:

Un “**evento del sistema**” es un hecho de alto nivel **generado por un actor externo**; es un evento de entrada externa. Se asocia a operaciones del sistema: las que emite en respuesta a los eventos del sistema.

Un Controlador es un objeto de interfaz no destinado al usuario que se encarga de manejar un evento del sistema. Define además el método de su operación.

Por ejemplo, cuando un cajero que usa un sistema de terminal en el punto de venta oprime el botón “**Terminar Venta**”, está generando un **evento sistémico** que indica que “**la venta ha terminado**”.

Patrón Controlador



Solución:

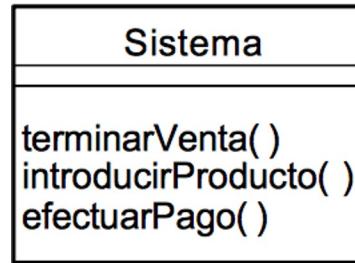
Asignar la responsabilidad del **manejo de un mensaje** de los eventos de un sistema a una clase que represente una de las siguientes opciones:

- el “sistema” global (controlador de fachada).
- la empresa u organización global (controlador de fachada).
- algo en el mundo real que es activo (por ejemplo, el papel de una persona) y que pueda participar en la tarea (controlador de tareas).
- un manejador artificial de todos los eventos del sistema de un caso de uso, generalmente denominados “Manejador<NombreCasodeUso>” (controlador de casos de uso).

Patrón Controlador

Ejemplo (1/3)

- En la aplicación del punto de venta se dan varias operaciones del sistema, como se advierte en la figura.



¿Quién debería ser el controlador de **eventos sistémicos** como introducirProducto y terminarVenta?

Patrón Controlador

Ejemplo (2/3)

De acuerdo con el patrón Controlador, disponemos de las siguientes opciones:

representa el “sistema” global

TPDV

representa la empresa u organización global

Tienda

representa algo en el mundo real que está activo
(por ejemplo, el papel de una persona) y que
puede intervenir en la tarea

Cajero

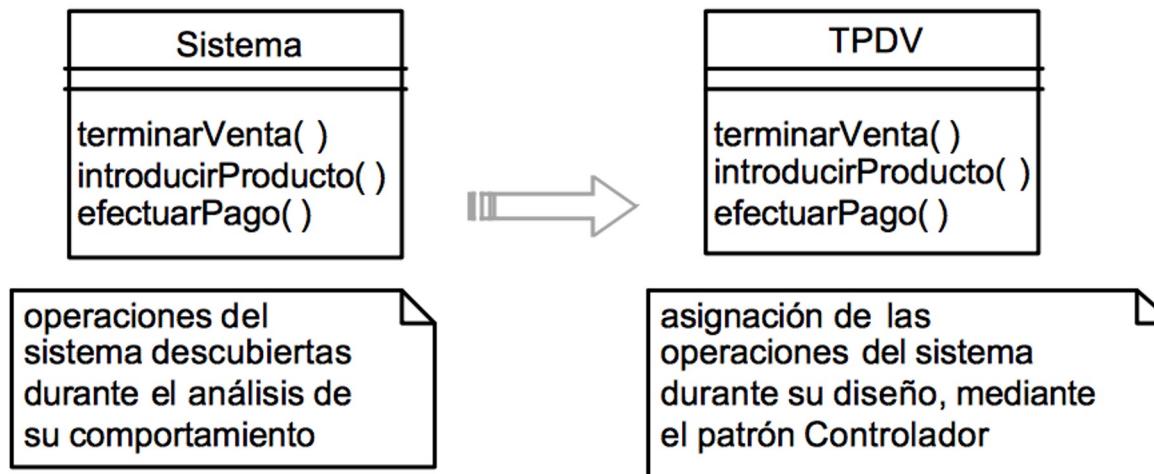
representa un manejador artificial de todas las
operaciones del sistema de un caso de uso.

