Diseño de Sistemas Software

Introducción a los patrones de diseño

Carlos Pérez
Cristina Cachero

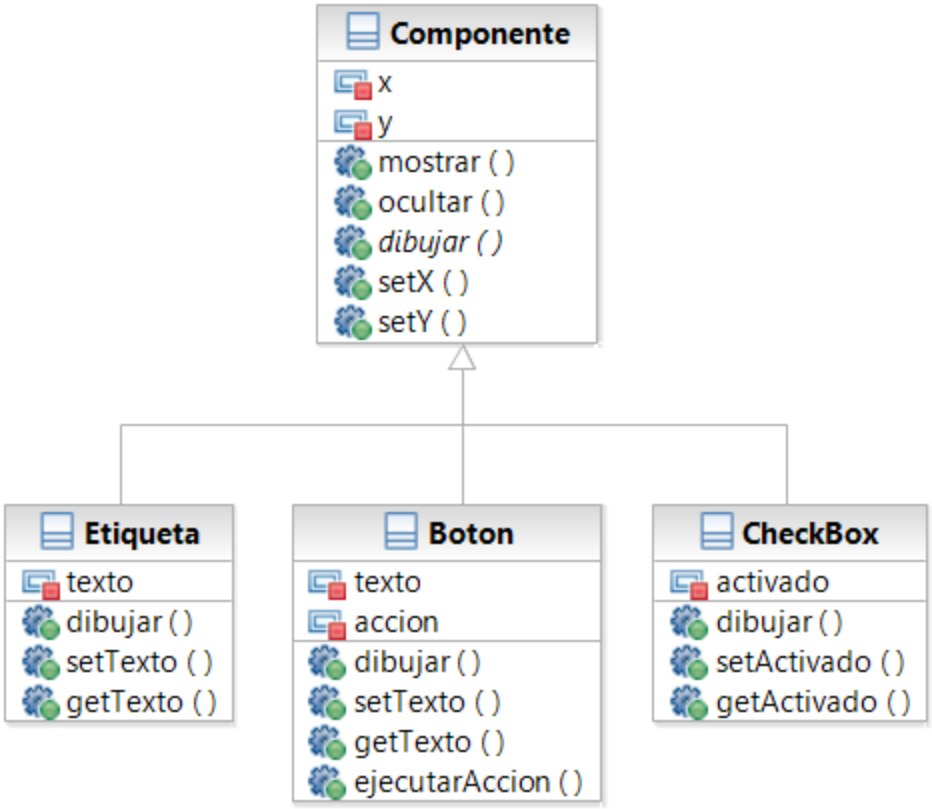
Patrones: situación de partida

- Widgets 1.0
  - Supongamos que queremos implementar un paquete de componentes para aplicaciones de entorno gráfico. Necesitamos los siguientes objetos para componer las pantallas:
    - Etiqueta: para mostrar pequeños textos explicativos
    - Botón: para que el usuario pueda iniciar acciones
    - Checkbox: para que el usuario pueda activar/desactivar opciones
  - Todos los componentes tienen unas propiedades comunes, como la posición en pantalla, y métodos para mostrarlos y ocultarlos.

DIBUJA EL **DIAGRAMA UML** PARA ESTE PROBLEMA

Patrones: situación de partida

- Widgets 1.0



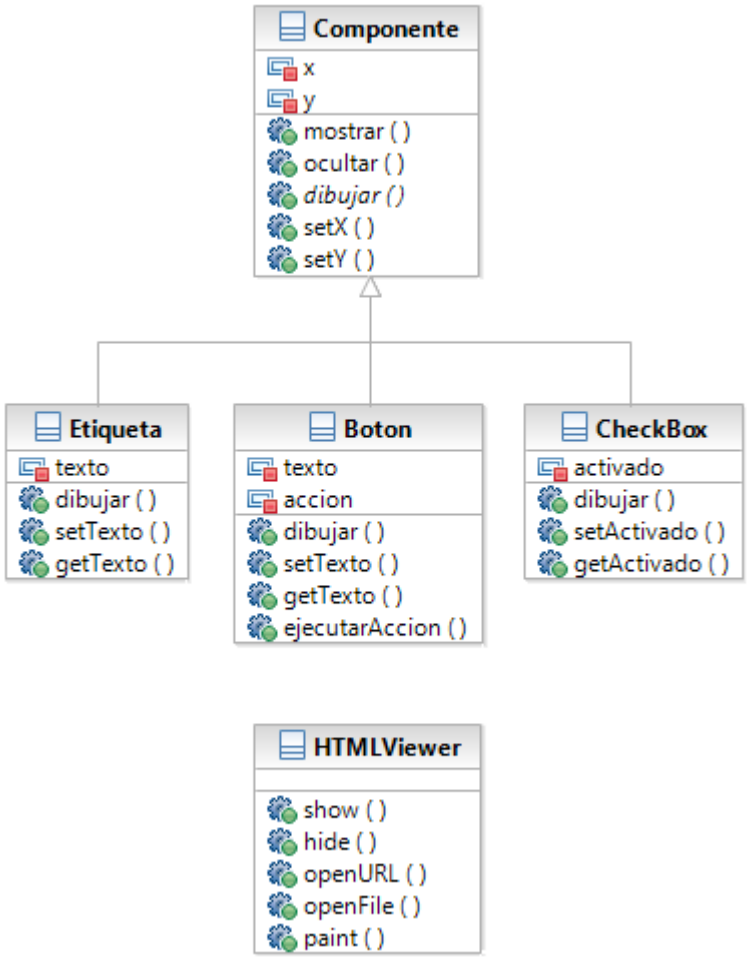
Patrones: situación de partida

- Widgets 1.1

  Tenemos que añadir dos nuevos componentes para ampliar las funcionalidades de nuestro software:

  - Cuadro de texto: permite al usuario escribir texto para luego procesarlo
  - Visor HTML: muestra páginas web

