Arquitectura y Diseño de Sistemas

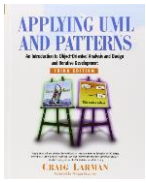
Lic. Ariel Trellini

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Principios de Diseño GRASP

Diseñando objetos con responsabilidad

Bibliografía



Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Desing and Iterative Development
Craig Larman
1995… 2004 – Addison Wesley

Craig Larman

Introducción

¿Cómo comenzamos a realizar un diseño orientado a objetos?

- Tradicionalmente en Programación Orientada a Objetos (POO), el diseño de objetos es pasado por alto. Por ejemplo:
  - *Luego de identificar los requerimientos y crear un modelo de dominio, comenzar a agregar métodos a clases y definir los mensajes entre los objetos para cumplir con los requerimientos.*
  - *Pensar en sustantivos y convertirlos a objetos; pensar en verbos y convertirlos a métodos*
- Pero… ¿cómo?
  - ¿Qué métodos pertenecen a qué objetos?
  - ¿Cómo deben interactuar los objetos?

Responsability-Driven Design (RDD)

*Una manera popular de pensar el diseño de objetos y también de componentes de mayor escala es en términos de **responsabilidades**, **roles** y **colaboraciones***

- Responsabilidades
  - En RDD, pensamos los objetos como teniendo responsabilidades, es decir, una abstracción de lo que hacen/conocen
  - Las responsabilidades están relacionadas a las obligaciones o comportamiento de un objeto de acuerdo a su **rol**.
  - Granularidad de la responsabilidad
    - Grandes responsabilidades toman cientos de clases y métodos
      - → Ejemplo: Proveer acceso a una base de datos relacional
    - Pequeñas responsabilidades podrían tomar un método
      - → Ejemplo: Crear una venta

- Responsabilidades (cont.)
  - Tipos de Responsabilidades
    - Hacer
      - → Hacer algo por sí mismo: crear un objeto o hacer un cálculo
      - → Iniciar una acción en otro objeto
      - → Controlar y coordinar actividades en otros objetos
    - Conocer
      - → Conocer acerca de los datos privados encapsulados
      - → Conocer acerca de los objetos relacionados
      - → Conocer acerca de las cosas que puede calcular
  - Las responsabilidades son asignadas a las clases durante el diseño. Ejemplo
    - Una Venta es responsable de crear ItemsDeVenta (hacer)
    - Una Venta es responsable de conocer su Total (conocer)
- Colaboraciones
  - Las responsabilidades son implementadas por medio de métodos que o bien actúan solos, o colaboran con otros métodos y objetos
    - Ejemplo: *Venta.getTotal()* podría colaborar con *ItemDeVenta.getSubtotal()*

RDD es una Metáfora

Pensar a los objetos de software de forma similar a personas con responsabilidades que colaboran con otras personas para realizar el trabajo. RDD conduce a ver un diseño orientado a objetos como una ***comunidad de objetos responsables colaborando****.*

GRASP

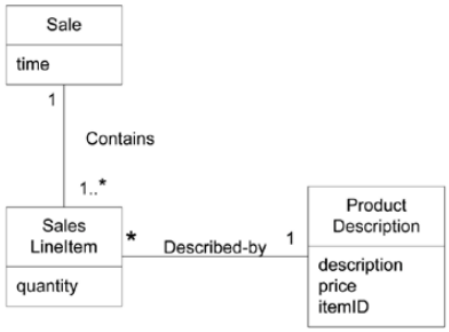
- Definición
  - Es una colección de principios de diseño orientados a objetos que **guían la asignación de responsabilidades** sobre los objetos.
  - Es un intento de documentar lo que diseñadores expertos probablemente conozcan intuitivamente.
- Acrónimo
  - **G**eneral **R**esponsability **A**ssignment **S**oftware **P**rinciples (or **P**atterns)
- Objetivo
  - Ayudar a entender el diseño de objetos esencial, aplicando el razonamiento de diseño de una manera metódica, racional y explicable.

- Principios
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion
  - Polymorphism
  - Pure Fabrication
  - Indirection
  - Protected Variations

Creator

- Problema
  - ¿Quién debería ser responsable de crear una nueva instancia de un clase?
    - Resolver bien esta pregunta ayuda al mantener un bajo acoplamiento, incrementar la claridad, el encapsulamiento y la reusabilidad
- Solución
  - Asignar a la clase B la responsabilidad de crear una instancia de la clase A si una de estas sentencias es verdadera:
    - B contiene o “agrega” a A
    - B registra a A
    - B usa a A
    - B tiene los datos de inicialización de A que serán pasados a A cuando sea creada. Así, B es un Experto con respecto a la creación de A.
  - Si más de una opción aplica, generalmente preferir una clase B que “agregue” o “contenga” a la clase A.