*ManejadordeComprarP
roductos*

Patrón Controlador

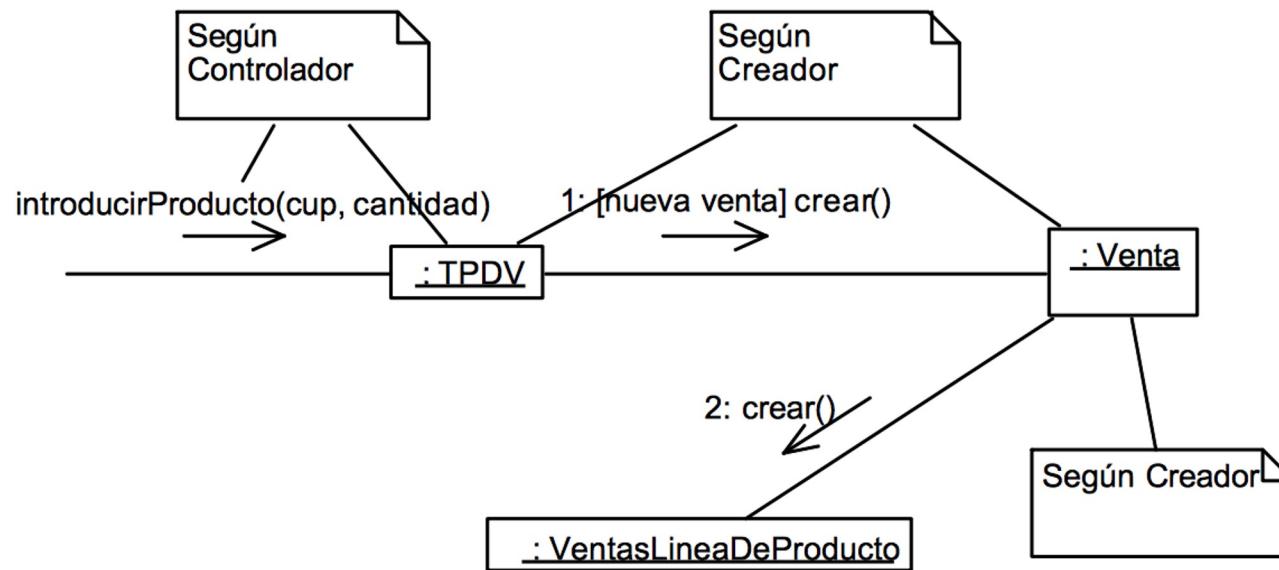
Ejemplo (3/3):

En la decisión de cuál de las cuatro clases es el controlador más apropiado influyen también otros factores como la cohesión y el acoplamiento.



Patrón Controlador

Recordamos esta figura cuando hablábamos del patrón creador?



Patrón Controlador

Beneficios:

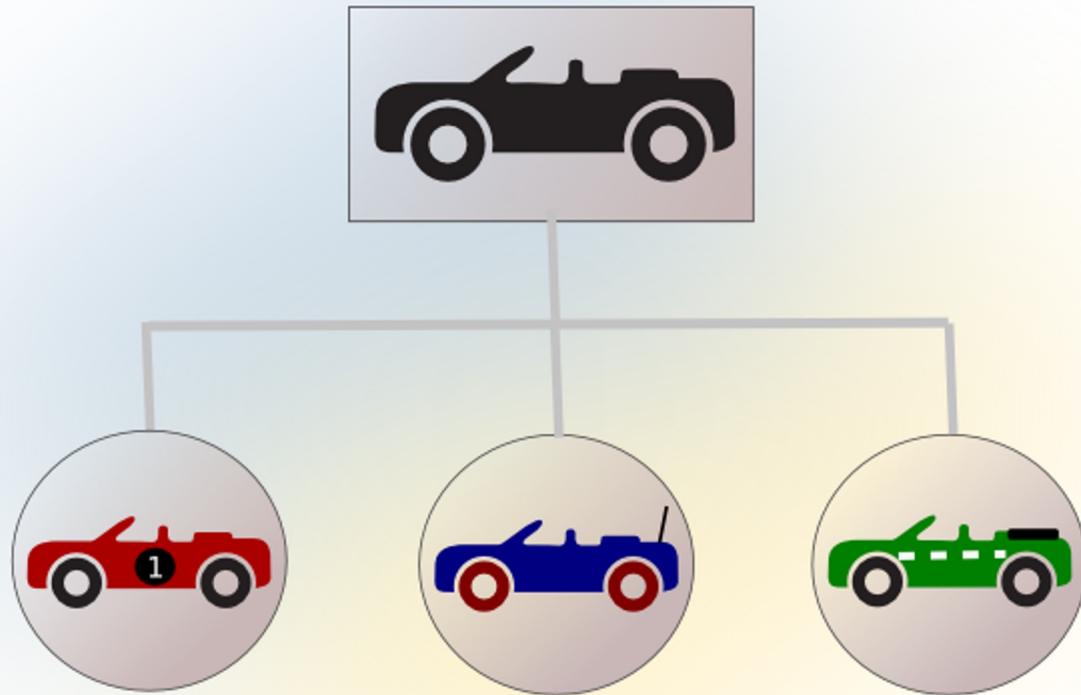
- Mayor potencial de los **componentes reutilizables**. Garantiza que la empresa o los procesos de dominio sean manejados por la capa de los objetos del dominio y no por la de la interfaz.
- **Reflexionar sobre el estado del caso de uso.** A veces es necesario asegurarse de que las operaciones del sistema sigan una secuencia legal o poder razonar sobre el estado actual de la actividad y las operaciones en el caso de uso subyacente.

