Patrones GRASP

Patrones GRASP – Caso Real de Uso

Los Casos Real de Uso presentan un diseño concreto de cómo se realizará el caso. Es una de las primeras actividades en el ciclo de desarrollo. Su creación depende de los casos esenciales generados con anterioridad.

Un Caso Real de Uso describe el diseño concreto del caso de uso a partir de una tecnología particular de entrada y salida, y su implementación global. Por ejemplo, si hay una interfaz gráfica para el usuario, el caso de uso real incluirá diagramas de las ventanas en cuestión y una explicación de la interacción.

Patrones GRASP.

PATRONES: Es una descripción de un problema y su solución que recibe un nombre y que puede emplearse en otros contextos, con una sugerencia sobre la manera de usarlo en situaciones nuevas.

No suelen contener ideas nuevas*:* Por el contrario, intentan codificar el conocimiento, las expresiones y los principios ya existentes y trillados.

Tienen nombre*: A*signar un nombre sugestivo a un patrón ofrece las siguientes ventajas:

Apoya el agrupamiento y la incorporación del concepto a nuestro sistema cognitivo y la memoria.

Facilita la comunicación.

Patrones GRASP

GRASP: *General Responsability Asignment Software Patterns*

Patrones de Principios Generales Para Asignar Responsabilidades.

- Se aplican con los Diagramas de Interacción (secuencia y comunicación)

Patrones GRASP, responsabilidades?

Responsabilidad

- Contrato u obligación de un tipo o clase
- Las responsabilidades se asignan a los objetos durante el **diseño**.

Las responsabilidades se dividen en:

- Responsabilidad de **Conocer**.
- Responsabilidad de **Hacer**.

Patrones GRASP. Responsabilidades de Conocer

Responsabilidades de un objeto relacionadas con conocer:

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

Patrones GRASP. Responsabilidades de Hacer

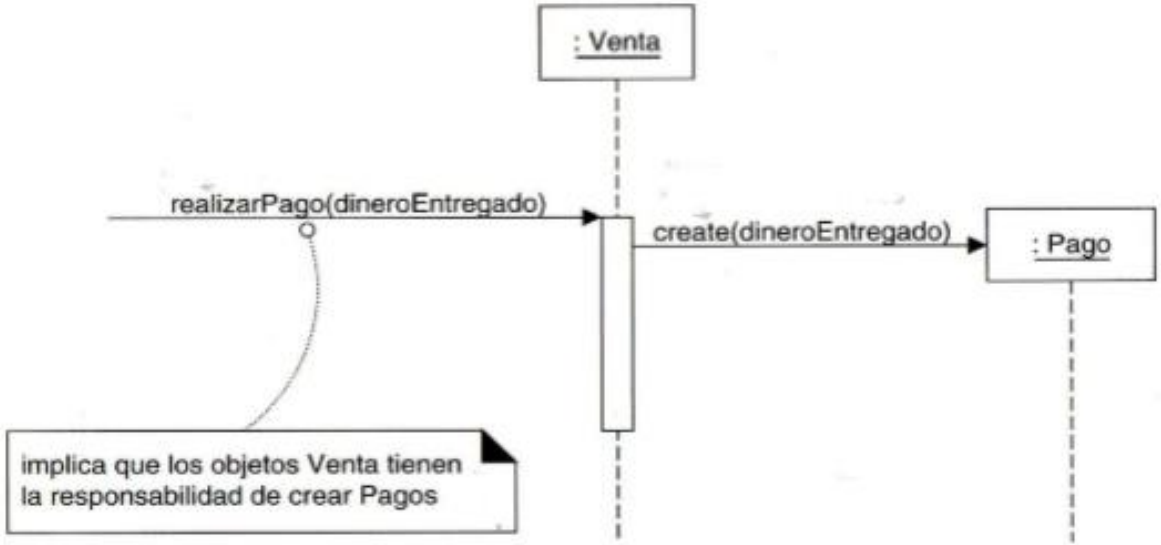
Responsabilidades de un objeto relacionadas con Hacer:

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

Patrones GRASP. Responsabilidades y Métodos

Responsabilidad

- Las responsabilidades y los métodos están relacionados



Patrones GRASP mas comunes

- **Experto**
- **Creador**
- **Alta Cohesión**
- **Bajo Acoplamiento**
- **Controlador**

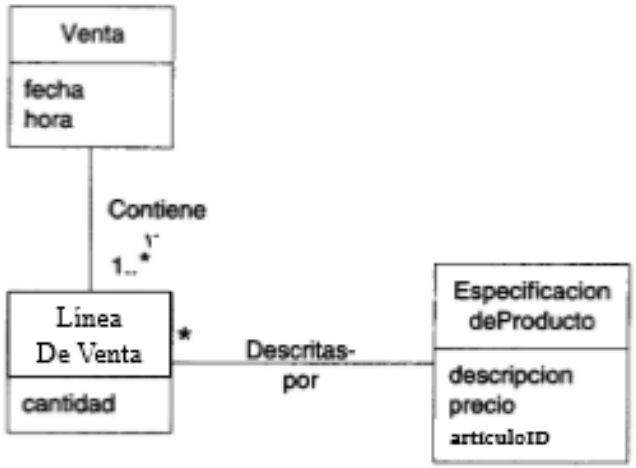
Patrones GRASP: *"Experto"*

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

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

Ejemplo PDV (Larman)

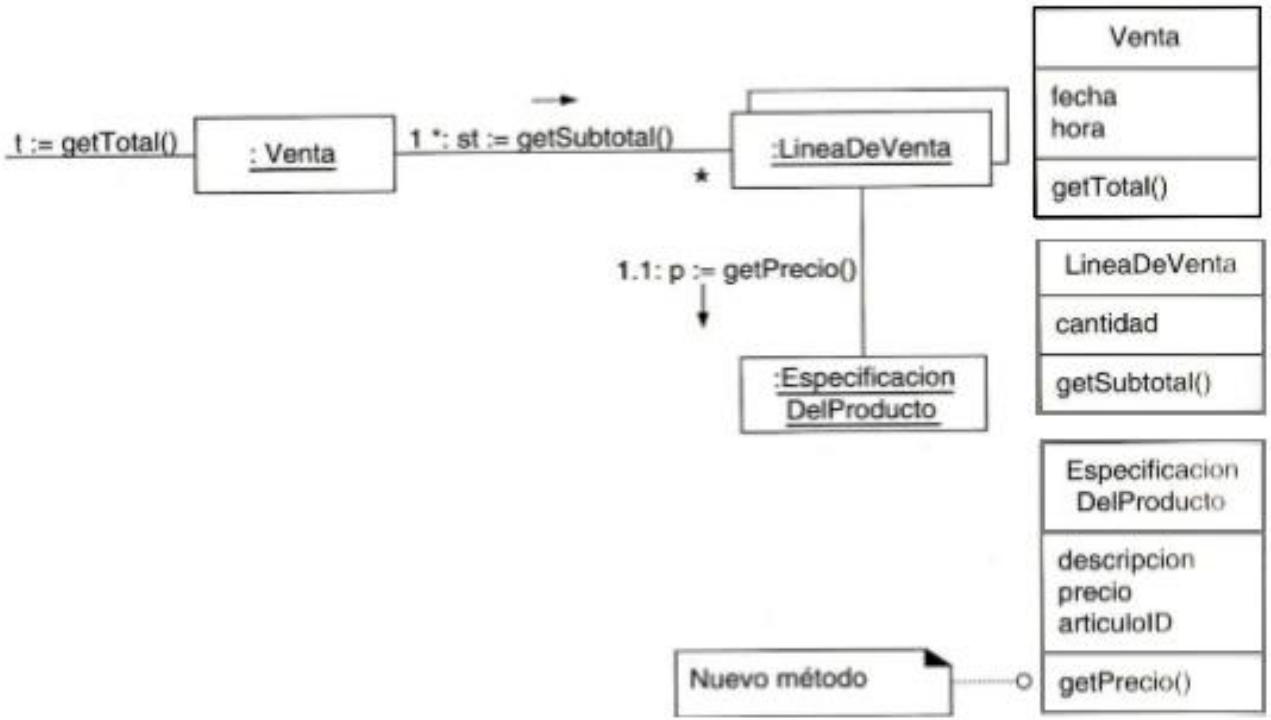
¿Quién es el responsable de conocer el total de una venta?