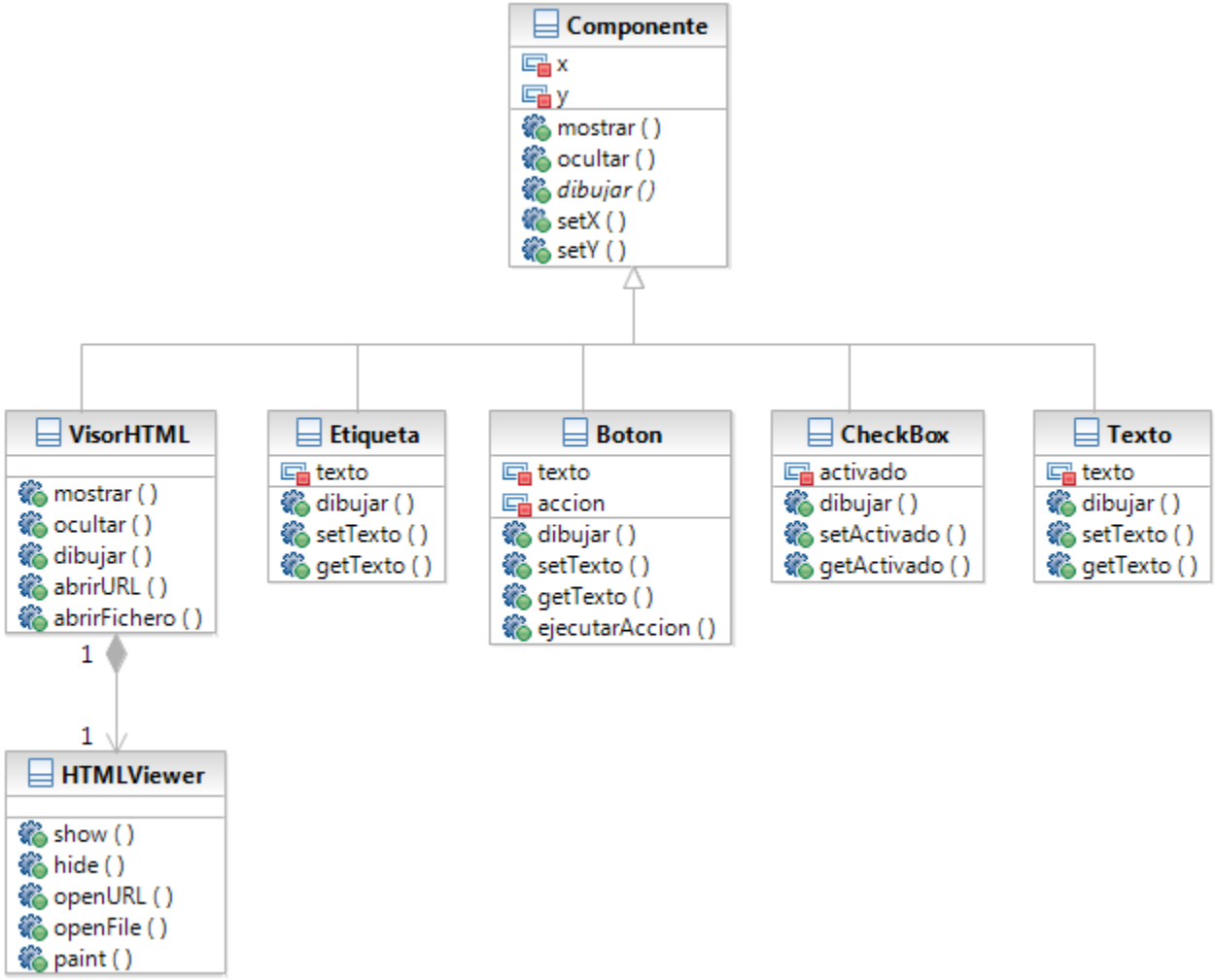
  Ya contamos con una implementación del visor HTML en una clase llamada `HTMLViewer`, desarrollada por una empresa externa, pero desgraciadamente la interfaz que ofrece no es compatible con nuestro sistema.



¿MODIFICACIONES?

Patrones: situación de partida

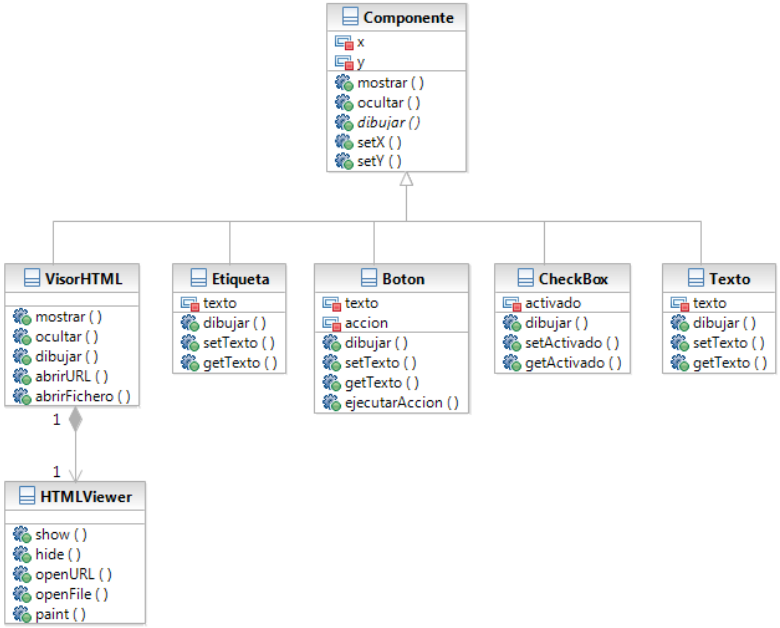
- Widgets 1.1



Patrones: situación de partida

- Widgets 1.2

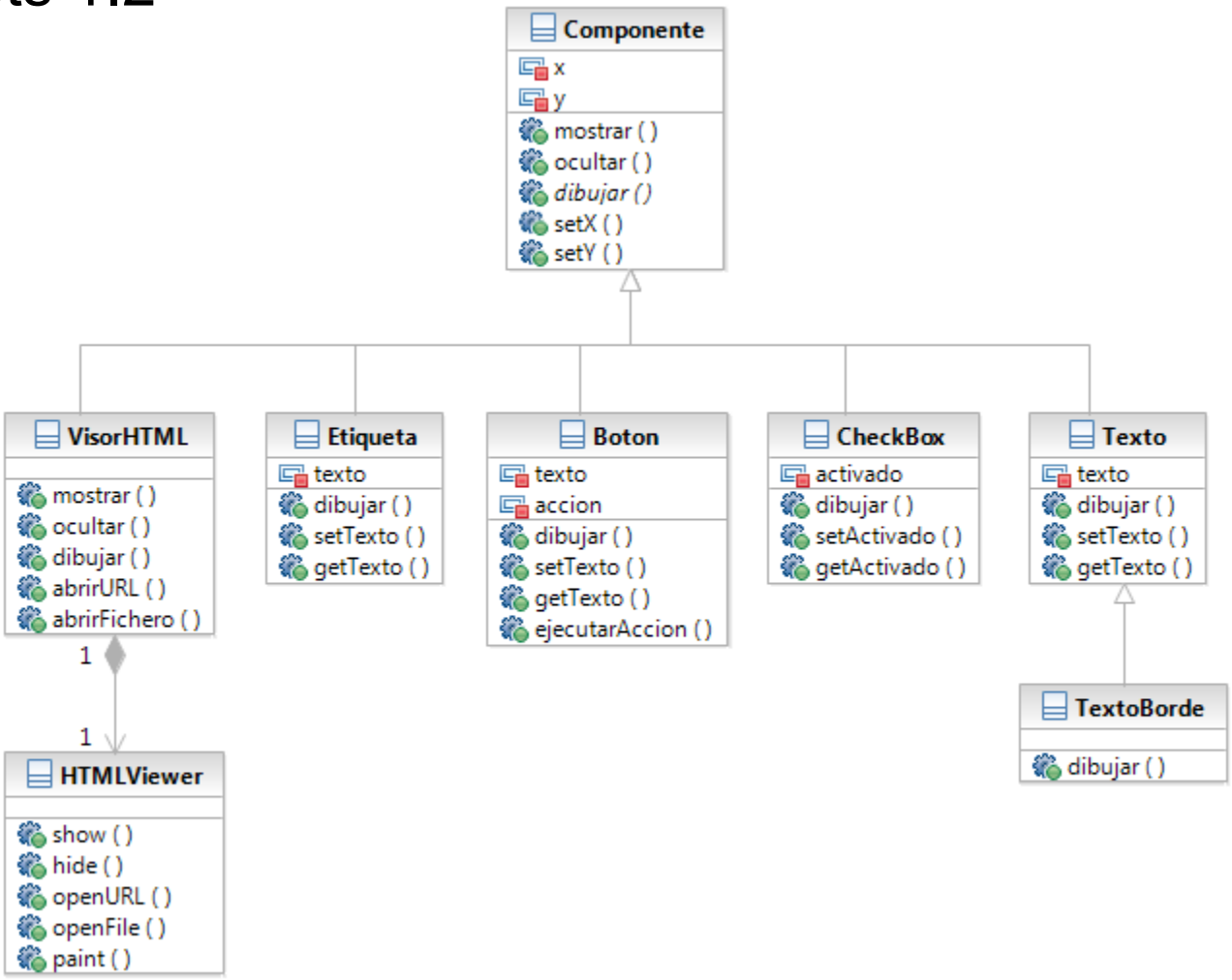
  Para hacer nuestro sistema un poco más atractivo, nos han pedido cuadros de texto con borde, para que el usuario distinga fácilmente los campos que son opcionales de los obligatorios en los formularios para introducir datos.



¿MODIFICACIONES?

