Diseño de responsabilidades con patrones (GRASP)

Diseño de Sistemas Software

Carlos Pérez
Cristina Cachero

Índice

- Introducción
- Patrones GRASP Básicos
  - Creador
  - Experto (en Información)
  - Bajo Acoplamiento
  - Controlador
  - Alta Cohesión
- Patrones GRASP Avanzados
  - Polimorfismo
  - Fabricación Pura
  - Indirección
  - Protección de Variaciones

Objetivos del tema

- Comprender el uso del diagrama de clases para reflejar conceptos a distintos niveles de abstracción

- Ser capaz de realizar un diseño software que sea respetuoso con los principios de asignación de responsabilidades de GRASP

Introducción

Perspectiva de análisis vs. perspectiva de diseño en diagramas de clase UML

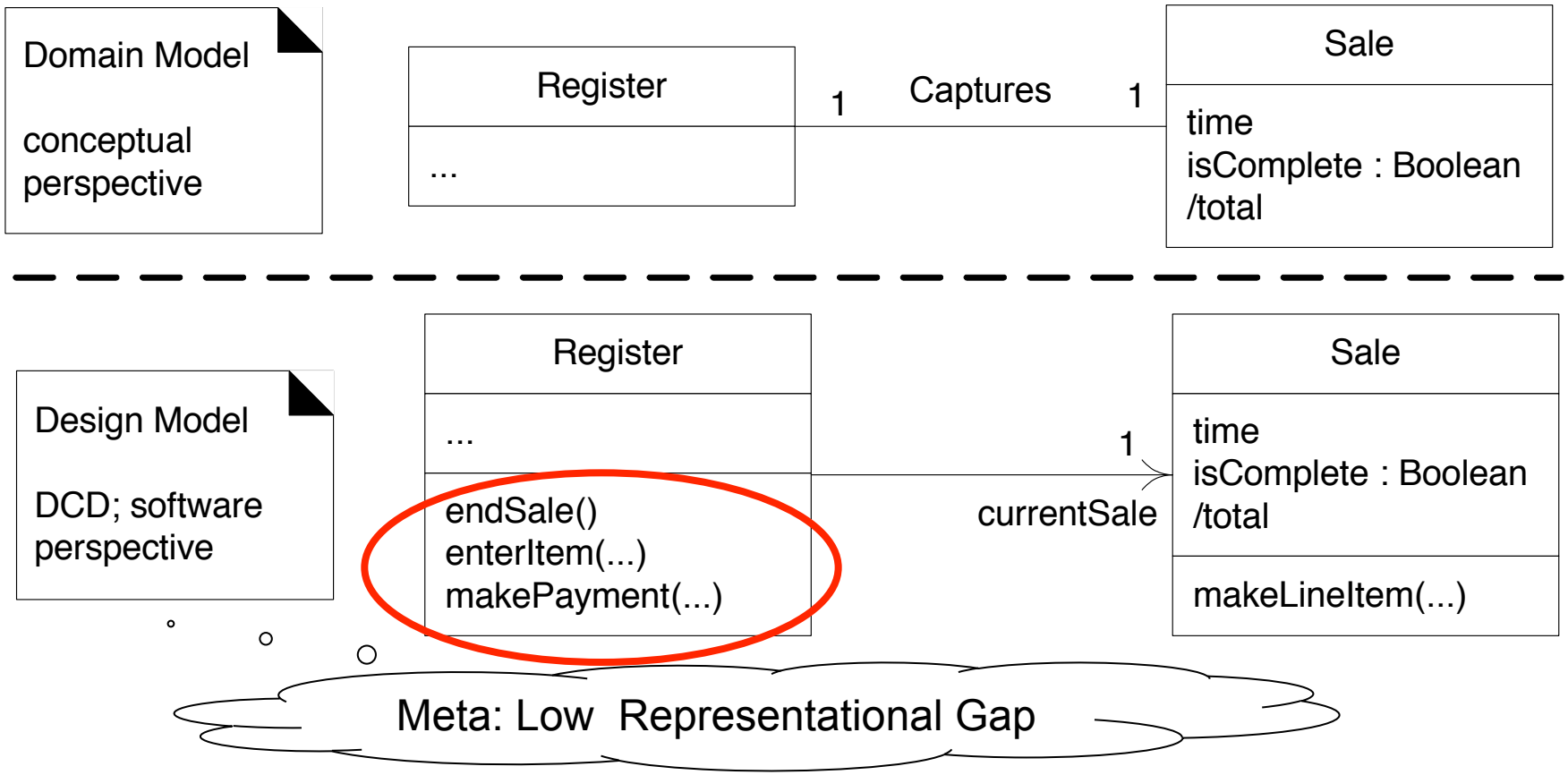
Introducción: Usos del diagrama de clases

- UML incluye los **diagramas de clases** para ilustrar clases, interfaces y sus asociaciones.
- Éstos se utilizan para el **modelado estático de objetos**.
- Los diagramas de clases se pueden usar tanto desde una perspectiva conceptual (para el modelo del dominio) como desde una perspectiva software (para el modelado de la CAPA DE DOMINIO).
  - Es común hablar de **"Diagrama de clases de Análisis"** para referirse al diagrama de clases que representa el modelo de dominio, y a **"Diagrama de Clases de Diseño"** para hablar del diagrama de clases que representa a las clases software, es decir, a las clases que finalmente van a ser implementadas.
  - EN LA ASIGNATURA NOS VAMOS A CENTRAR EN EL MODO DE DEFINIR LOS MODELOS DE DISEÑO Y, DENTRO DE ELLOS, EL DIAGRAMA DE CLASES DE DISEÑO

Introducción: Diseño de objetos

- Los diagramas de diseño de clases se derivan del modelo de dominio, añadiendo los métodos y secuencias de mensajes necesarios para satisfacer los requisitos.
- Por lo tanto tenemos que:
  - Decidir qué operaciones hay que asignar a qué clases
  - Cómo los objetos deberían interactuar para dar respuesta a los casos de uso
- El artefacto más importante del flujo de trabajo de diseño es el **Modelo de Diseño**, que incluye el **diagrama de clases software** (no conceptuales) y **diagramas de interacción**.

Introducción: Diseño de objetos



Pasando del análisis al diseño

- Los diagramas de diseño de clases normalmente contienen nuevas clases que no están presentes en el modelo de dominio
  - **Clases de utilidad:** p.ej. clases que encapsulan algoritmos genéricos que pueden ser accedidos por más de una clase
  - **Librerías:** referencias a librerías proporcionadas por el entorno (p.ej. STL)
  - **Interfaces:** definiendo comportamientos abstractos
  - **Clases de ayuda:** aparecen por la descomposición de clases grandes

Introducción: Responsabilidades

- UML define una **responsabilidad** como **"un contrato u obligación de un clasificador"**.
- Las responsabilidades se asignan a las clases durante el diseño de objetos
  - **Hacer:**
    - Hacer algo él mismo (e.g. crear un objeto, realizar un cálculo)
    - Iniciar la acción en otros objetos.
    - Controlar y coordinar actividades en otros objetos.
  - **Saber o Conocer:**
    - Conocer sus datos privados (encapsulados).
    - Conocer los objetos con los que se relaciona.
    - Conocer las cosas que puede derivar o calcular.

Patrones GRASP

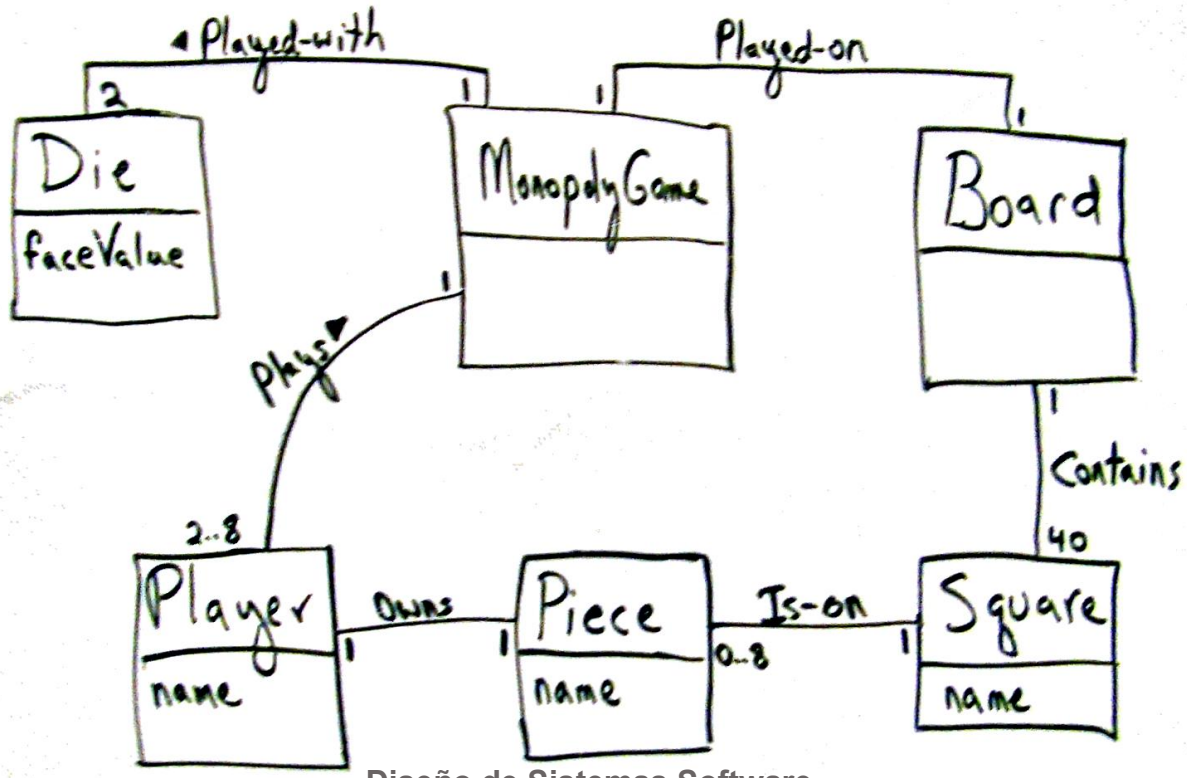
GRASP: Introducción

- **GRASP** es un acrónimo para General Responsibility Assignment Software Patterns.
- Describen **9 principios fundamentales del diseño de objetos** y de la **asignación de responsabilidades**, expresado en términos de patrones.
- Se dividen en dos grupos:

BÁSICOS	AVANZADOS
• Creador • Experto (en Información) • Bajo Acoplamiento • Controlador • Alta Cohesión	• Polimorfismo • Fabricación Pura • Indirección • Protección de Variaciones

GRASP: Introducción

- Para ilustrar los patrones vamos a suponer que queremos modelar un monopoly
- Dibujad su modelo de dominio (perspectiva conceptual)
  - Nota: asume dos dados
  - Nota: comienza modelando el tablero, las casillas, y el acto de hacer una tirada y mover las piezas por parte de un jugador



Patrones GRASP básicos

Creador

Experto (en información)

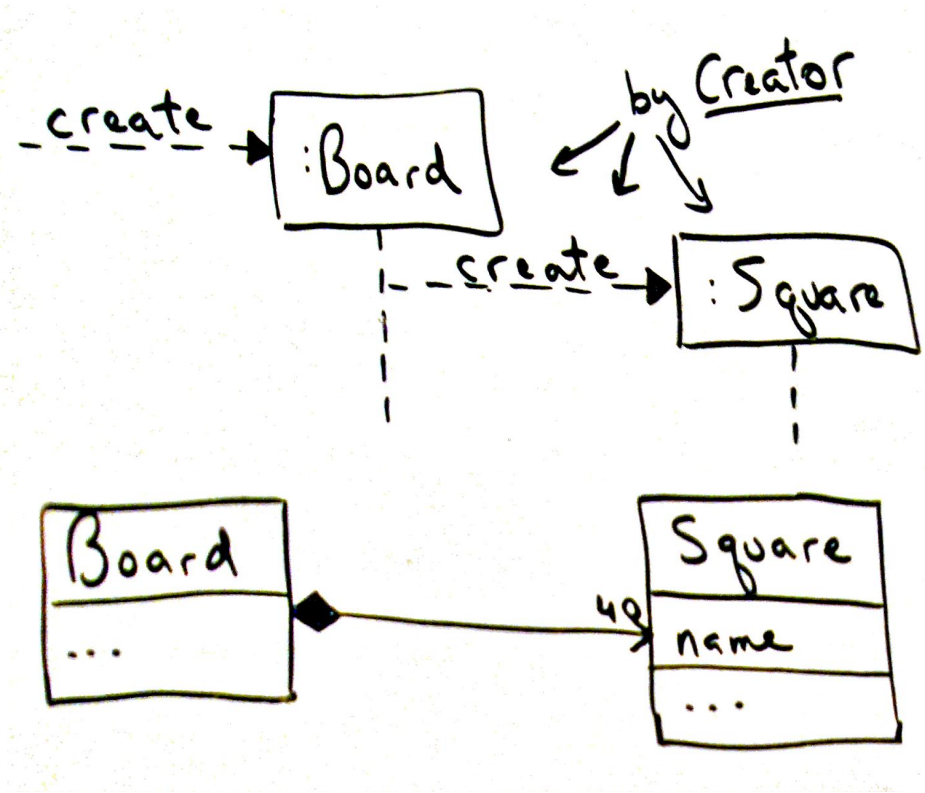
Bajo acoplamiento

Controlador

Alta cohesión

GRASP: Creador

- En el Monopoly, ¿quién debería ser el responsable de crear Casillas?
  - Puesto que un Tablero contiene casillas, por el patrón Creator éste debería ser el que las creara.
  - Además, una casilla sólo está en un tablero, por lo que la relación es de composición (refinamiento del DCD)



GRASP: Creador

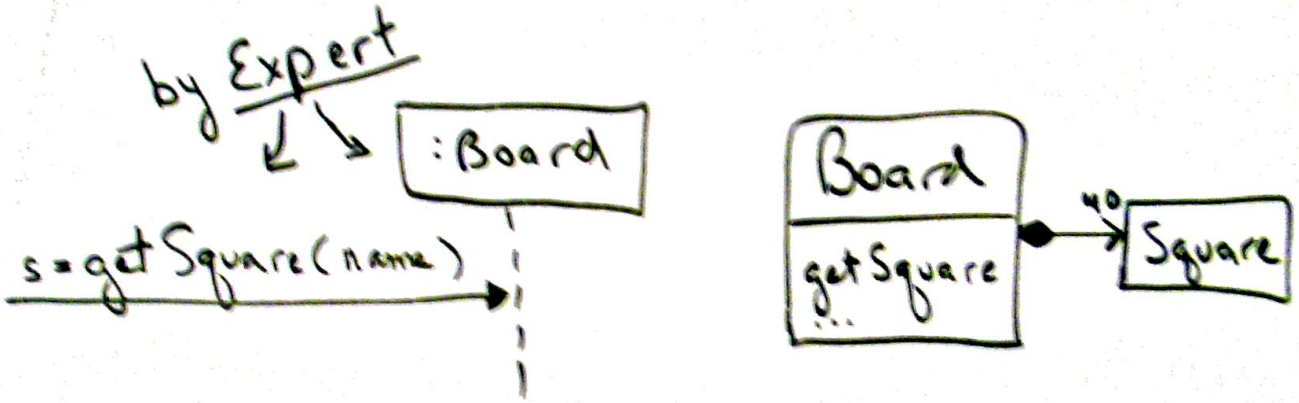
- **Problema:** ¿Quién debería ser el responsable de crear una nueva instancia de alguna clase?
- **Solución:** Asigna a la clase B la responsabilidad de crear una instancia de la clase A si una o más de las siguientes afirmaciones es cierta:
  1) B agrega objetos de tipo A.
  2) B contiene objetos de tipo A.
  3) B graba objetos de tipo A.
  4) B tiene datos inicializadores que serán pasados a A cuando sea necesario crear un objeto de tipo A (por tanto B es un Experto con respecto a la creación de A).

GRASP: Creador

- **Beneficios**
  - Promueve el bajo acoplamiento, lo que implica menos dependencias (mejor mantenimiento) y mayores oportunidades de reutilización.
- **Contraindicaciones**
  - A veces la creación de un objeto requiere cierta complejidad, como por ejemplo el uso de instancias recicladas para aumentar el rendimiento o la creación condicional de una instancia perteneciente a una familia de clases. En estos casos es preferible delegar la creación a una clase auxiliar, mediante el uso de los patrones *Concrete Factory* o *Abstract Factory*.

GRASP: Experto (en información)

- En el Monopoly, imaginad que necesitamos ser capaces de referenciar una casilla particular, dado su nombre (la calle que representa). ¿A quién le asignamos la responsabilidad?
- El patrón Information Expert nos indica que debemos asignar esa responsabilidad al objeto que conoce la información necesaria, i.e. los nombres de todas las casillas: este objeto es el objeto TABLERO



- ¿Por qué no poner el método getSquare(name) como estático en la clase Square?

GRASP: Experto (en información)

- **Problema**: ¿cuál es el principio general de asignación de responsabilidades a objetos?

- **Solución**: Asigna cada responsabilidad al experto de información: la clase que tiene (la mayor parte de) la información necesaria para cubrir la responsabilidad.

GRASP: Experto (en información)

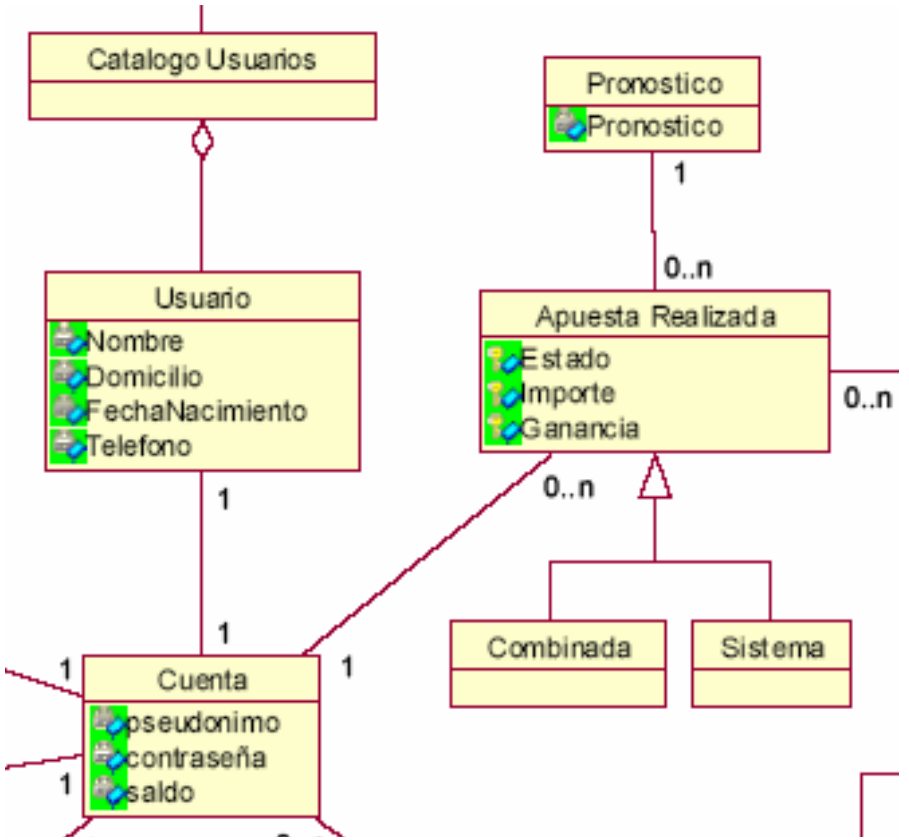
- **¡ATENCIÓN!**

  A veces hay que aplicar el Information Expert en cascada.

  - Ejemplo: Casa de apuestas

    Suponed que queremos asignar la responsabilidad de calcular la ganancia/pérdida neta de un usuario en el siguiente diagrama.

    Dibujad el diagrama de secuencia e indicad los cambios en el diagrama de clases.