Patrón Controlador

Importante:



Los objetos de la interfaz (por ejemplo, objetos de ventanas, front-end) y la capa de presentación **no** deberían encargarse de manejar los eventos del sistema.



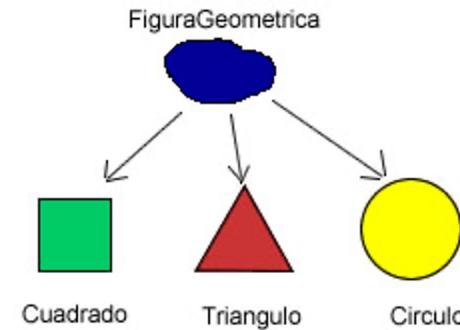
Polimorfismo

Polimorfismo

La capacidad que tienen los objetos de diferentes clases de **responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación.**

Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.

Conseguir que un objeto de una clase se comporte como un objeto de cualquiera de sus subclases, dependiendo de la forma de llamar a los métodos de dicha clase o subclases



Polimorfismo

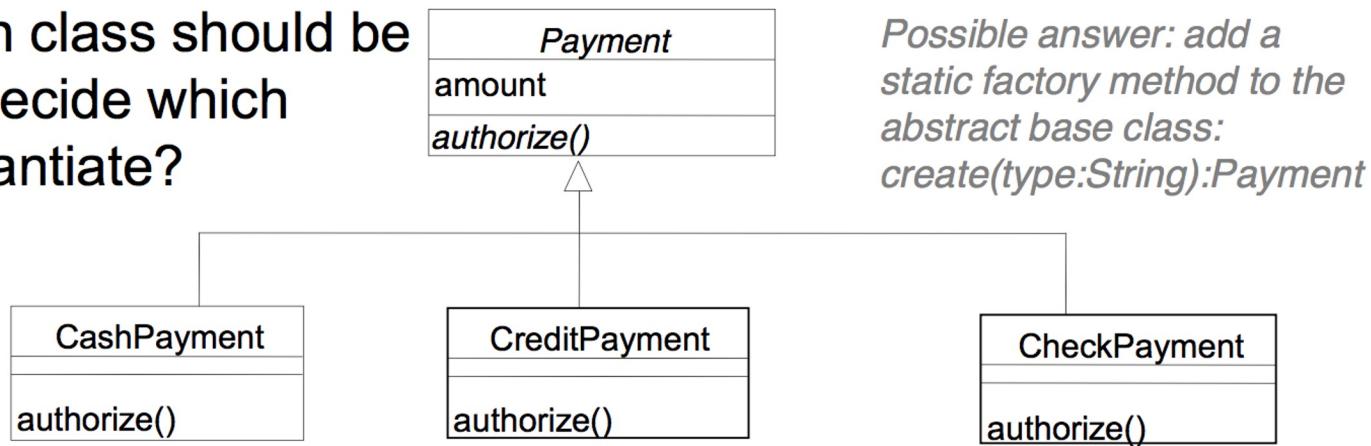
Cuando tenga que designar una **responsabilidad que dependa del tipo**, se tiene que hacer uso del **polimorfismo**.