Patrones: situación de partida

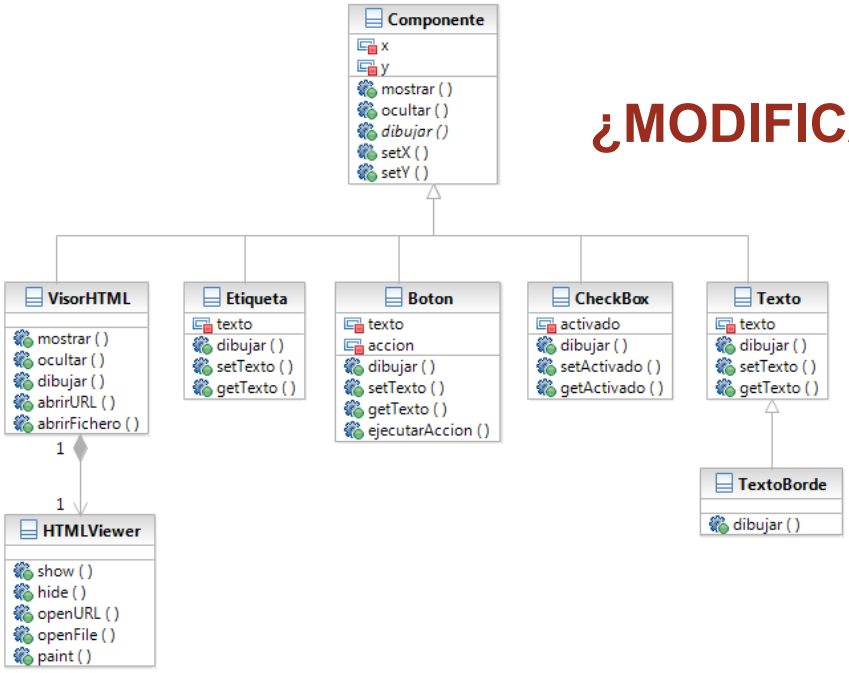
- Widgets 1.2



Patrones: situación de partida

- Widgets 1.3

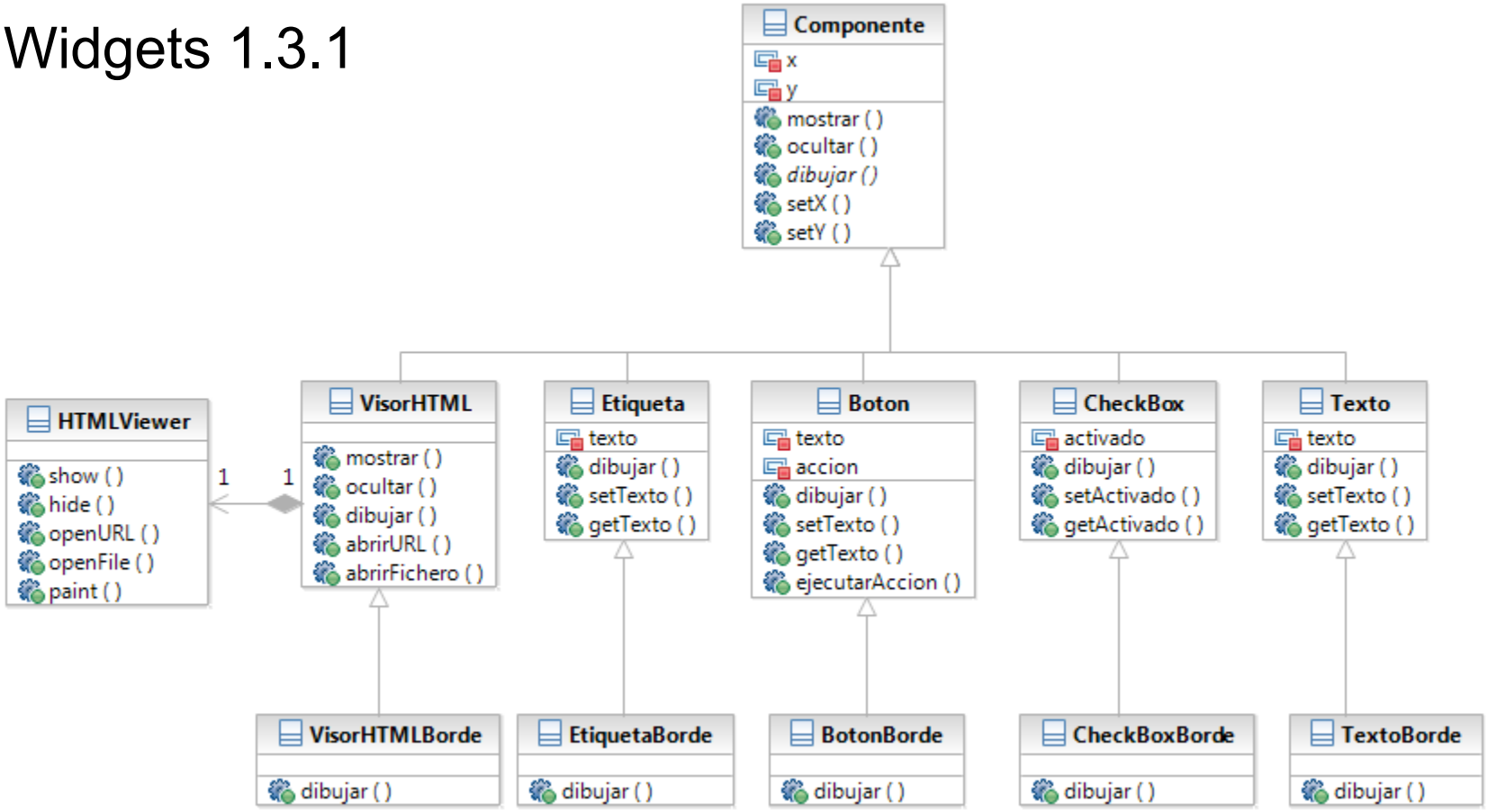
  Nuestra última actualización ha gustado mucho, así que ahora nos han pedido que todos los componentes puedan tener borde, igual que los cuadros de texto.



¿MODIFICACIONES?

Patrones: situación de partida

- Widgets 1.3.1

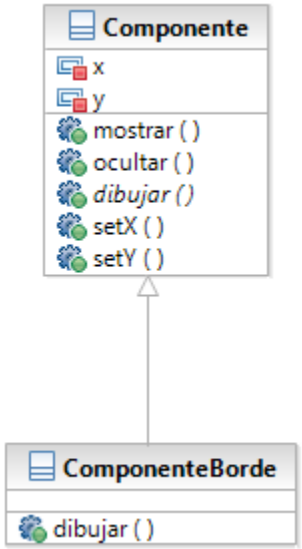


¿PROBLEMAS?