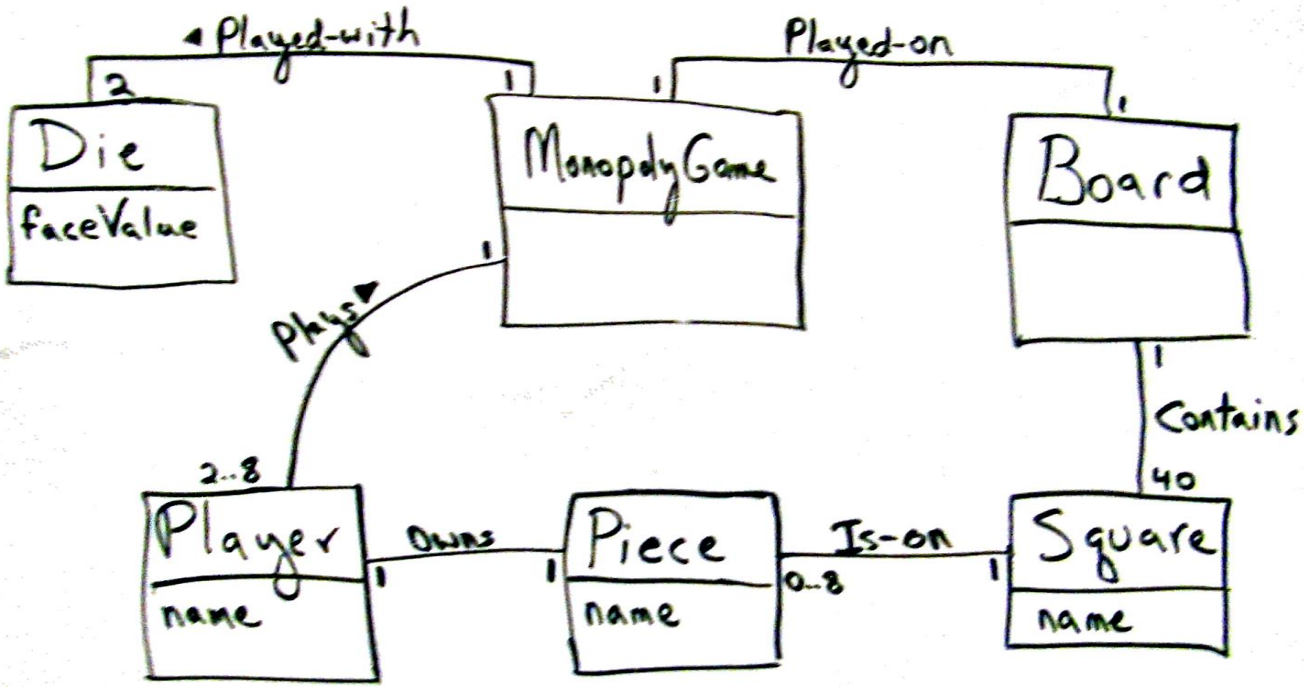


GRASP: Experto (en información)

- **Beneficios**
  - Se respeta la encapsulación de información, ya que los objetos usan su propia información para completar las tareas. Esto implica normalmente un bajo acoplamiento.
  - El comportamiento se distribuye entre las clases que contienen la información necesario, consiguiendo clases más ligeras.
- **Contraindicaciones**
  - ¿Quién debería ser el resposable de almacenar una apuesta en la base de datos? Añadir esta resposabilidad a la clase Apuesta aumentaría sus responsabilidades añadiendo lógica de acceso a datos, disminuyendo así su cohesión.

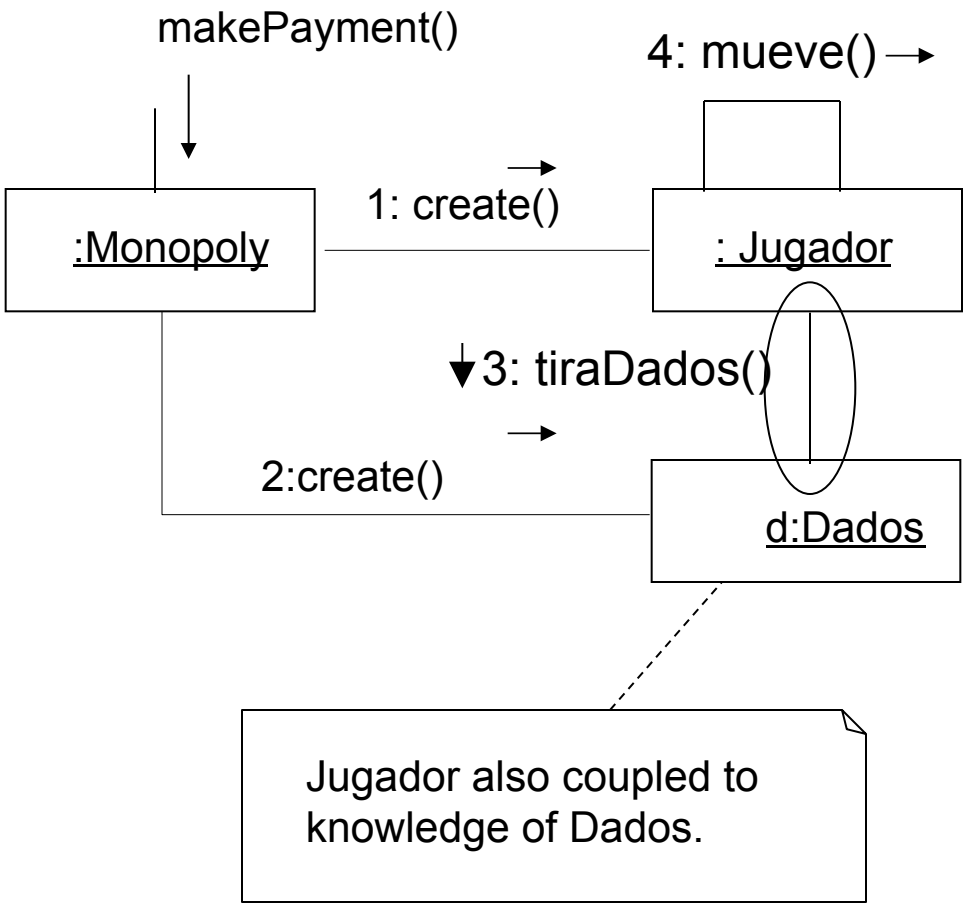
GRASP: Bajo acoplamiento

- Asume que necesitamos modelar el acto de tirar los dos dados de un jugador en el monopoly. ¿A quién asignamos la responsabilidad?



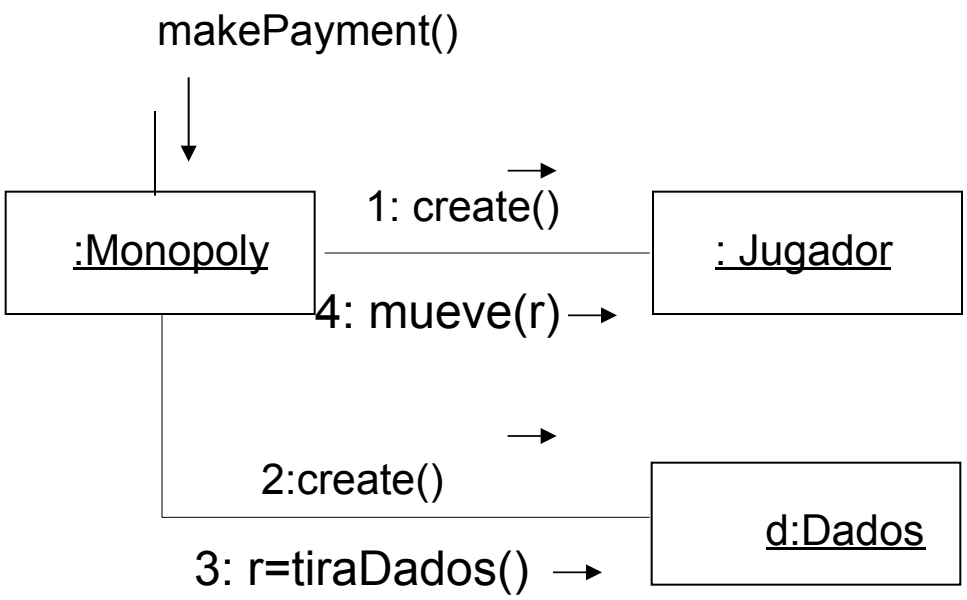
- Según el patrón Creador, Jugador es el mejor candidato (ya que Jugador “tira” los dados en el mundo real).
- ¿Qué problema tiene esta solución?

GRASP: Bajo acoplamiento



- Si el jugador tira los dados, es necesario acoplar al Jugador con conocimiento sobre los Dados (acoplamiento que antes no existía)

GRASP: Bajo acoplamiento



- Una solución alternativa es que sea el propio Juego el que tire los dados y envíe el valor que ha salido al Jugador.
- Se disminuye el acoplamiento entre Jugador y Dados y, desde el punto de vista del "acoplamiento" es un mejor diseño.

GRASP: Bajo acoplamiento

- **Problema:** ¿Cómo soportar una baja dependencia, un bajo impacto de cambios en el sistema y un mayor reuso?

- **Solución:** Asigna una responsabilidad de manera que el acoplamiento permanezca bajo

GRASP: Bajo acoplamiento

- **Acoplamiento:** medida que indica cómo de fuertemente un elemento está conectado a, tiene conocimiento de, o depende de otros elementos.
  - Una clase con un alto acoplamiento depende de muchas otras clases (librerías, herramientas, etc.)
- **Problemas del alto acoplamiento:**
  - Cambios en clases relacionadas fuerzan cambios en la clase afectada por el alto acoplamiento.
  - La clase afectada por el alto acoplamiento es más difícil de entender por sí sola: necesita entender otras clases.
  - La clase afectada por el alto acoplamiento es más difícil de reutilizar, porque requiere la presencia adicional de las clases de las que depende.