Patrones GRASP: *"Experto"*

Ejemplo PDV (Larman)

¿Qué información hace falta para calcular el total?



Patrones GRASP: “Experto”

- El patrón **Experto** expresa la **“intuición”** de que los objetos hacen cosas relacionadas con la información que poseen.

- Nótese que el cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos (hay muchos expertos “parciales” que colaboran en la tarea).

- El patrón experto ofrece una analogía con el mundo real. Acostumbramos a asignar responsabilidades a individuos que disponen de la información necesaria para llevar a acabo una tarea.

Patrones GRASP: “Experto”

- Otras formas de designar el patrón

  “Juntar responsabilidad y la información”

  “Lo hace el que conoce”

  “Lo hago yo mismo”

  “Unir los servicios a los atributos sobre los que operan”
- 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 el hecho de tener sistemas más robustos y de fácil mantenimiento.
  - El comportamiento se distribuye entre las clases que cuentan con la información requerida, alentando 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**.

Patrones GRASP: "*Creador*"

Problema

¿Quién es responsable de crear una nueva instancia de alguna clase?.

Solución

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

- B "agrega" objetos A.
- B "contiene" objetos A.
- B "registra" las instancias de los objetos de A.
- B "utiliza" estrechamente los objetos de A.
- B "tiene los datos de inicialización" que pasarán a A cuando sea creado (por tanto, B es un Experto con respecto ala creación de A).
- B es un creador de los objetos de A.

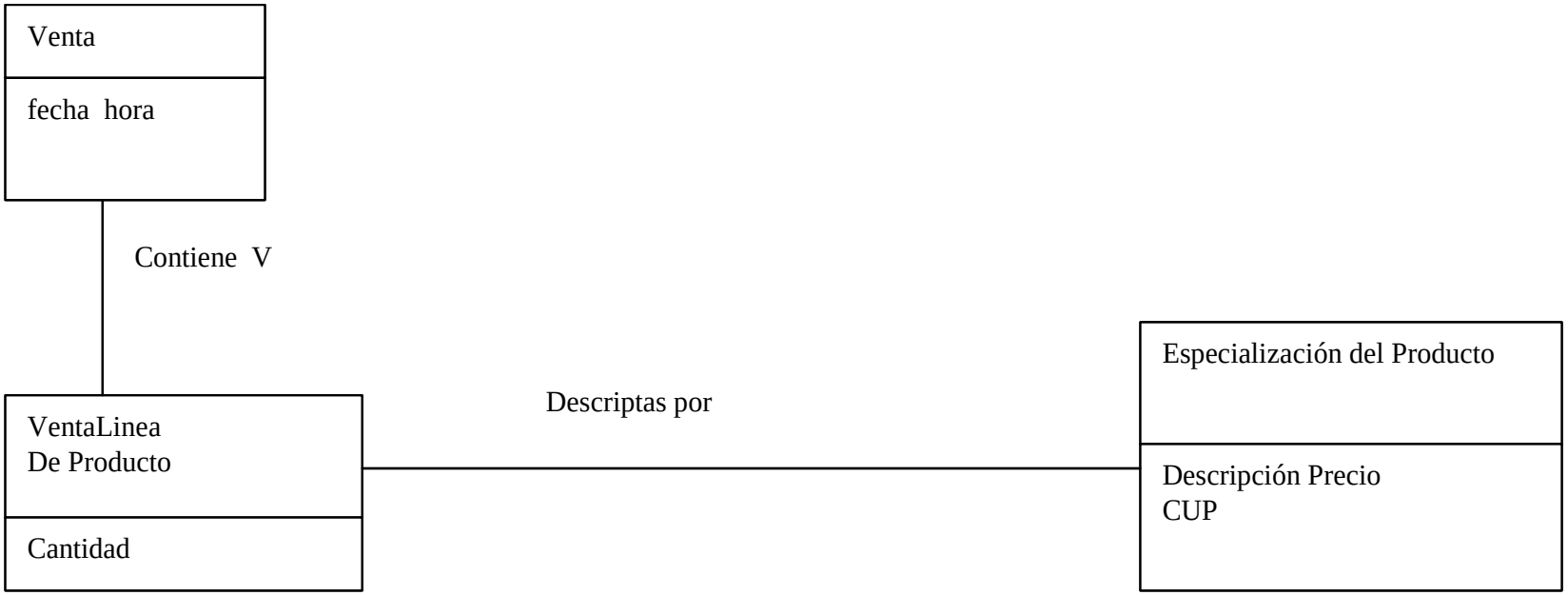
Patrones GRASP: "*Creador*"

Ejemplo (punto de ventas de Larman)

- En el ejemplo de punto de ventas. ¿Quién debería encargarse de crear una instancia Línea de Venta?.

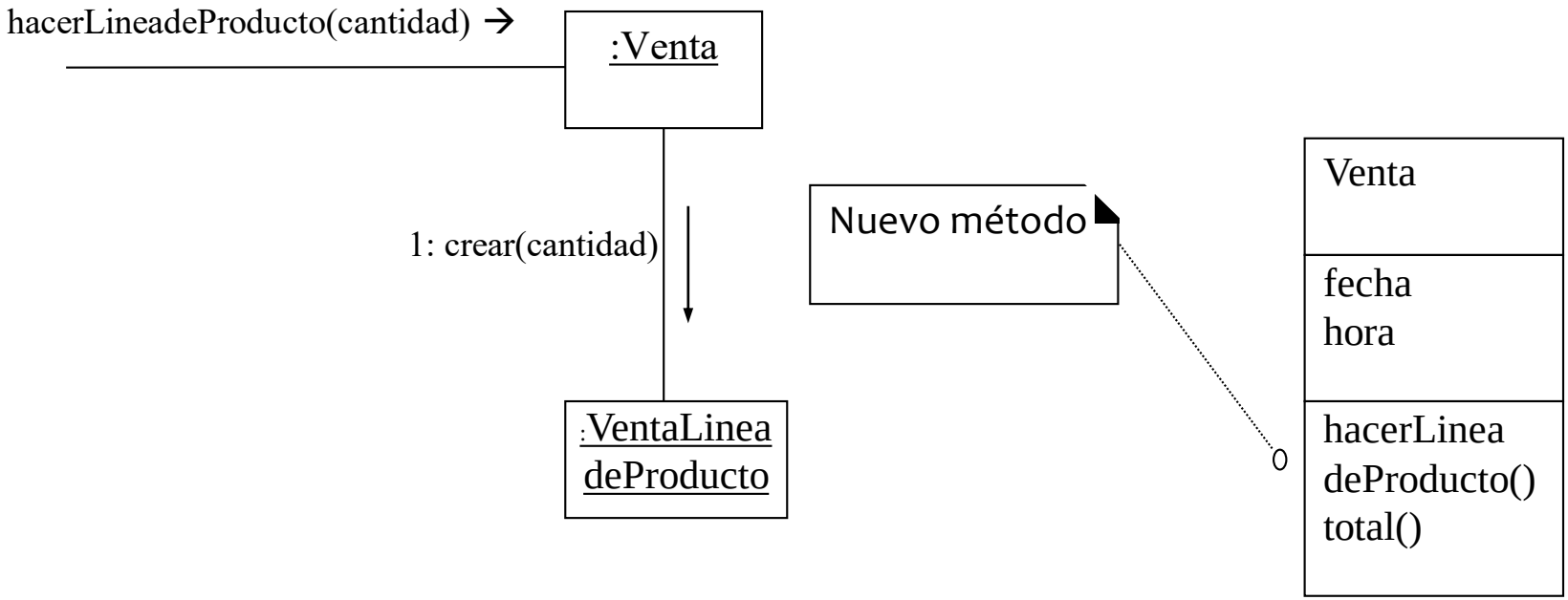
- Desde el punto de vista del patrón Creador, deberíamos buscar una clase que agregue, contenga y realice otras operaciones sobre este tipo de instancias.

Patrones GRASP: "*Creador*"



En este Modelo Conceptual una Venta contiene muchos objetos VLP; por ello el patrón ***Creador*** sugiere que ***Venta*** es idónea para asumir la responsabilidad de crear la instancia VLP, y a partir de aquí podemos diseñar las interacciones de los objetos en un diagrama de colaboración.