DUPLICACIÓN DE CÓDIGO: todos dibujan el mismo tipo de borde

Patrones: situación de partida

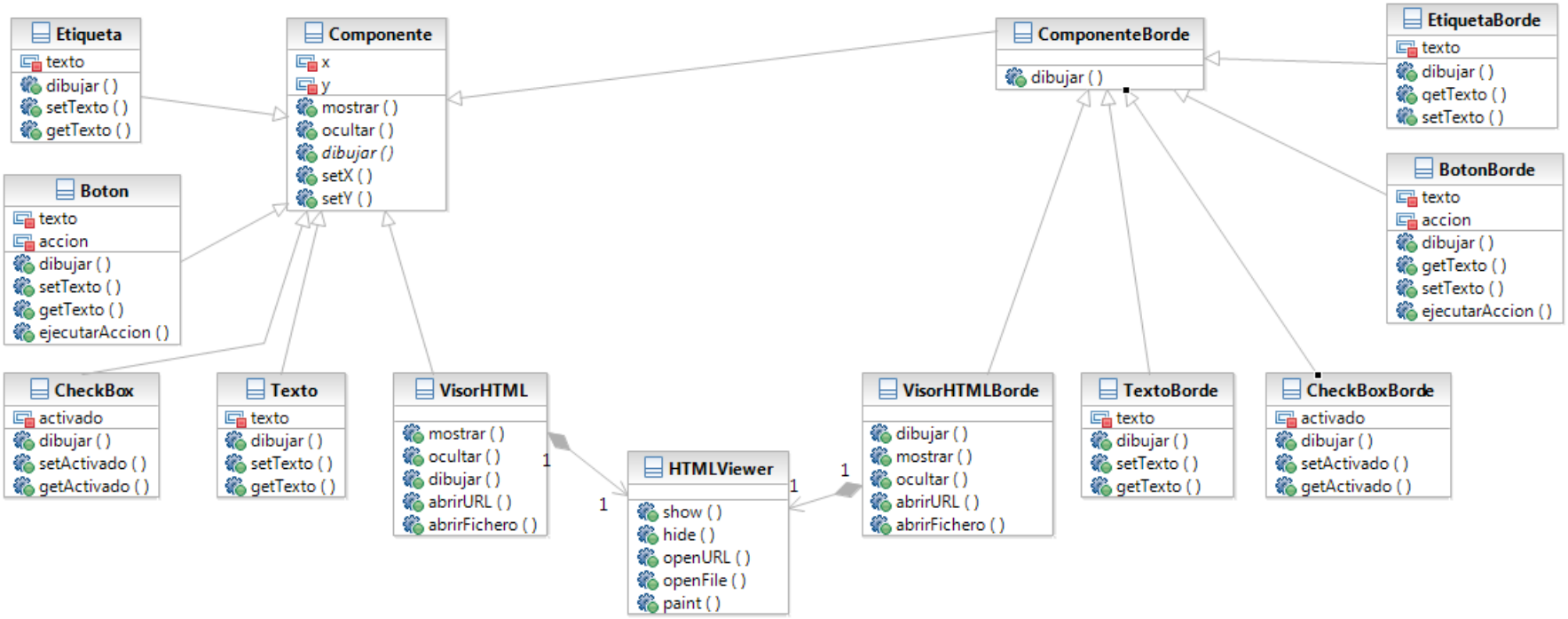
- Widgets 1.3.2



¿Seguro que hemos eliminado la duplicación de código?

Patrones: situación de partida

- Widgets 1.3.2



La mayoría de lenguajes de programación OO no permiten herencia múltiple

¿Y si hacemos que la clase Componente dibuje el borde?

Patrones: situación de partida

- Widgets 1.3.3

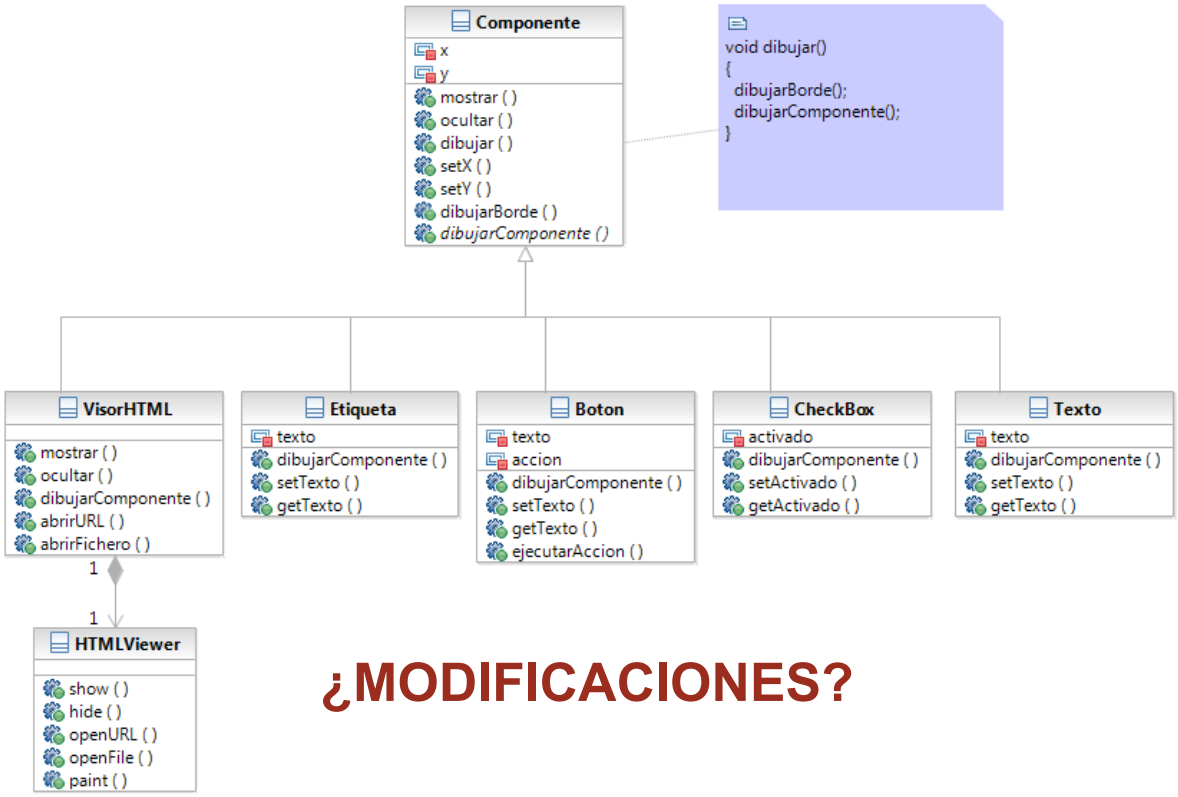
¿PROBLEMAS?

Baja la cohesión de la clase Componente.
Dificulta el mantenimiento, p.ej. si queremos añadir nuevos tipos de borde.

Patrones: situación de partida

- Widgets 1.4

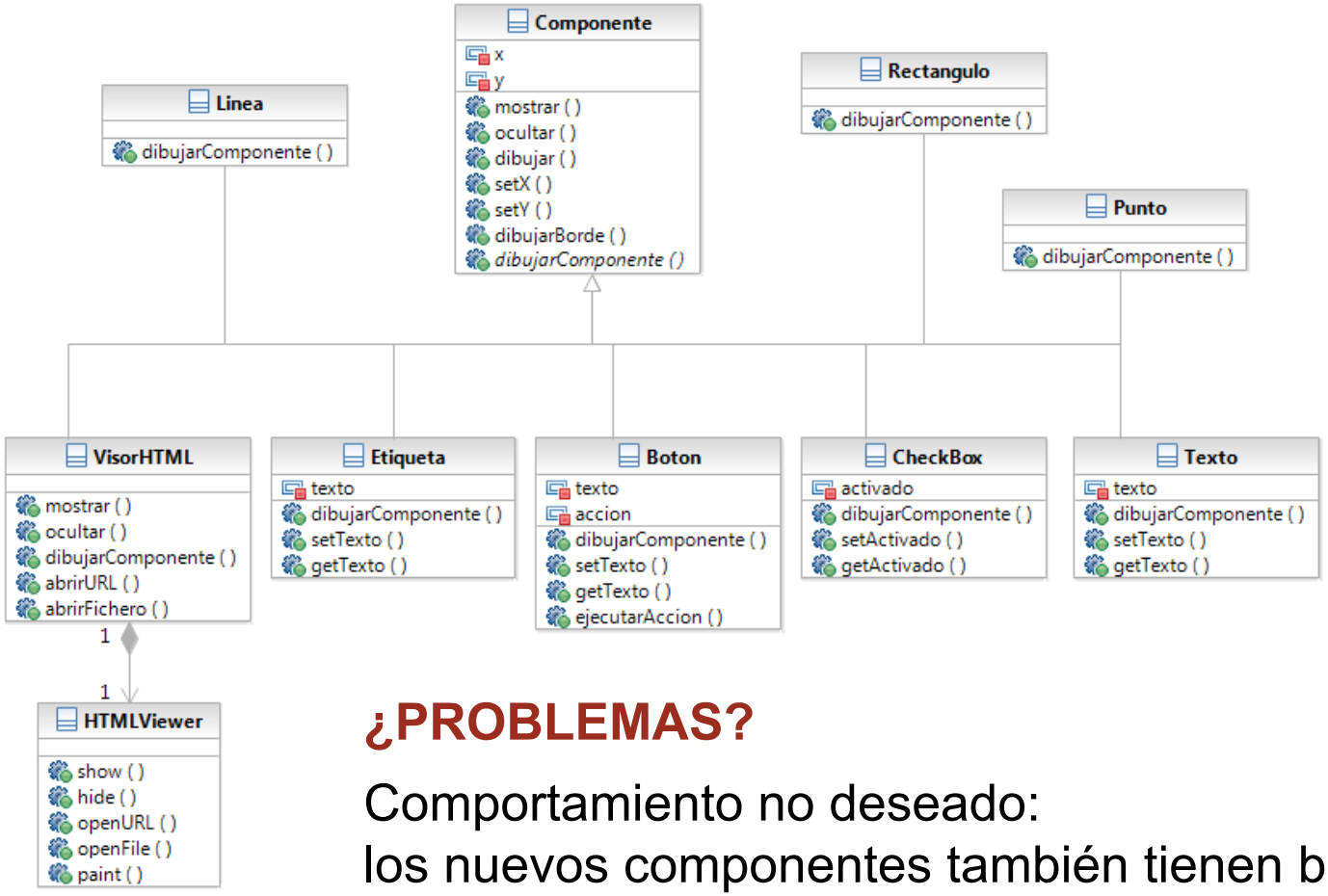
  Para enriquecer aún más el interfaz, nos piden añadir líneas, puntos y algunas formas geométricas para aumentar las posibilidades de diseño gráfico.



