Diseño de Software Orientado a Objetos

Ingeniería de Software – IIC2143

El diseño pone el énfasis en una solución conceptual que satisface los requisitos

Una solución *conceptual* tanto en software como en hardware —no mira la implementación de la solución

Excluye detalles de bajo nivel u obvios —obvios para los usuarios

En último término los diseños pueden ser implementados, y la implementación (p.ej., código en Java) expresa el verdadero diseño llevado a la práctica

Durante el **diseño orientado a objetos**, el énfasis se pone en definir objetos de software,

… sus responsabilidades,

… y cómo ellos colaboran para satisfacer los requisitos

Cómo se enseña a veces el diseño de software

"Después de identificar los requisitos y crear un modelo del dominio, agregue métodos a las clases apropiadas y defina los mensajes entre los objetos para cumplir con los requisitos"

Un consejo tan vago no ayuda

Hay involucrados principios y temas de discusión profundos

Decidir qué métodos van dónde y cómo debieran interactuar los objetos tiene consecuencias

Hacer diseño OO involucra un conjunto grande de principios “soft”, con varios grados de libertad

No hay magia —los principios (y patrones) pueden ser nombrados, explicados y aplicados

Los ejemplos ayudan; la práctica ayuda

Saber los detalles de UML no enseña a pensar en términos de objetos

El UML es sólo un lenguaje estándar de modelación visual:

- a veces se lo describe como una “herramienta de diseño”
- … pero esto no es realmente correcto

La herramienta de diseño crítica para desarrollo de software es

… una **mente bien educada en los principios de diseño**

¿Cuáles son los inputs para el diseño de software?

Hay procesos:

- hicimos un levantamiento de requisitos (funcionales)
- analizamos en detalle los casos de uso más importantes

Hay artefactos:

- los casos de uso definen el comportamiento visible que los objetos de software deben proporcionar —los objetos son diseñados para implementar los casos de uso
- los DSS's (diap.#13) identifican las operaciones del sistema —los mensajes iniciales (*found messages*) en los diagramas de secuencia
- el modelo de dominio sugiere algunos nombres y atributos de los objetos de software

¿Cuáles son las actividades del diseño de software?

A partir de los inputs anteriores

- … podemos escribir código inmediatamente
- … o a hacer modelamiento con UML para el diseño de objetos
- … o emplear alguna otra técnica de modelamiento

El punto de UML es modelamiento visual —usar un lenguaje que nos permita explorar más visualmente que lo que permite el texto:

- dibujamos diagramas de secuencia y diagramas de clase complementarios
- … y aplicamos principios y patrones de diseño (que ya veremos)
- … basados en la metáfora de *diseño guiado por responsabiidades* (RDD) — cómo asignamos responsabilidades a objetos que colaboran

El **diseño guiado por responsabilidades** (RDD) cubre tres aspectos

Responsabilidades:

- los objetos tienen responsabilidades —una abstracción de lo que hacen

Roles:

- las responsabilidades representan el comportamiento o las obligaciones de un objeto según su rol en el sistema

Colaboraciones:

- para cumplir sus responsabilidades, muchas veces un objeto debe colaborar con otros objetos

RDD nos lleva a mirar un diseño OO como una *comunidad de objetos colaboradores responsables*

En RDD, hay dos tipos de responsabilidades: hacer y saber

Responsabilidades de **hacer** de un objeto:

- hacer algo uno mismo —crear otro objeto, hacer un cálculo
- iniciar las acciones de otros objetos
- controlar y coordinar las actividades en otros objetos

Responsabilidades de **saber** de un objeto:

- saber acerca de sus datos internos propios
- saber acerca de los objetos con los que está relacionado
- saber acerca de lo que puede deducir o calcular

Las responsabilidades son asignadas a las clases durante el diseño

P.ej.

- una *Venta* es responsable de crear una *LineadeVenta*
- una *Venta* es responsable de saber su total

La traducción de responsabilidades a clases y métodos depende de la granularidad de la responsabilidad:

- “proporcionar acceso a una base de datos” puede requerir muchísimas clases y métodos
- “crear una orden” involucra sólo un método en una clase