Patrones GRASP: "*Creador*"



Al asignar responsabilidades en un diagrama de interacción=> definimos en Venta un Método de ***"hacerlíneadeProducto"***.

Patrones GRASP: "*Creador*"

Explicación

- El propósito de este patrón es asignar responsabilidades relacionadas con la **creación de objetos producidos en cualquier evento**.
- El patrón **Creador** indica que una clase es idónea para asumir la responsabilidad de crear la **cosa contenida o registrada**.

Beneficios

- Facilita un **Bajo Acoplamiento**, => supone menos dependencias respecto al mantenimiento y mejores oportunidades de reutilización.

•NOTA: Un constructor por defecto es una subrutina cuya misión es inicializar un objeto de una clase sin parámetros, que no hace nada.
Sin embargo será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores predeterminados por el sistema (los atributos numéricos a ceros, los alfanuméricos a nulos, y las referencias a objetos a null).

Patrones GRASP: "*Bajo Acoplamiento*"

Problema:

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

El acoplamiento es una medida de la fuerza con que una clase está conectada a otras clases, con que las conoce y con que recurre a ellas. Una clase con bajo (o débil) acoplamiento no depende de "muchas otras".

Una clase con alto (o fuerte) acoplamiento (depende de "muchas otras") presentan los siguientes problemas:

- Los cambios de las clases afines ocasionan cambios locales.
- Son más difíciles de entender.
- Son más difíciles de reutilizar porque se requiere la presencia de otras clases de las que dependen.

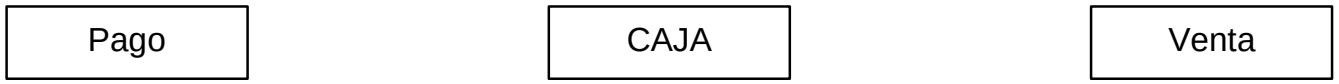
Solución:

Asignar una responsabilidad para mantener BAJA ACOPLAMIENTO

Patrones GRASP: "*Bajo Acoplamiento*"

EjemploSolución

Asignar responsabilidad para mantener bajo el acoplamiento.

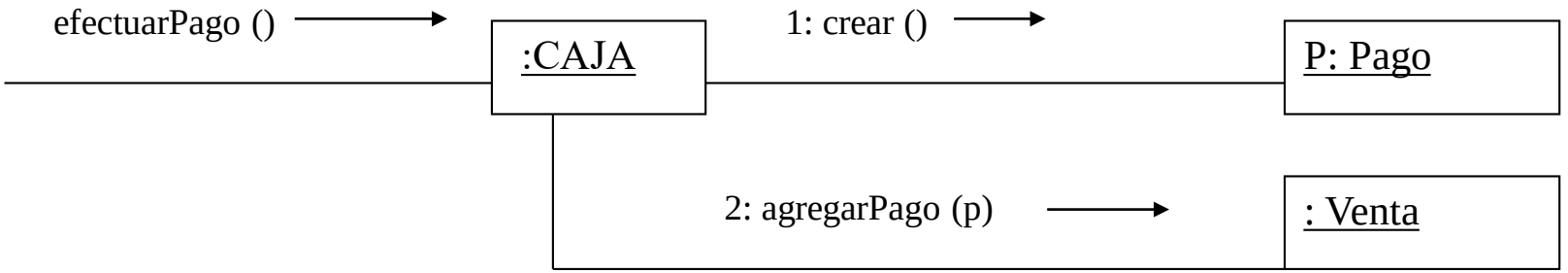


Suponga que necesitamos crear una instancia *Pago* y asociarla a *Venta*. ¿Qué clase se encargará de hacer esto?

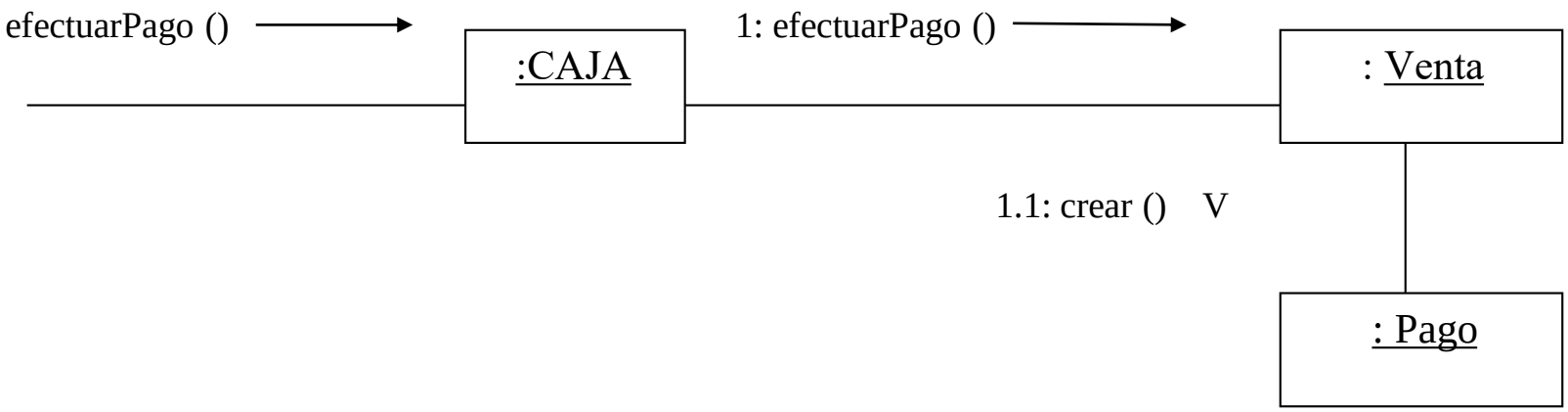
Como una instancia de *CAJA* **"registra"** un *Pago*, el patrón **Creador** indica que *CAJA* es un buen candidato para producir el *Pago*. La instancia *CAJA* podría entonces enviarle a *Venta* el mensaje: *agregarPago*, y con *Pago* como parámetro.

Patrones GRASP: "*Bajo Acoplamiento*"

Solución que acopla la clase CAJA



Solución Alternativa



Patrones GRASP: "*Bajo Acoplamiento*"

Explicación

- El Bajo Acoplamiento es un patrón **evaluativo** que el diseñador aplica al juzgar sus decisiones de diseño.
- El Bajo Acoplamiento soporta el diseño de clases **más independientes**, que reducen el **impacto de los cambios**, y también **más reutilizables**, que acrecientan la oportunidad de una mayor productividad.
- No puede considerarse en forma independiente como Experto o Alta Cohesión.
- En términos generales, han de tener escaso acoplamiento las clases muy genéricas y con grandes probabilidades de reutilización.

- El caso extremo de dicho patrón ´bajo acoplamiento´ **ocurre cuando existe poco o nulo acoplamiento**. **No conviene, pues se tendrá objetos que hacen de todo, complejos, o muy pasivos**

Patrones GRASP: "*Alta Cohesión* "

Problema

¿Cómo mantener la complejidad dentro de límites manejables?:

En el DOO, la **cohesión** es una medida de cuán **relacionados** y enfocados están las **responsabilidades** de una **clase**.

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

A menudo representan un alto grado de abstracción o han asumido responsabilidades que deberían haber delegado.

Patrones GRASP: "*Alta Cohesión* "

Una clase con **baja cohesión** hace muchas cosas no afines o un trabajo excesivo.