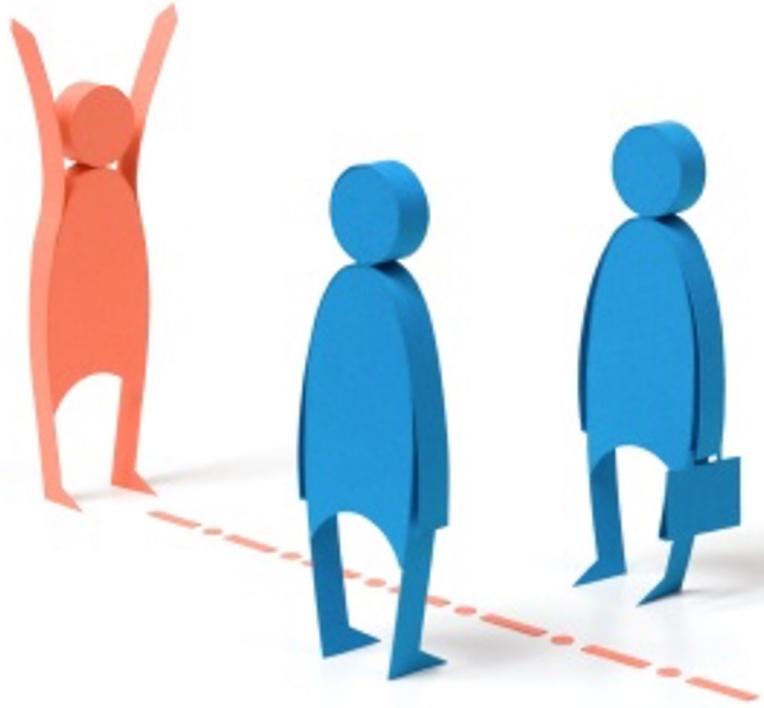
Cuando las **alternativas o comportamientos** relacionados varían según el tipo (clase), asigne la responsabilidad para el comportamiento- utilizando **operaciones polimórficas**- a los tipos para los que varía el comportamiento.

Asigna el mismo nombre a servicios en diferentes objetos.

- How to handle alternatives based on types?
- Use **polymorphic method calls** to select the behaviour
 - rather than using “if” statements to test for the type
 - at various places throughout the code
 - easier to add additional behaviours later on
 - disadvantage is –
 - an increased number of classes in a design
 - ➔ *classic trade-off: efficiency vs. understandability/maintainability*

■ Question: Which class should be responsible to decide which subclass to instantiate?





Indirección

Inspira Crea Transforma

UNIVERSIDAD
EAFIT[®]

Indirección

Problema

¿Dónde asignar responsabilidades para **evitar/reducir el acoplamiento directo entre elementos** y mejorar la reutilización?