GRASP: Bajo acoplamiento

- Tipos de acoplamiento

  El acoplamiento dentro de los D. Clases puede ocurrir por varios motivos:

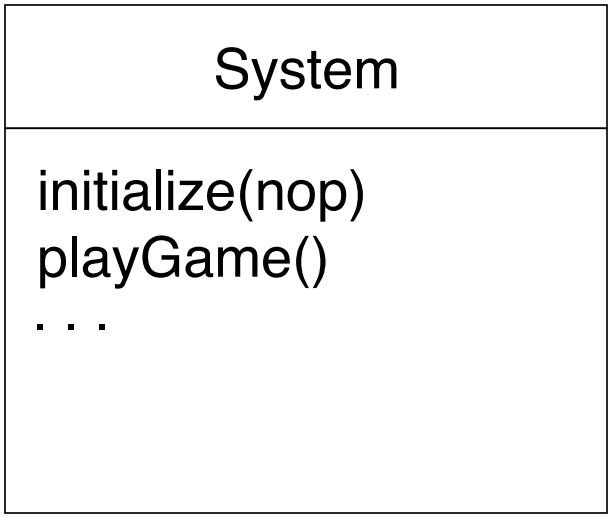
  - **Definición de Atributos:** X tiene un atributo que se refiere a una instancia Y.
  - **Definición de Interfaces de Métodos:** p.ej. un parámetro o una variable local de tipo Y se encuentra en un método de X.
  - **Definición de Subclases:** X es una subclase de Y.
  - **Definición de Tipos:** X implementa la interfaz Y.

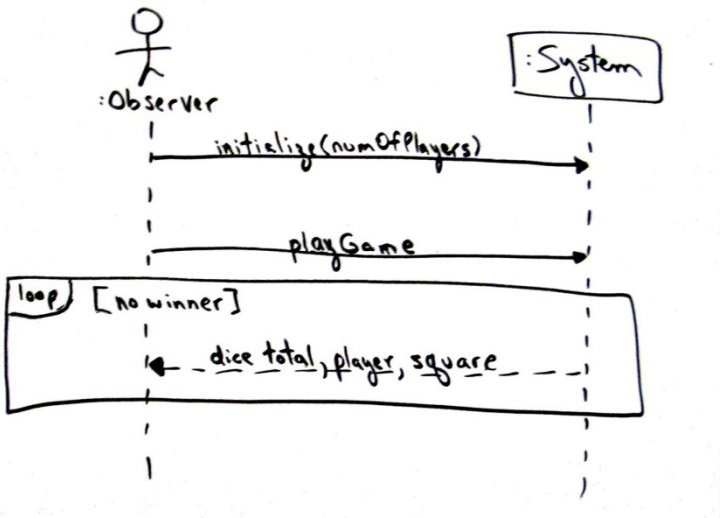
¿CUÁL ES MEJOR?

GRASP: Bajo acoplamiento

- **Beneficios**
  - Las clases con bajo acoplamiento normalmente…
    - No les afectan los cambios en otros componentes
    - Son sencillas de comprender de forma aislada
    - Es fácil reutilizarlas
- **Contraindicaciones**
  - El alto acoplamiento con elementos estables es raramente un problema, ya raramente habrá cambios que puedan afectar a estas clases, por lo que no merece la pena el esfuerzo de evitar este acoplamiento.

GRASP: Controlador

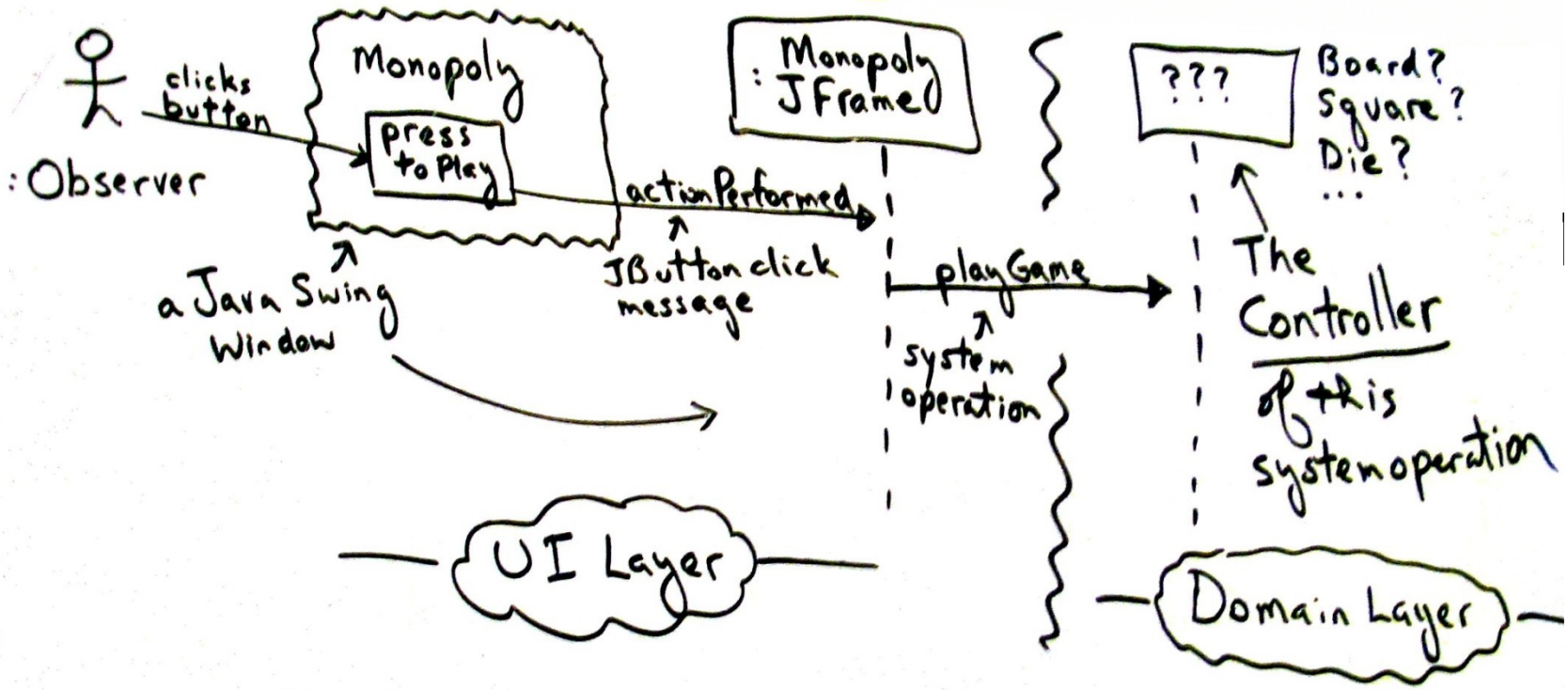




- En el Monopoly puede haber múltiples operaciones del sistema, que en un principio se podrían asignar a una clase **System**.
- Sin embargo esto no significa que finalmente deba existir una clase software **System** que satisfaga este requisito; es mejor asignar las responsabilidades a uno o más **Controllers**.

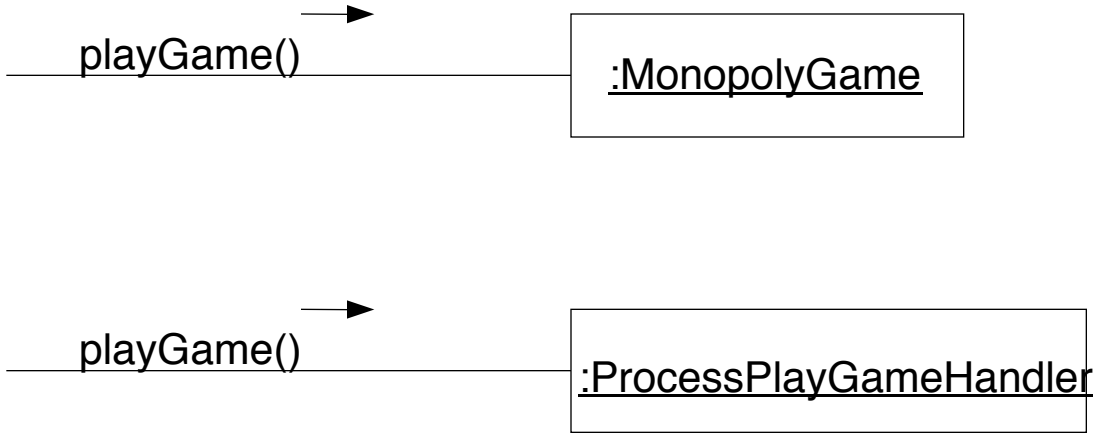
GRASP: Controlador

- ¿En el monopoly, quién debería ser el **Controlador** para los eventos de sistema, tales como *playGame* e *initialize*?