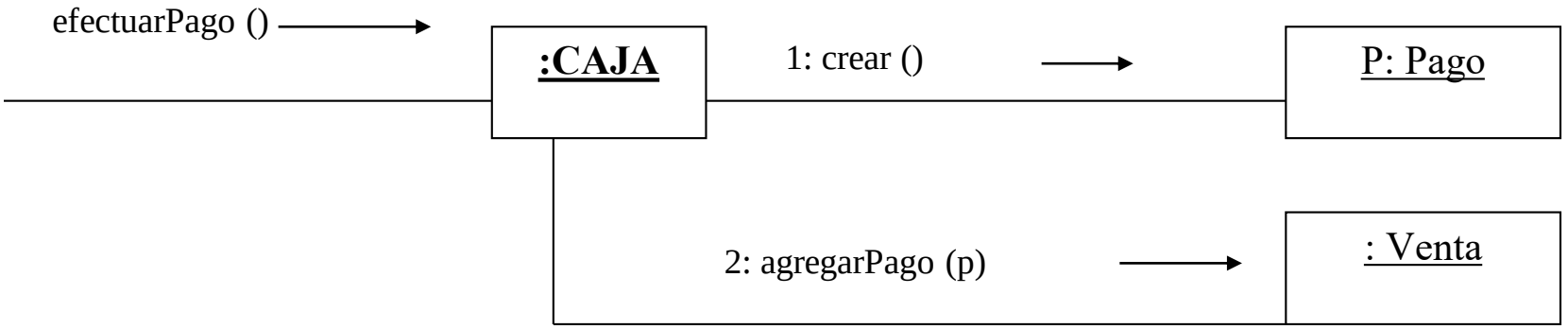
No conviene este tipo de clases, porque:

- Son difíciles de comprender.
- Son difíciles de reutilizar.
- Son difíciles de conservar.
- Son delicadas: las afectan constantemente los cambios.

Solución:

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

Patrones GRASP: "*Alta Cohesión* "

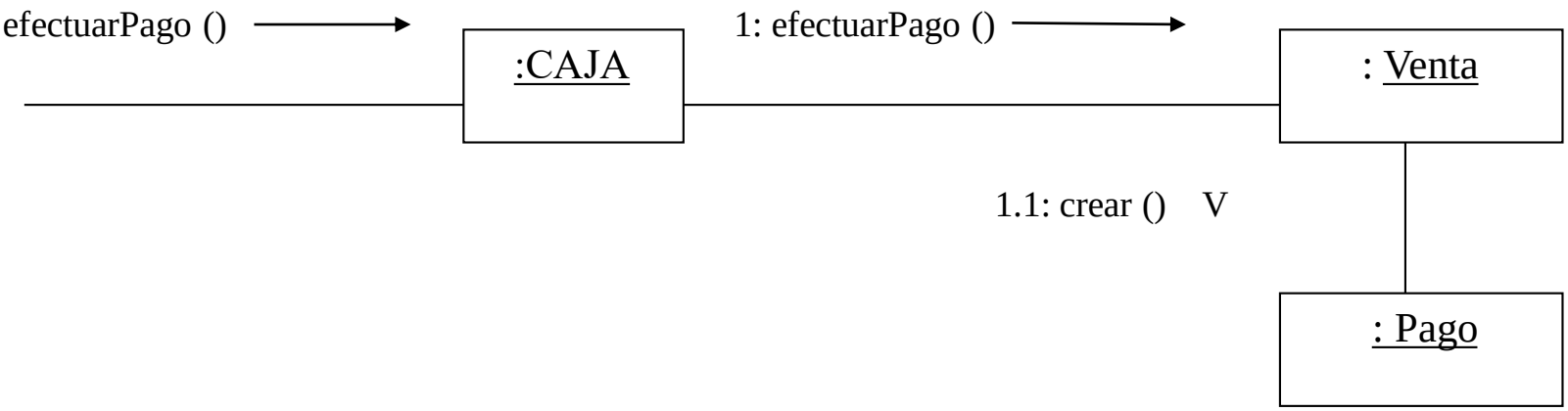


Esta asignación de responsabilidad coloca en la clase *CAJA* la generación del pago. (patrón ***Creador***).

Esto es aceptable. Pero si seguimos haciendo que la clase *CAJA* se encargue de efectuar algún trabajo con un número creciente de operaciones del sistema, se irá saturando con tareas y terminará por perder la cohesión.

En cambio, en el 2º ejemplo, se delega a la *Venta* la responsabilidad de ***crear*** el pago. Esto brinda un soporte a una **Alta Cohesión** y a un **Bajo Acoplamiento**.

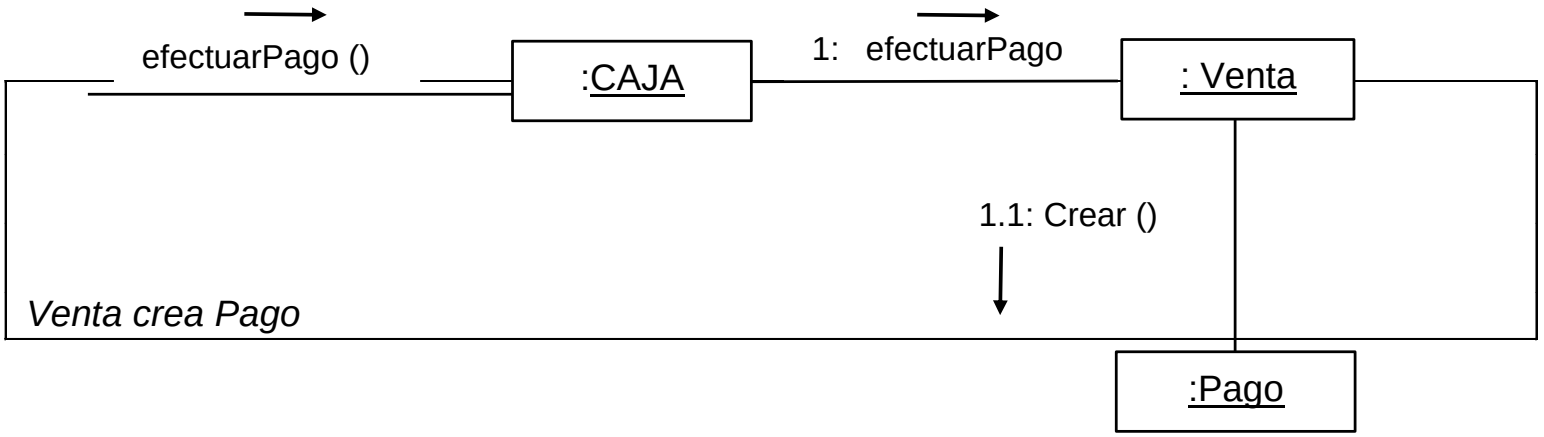
Patrones GRASP: "*Alta Cohesión*"



En cambio, en el 2º ejemplo, se delega a la *Venta* la responsabilidad de ***crear*** el pago. Esto brinda un soporte a una **Alta Cohesión** y a un **Bajo Acoplamiento**.

Patrones GRASP: "*Alta Cohesión*"

El nivel de cohesión no puede ser considerado independientemente del **Experto** y del **Bajo Acoplamiento**.



Patrones GRASP: "*Alta Cohesión*"

Grados de cohesión:

- **Muy baja Cohesión**: una **clase** es la **única responsable** de muchas cosas distintas en áreas funcionales diferentes (se deben dividir las responsabilidades en una familia de clases relacionadas).

- **Baja Cohesión**: una **clase** tiene la **responsabilidad exclusiva** de una **tarea compleja** dentro de una área funcional (dividir la clase en una familia de clases ligeras que compartan el trabajo).

Alta Cohesión: una **clase tiene responsabilidades moderadas** en un área y colabora con las otras para realizar las tareas (supuesto ideal

Patrones GRASP: "*Alta Cohesión* "

Una **clase de Alta Cohesión** posee un número relativamente pequeño, con una importante funcionalidad relacionada y poco trabajo por hacer. **Colabora con otros objetos para compartir el esfuerzo si la tarea es grande**.

Una **clase** con mucha **Cohesión** es útil porque es bastante **fácil** darle **mantenimiento, entenderla y reutilizarla**.

El patrón Alta Cohesión presenta semejanzas con el mundo real. Sabemos que si alguien asume demasiadas responsabilidades, no será eficiente (hay que **delegar**).

Patrones GRASP: "*Alta Cohesión* "

Beneficios:

- Mejorar la claridad y la facilidad con que se entiende el diseño.
- Se simplifican el mantenimiento y las mejoras en .
- A menudo se genera un Bajo Acoplamiento.
- Mayor capacidad de reutilización.

Patrones GRASP: "*Controlador*"

Problema:

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

Un **evento del sistema** es un suceso generado por un actor externo (evento de entrada externo). Se asocia a **operaciones del sistema** => respuestas a los **eventos del sistema**.

Por ejemplo: cuando un cajero oprime el botón ***"Terminar Venta"***, está generando un evento que indica que ***"la venta ha terminado"***.