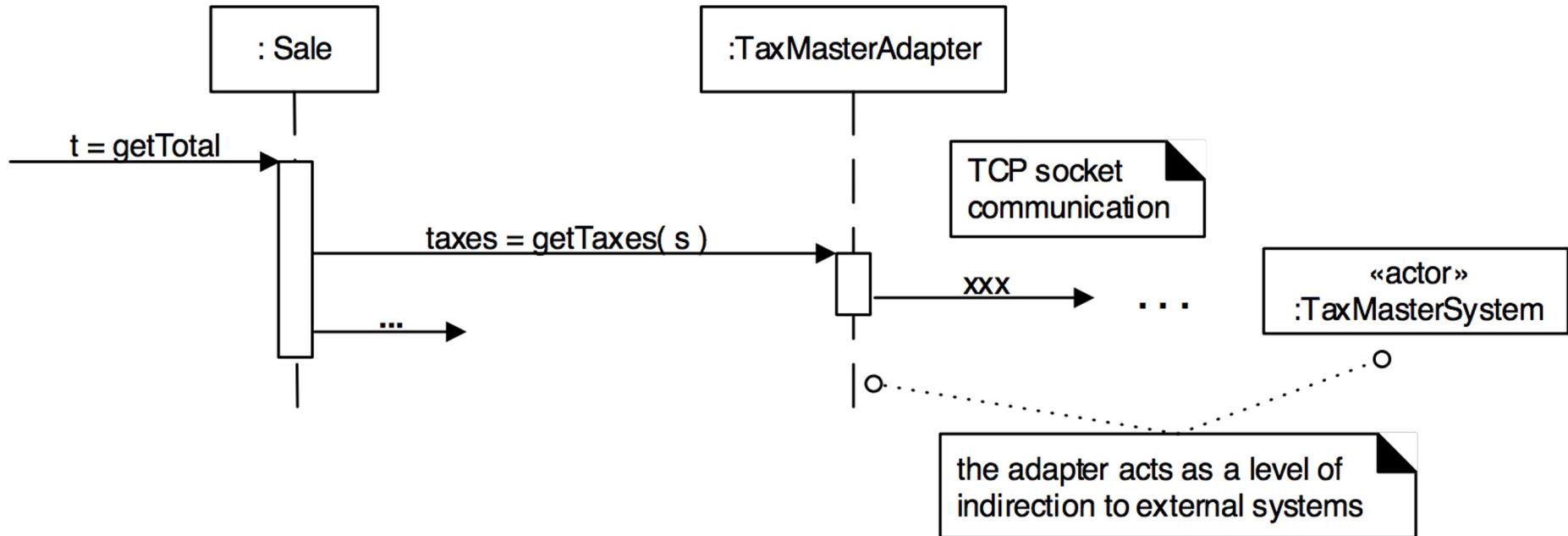
Solución

- **Asigne la responsabilidad a un objeto que medie entre los elementos.**
- Identifique los de variación predilectos o inestables. Asigne responsabilidades para crear una “**interfaz**” alrededor de ellos.

Explicación

El patrón de indirección nos aporta mejorar el bajo acoplamiento entre dos clases **asignando la responsabilidad de la mediación entre ellos a un tercer elemento (clase) intermedio.**

Por ejemplo, en un TPV (Terminal de Punto de Venta) usar una Interfaz "**Adaptador**" a través de la cual, con las clases que la realicen, podamos establecer los distintos métodos de pago.





Fabricación Pura

Inspira Crea Transforma

UNIVERSIDAD
EAFIT[®]

Fabricación Pura

¿Quien es el responsable cuando tu estas desesperado y no quieres violar la alta cohesión y el bajo acoplamiento?



Fabricación Pura

La fabricación pura se da en las clases que no representan un ente u objeto real del dominio del problema, sino que se ha creado **intencionadamente** para disminuir el acoplamiento, aumentar la cohesión y/o potenciar la reutilización del código.

Es la solución cuando el diseñador se encuentre con una clase poco cohesiva y no tenga otra clase en la que implementar algunos métodos. Es decir que es una clase "**inventada**" o que no existe en el problema como tal, pero que añadiéndola se logra mejorar estructuralmente el sistema.

Advertencia: al abusar de este patrón suele aparecer clases función o algoritmo (que tienen un solo método).

- Where to assign responsibilities if no appropriate domain object exists?
 - ➔ *invent an artificial class and assign cohesive responsibilities to it*
 - preserves high cohesion and low coupling
- Example
 - *support needed to save Sale instances in a relational database*
 - there would be some justification to assign this responsibility to Sale
 - justified by the Expert pattern
 - however, this requires a relatively large number of supporting database-oriented operations
 - ➔ the Sale class would become incohesive
 - on the other hand, this is a general task for which many classes need support
 - ➔ create a class PersistentStorage (or EntityManager etc.) to solve this
 - that goes into the technical services layer
 - as it is perhaps reusable in other contexts



Variaciones Protegidas

Inspira Crea Transforma

UNIVERSIDAD
EAFIT[®]