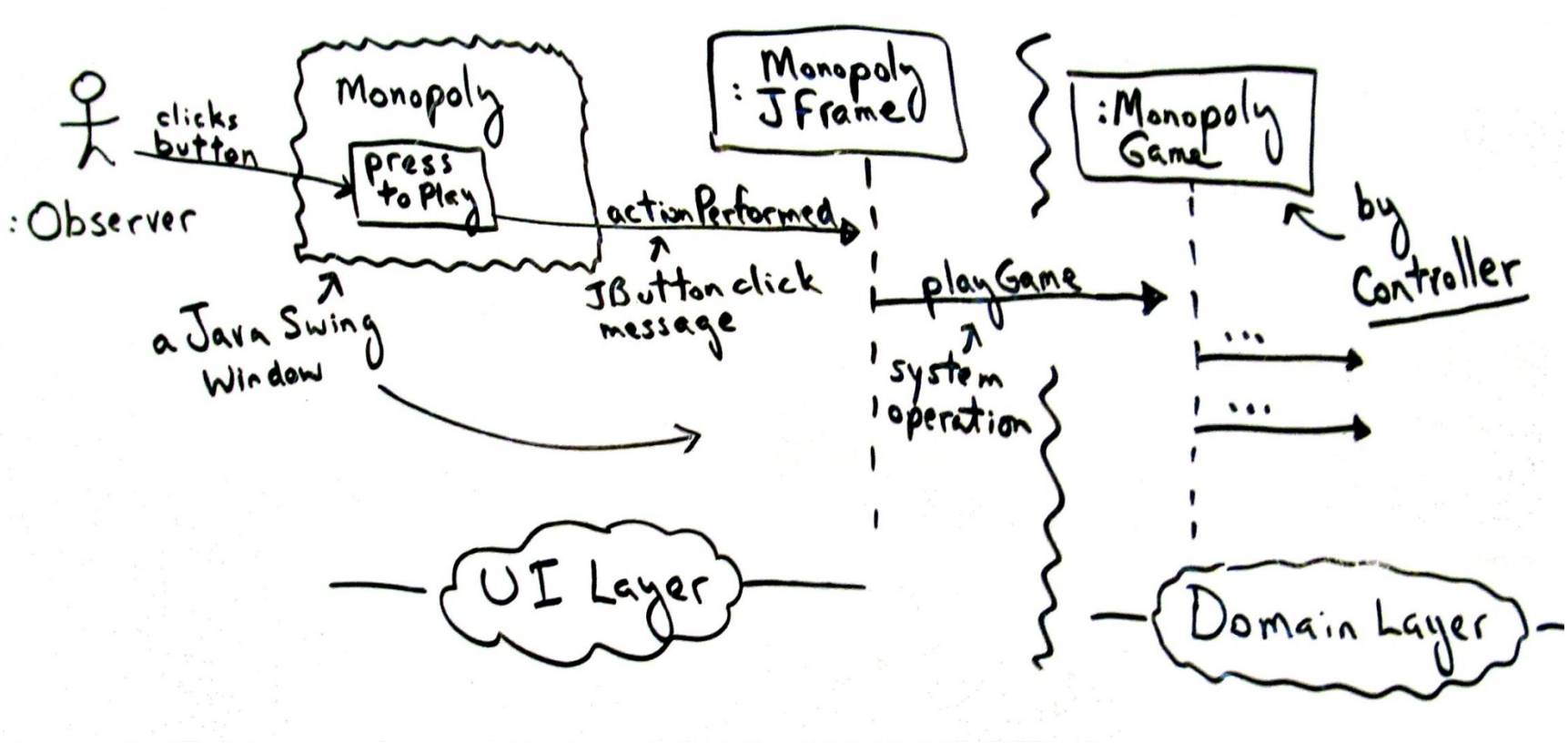


GRASP: Controlador

- Existen dos posibilidades:
  - MonopolyGame
  - ProcessInitializeHandler (esta solución requiere que haya otros controladores como ProcessPlayGameHandler, etc.).



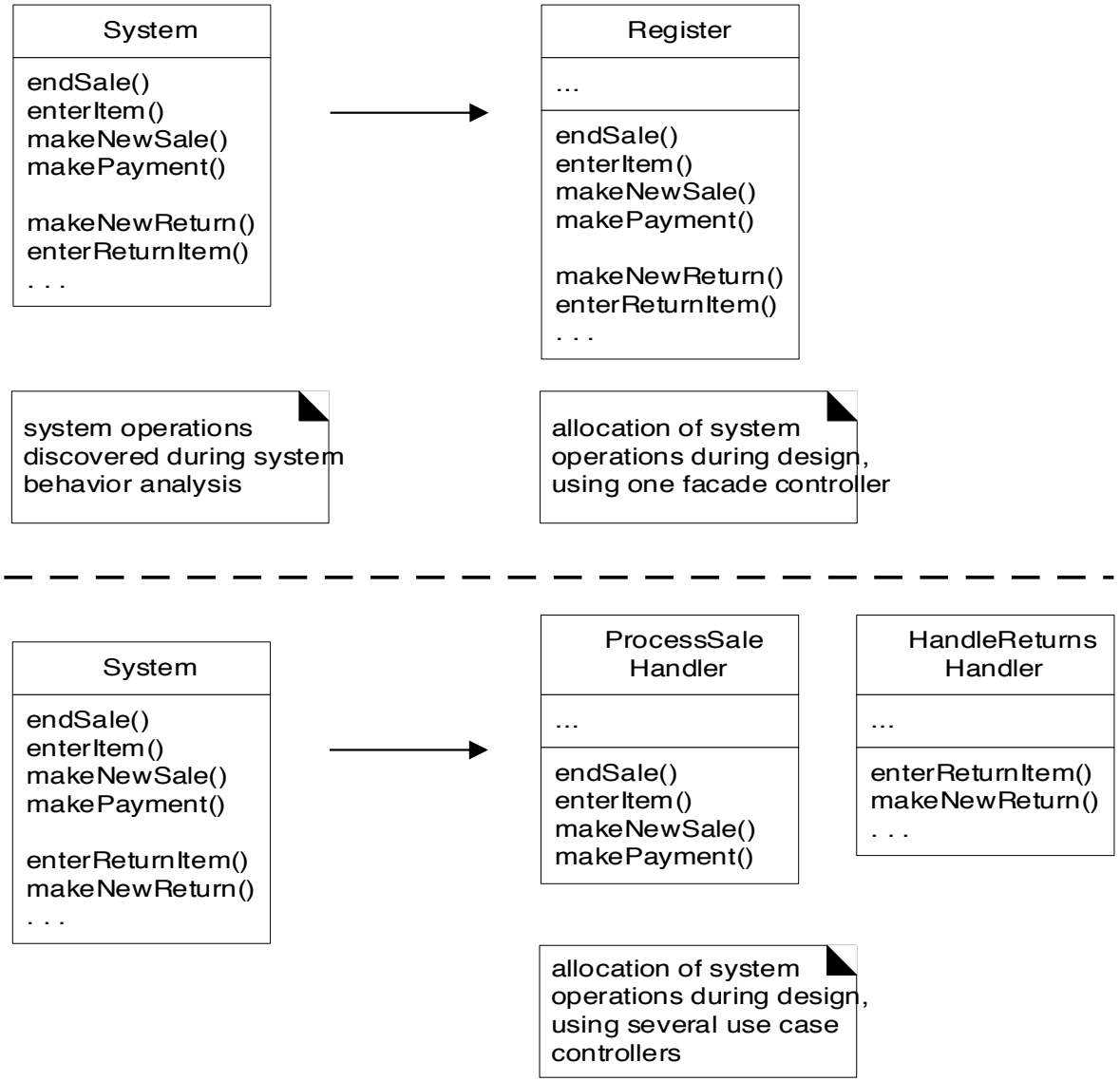
GRASP: Controlador



GRASP: Controlador

- **Problema:** ¿Quién debería ser el responsable de manejar un evento de entrada al sistema?
  - ¿Quién es el primer objeto de la capa de dominio que recibe los mensajes de la interfaz?
- **Solución:** Asigna la responsabilidad de recibir o manejar un evento del sistema a una clase que represente una de estas dos opciones:
  - El sistema completo (Control 'fachada').
  - Un escenario de un Caso de Uso (estandariza nomenclatura: ControladorRealizarCompra, CoordinadorRealizarCompra, SesionRealizarCompra, ControladorSesionRealizarCompra.)

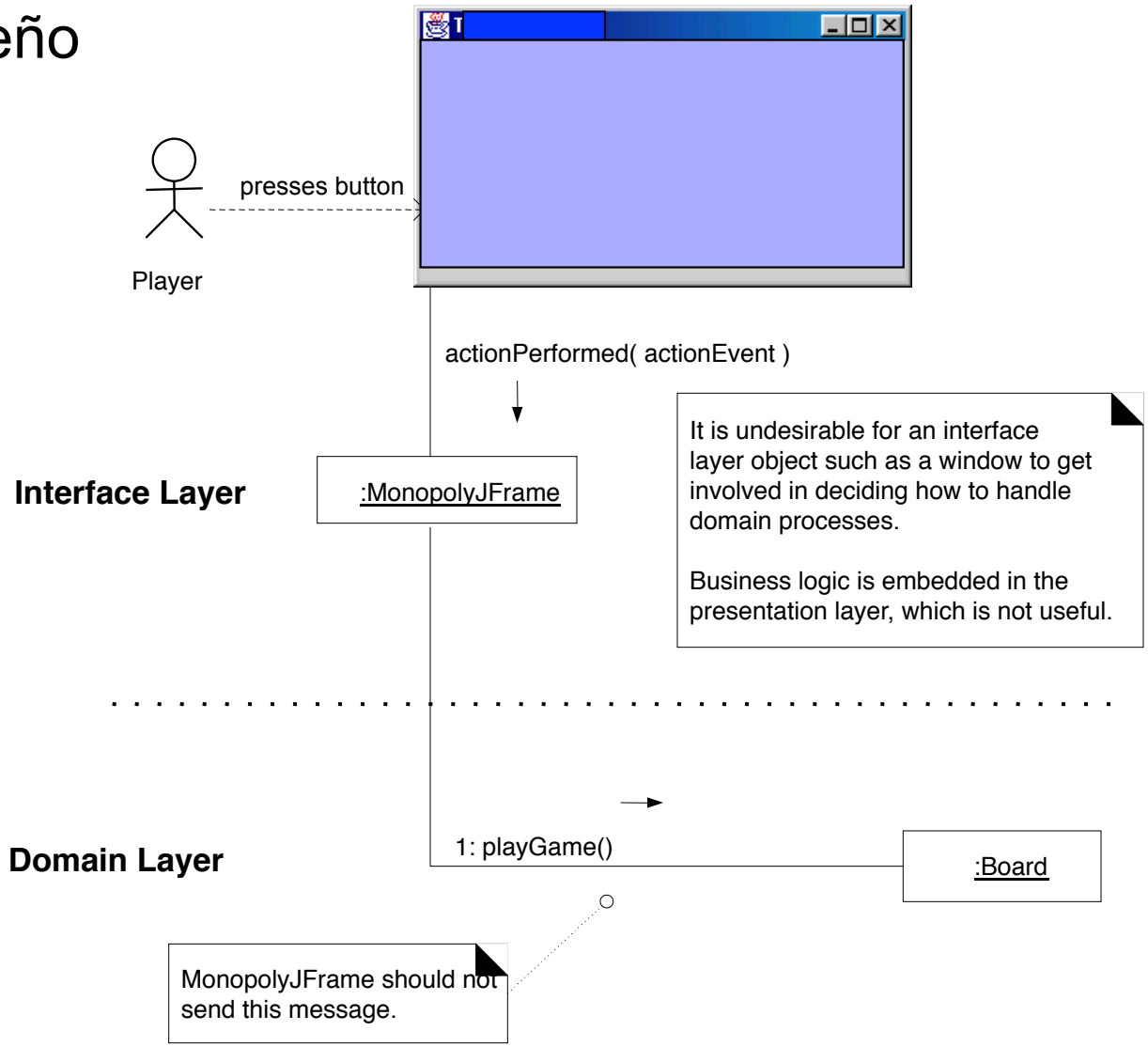
GRASP: Controlador



- El **Controlador** del que habla este patrón es un objeto que NO pertenece a la capa de interfaz y que define el método para la operación del sistema.
- Normalmente ventanas, applets etc reciben eventos mediante sus propios controladores de interfaz, y los DELEGAN al tipo de controlador del que hablamos aquí.