- Ejemplo – Punto de Venta
  - ¿Quién debiera ser responsable de crear una instancia de SalesLineItem?
  - Puesto que Sale contiene (de hecho, "agrega") muchos objetos SalesLineItem, *Creator* sugiere que Sale es un buen candidato para tener la responsabilidad de crear instancias de SalesLineItem



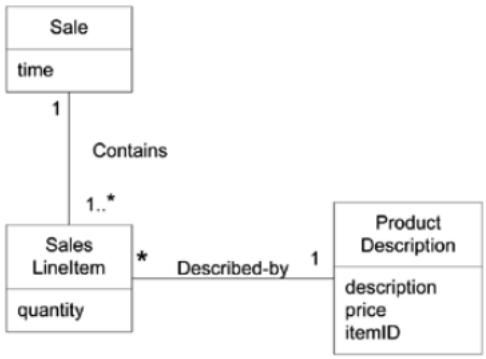
- Contraindicaciones
  - Cuando la creación requiere una complejidad significativa, es aconsejable delegarla a una clase dedicada que implemente Concrete Factory o Abstract Factoy (GoF)
    - Cuando se usan instancias recicladas por un tema de performance
    - Cuando se crea una instancia de una familia de clases similares
- Beneficios
  - Mantiene el acoplamiento bajo
    - Menor mantenimiento de dependencias
    - Mayores oportunidades de reuso

Information Expert

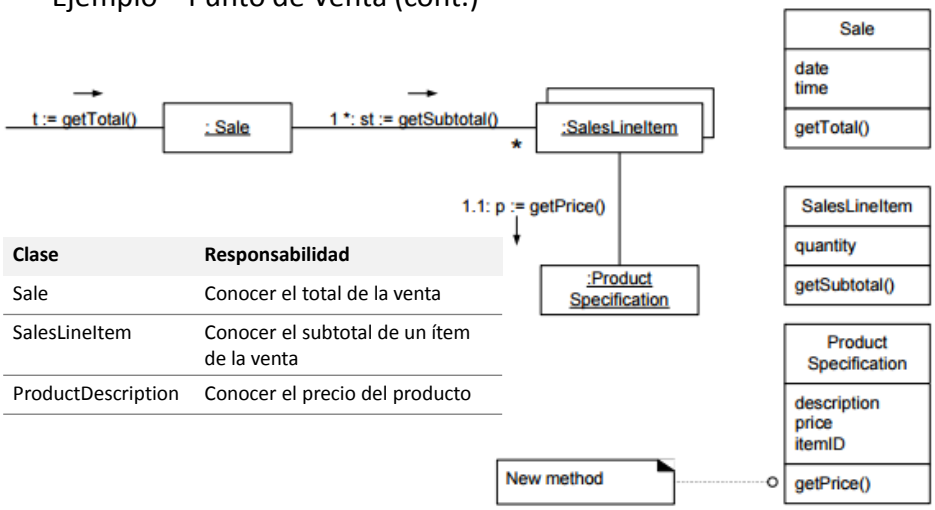
- Problema
  - ¿Cuál es el principio general para asignar responsabilidades a los objetos?
    - Resolver bien esta pregunta ayuda al mantener un bajo acoplamiento, incrementar la claridad, el encapsulamiento y la reusabilidad

- Solución
  - Asignar una responsabilidad a la clase que tiene la información necesaria para cumplir dicha responsabilidad
  - Proceso
    - Establecer claramente la responsabilidad
    - Si ya existen clases relevantes en nuestro modelo de diseño, buscar ahí
    - Si no, inspirarse en un modelo de más alto nivel (por ejemplo, modelo surgido de la etapa de análisis) para crear la clase correspondiente a la entidad identificada.

- Ejemplo – Punto de Venta
  - ¿Quién sería responsable de conocer el monto total de una venta?
    - ¿Qué información se necesita conocer? ¿Quién la conoce?



- Ejemplo – Punto de Venta (cont.)



Clase	Responsabilidad
Sale	Conocer el total de la venta
SalesLineItem	Conocer el subtotal de un ítem de la venta
ProductDescription	Conocer el precio del producto

- Contraindicaciones
  - ¿Quién sería responsable de guardar una Sale en la base de datos?

    Information Expert → Sale **ERROR !**

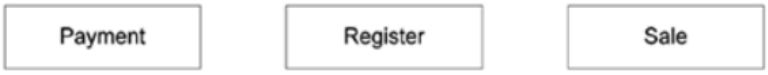
    Extendiendo esta decisión, todas las clases de dominio deberían tener lógica de acceso a datos, lo cual genera problemas de separación de intereses, cohesión, acoplamiento y duplicación.

- Beneficios
  - Mantiene el encapsulamiento de la información, ya que los objetos usan su propia información para realizar las tareas. Esto ayuda a tener un bajo acoplamiento
  - El comportamiento es distribuido a través de las clases que tienen la información requerida, propiciando clases más cohesivas, fáciles de entender y mantener.