¿MODIFICACIONES?

Patrones: situación de partida

- Widgets 1.4



¿PROBLEMAS?

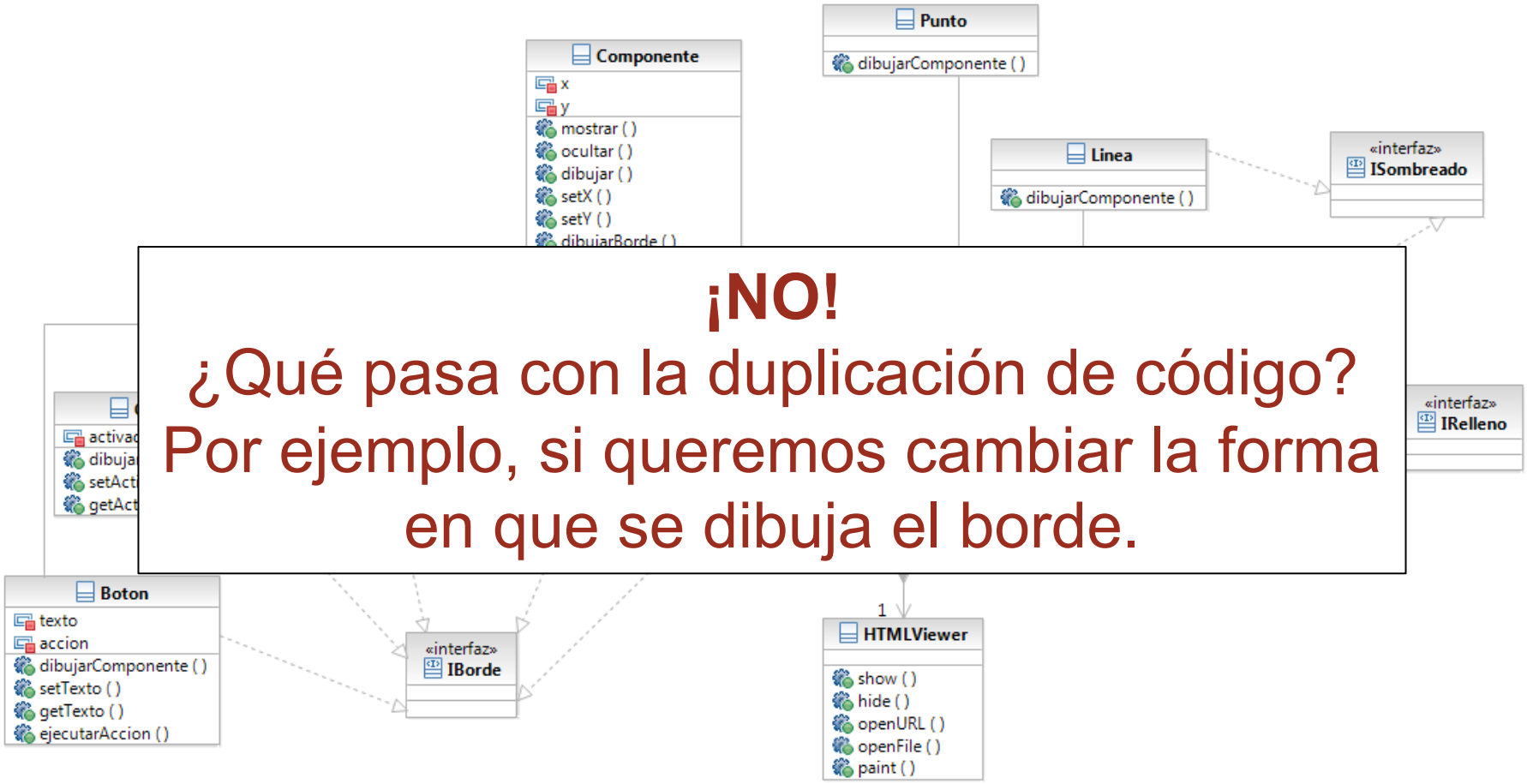
Comportamiento no deseado:
los nuevos componentes también tienen bordes.

Patrones: situación de partida

- En general, los problemas de las jerarquías de herencia son...
  - El código se duplica en todas las subclases
  - Puede causar una baja cohesión: ¿qué ocurre si empezamos a definir más y más tipos de nuevos bordes? ¿Y si queremos introducir la posibilidad de que las nuevas formas se dibujen con un patrón determinado (rallado, puntitos, etc.)? Las clases se están preocupando de cosas que no tienen que ver con el propio componente.
  - Reduce la posibilidad de reuso, si queremos extender la funcionalidad de una clase a las demás.
  - No escala bien con variaciones: ¿y si ahora tenemos además dos tipos de sombreado posible?
  - Los cambios en el comportamiento en tiempo de ejecución son difíciles
  - Los cambios pueden tener efectos secundarios, afectando el comportamiento de otras clases.

Patrones: situación de partida

- ¿Y si usamos interfaces?



¡NO!

¿Qué pasa con la duplicación de código?
Por ejemplo, si queremos cambiar la forma en que se dibuja el borde.

Patrones: situación de partida

- Diseñar un sistema desde cero es un proceso iterativo, en el que vamos aprendiendo de los errores previos hasta lograr un buen diseño
- ¿Cuáles son las características de un buen diseño?
  - **Abstracción**: debe mostrar únicamente los detalles relevantes, ocultando los detalles de implementación
  - **Flexibilidad**: debe estar preparado para ser usado de distintas maneras, incluso de formas no previstas
  - **Reutilización**: se debe poder reutilizar en distintos contextos
  - **Calidad**: debe evitar los errores en la implementación
  - **Modularidad**: debe dividir el problema en partes más pequeñas, favoreciendo el bajo acoplamiento y la alta cohesión

Patrones: situación de partida

- ¿Cómo conseguir la calidad en los diseños?
  - Estudiar software existente (p.ej. proyectos de código abierto)
  - Experiencia, a través de la práctica y la repetición
  - Recopilar la experiencia personal de otros diseñadores en reuniones, internet, etc.
- Lleva mucho tiempo

Patrones: introducción

- ¿Cómo podemos acelerar el proceso de aprendizaje?