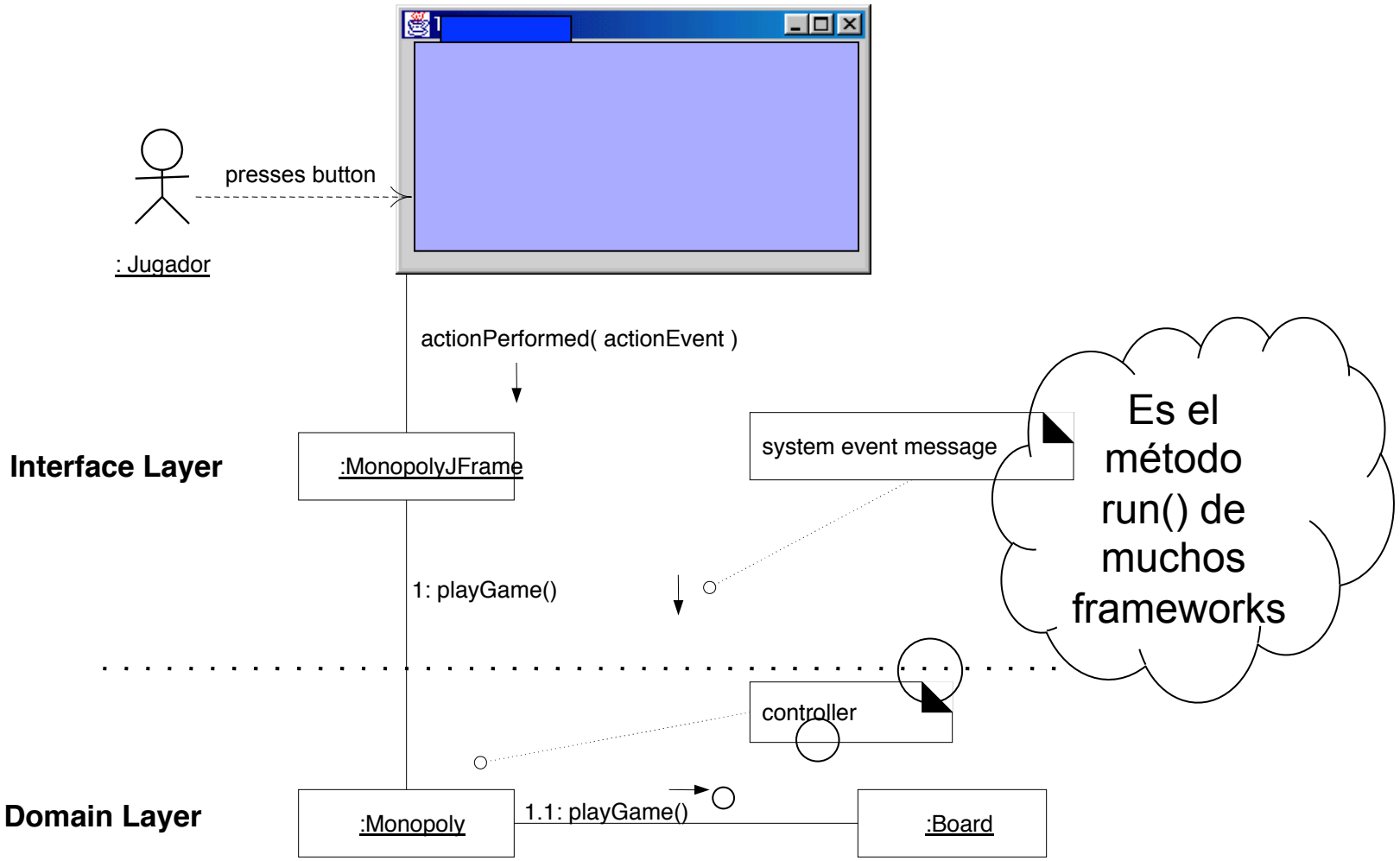
GRASP: Controlador

- Mal diseño



GRASP: Controlador

- Buen diseño

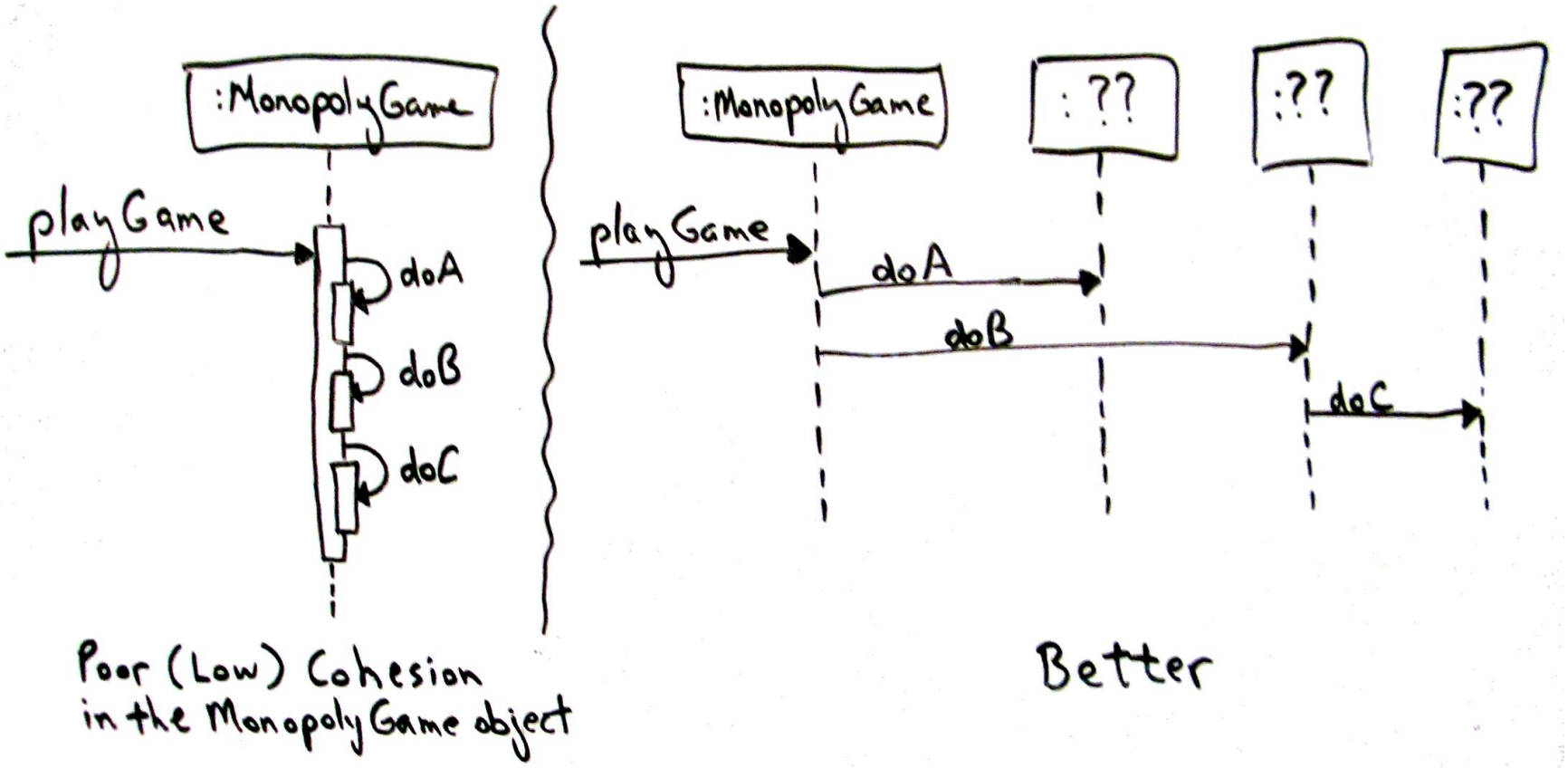


GRASP: Alta cohesión

- ¿Cómo reescribiríais el siguiente código del monopoly?

```
Monopoly::PlayGame(){
    turno=random(numJug);
    Dados d[2]=cub.getDados();
    int punt=0;
    for (i=1 to 2)
        punt+=dado[i].tirarDado();
  Casilla cAct=jug[turno].getCasillaAct();
  Casilla cNueva=cAct+punt;
  jug[t].colocaEnCasilla(cNueva)
  …
  turno=turno+1 mod numJug;
  …
}
```

GRASP: Alta cohesión



GRASP: Alta cohesión

- **Cohesión:** medida de cómo de fuertemente se relacionan y focalizan las responsabilidades de un elemento. Los elementos pueden ser clases, subsistemas, etc.
- Una clase con baja cohesión hace muchas actividades poco relacionadas o realiza demasiado trabajo.
- **Problemas** causados por un diseño con **baja cohesión**:
  - Difíciles de entender.
  - Difíciles de usar.
  - Difíciles de mantener.
  - Delicados: fácilmente afectables por el cambio.

GRASP: Alta cohesión

- **Problema:** ¿Cómo mantener la complejidad manejable?

- **Solución:** Asigna las responsabilidades de manera que la cohesión permanezca alta.
  - Clases con baja cohesión a menudo representan abstracciones demasiado elevadas, o han asumido responsabilidades que deberían haber sido asignadas a otros objetos.