Patrones GRASP: "*Controlador*"

Problema:

¿Quién debería ser el responsable de gestionar un evento de entrada al sistema?

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:

1) **El "sistema global" (controlador de fachada).**
2) **La empresa u organización global** (**controlador de fachada**).
3) **Algo en el mundo real que es activo** y que pueda participar en la tarea (**controlador de tareas**).
4) **Un manejador artificial** de todos los eventos del sistema de un caso de uso: ***"Manejador < NombreCasodeUso"*** (**controlador de casos de uso**).

Use la misma clase de controlador con todos los eventos del sistema en el mismo Caso de Uso.

Patrones GRASP: "*Controlador*"

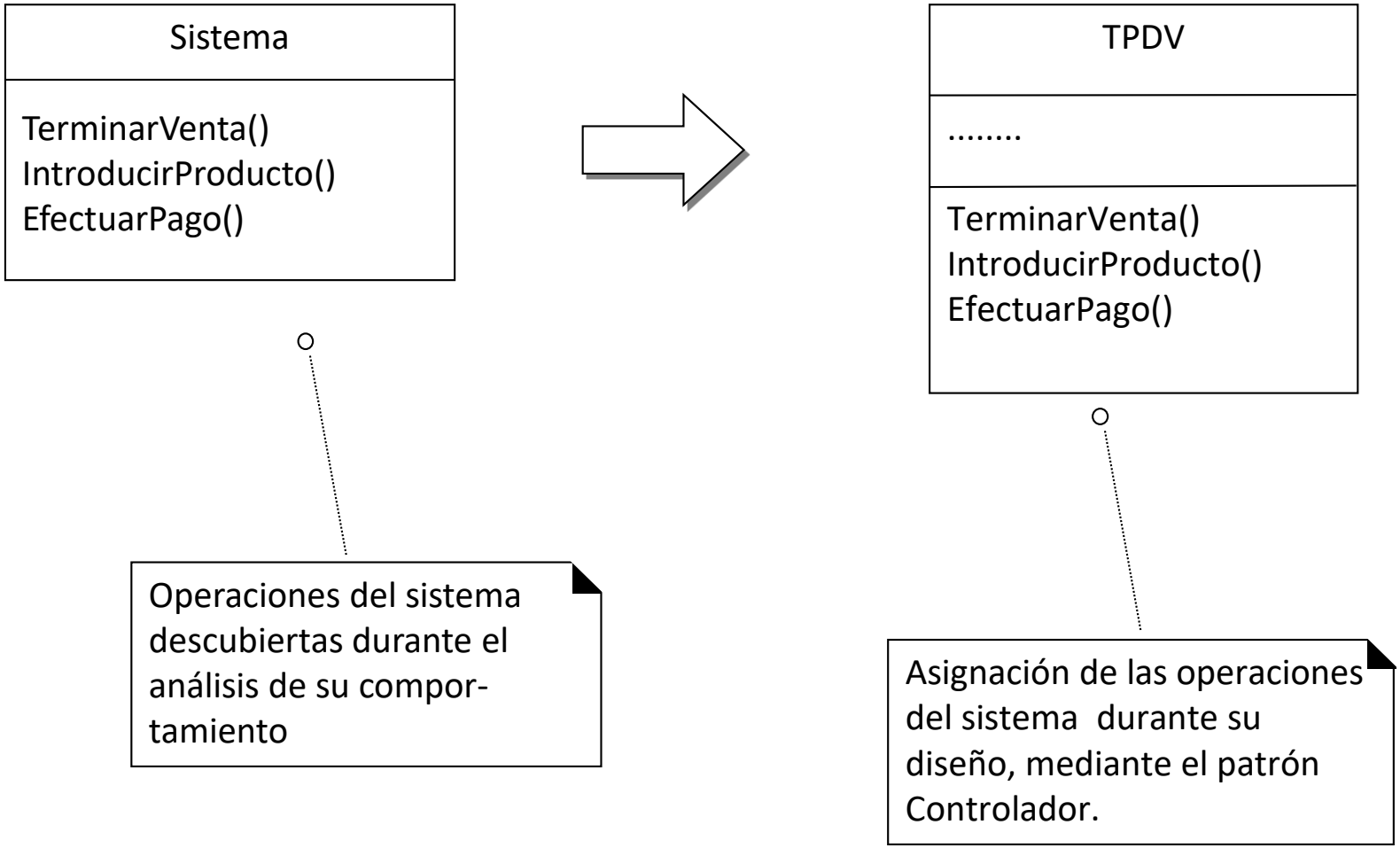
ejemplo:
¿Quién debería ser el responsable de gestionar un evento de entrada al sistema?

Sistema
finalizarVenta() IntroducirArtículo() crearNuevaVenta() realizarPago()

Durante el **análisis**, estas operaciones son asignadas al tipo **Sistema** para indicar que son operaciones del sistema. Pero ello no significa que una clase llamada Sistema las ejecute durante el diseño.

Durante el **diseño**, a la clase **Controlador** se le asigna la **responsabilidad** de las operaciones del **sistema**.

Patrones GRASP: "*Controlador*"



Patrones GRASP: "*Controlador*"

En **DOO** hay que **elegir los controladores** que manejen los eventos de entrada. Este patrón ofrece una guía para tomar decisiones apropiadas.
La misma clase controlador debería utilizarse con todos los eventos de **un Caso de Uso,** es decir **un controlador por CU** (recomendable**).**

Un defecto frecuente al diseñar controladores consiste en asignarles demasiadas responsabilidades. Normalmente, un controlador debería delegar a otros objetos el trabajo que ha de realizarse mientras coordina la actividad.

El **controlador de fachada** que representa al **"sistema"** global, es una clase que, para el diseñador, representa de alguna manera al sistema entero (unidad física: TPDV, Sistema, etc.). Son adecuados cuando el sistema sólo tiene pocos eventos o cuando es imposible redirigir los mensajes de los eventos del sistema a otros controladores.

Patrones GRASP: "*Controlador*"

En el "***Manejador artificial de Casos de Uso***", hay **un controlador para cada Caso de Uso**. No es un objeto del dominio, sino un concepto artificial cuyo fin es dar soporte al sistema (una fabricación pura).

Por ejemplo, en el Caso de Uso ***ComprarProducto*** habrá una clase:

ManejadordeComprarProducto.

Es **conveniente** elegirlo, cuando un **controlador** comienza a "**saturarse**" con demasiadas responsabilidades. Este controlador constituye una buena alternativa cuando hay muchos eventos de sistemas entre varios procesos: asigna su manejo a clases individuales controlables.

Patrones GRASP: "*Controlador*"

Beneficio Principal:

Mayor potencial de los **componentes reutilizables**. Garantiza que los eventos del sistema sean manejados por la capa de los objetos del dominio y no por la de la interfaz.
Un diseño de interfaz como controlador reduce la posibilidad de reutilización, por estar ligado a una interfaz determinada que rara vez puede reutilizarse en otras aplicaciones.
En cambio, el hecho de **delegar en un controlador la responsabilidad de la operación de un sistema, entre las clases del dominio, permite una mejor reutilización de la lógica.**

Patrones GRASP: "*Controlador*"

Controladores Saturados:

Es una clase de controlador mal diseñada, presenta Baja Cohesión: está dispersa y tiene demasiadas áreas de responsabilidad.

Signos de Saturación:

- Una **sola** clase controlador recibe todos los eventos que son excesivos. Ejemplo: suele ocurrir con los controladores de papeles o de fachada.

- El **controlador** realiza él mismo muchas tareas necesarias para cumplir el evento del sistema, sin que delegue trabajos (violación de los patrones Experto y Alto Acoplamiento).

- Un controlador posee muchos atributos y conserva información importante sobre el sistema que debería haber sido redistribuida entre objetos – y duplica la información.

Patrones GRASP: "*Controlador*"

- **Soluciones:**

- **Agregar mas controladores:** un sistema no necesariamente debe tener uno solamente. Además de los controladores de fachada, se recomienda emplear los controladores de papeles o los de Casos de Uso.

- Diseñe el controlador para que **delegue fundamentalmente** a otros objetos el desempeño de las responsabilidades de la operación del sistema.