Low Coupling

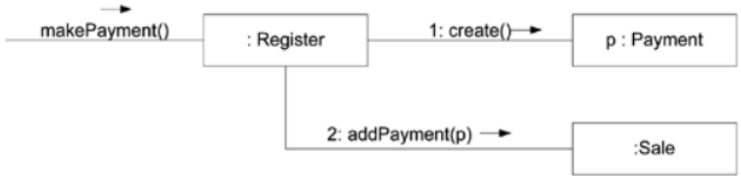
- Problema
  - ¿Cómo lograr baja dependencia, bajo impacto de cambio y un reuso mayor?
    - Una clase con alto acoplamiento sufre de los siguientes problemas:
      - → Cambios en sus clases relacionadas se propagan a la clase en cuestión.
      - → Es más difícil de entender de manera aislada
      - → Es más difícil de reusar, ya que requiere a sus muchas dependencias.
- Solución
  - Asignar una responsabilidad tal que el acoplamiento se mantenga bajo. Usar este principio para evaluar alternativas.
- Consideraciones
  - Low Coupling fomenta asignar una responsabilidad de forma tal que no incremente el acoplamiento a un nivel que produzca resultados negativos típicos del alto acoplamiento.

- Ejemplo – Punto de Venta
  - Considerando el siguiente diagrama de clases (parcial)

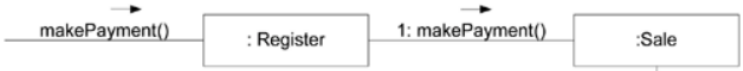


Supongamos que tenemos que crear una instancia de Payment y asociarla con la Sale… ¿Qué clase debería ser responsable de esto?

- Ejemplo – Punto de Venta (cont.)
  - **Diseño A:** Puesto que la registradora registra un pago en el mundo real, el patrón *Creator* sugiere la clase Register



  - **Diseño B:** Asigna la responsabilidad a Sale, quien se debe quedar con una referencia.



¿Qué diseño basado en la asignación de responsabilidades tiene un menor acoplamiento? **Diseño B**

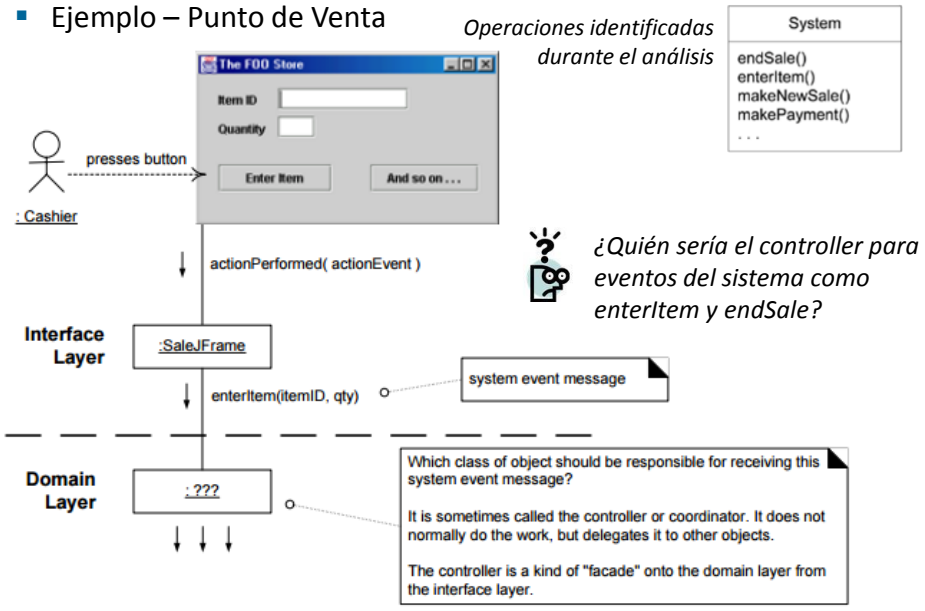
En la práctica, el nivel de acoplamiento no puede ser considerado de manera aislada de otros principios como Information Expert y High Cohesion. Sin embargo, es un factor a considerar para mejorar el diseño.