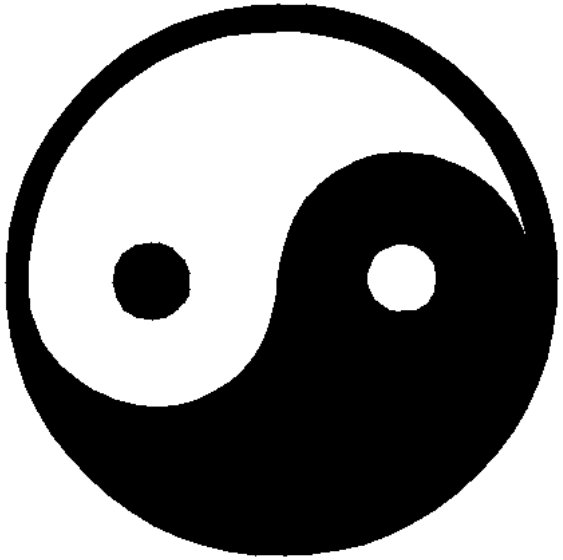
GRASP: Alta cohesión

- Existen diversos grados de cohesión funcional
  - **Cohesión muy baja**: clase responsable de muchas cosas en muchas áreas distintas.
    - ej.: una clase responsable de hacer de interfaz con una base de datos y con un servicio web.
  - **Baja cohesión**: clase responsable de una tarea compleja en un área funcional.
    - ej.: una clase responsable de interactuar con una base de datos completa
  - **Alta cohesión**: clase que tiene responsabilidades moderadas en un área funcional y colabora con otras clases para realizar una tarea.
    - ej.: una clase responsable de una parte de la interacción con la BD.

GRASP: Alta cohesión

- **Beneficios**
  - Aumenta la claridad y comprensibilidad del diseño.
  - Se simplifican el mantenimiento y las mejoras.
- **Contraindicaciones:** en algunas ocasiones se puede aceptar una menor cohesión:
  - Agrupar responsabilidades o código en una única clase o componente puede simplificar el mantenimiento por una persona (p.ej. Un experto en bases de datos)
  - Objetos distribuidos entre servidores. Debido al sobrecoste y las implicaciones en el rendimiento, a veces es deseable crear objetos menos cohesivos que provean un interfaz para numerosas operaciones.

GRASP: Cohesión vs. acoplamiento



- Una mala cohesión normalmente implica un mal acoplamiento, y viceversa. Una clase con demasiadas responsabilidades (baja cohesión), normalmente se relaciona con demasiadas clases (alto acoplamiento).

Patrones GRASP avanzados

Polimorfismo

Fabricación pura

Indirección

Protección de variaciones

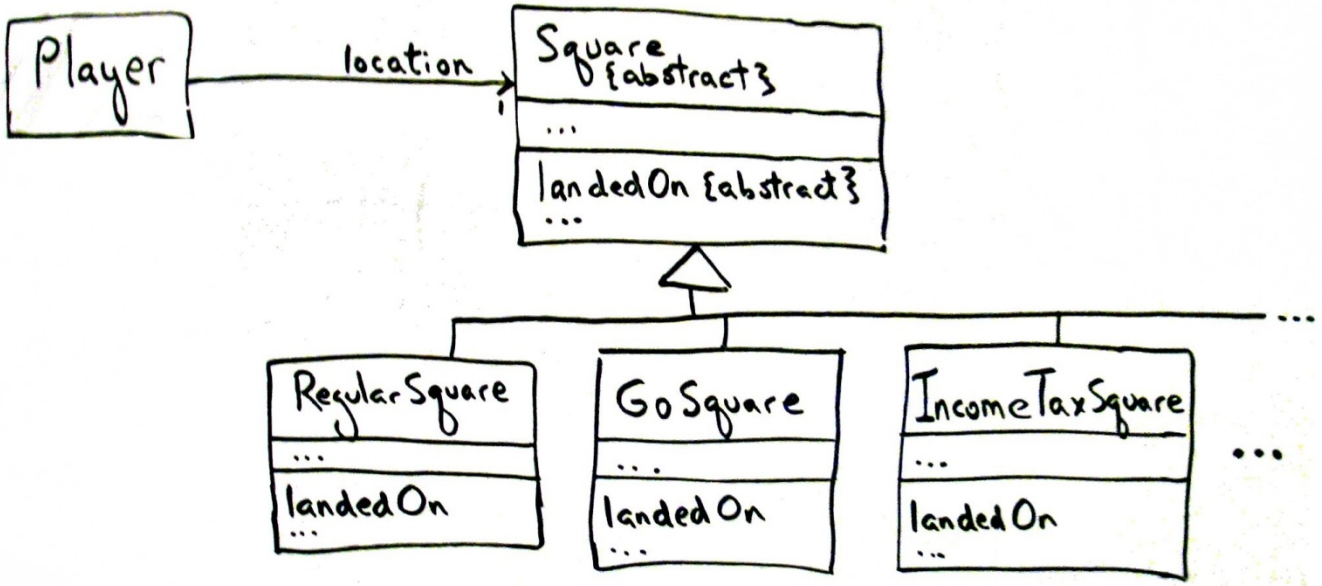
GRASP: Polimorfismo

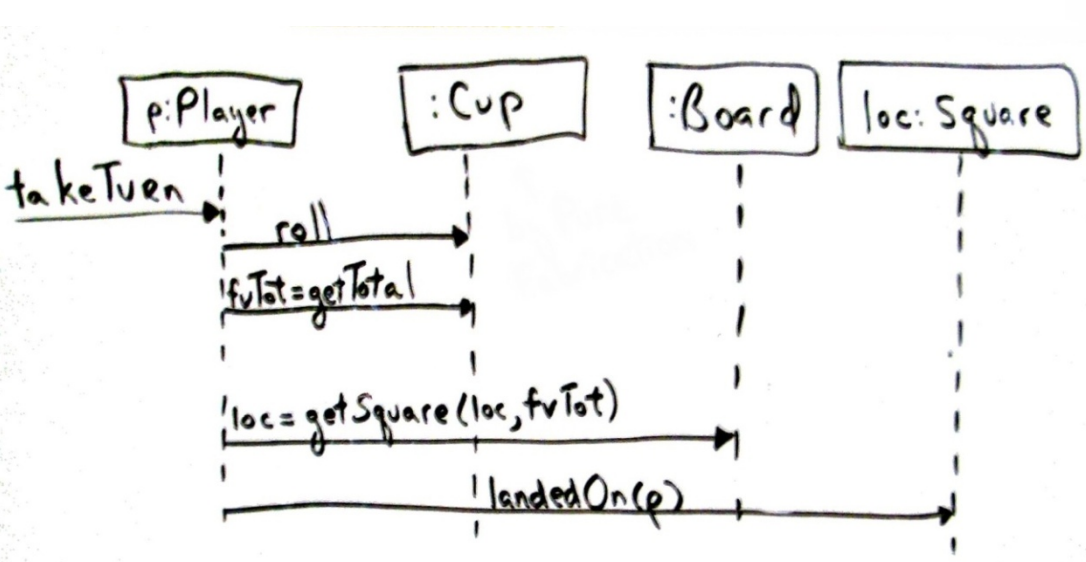
- Vamos ahora a incluir el concepto de tipos de casillas en el Monopoly.
- Distintos tipos de casillas. En función del tipo de casilla, el comportamiento del método caerEn() varía:
  - Casillas de suerte: coger una carta de la suerte
  - Casillas de propiedades: depende de si la casilla tiene propietario o no, y si lo tiene si soy yo o es un contrincante del juego
  - Casilla de Vaya a la Cárcel: ir a la cárcel
  - Casilla de Tasas: pagar tasas
  - …
- ¿A quién asigno la responsabilidad caerEn()?¿Cómo lo implemento?

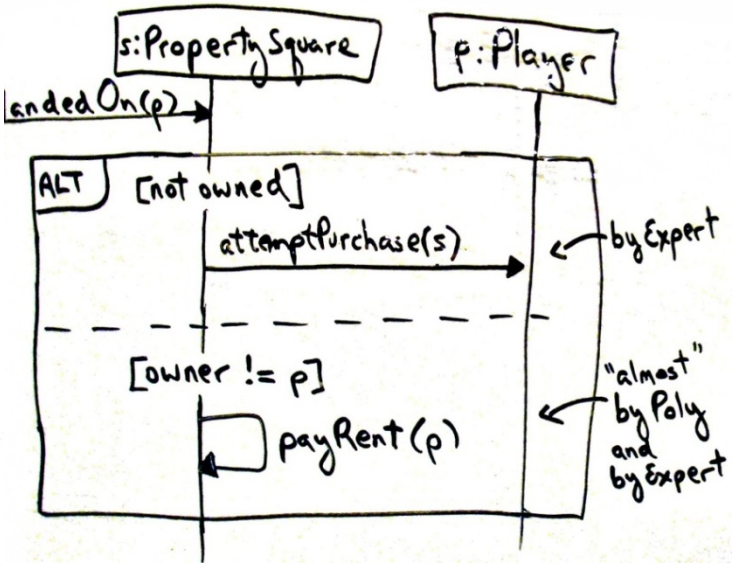
GRASP: Polimorfismo

- **Problemas:**
  - ¿Cómo manejar alternativas basadas en un tipo sin usar sentencias condicionales if-then o switch que requerirían modificación en el código?
  - ¿Cómo crear componentes software “conectables”?
    - Viendo los componentes software en una relación cliente-servidor ¿Cómo reemplazar un componente servidor con otro sin afectar al cliente?
- **Solución:**
  - Cuando alternativas o comportamientos relacionados varían por el tipo (clase), asigna la responsabilidad del comportamiento usando “operaciones polimórficas” a los tipos para los cuales el comportamiento varía.
  - **Corolario:** No preguntes por el tipo del objeto usando lógica condicional para realizar las alternativas variantes basadas en el tipo.