→ USO DE PATRONES

- Uno de los mecanismos existentes para capturar conocimiento sobre problemas y soluciones exitosas en el desarrollo del software
- Permiten reutilizar la experiencia de otros diseñadores, y reducir el esfuerzo asociado a la producción de sistemas más efectivos y flexibles (más adaptables al cambio)

Patrones: orígenes e historia

- Origen: escritos del arquitecto Christopher Alexander

  Acuñó el uso del término "pattern" (1977-1979)

- Ejemplo: lugar para ubicar una ventana
  - Fuerzas
    - El usuario quiere sentarse y sentirse cómodo
    - El usuario es atraído por la luz
  - Solución
    - En cada habitación, coloca al menos una ventana en un lugar apropiado

Patrones: orígenes e historia

- Uso y evolución en el software
  - Kent Beck y Ward Cunningham, Textronix, OOPSLA'87

    Usaron las ideas de patrones de Alexander para el diseño de GUI's en Smalltalk)
  - Erich Gamma, Ph.D. thesis, 1988-1991
  - James Coplien, Advanced C++ Idioms book, 1989-1991
  - Gamma, Helm, Johnson, Vlissides (“Gang of Four” - GoF)

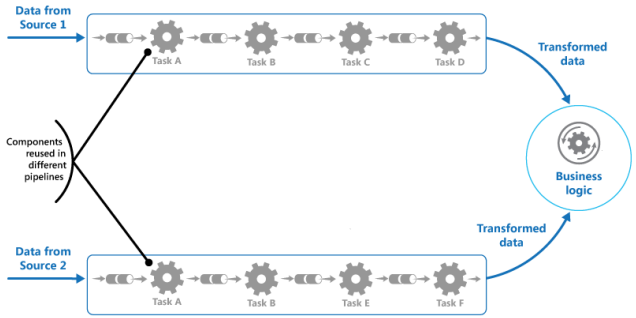
    Design Patterns: Elements of Reusable Object-Oriented Software, 1991-1994
  - Conferencias PLoP y libros, desde 1994 hasta la actualidad
  - Buschmann, Meunier, Rohnert, Sommerland, Stal, Pattern-Oriented Software Architecture: A System of Patterns (“libro POSA”)

Patrones: tipos de patrones

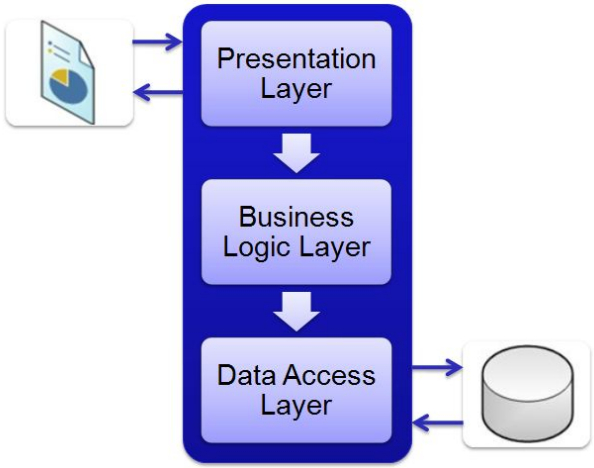
- Patrones de análisis (Fowler)
  - Analysis Patterns: Reusable Object Models [Fowler97].
    En su web hay actualizaciones de varios capítulos de ese libro
- Patrones de diseño
  - Patrones GRASP (Larman)
  - Microarquitectura (diseño detallado) (Gamma)
  - Arquitecturales y microarquitecturales (Buchmann, Fowler)
- Patrones de implementación
  - Idioms (específicos de paradigma y lenguaje programación)
- Patrones de Integración de Aplicaciones:
  - Enterprise Integration Patterns [Hophe03].
- Patrones de proceso de desarrollo software (diseño de procesos)
- Patrones de organización (estructura de organizaciones/proyectos)
- Patrones de UI: Patrones referentes a interfaces de usuarios.
  - Existen distintas categorías bien diferenciadas: algunas se encargan de detalles relacionados con la cognición, memoria a corto plazo y mejoras en la experiencia del usuario, mientras que otros describen técnicas de ingeniería para crear interfaces de usuario.
- Patrones de Pruebas: Patrones para diseñar y realizar pruebas.
- Antipatrones (de cualquiera de estas categorías)
- ...

Patrones: tipos de patrones

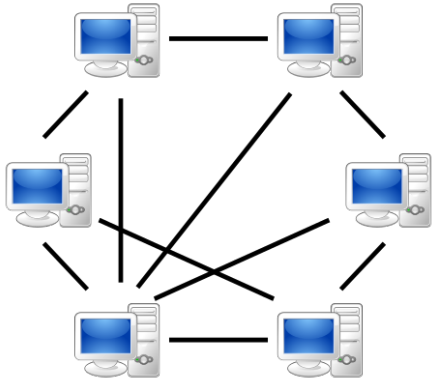
- **Arquitecturales**: estructura global del sistema
- Algunos ejemplos:



Pipes and filters



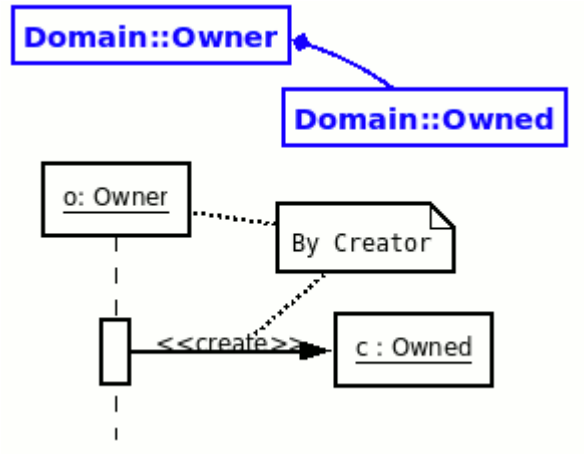
Arquitectura en capas



Peer-to-peer

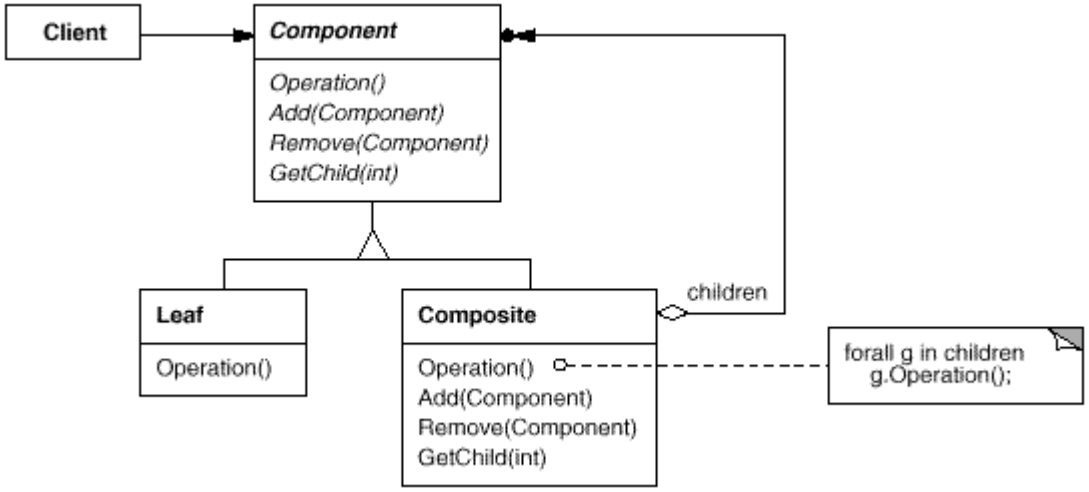
Patrones: tipos de patrones

- Patrones GRASP (Larman): asignación de responsabilidades a las clases del sistema
- Especialmente útiles para diseñar y refinar el modelo de dominio



Patrones: tipos de patrones

- Patrones GOF: diseño detallado para el día a día



Patrones: principios subyacentes

- En Larman se proponen varios principios fundamentales que sustentan los patrones :
  - Prever la variación y evolución del software
  - Favorecer el bajo acoplamiento, la alta cohesión y el reuso del software

- En el libro *Design Patterns* (GOF) también se propone unos principios fundamentales de diseño subyacentes a los patrones que permiten crear aplicaciones más flexibles y robustas:
  - Programar para interfaces (o clases abstractas) y no para una implementación.
    - Ved la diferencia...