- Contraindicaciones
  - Un alto acoplamiento con elementos estables y generalizados rara vez es un problema. Por ejemplo, una aplicación JEE puede acoplarse de manera segura a las bibliotecas de Java (java.util, etc.), ya que son estables y generalizadas.
- Beneficios
  - Reduce el impacto de cambio en los componentes
  - Es simple de entender de manera aislada
  - Facilita el reuso

Controller

- Problema
  - ¿Cuál es el primer objeto más allá de la capa de UI que recibe y coordina (controla) una operación del sistema?
    - Un controller es ese primer objeto. ¿A quién se le asigna dicha responsabilidad?
- Solución
  - Asignar una responsabilidad a una clase que represente una de las siguientes alternativas:
    - *Facade Controller*: Representa al sistema general, a un objeto raíz, a un dispositivo dentro del cual está corriendo el software, o a un subsistema mayor
    - *Use Case Controller*: Representa un escenario de caso de uso dentro del cual se manejan eventos del sistema
      - → Frecuentemente denominado <*UseCaseName*>Handler, <*UseCaseName*>Coordinator o <*UseCaseName*>Session.

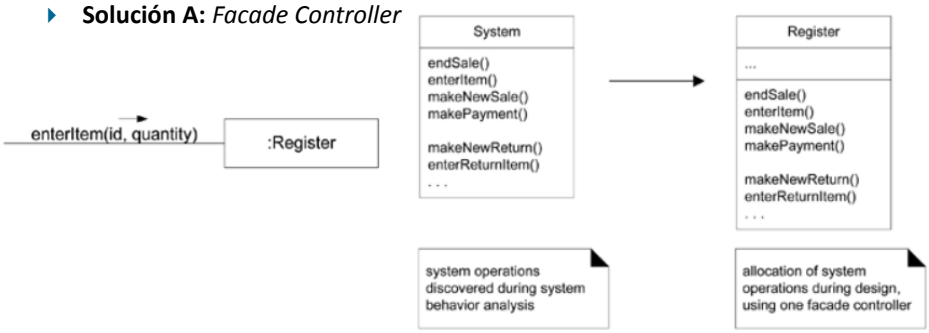
- Ejemplo – Punto de Venta



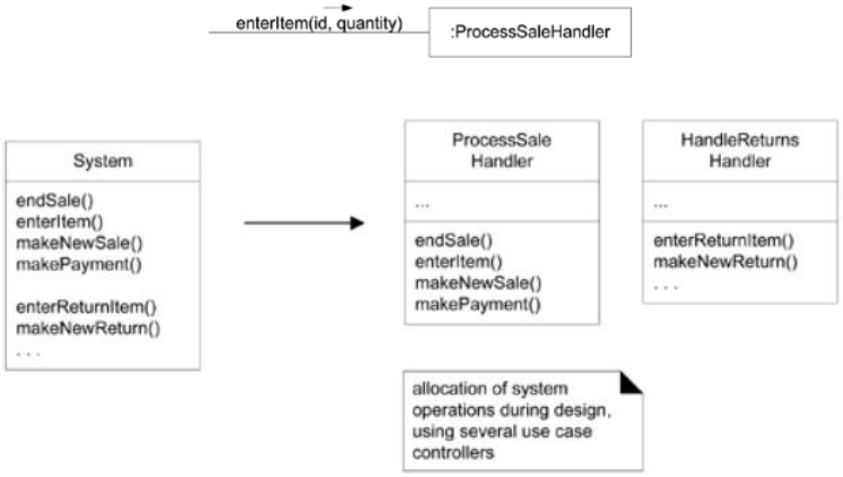
- Ejemplo – Punto de Venta (cont.)
  - De acuerdo a *Controller*, habría un par de opciones

Tipo	Descripción	Controller
Facade Controller	Representa al sistema general, a un objeto raíz, a un dispositivo o a un subsistema	Register, POSSystem
Use Case Controller	Representa a un receptor de todos los eventos de un escenario de caso de uso.	ProcessSaleHandler

  - **Solución A:** *Facade Controller*



- Ejemplo – Punto de Venta (cont.)
  - **Solución B:** *Use Case Controller*



- Consideraciones
  - Controller es un patrón de delegación
    - La Capa UI delega la atención de requerimientos a la Capa de Negocio
  - Generalmente se quiere usar la misma clase controller para todos los eventos del sistema provenientes de un caso de uso.
    - Esto es útil cuando se quiere identificar eventos fuera de secuencia (wizard)
  - Defecto común: Sobre-asignar responsabilidades

Normalmente un controller debería delegar a otros objetos el trabajo que necesita ser hecho. Solamente debería coordinar o controlar la actividad. No debería hacer mucho trabajo por sí mismo

- Facade Controller
  - La idea es elegir una abstracción (nombre de clase) que sugiera un punto de entrada a las otras capas de la aplicación
  - Ejemplos: Register, TelecommSwitch, Phone, Robot, POSSystem, ChessGame.
  - Son apropiados cuando…
    - No hay demasiados “eventos del sistema”
    - La UI no puede redirigir los mensajes de eventos del sistema a distintos controllers.
- Use Case Controller
  - Se tendrá un controller para cada caso de uso
  - Son apropiados cuando, por ejemplo, hay muchos eventos del sistema a lo largo de diferentes procesos. Use case controllers factorizan su atención en clases separadas y manejables, proveyendo las bases para conocer y razonar acerca del estado del escenario actual en progreso.