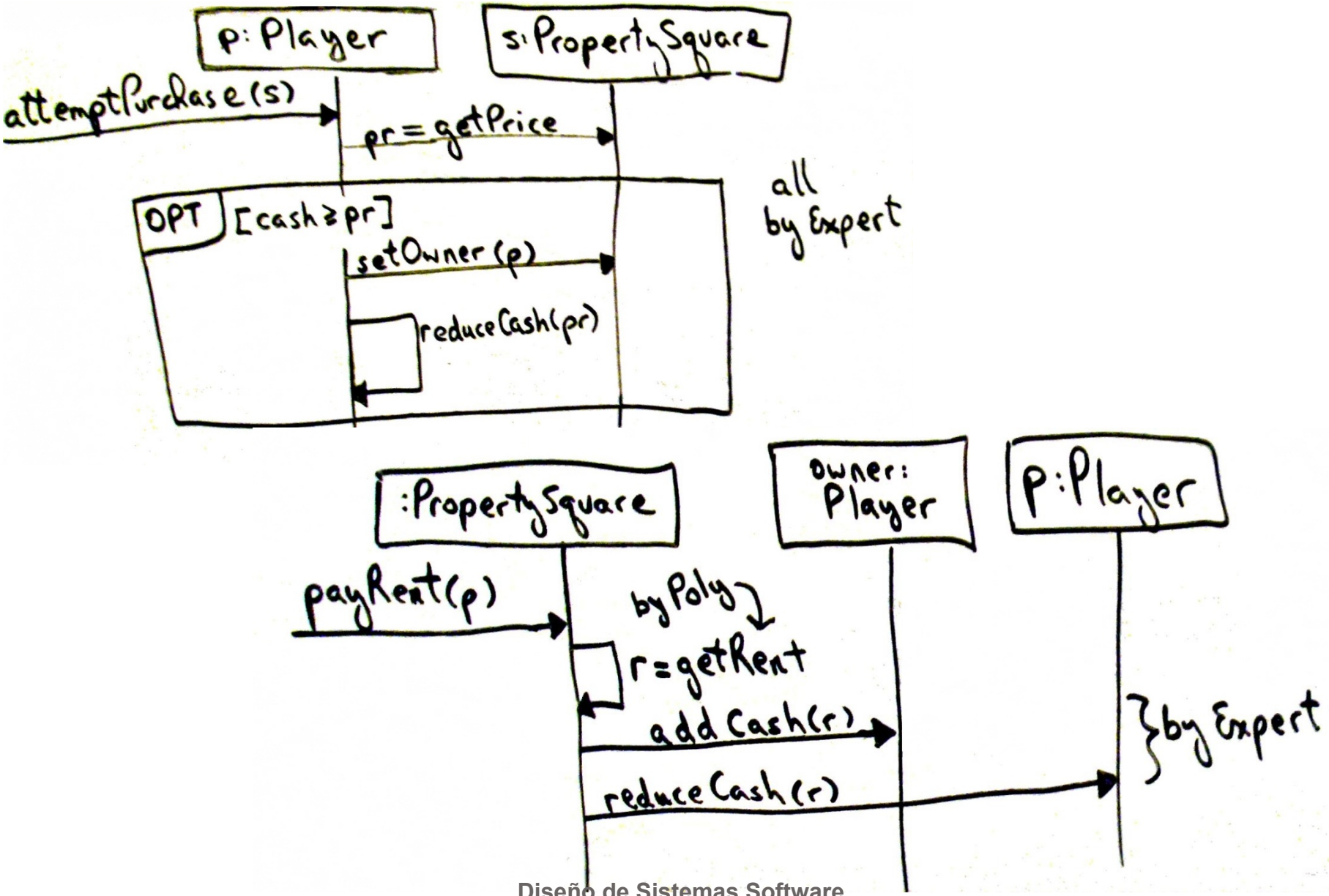
GRASP: Polimorfismo







GRASP: Polimorfismo



GRASP: Polimorfismo

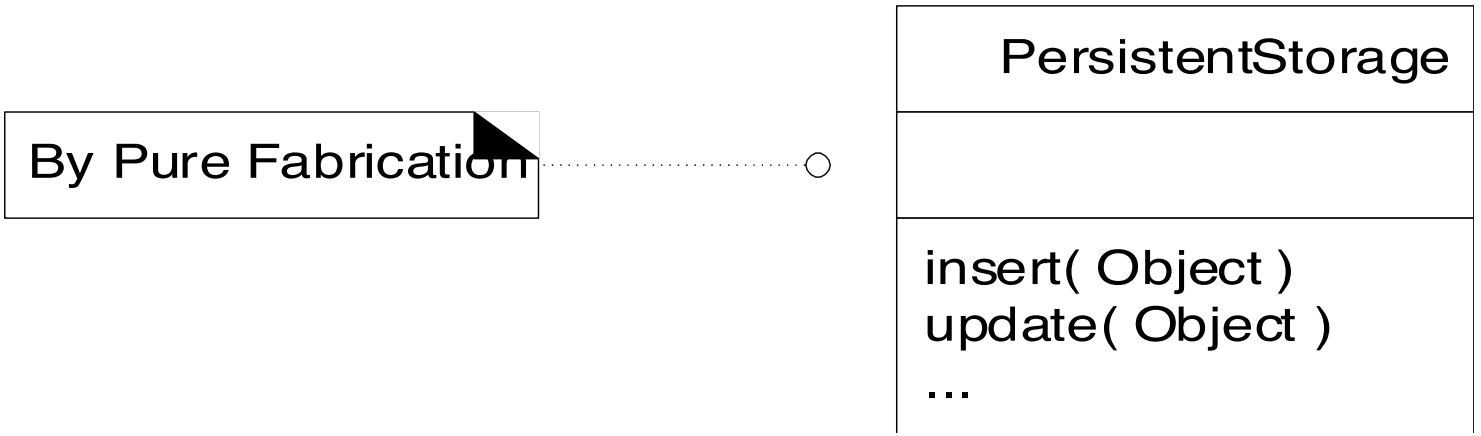
- **Beneficios**
  - Es fácil extender el sistema con nuevas variaciones.
  - Se pueden introducir nuevas implementaciones sin afectar a las clases cliente.
- **Contraindicaciones**
  - No es raro dedicar demasiado tiempo a la realización de diseños preparados para cambios poco probables, mediante el uso de herencia y polimorfismo.

GRASP: Fabricación pura

- Es necesario guardar instancias del *Monopoly* en una base de datos relacional. ¿Quién debería tener esa responsabilidad?
- Por **Expert** la clase Monopoly debería tener esta responsabilidad, sin embargo:
  - La tarea requiere un número importante de operaciones de base de datos, ninguna relacionada con el concepto de Monopoly, por lo que Monopoly resultaría **incohesiva**.
  - Monopoly quedaría acoplado con la interface de la base de datos (ej.- JDBC en Java, ODBC en Microsoft) por lo que el **acoplamiento aumenta**. Además el acoplamiento sería a una tecnología específica de base de datos, no a otro objeto del dominio (que sería menos grave).
  - Guardar objetos en una base de datos relacional es una tarea muy general para la cual se requiere que múltiples clases le den soporte. Colocar éstas en Monopoly sugiere **pobre reuso** o gran cantidad de duplicación en otras clases que hacen lo mismo.

GRASP: Fabricación pura

- **Solución:** Crear una clase (*PersistentStorage*) que sea responsable de guardar objetos en algún tipo de almacenamiento persistente (tal como una base de datos relacional).



GRASP: Fabricación pura

- Problemas resueltos:
  - La clase Monopoly continua bien definida, con alta cohesión y bajo acoplamiento.
  - La clase PersistentStorage es, en sí misma, relativamente cohesiva, tiene un único propósito de almacenar o insertar objetos en un medio de almacenamiento persistente.
  - La clase PersistentStorage es un objeto genérico y reusable.

GRASP: Fabricación pura

- **Problema:** ¿Qué objeto debería tener la responsabilidad, cuando no se desean violar los principios de “Alta Cohesión” y “Bajo Acoplamiento” o algún otro objetivo, pero las soluciones que sugiere Expert (por ejemplo) no son apropiadas o cuando no es apropiado asignarlo a una clase software inspirada a partir de una clase conceptual?
- **Solución:** Asigne un conjunto “altamente cohesivo” de responsabilidades a una clase artificial conveniente que no represente un concepto del dominio del problema, algo producto de la “imaginación” para soportar “high cohesion”, “low coupling” y reuso.
  - Éste es precisamente el principio que se aplica cuando se introducen clases ‘Helper’ y clases ‘Utility’

GRASP: Fabricación pura

- En sentido amplio, los objetos pueden dividirse en dos grupos:
  - Aquellos diseñados por/mediante **descomposición representacional**. (Ej.- *Monopoly* representa el concepto “partida”)
  - Aquellos diseñados por/mediante **descomposición conductual**. Este es el caso más común para objetos **Fabricación pura.**

GRASP: Indirección

- ¿Cómo podemos desacoplar el juego del hecho de que se juega con 2 dados?
- Solución: Cubilete
  - El objeto Cubilete es un ejemplo de indirección: un elemento que no existía en el juego real pero que introducimos para aislarnos del número de dados que usa el juego.
- ¿Cuándo hemos visto esto antes?

GRASP: Indirección

- **Problema:**
  - ¿Dónde asignar una responsabilidad para **evitar acoplamiento directo entre dos o más cosas**? ¿Cómo desacoplar objetos de tal manera que el bajo acoplamiento se soporte y el reuso potencial se mantenga alto?
- **Solución:**
  - **Asignad la responsabilidad a un objeto intermedio** que medie entre otros componentes o servicios, de tal manera que los objetos no estén directamente acoplados. El objeto intermedio crea una **indirección** entre los componentes.

GRASP: Indirección

- ***"Muchos problemas en ciencias de la computación pueden resolverse mediante otro nivel de indirección"*** es un viejo adagio con relevancia particular en diseño orientado a objetos (David Wheeler).
- Así como muchos patrones de diseño son especializaciones de **Fabricación Pura**, muchos otros también lo son de **Indirección** (Adapter, Facade, Observer, entre otros).
- Además muchas Fabricaciones Puras son generadas por causa de Indirección.
- **La motivación principal** es el **bajo acoplamiento**; por lo que un intermediario se agrega para desacoplar otros componentes o servicios.

GRASP: Protección de variaciones

- **Problema:**
  - ¿Cómo diseñar objetos, subsistemas y sistemas de tal manera que las variaciones o inestabilidad en estos elementos no tenga un impacto indeseable sobre otros elementos?
- **Solución:**
  - *Identifique los puntos de variación o inestabilidad; asigne responsabilidades para crear una interfaz estable a su alrededor.*
  - Nota: el término interfaz se usa en su sentido más amplio, refiriéndose a los métodos que expone una clase; no implica el uso de interfaces de lenguajes de programación.