Patrones GRASP: "*Controlador*"

DOS Advertencias Importantes:

1º) Los controladores de papeles pueden conducir a la obtención de diseños deficientes:

Asignar una responsabilidad a un objeto de papeles humano es una forma de imitar lo que ese papel realiza en el mundo real, ejemplo la clase registro (para un registro contable). Ello es aceptable, pero si escoge un controlador de papeles, no caiga en la trampa de diseñar objetos de tipo humano que realicen todo el trabajo, opte más bien por delegar. Por eso, no se aconseja utilizar mucho los controladores de papeles.

2º) La capa de presentación no debiera manejar los eventos del sistema:

Los objetos de la interfaz (objetos de ventanas, applets) y la capa de presentación no deberían encargarse de manejar los eventos del sistema. Para ello deben usarse controladores separados.

Ejercicio

Considera el siguiente cod.:

```
Public Class Employee
{
    Public double calculaNomina();
    Public double calculaImpuestos();
    Public void escribeADisco();
    Public void leeADisco();
    Public String creaXML(String xml);
    Public void muestraEnInformeEmpleado();
    Public void muestraEnInformeNomina();
    Public void muestraEnInformeImpuestos();
}
```

Modele el diag de clases

NOTA:
NOMINA: sueldo, paga, haberes, retribución

Ejercicio

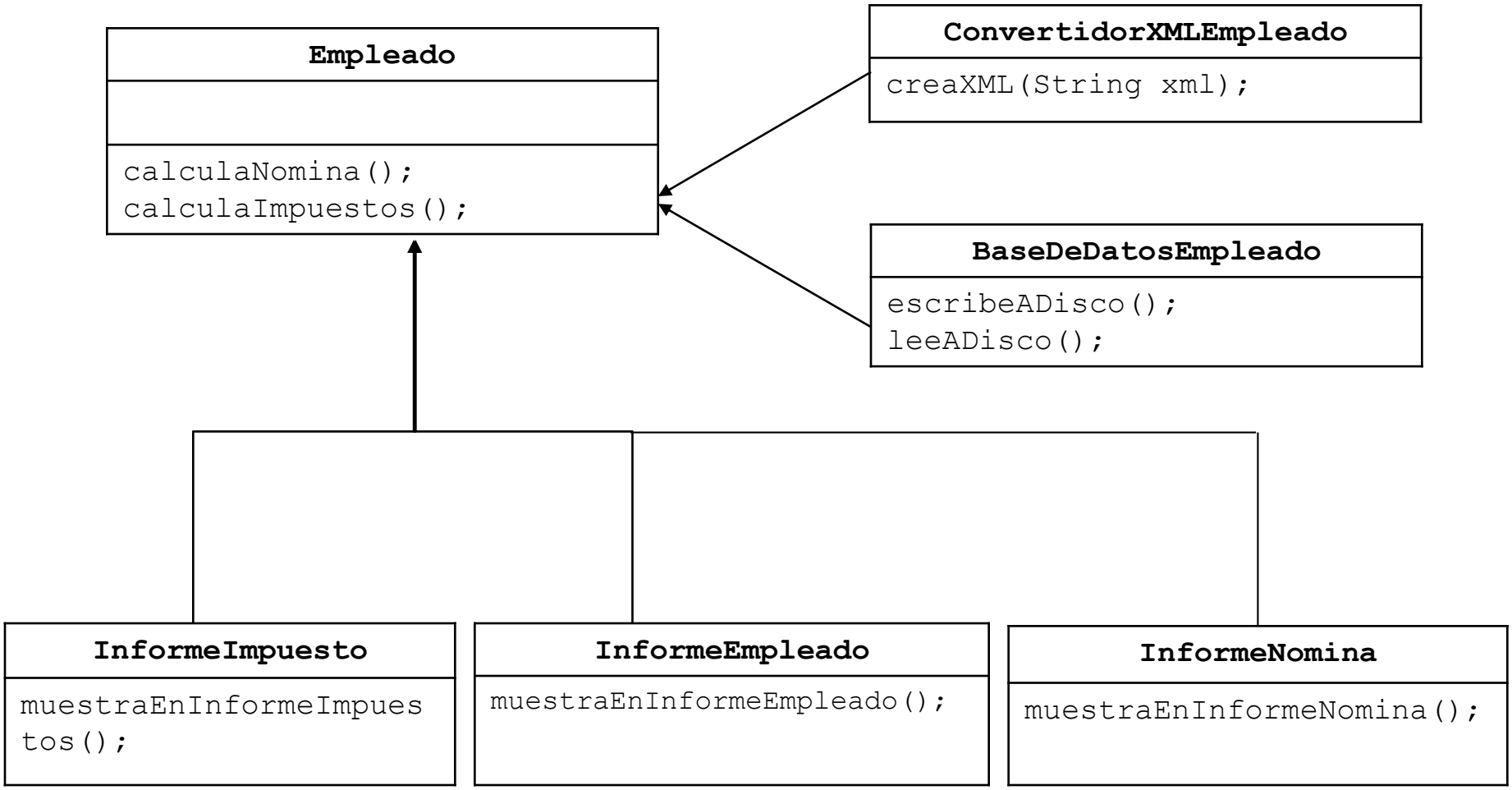
¿Qué problemas de diseño tiene el siguiente modelo?

Empleado

`calculaNomina();`
`calculaImpuestos();`
`escribeADisco();`
`leeADisco();`
`creaXML(String xml);`
`muestraEnInformeEmpleado();`
`muestraEnInformeNomina();`
`muestraEnInformeImpuestos();`

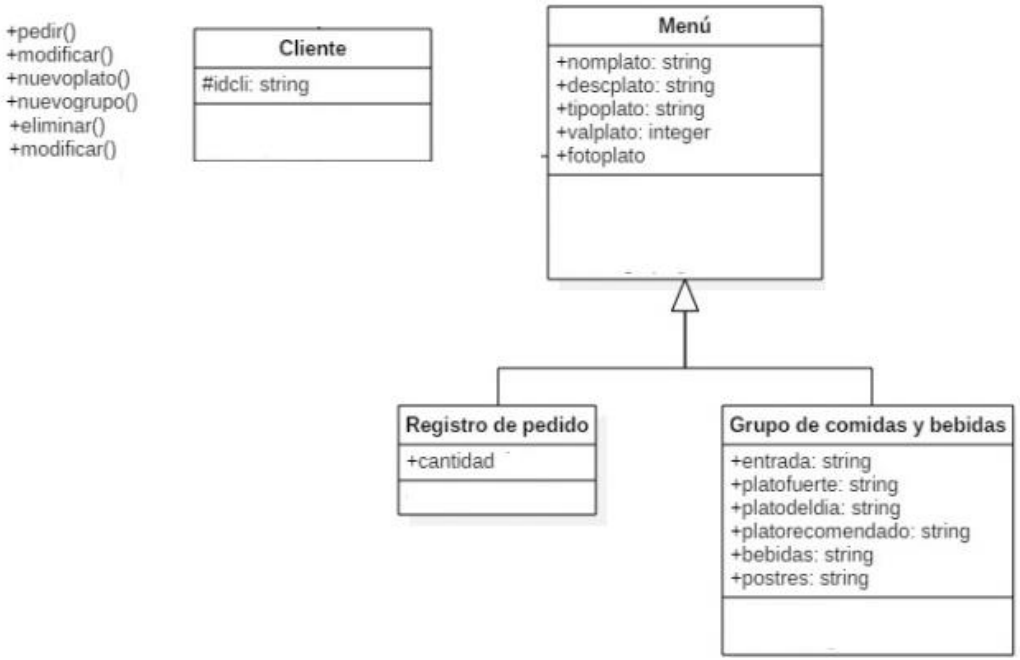
Modele una estructura, que cumpla con los patrones GRASP

Posible solución



¿Como podemos seguir mejorándolo?