- Beneficios
  - Incrementa el potencial para el reuso y para interfaces pluggables.
    - La lógica de aplicación/negocio no se maneja a nivel UI
    - Permite reusar lógica de aplicación en futuras aplicaciones, ya que no está atada a la UI
    - Permite reemplazar la capa UI
  - Permite razonar acerca del estado del casos de uso
    - Para asegurar el secuenciamiento de determinadas operaciones de negocio

- Ejemplo de código - Struts

```
package com.craiglarman.nextgen.ui.web;

// In Struts, an Action object is associated with a web browser button click,
// and invoked (on the server) when the button is clicked.
public class EnterItemAction extends Action {
    // This is the method invoked on the server when the button is clicked on the client browser
    public ActionForward execute(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response) throws Exception {
        // The server has a Repository object that holds references to several things, including
        // the POS "register" object
        Repository repository = (Repository)getServlet().
                getServletContext().getAttribute(Constants.REPOSITORY_KEY);
        Register register = repository.getRegister();
        // Extract the itemID and qty from the web form
        String txtId = ((SaleForm)form).getItemID();
        String txtQty = ((SaleForm)form).getQuantity();
        // Transformer is a utility class to transform Strings to other data types
        ItemID id = Transformer.toItemID(txtId);
        int qty = Transformer.toInt(txtQty);
        // here we cross the boundary from the UI layer to the domain layer delegate to the
        // 'domain controller'
        register.enterItem(id, qty);
        // ...
    } // end of method
} // end of class
```

- Ejemplo de código – Java Swing

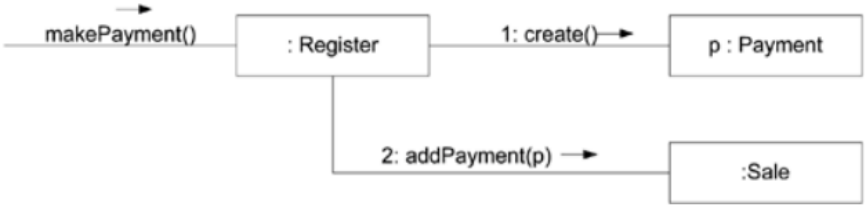
```
package com.craiglarman.nextgen.ui.swing;

public class ProcessSaleJFrame extends Jframe {
    private Register register;
    public ProcessSaleJFrame(Register _register) {
        register = _register;
    }
    private JButton btnEnterItem;
    private JButton getEnterItemButton() {
        if (btnEnterItem != null)
            return btnEnterItem;
        btnEnterItem = new JButton();
        btnEnterItem.setText("Enter Item");
        btnEnterItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                ItemID id = Transformer.toItemID(getTxtId().getText());
                int qty = Transformer.toInt(getTxtQty().getText());
                // Here we cross the boundary from the UI layer to the domain layer
                register.enterItem(id, qty);
            }
        } ); // end of the addActionListener call
        return btnEnterItem;
    } // end of method
// ...
} // end of class
```


High Cohesion

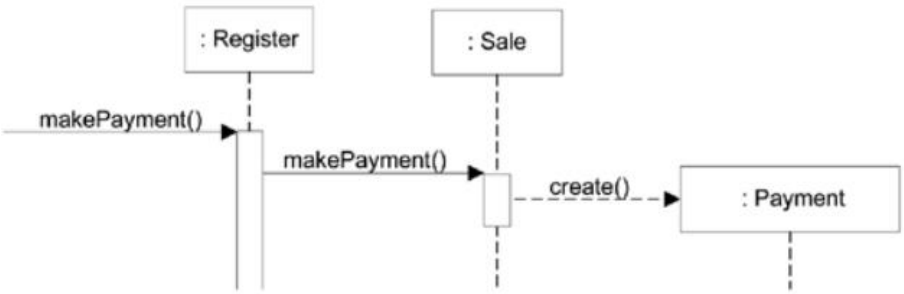
- Problema
  - ¿Cómo mantener los objetos enfocados, entendibles, manejables y, como efecto colateral, soportar Low Coupling?
    - Un elemento con responsabilidades altamente relacionadas y que no hace una excesiva cantidad de trabajo, tiene alta cohesión.
- Solución
  - Asignar una responsabilidad tal que la cohesión permanezca alta.
  - Usarla para evaluar alternativas
  - Una clase con baja cohesión, sufre los siguientes problemas:
    - Difícil de comprender
    - Difícil de reusar
    - Difícil de mantener
    - Constantemente afectada por los cambios.