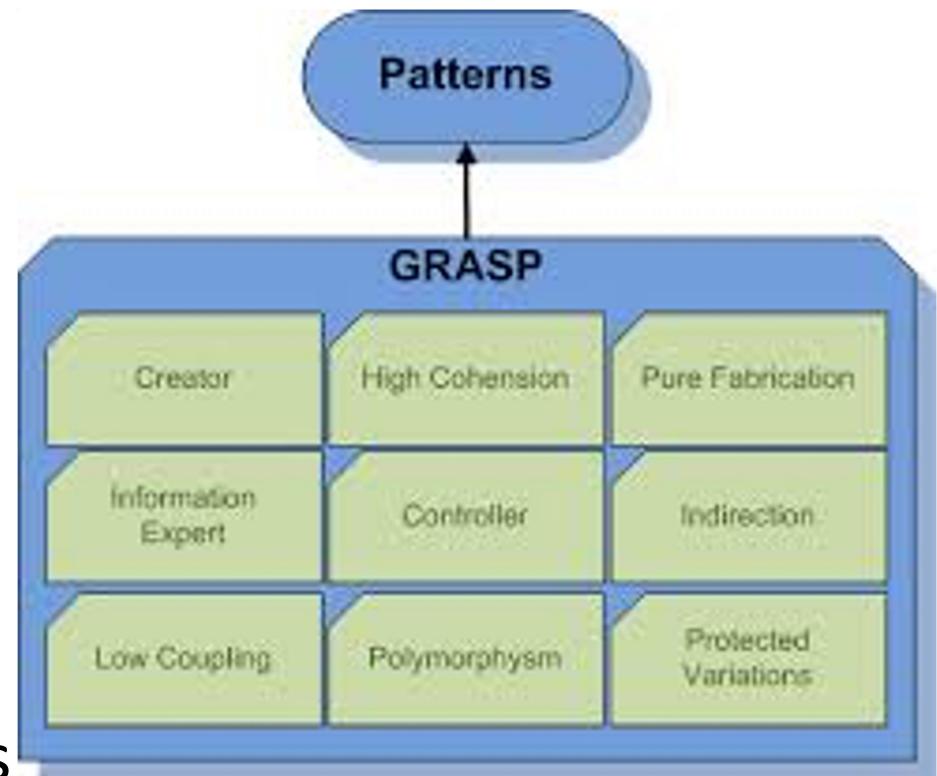
Variaciones Protegidas

Es el principio fundamental de **protegerse del cambio**. Aquello que es susceptible de modificaciones, lo envolvemos en una interfaz, utilizando el **polimorfismo** para crear varias implementaciones y posibilitar implementaciones futuras, de manera que quede lo menos ligado posible a nuestro sistema, de forma que cuando se produzca la variación, nos repercuta lo mínimo.

- General principle for a variety of mechanisms to preserve objects and (sub-) systems from variations
 - information hiding, interfaces and polymorphism are some simple examples
 - Don't talk to strangers (Law of Demeter)
 - A method should only talk to the following objects
 - this/self
 - parameters of the method
 - attributes of this/self
 - objects created in the method
 - service lookup
 - use of reflection
 - use of standards such as SQL
- Sometimes also known as open/closed principle
- Based upon Parnas' Law: [Endres/Rombach03]
Only what is hidden can be changed without risk.

GRASP en Resumen

1. Experto
2. Creador
3. Controlador
4. Alta Cohesión
5. Bajo Acoplamiento
6. Polimorfismo
7. Fabricación Pura
8. Indirección
9. Variaciones Protegidas



[Larman] C. Larman
Applying UML and Patterns
Prentice Hall, 2005

Referencia y links

- [1] SRP, *Robert C. Martin*.
https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk
- [2] OCP, *Robert C. Martin*. <http://www.objectmentor.com/resources/articles/ocp.pdf>
- [3] LSP, *Robert C. Martin*.
https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk
- [4] ISP, *Robert C. Martin*. <http://www.objectmentor.com/resources/articles/isp.pdf>
- [5] DIP, *Robert C. Martin*. <http://www.objectmentor.com/resources/articles/dip.pdf>
- [6] InversionOfControl, *Martin Fowler*.
<http://martinfowler.com/bliki/InversionOfControl.html>
- [7] Tell, Don't Ask, *Andy Hunt, Dave Thomas*. <http://pragprog.com/articles/tell-dont-ask>
- [8] Smalltalk by Example, *Alec Sharp*.
<http://stephane.ducasse.free.fr/FreeBooks/ByExample/SmalltalkByExampleNewRelease.pdf>
- [9] TellDontAsk, *Martin Fowler* <http://martinfowler.com/bliki/TellDontAsk.html>
- [10] GetterEradicator, *Martin Fowler* <http://martinfowler.com/bliki/GetterEradicator.html>

Referencia y links

[11] Demeter: Aspect-Oriented Software Development, *Karl J. Lieberherr*
<http://www.ccs.neu.edu/research/demeter/>

[12] Introducing Demeter and its Laws, *Brad Appleton* <http://www.bradapp.com/docs/demeter-intro.html>

[13] Software Design Pattern, *Wikipedia* <http://en.wikipedia.org/wiki/Softwaredesignpattern>

[14] Design Patterns: Elements of Reusable Object-Oriented Software, *E. Gamma y otros.* ISBN 0201633612.

[15] Refactoring to Patterns, *Josua Kerievsky.* ISBN 0321213351.

[16] Refactoring: Improving the Design of Existing Code, *Martin Fowler.* ISBN 0201485672.

[17] Notas de clase profesor Jose Patricio Bovet Derpich
<http://www.slideshare.net/josebovet/idss5501-principios-diseno-del-software>

[18] Notas de clase Marcello Visconti y Hernán Astudillo
<http://www.inf.utfsm.cl/~visconti/ili236/Documentos/08-Patrones.pdf>

[19] <http://www.genbeta.dev/metodologias-de-programacion/doce-principios-de-diseno-que-todo-desarrollador-deberia-conocer>



"That's all Folks!"