- Asignar responsabilidades al siguiente modelo inicial, colocar clases y comportamiento
- Modelar el diag de secuencial :
  - Hacer pedido
  - Crear menú
  - Ejecutar pago



Campo de caso de uso	Descripción	
Nombre del caso de uso	Pedido Comida	
Version	1.0	
Actores	Cliente - Restaurante	
Objetivos Asociados	implementar una interface para que el cliente realice su pedido según el menú escogido.	
Descripción del caso de uso	este caso de uso permite que el usuario realice el pedido de su comida según el menú escogido anteriormente, posteriormente el pedido ingresa al sistema para que el chef pueda cocinar su orden y ser entregado en el tiempo estipulado para dicho pedido.	
Secuencia Normal	Paso	Accion
	1	Sistema solicita al cliente que seleccione el menu que desea
	2	El cliente selecciona su plato del menu proporcionado.
	3	Cliente confirma envio de pedido.
	4	Sistema procesa pedido enviado e informa tiempo de entrega del pedido.
	5	Sistema envia informacion de pedido a la base de datos para su posterior pago.
Excepciones	Paso	Accion
	Escoger menú	Realizar pedido

Patrones GRASP: "*Controlador*"

MVC (Model–View –Controller)

- Origen a principios de 8Os para Smalltalk
- Divide la aplicación en 3 tipos de componentes: modelos, Vistas y controladores.

- ✓ **Modelo**: principal funcionalidad e información
- ✓ **Vista**: muestra información al usuario
- ✓ **Controlador**: controla los inputs del usuario

Patrones GRASP: "*Controlador*" y MVC

El MVC se basa en un concepto introducido en el año de 1970, por Dijkstra, el concepto de ***"separación de incumbencias"*** *en OO*

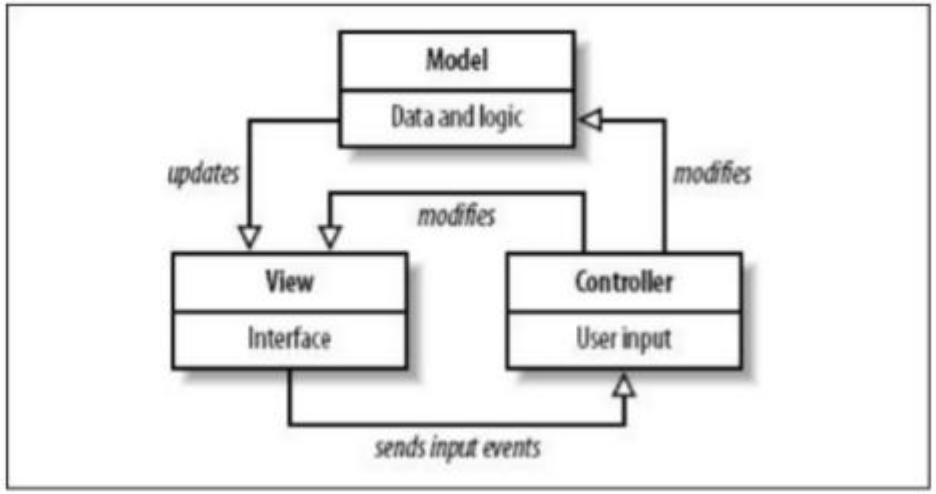
Implica que la solución de problema, involucra varias incumbencias o asuntos, los que deben ser **identificadas** y analizados en forma **independiente**.

Separando las incumbencias, se disminuye la complejidad.
Se puede cumplir con requerimientos relacionados con la calidad.

Las técnicas de modelado que se usan en la etapa de diseño de un sistema se basan en partirlo en varios subsistemas que resuelvan parte del problema o correspondan a una parte del dominio sobre el que trata.

Patrones GRASP: "*Controlador*"

MVC (Model–View –Controller)



Patrones GRASP: "*Controlador*" y MVC

El **Modelo** es el responsable de:

- Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.
- Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser: "Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor".
- Lleva un registro de las vistas y controladores del sistema.

  Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero batch que actualiza los datos, un temporizador que desencadena una inserción, etc.).

  Un ejemplo de MVC con un modelo pasivo (aquel que no notifica cambios en los datos) es la navegación web, que responde a las entradas del usuario, pero no detecta los cambios en datos del servidor.

Patrones GRASP: "*Controlador*" y MVC

Vista
La vista es la capa de la aplicación que ve el usuario en un formato adecuado para interactuar, es nuestra interfaz grafica.
Las **vistas** son responsables de:

- Recibir datos del modelo y los muestra al usuario.
- Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
- Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

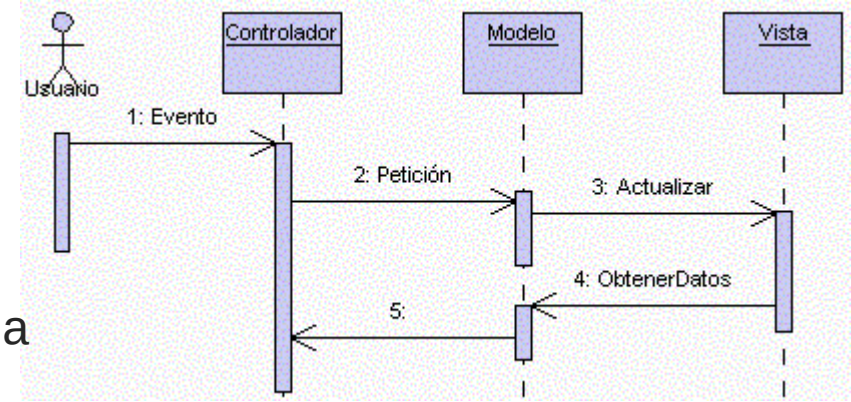
Patrones GRASP: "*Controlador*" y MVC

Diag de secuencia

Pasos:

1. El usuario introduce el **evento**.
2. El Controlador recibe el evento y lo traduce en una **petición** al **Modelo** (aunque también puede llamar directamente a la vista).
3. El **modelo** (si es necesario) llama a la vista para su **actualización**.
4. Para cumplir con la **actualización** la Vista puede **solicitar** datos al **Modelo**.
5. El Controlador recibe el control.

NOTA: algunos MVC, el modelo actualiza directamente a la vista



Ejercicio

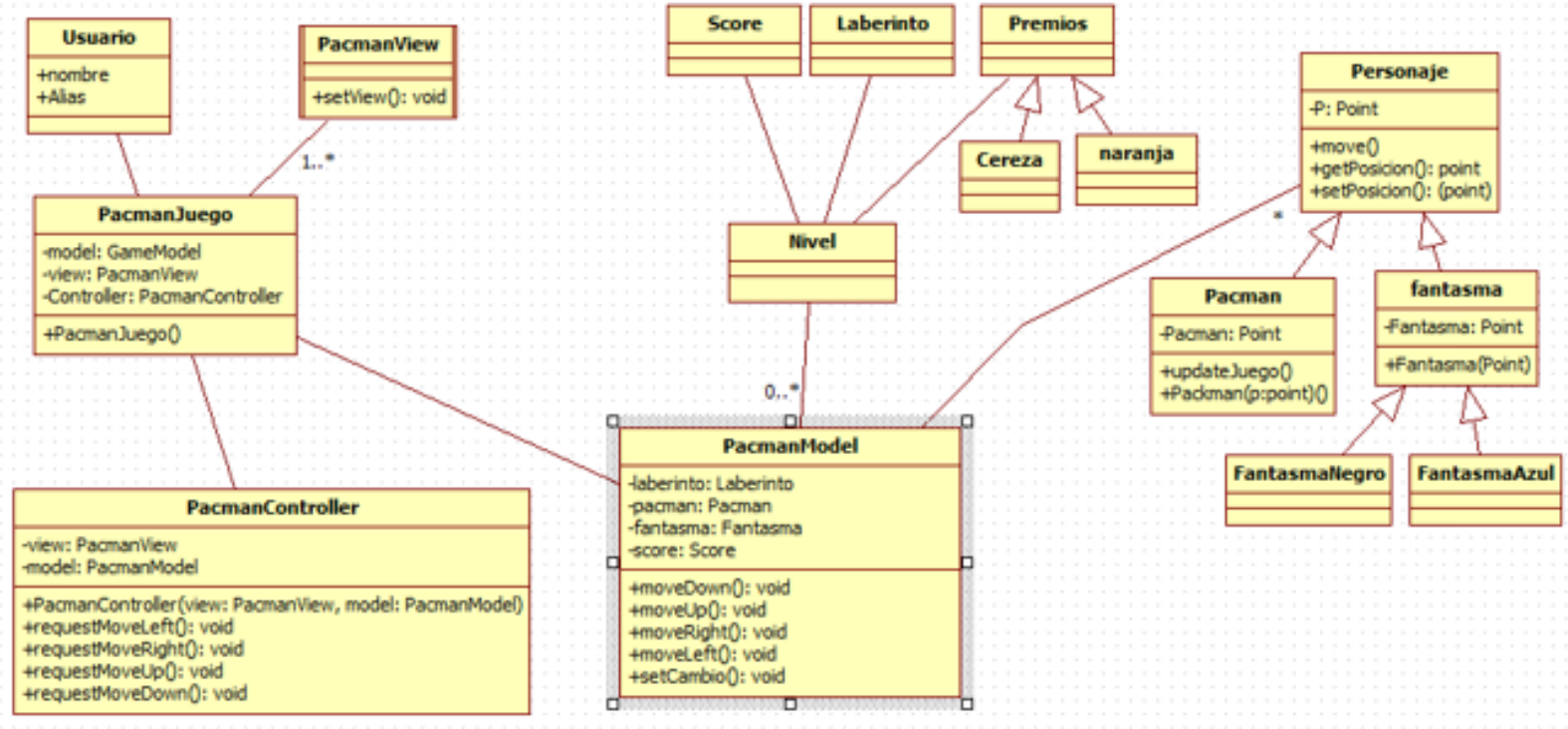
Se solicita implementar un juego (**pacman**) para formato WEB y Apps (tablets y celulares),
Para llevar a cabo el modelo, se realizó una breve investigación del juego, donde se definió:
PacMan , es el personaje principal es un círculo amarillo al que le falta un sector por lo que parece tener boca.
Aparece en laberintos donde debe comer puntos pequeños (llamados Pac-dots en inglés), puntos mayores y otros premios con forma de frutas y otros objetos.
El objetivo del personaje es comer todos los puntos de la pantalla, momento en el que se pasa al siguiente nivel o pantalla. Sin embargo, cuatro fantasmas o monstruos, Shadow (Blinky), Speedy (Pinky), Bashful (Inky) y Pokey (Clyde), recorren el laberinto para intentar capturar a Pac-Man. Estos fantasmas son, respectivamente, de colores rojo, rosa, cyan y naranja.
Cada fantasmas tiene un comportamiento diferente :
Blinky es muy rápido, y tiene la habilidad de encontrar a Pac-Man. Después de que Pac-Man come cierta cantidad de puntos su velocidad incrementa considerablemente