- Ejemplo – Punto de Venta
  - Se pretende crear una instancia de Payment y asociarla con Sale. ¿Qué clase debería ser responsable de esto?
  - Puesto que la registradora registra un pago en el mundo real, el patrón *Creator* sugiere la clase Register



  - Esto es aceptable como un ejemplo aislado. Sin embargo…
  - ¿Qué sucede si Register tiene que tomar la responsabilidad de otras 50 operaciones?
    - Register pasa a ser un objeto poco cohesivo e inflado.

- Ejemplo – Punto de Venta (cont.)
  - El siguiente diseño delega la creación del Payment a Sale, mejorando la cohesión de Register.



  - Puesto que este último diseño soporta tanto alta cohesión como bajo acoplamiento, es el más deseable.

En la práctica, el nivel de cohesión no puede ser considerado de manera aislada de otros principios como Information Expert y Low Coupling.

- Contraindicaciones
  - En pocos casos, aceptar una baja cohesión puede estar justificado:
    - Agrupar responsabilidades o código para facilitar el mantenimiento
      - → Ejemplo: Poner las sentencias SQL en una clase.
    - Objetos distribuidos: Dado el overhead y performance asociados a los objetos remotos, a veces es deseable crear pocos, grandes y poco cohesivos objetos remotos que proveen una interfaz para muchas operaciones (*Coarse-Grained Remote Interface*)
      - → Ejemplo: o.setName, o.setSalary, o.setHireDate ➔ o.setData

- Beneficios
  - Mejora la claridad y la facilidad de comprensión del diseño
  - Simplifica el mantenimiento y la introducción de mejoras
  - Ayuda al bajo acoplamiento
  - Incrementa el reuso de funcionalidad altamente relacionada, ya que una clase cohesiva se puede usar para un propósito bien específico.

Polymorphism

- Problema
  - ¿Cómo manejar alternativas basadas en el tipo?
  - ¿Cómo crear componentes de software pluggables?

- Solución
  - Cuando alternativas o comportamientos relacionados varían por tipo (clase), asignar responsabilidades para el comportamiento (usando operaciones polimórficas) a los tipos para los cuales el comportamiento varía.
  - No chequear el tipo del un objeto, usando lógica condicional para realizar distintas alternativas en base dicho tipo

```
Switch creatureType
    Case batType: print "Screech!"
    Case cowType: print "Moooooo..."
    Case humanType: print "Let's watch TV!"
    [...]
```

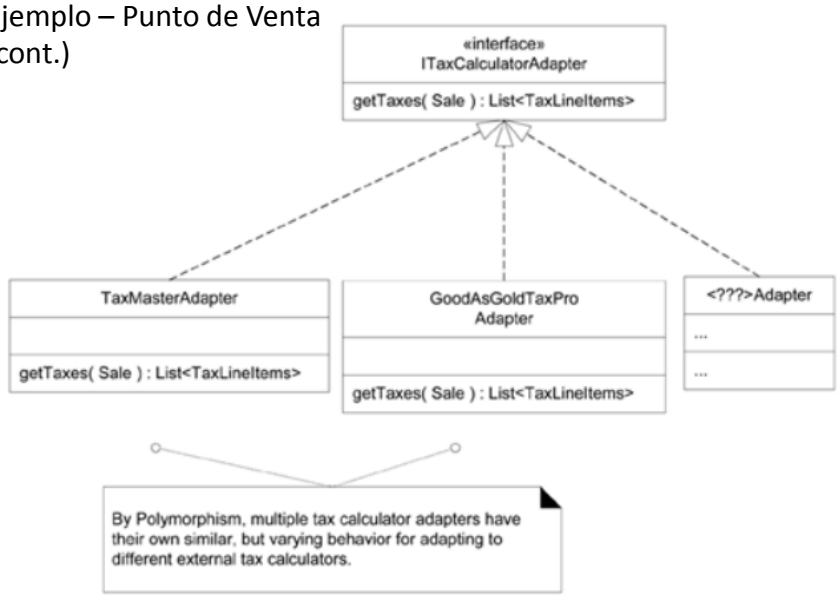
NOOO !

- Ejemplo – Punto de Venta
  - Existen múltiples calculadoras de impuestos de terceros que deben ser soportadas por el sistema. Cada calculadora tiene una interfaz diferente (SOAP, TCP, HTTP, Java RMI)

    ¿Qué objetos serían responsables de manejar estas variaciones en las interfaces de la calculadora de impuestos externa?

  - Por Polymorphism debemos asignar la responsabilidad para adaptarse a diferentes calculadoras a los objetos en sí mismo, e implementarla dentro de la operación polimórfica *getTaxes*

- Ejemplo – Punto de Venta (cont.)