Perro *p=new Perro(); **p.Ladra();**	**Animal *p=new Perro();** **p.hazSonido();**

    - Se os ocurre alguna manera de hacer este código todavía más flexible?
  - Favorecer la composición de objetos frente a la herencia de clases.

Patrones: principios subyacentes

- Según Buchmann (POSA) los principios subyacentes son:

Funcionales:

- •Abstracción
- •**Encapsulación**
- •Ocultación Información
- •Modularización
- •Separación de preocupaciones
- •Acoplamiento y cohesión
- •Suficiencia, completitud y primitividad
- •Separación entre política e implementación
- •Separación entre interfaz e implementación
- •Único punto de referencia
- •Divide y vencerás

No Funcionales:

- •Facilidad para realizar cambios (mantenibilidad, extensibilidad, restructuración y portabilidad)
- •Interoperabilidad
- •Eficiencia
- •Confiabilidad
- •Testabilidad
- •Reusabilidad

Patrones: principios subyacentes

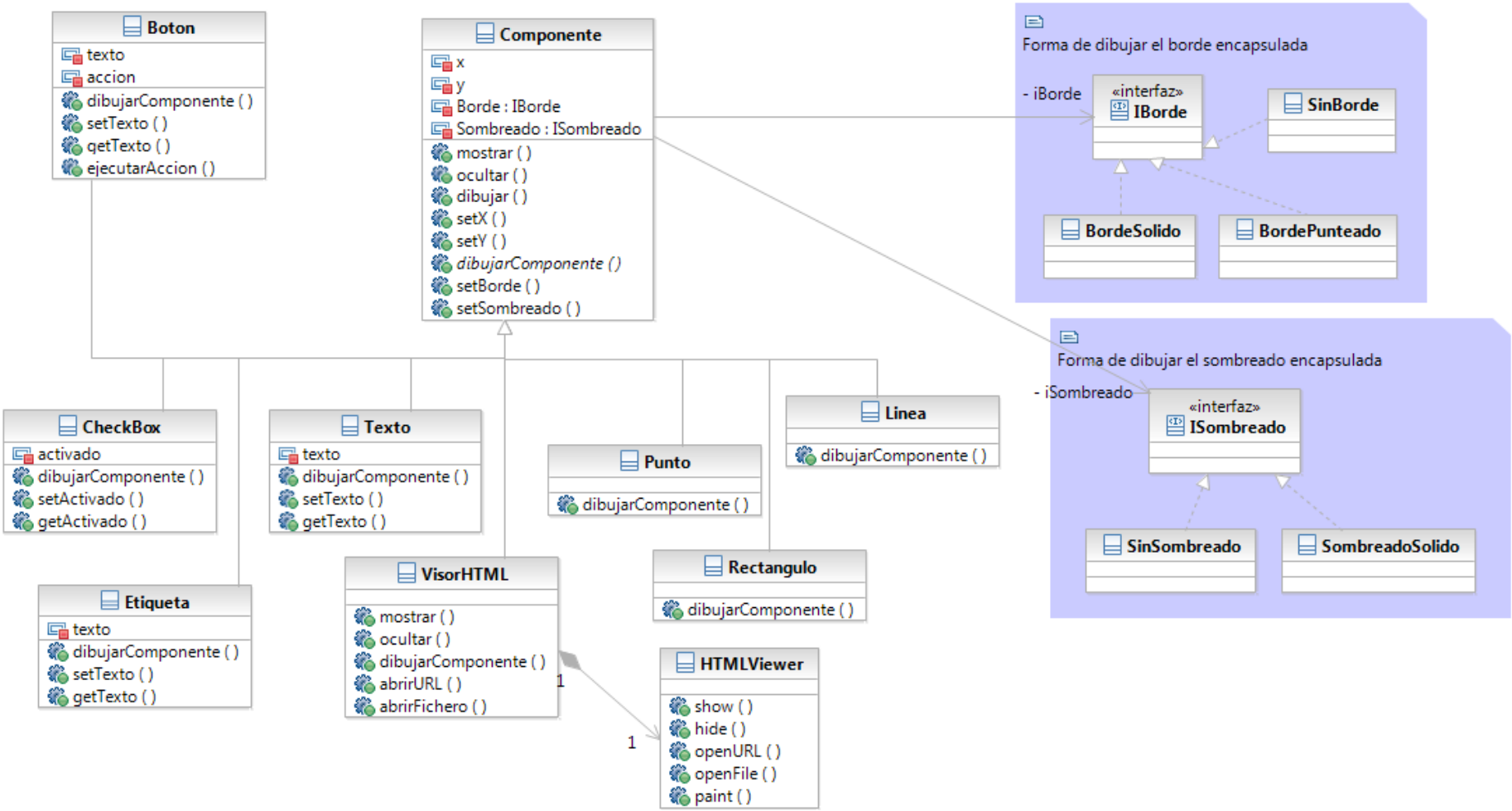
- En todos estos principios subyace la necesidad de ENCAPSULAR LAS VARIACIONES PROBABLES DEL SOFTWARE tras una fachada estable
- Además, cuando trabajamos con patrones nos damos cuenta de que los objetos no son un conjunto de datos y métodos, sino cosas con responsabilidades
- Por tanto:
  - La encapsulación no consiste en 'ocultar datos', sino en 'ocultar cualquier cosa': implementaciones, clases derivadas, detalles de diseño, reglas de instanciación, ... Es de hecho especialmente importante que la encapsulación pueda ser utilizada para contener variaciones en el comportamiento:
    ENCUENTRA LO QUE VARÍA Y ENCAPSÚLALO
  - Daos cuenta de que un comportamiento es igual de encapsulable que un dato. De hecho, un comportamiento puede tener 'propiedades' asociadas. p.ej. Operación calcularInterés() puede tener asociado una propiedad 'tipoInterésAplicable', que pueda variar de objeto a objeto.

Patrones: principios subyacentes

- ¿Cómo encapsulamos el comportamiento de nuestros componentes?
  - Parece que la clase Componente, aparte de los distintos adornos, funciona bien… Vamos a encapsular por tanto la parte que está variando, ¿cómo?
  - Queremos mantener un diseño flexible
    - Queremos poder asignar distintos tipos de adornos a los diferentes componentes
    - Y, ya que estamos, ¿por qué no permitir poder cambiar la forma de dibujar un componente en tiempo de ejecución?

Patrones: principios subyacentes

- Widgets 2.0 (patrón *Strategy*)



Patrones: análisis de variabilidad

- Daos cuenta de cómo, una vez que eliminamos las partes que varían de la jerarquía de herencia, podemos beneficiarnos de sus ventajas (reuso de propiedades y código) sin sus inconvenientes.