Ejercicio

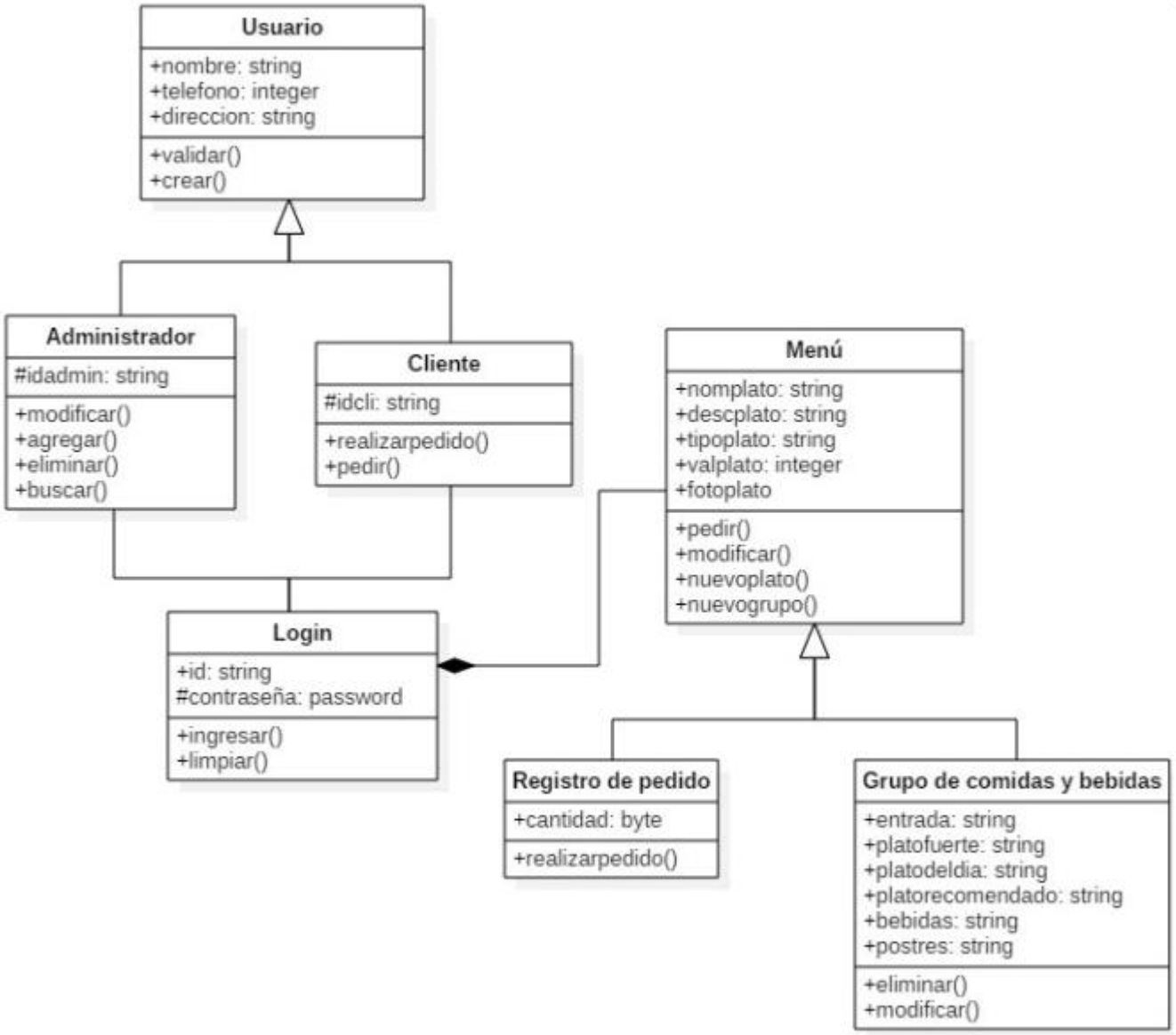
Pinky. Rodea los obstáculos al contrario de las manecillas del reloj
Inky es muy lento y muchas veces evitará el encuentro con Pac-Man, actúa de manera errática así que es difícil predecir cómo va a reaccionar.
Clyde. Él no persigue a Pac-Man, sino que deambula sin una ruta específica.
Hay cuatro puntos más grandes de lo normal situados cerca de las esquinas del laberinto (Power Pellets), y que proporcionan a Pac-Man la habilidad temporal de comerse a los monstruos (todos ellos se vuelven azules mientras Pac-Man tiene esa habilidad). Después de haber sido tragados, los fantasmas se regeneran en una caja situada en el centro del laberinto. El tiempo en que los monstruos permanecen vulnerables varía según la pantalla, pero tiende a decrecer a medida que progresa el juego.
Además de comer los puntos, Pac-Man puede obtener puntuación adicional si se come alguno de los objetos que aparecen dos veces por pantalla justo debajo de la caja en el centro del laberinto de donde salen los monstruos. El objeto cambia cada pantalla o dos, y su valor en puntos aumenta, de forma que dos cerezas (el premio de la primera pantalla) valen 100 puntos, mientras que el último objeto, la llave, vale 5.000.

Ejercicio

El modelo de diseño inicial es el siguiente:



Se pide modelar la solución contemplando controladores.
Modelar la solución con MVC, sumando la funcionalidad de gestión de usuarios y gestión de puntajes


Usuario
+nombre: string
+telefono: integer
+direccion: string
+validar()
+crear()
Administrador
#idadmin: string
+modificar()
+agregar()
+eliminar()
+buscar()
Cliente
#idcli: string
+realizarpedido()
+pedir()
Menú
+nomplato: string
+descplato: string
+tipoplato: string
+valplato: integer
+fotoplato
+pedir()
+modificar()
+nuevoplato()
+nuevogrupo()
Login
+id: string
#contraseña: password
+ingresar()
+limpiar()
Registro de pedido
+cantidad: byte
+realizarpedido()
Grupo de comidas y bebidas
+entrada: string
+platofuerte: string
+platodeldia: string
+platorecomendado: string
+bebidas: string
+postres: string
+eliminar()
+modificar()