- Contraindicaciones
  - Sobre-diseño: Diseñar sistemas con interfaces y polimorfismos para eventuales posibles variaciones futuras.
    - Si el punto de variación es muy probable, entonces el esfuerzo es correcto. De lo contrario, no.
    - Ser realistas sobre la probabilidad de variación antes de invertir en flexibilidad.
- Beneficios
  - Las extensiones requeridas para nuevas variaciones son fáciles de agregar.
  - Nuevas implementaciones pueden ser introducidas sin afectar a los clientes.

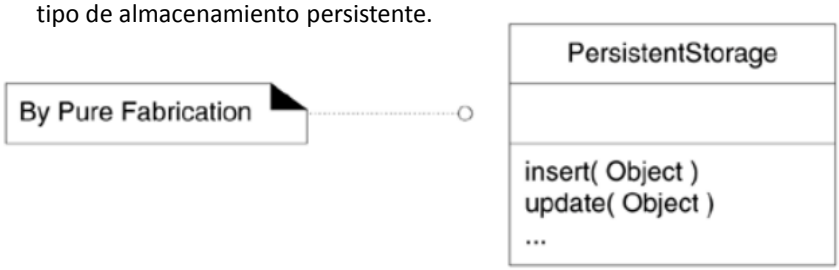
Pure Fabrication

- Problema
  - ¿Qué objetos deberían tener la responsabilidad cuando no se quiere violar High Cohesion, Low Coupling o otras metas, pero la solución provista por Information Expert (por ejemplo) no es apropiada?

- Solución
  - Asignar un conjunto de responsabilidades altamente cohesivas a una clase artificial (o por conveniencia) para el dominio, es decir, que no represente un concepto del dominio del problema
  - Esto tiene como objetivo obtener una solución con más alta cohesión, menor acoplamiento y mejor reuso.

- Ejemplo – Punto de Venta
  - Supongamos que es necesario guardar las instancias de Sale en una base de datos relacional. ¿Quién debería tomar dicha responsabilidad?
  - **Solución – Information Expert**
    Sería la misma clase Sale.
    Consecuencias
    - *Pérdida de cohesividad:* La tarea requiere muchas operaciones orientadas a la base de datos que no están relacionadas al concepto de venta.
    - *Mayor acoplamiento:* La clase Sale tiene que acoplarse a la interfaz de la base de datos (por ej., JDBC)
    - *Poco reuso:* La persistencia en una base de datos relacional es una tarea que muchas clases necesitarían. Ubicar la responsabilidad en Sale implica poco reuso o gran duplicación de código.

- Ejemplo – Punto de Venta
  - **Solución – Pure Fabrication**
    Crear una nueva clase que sea responsable de guardar objetos en algún tipo de almacenamiento persistente.



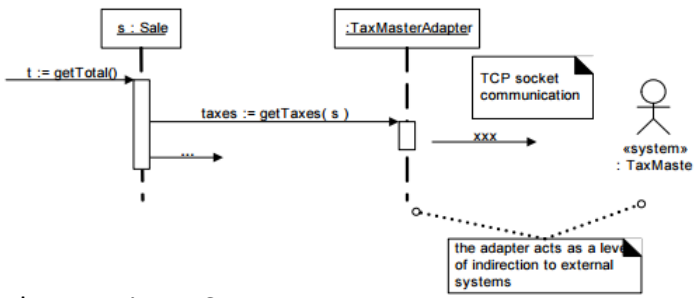
  - Consecuencias
    - *Sale* permanece bien diseñada, con alta cohesión y bajo acoplamiento
    - PersistentStorage es relativamente cohesiva, teniendo como único objetivo el almacenamiento de los objetos
    - PersistentStorage es una clase muy genérica y reusable.

- Otros Ejemplos
  - *TableOfContentsGenerator* para generar la tabla de contenido de un documento (por algoritmo)
  - Muchos *patrones de diseño GoF* son Pure Fabrications: Adapter, Strategy, Command, etc.
- Contraindicaciones
  - Sobre-uso: Especialmente por aquellas personas nuevas al OOD y más familiares con la descomposición del sw en término de funciones.
    - Puede conducir a tener muchos objetos de comportamiento que no tienen responsabilidades co-locadas con la información necesaria para su cumplimiento… afectando fuertemente el acoplamiento.
- Beneficios
  - Soporta alta cohesión porque las responsabilidades son factorizadas en clases separadas, enfocándose cada una en un conjunto de tareas relacionadas.
  - Incrementa la posibilidad de reuso.