- COPLIEN: Para diseñar OO hay que detectar dónde varían las cosas y dónde no (análisis de lo común) y CÓMO VARÍAN (análisis de variabilidad)
  - Conceptos comunes: clases abstractas, perspectiva de análisis y diseño
  - Variaciones: clases concretas, perspectiva de implementación

Patrones: documentación

- Los patrones se documentan mediante el uso de plantillas (*templates*)
- Cualquier plantilla debería incluir al menos las siguientes secciones (**forma canónica** o Alejandrina)
  - Nombre: significativo y corto
  - Problema: intención del patrón
  - Contexto: precondiciones de aplicabilidad
  - Fuerzas: aspectos que deben ser considerados en la solución
  - Solución: descripción de relaciones estáticas y dinámicas entre los componentes del patrón
- Otras secciones posibles:
  - Ejemplos: ejemplos de aplicaciones del patrón
  - Contexto resultante:el estado del sistema después de aplicar el patrón
  - Fundamentos: explicación de los pasos o reglas en el patrón
  - Patrones relacionados: relaciones estáticas y dinámicas
  - Usos reales: usos del patrón y su aplicación en sistemas existentes

Patrones: catálogo/sistema de patrones

[Buschmann, POSA]

- Catálogo de Patrones: un catálogo es un grupo de patrones relacionados, normalmente divididos en subcategorías, que pueden ser utilizados en conjunto o por separado

Patrones: patrones vs. frameworks

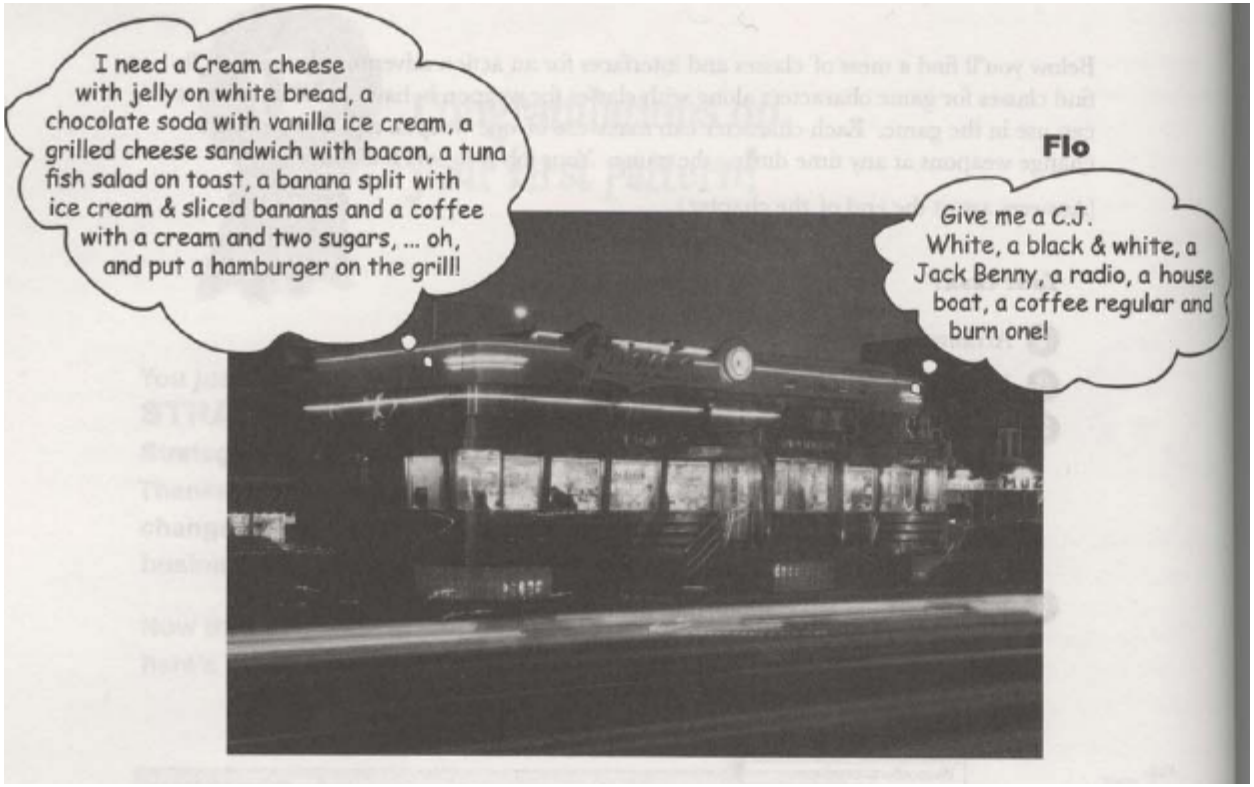
- Los patrones son más abstractos y generales que los frameworks.
  - Un patrón es una descripción de cómo se puede resolver un problema, pero no es la solución en sí
  - Un patrón no puede ser directamente implementado en un entorno software particular.
    - Una implementación exitosa es solo un ejemplo de un patrón de diseño
  - Los patrones son más sencillos que los frameworks.
    - Un patrón no puede incorporar un framework, aunque un framework sí puede hacer uso de varios patrones.

Patrones: cómo y cuándo utilizar patrones

- Un patrón no es una solución prescriptiva, sino una guía sobre cómo encontrar una solución adecuada
  - Es muy probable que un patrón se utilice de manera distinta en función de las circunstancias. p.ej. *Singleton* sobre clases hijas en lugar de sobre clases padre en una jerarquía de herencia.
- Los diseñadores pueden capturar su experiencia en la forma de nuevos patrones (*pattern mining*)

Patrones: beneficios del uso de patrones

- Reutilización de soluciones genéricas de probada eficacia
  - Evita errores a los diseñadores noveles
- Vocabulario paar discutir el dominio del problema a un nivel de abstracción más elevado



Patrones: peligros del uso de patrones

- Puede limitar la creatividad
- Pueden llevar al sobre-diseño
  - Un patrón debe siempre ser utilizado en el contexto adecuado, y sólo tras haber evaluado cuidadosamente sus ventajas e inconvenientes
- Sólo son aplicables dentro de una cultura de reuso
- Sólo afrontan algunos de los problemas que ocurren durante el desarrollo de sistemas
  - ¡No son la solución a todos los problemas de desarrollo de sistemas!

Patrones: guías de uso

- Si detectamos un alto acoplamiento, una clase compleja, etc.
  - ¿Hay algún patrón que resuelva un problema similar?
  - ¿Ese patrón proporciona una solución alternativa que puede ser más aceptable?
  - ¿Hay una solución más sencilla? Si la hay, mejor olvidarse del patrón
  - ¿Es el contexto del patrón consistente con respecto al del problema?
  - ¿Son las consecuencias de utilizar el patrón aceptables?
  - ¿Existen restricciones impuestas por el entorno de software que entrarían en conflicto con el uso del patrón?

Bibliografía

- PATRONES ANÁLISIS
  - [Fowler97] Fowler, Martin: Analysis Patterns: Reusable Object Models, Adisson Wesley, 1997
- PATRONES DISEÑO
  - Head-Up Patterns
  - **[LAR04]. GRASP**
  - **[GAM95] E. Gamma et al. Design Patterns. Elements of Software Oriented Reuse. Addison-Wesley 1995**
  - [Vlissides98]Vlissides, John: Pattern Hatching: Design Patterns Applied, Addison Wesley, 1998.
  - [SHA05] Design Patterns Explained. Alan Shalloway, James R. Trott. Addison Wesley 2005
- PATRONES ARQUITECTURA
  - [Buschman96] Buschmann, Frank et al.: Pattern Oriented Software Architecture, Volume 1: A System of Patterns, Willey & Sons, 1996.[
  - **[Fowler03] Fowler, Martin: Patterns of Enterprise Application Architecture, Addison Wesley, 2003.**
  - [Hophe03]Hophe, Gregor, Woolf, Robert: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addisson Wesley, 2003.
- ANTIPATRONES
  - [BRO98] W.J. Brown, R.C. Malveau, H.W. “Skip” McCormick III, T. J. Mowbray. Anti Patterns. Refactoring software, Architectures and Projects in Crisis.. Wiley, 1998.

¿Dudas?