Indirection

- Problema
  - ¿Dónde asignar una responsabilidad para evitar un acoplamiento directo entre dos (o más) cosas?
  - ¿Cómo desacoplar objetos para lograr bajo acoplamiento y alto reuso?
- Solución
  - Asignar la responsabilidad a un objeto intermedio para mediar entre otros componentes o servicios tal que no queden directamente acoplados
    - El intermediario crea una indirección entre los componentes involucrados

- Ejemplo – TaxCalculatorAdapter
  - Estos objetos actúan como intermediarios a las calculadoras de impuestos externas.
  - A través del polimorfismo proveen una interfaz consistente, ocultando las variaciones de las APIs



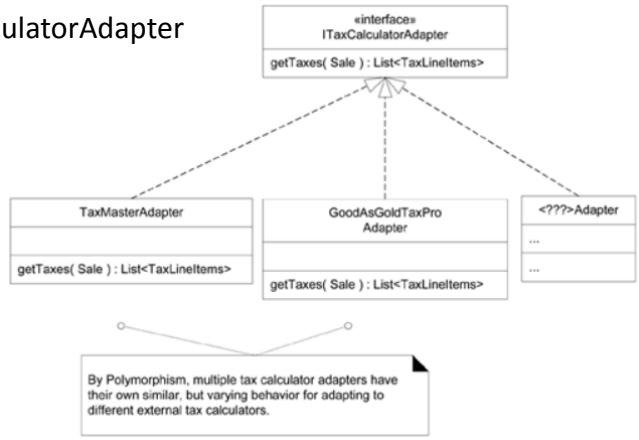
- Ejemplo – PersistentStorage
  - La clase *PersistentStorage* actúa como un intermediario entre *Sale* y la base de datos.

- Consideraciones
  - Muchos patrones de diseño GoF son especializaciones de Indirection: Adapter, Facade, Observer, Bridge, Mediator etc.
- Beneficios
  - Disminuye el acoplamiento entre componentes.

Protected Variations

- Problema
  - ¿Cómo diseñar objetos, subsistemas y sistemas tal que variaciones e inestabilidad en estos elementos no produzca un impacto indeseable en otros elementos?
- Solución
  - Identificar puntos de probable variación o inestabilidad
  - Asignar responsabilidades para crear una interfaz estable alrededor de ellos.
  - PV es un principio raíz que motiva a muchos mecanismos y patrones en la programación y diseño para proveer flexibilidad y protección de variaciones en datos, comportamiento, hardware, componentes de software, sistemas operativos, etc.

- Ejemplo – TaxCalculatorAdapter



  - El punto de inestabilidad es las diferentes interfaces (o APIs) de las calculadoras de impuestos externas.
  - La protección contra cambios en las APIs externas se realiza agregando una interfaz y usando polimorfismo

- Mecanismos Motivados por Protected Variations
  - Mecanismos Básicos
    - Encapsulamiento de datos, interfaces, polimorfismo, indirección y especificación y uso de estándares son motivados por PV.
  - Diseños Manejados por Datos
    - Incluyen varias técnicas como leer códigos, valores, path de archivos de clases, etc, desde un origen externo para cambiar el comportamiento o parametrizar el sistema en ejecución
    - El sistema queda protegido del impacto de los datos, metadatos o variaciones declarativas **externalizando la variación**, leyéndola y actuando en base a ella.
  - Búsqueda de Servicios
    - Los clientes de un servicio se protegen de **variaciones en la ubicación del servicio**, utilizando una interfaz estable de un servicio de búsqueda (Java-JNDI, UDDI para web services)
  - Diseños Manejados por Intérpretes
    - Permite parametrizar o cambiar el comportamiento de un sistema a través de **expresiones lógicas externas**.
    - El sistema se protege del impacto en variaciones de la lógica externalizando la lógica, leyéndola y usando un intérprete para ejecutarla.

- Mecanismos Motivados por Protected Variations (cont.)
  - Diseños Reflectivos
    - Utilzan *reflection* para acceder a la estructura interna de un objeto: campos, propiedades, métodos, etc.
    - Protegen al sistema del impacto de variaciones en la lógica o estructura de los objetos a través de algoritmos reflectivos que usan introspección.
  - Acceso Uniforme
    - Algunos lenguajes de programación (Ada, Eiffel y C#) soportan construcciones sintácticas que permiten invocar de la misma manera a un método o a un campo.
    - Esto permite pasar de un campo a un método sin cambiar al cliente.
  - Lenguajes Estándar
    - Lenguajes como SQL proveen protección contra la proliferación de diferentes lenguajes para acceder a bases de datos.

- Contraindicaciones
  - Puntos de evolución especulativos
    - A veces el costo de protegerse de futuros cambios sobrepasa al costo de retrabajo necesario para modificar un diseño simple en respuesta a un cambio real.
    - Si la necesidad por flexibilidad y protección del cambio es real, entonces el uso de PV está fundamentado. Pero si es para protegerse de futuros cambios o reuso con probabilidades muy inciertas, se debe reflexionar seriamente si es necesario.
- Beneficios
  - Extensiones requeridas para nuevas variaciones son fáciles de agregar
  - Nuevas implementaciones pueden ser introducidas sin afectar a los clientes.
  - Se disminuye el acoplamiento
  - Se disminuye el impacto y el costo de cambio