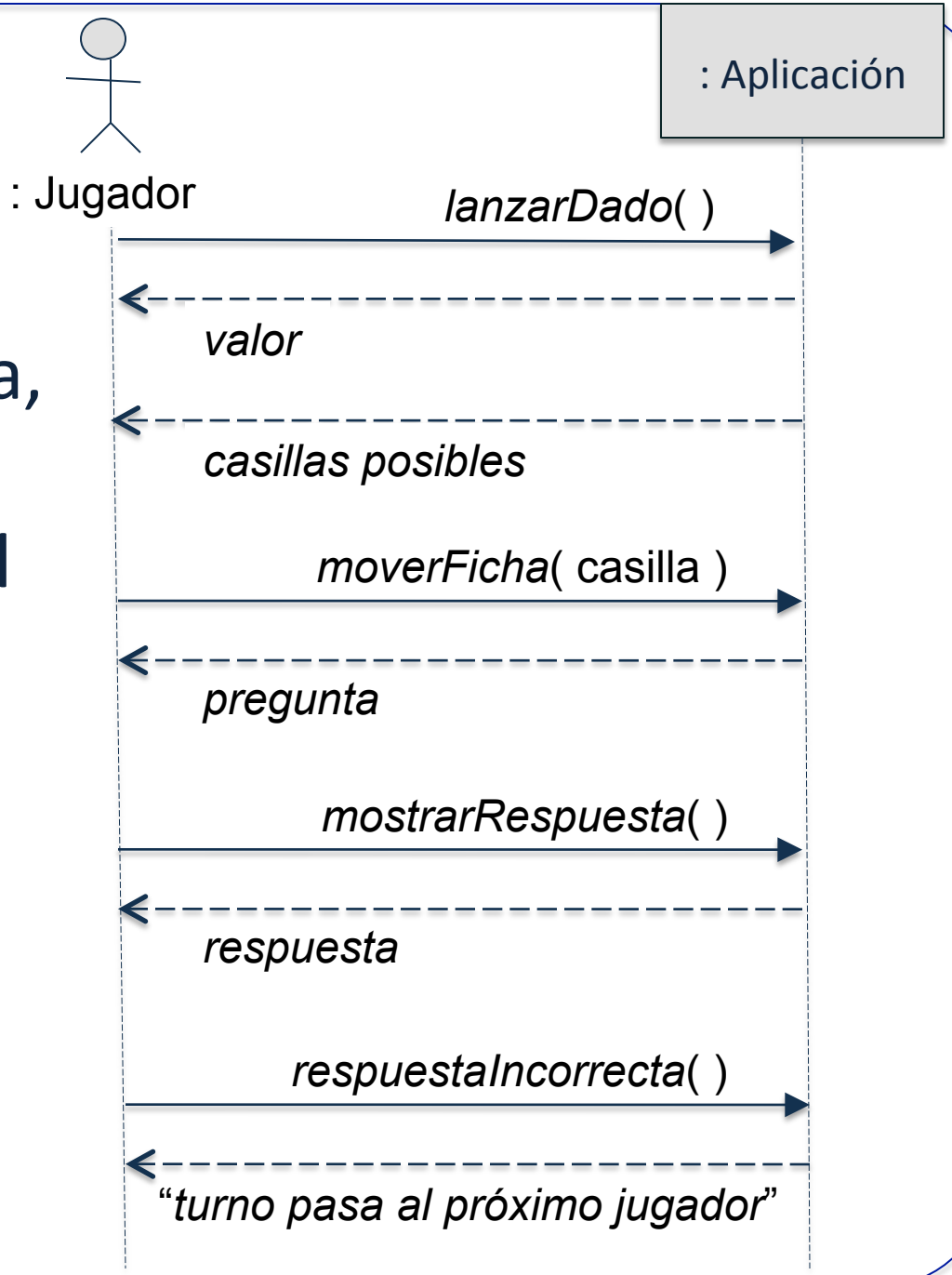
Una responsabilidad no es lo mismo que un método —es una abstracción:

- los métodos permiten cumplir las responsabilidades

El juego “The Trivium”: Caso de uso *Jugar*

1. El Jugador que tiene el turno lanza el dado
2. El Juego muestra el resultado del lanzamiento del dado
3. El Juego muestra las casillas a las que el Jugador puede mover su ficha
4. El Jugador mueve su ficha a una de esas casillas
5. El Juego hace una pregunta correspondiente al tema de la casilla
6. El Jugador (responde y) pide al Juego que muestre la respuesta correcta
7. El Juego muestra la respuesta correcta
8. El Jugador indica que su respuesta ha sido incorrecta
9. El Juego pasa el turno a otro jugador

Diagrama de secuencia, a nivel de la aplicación (o del sistema), para el caso de uso *Jugar*



¿Cómo diseñamos la aplicación para que responda al caso de uso?

Podríamos diseñar una aplicación monolítica, compuesta por una sola clase, que pueda hacerlo todo:

- "lanzar el dado" y mostrar el resultado
- determinar las nuevas casillas posibles a partir de la casilla actual y el valor del dado
- mover la ficha del jugador a una nueva casilla
- elegir una nueva pregunta, y respuesta, según el tema de la nueva casilla
- mostrar la pregunta
- …

… pero

El principio de diseño ***Alta Cohesión***

Problema: ¿Cómo mantener las clases focalizadas, inteligibles y manejables?

Solución: Asignemos las responsabilidades de modo que la cohesión de las clases permanezca alta

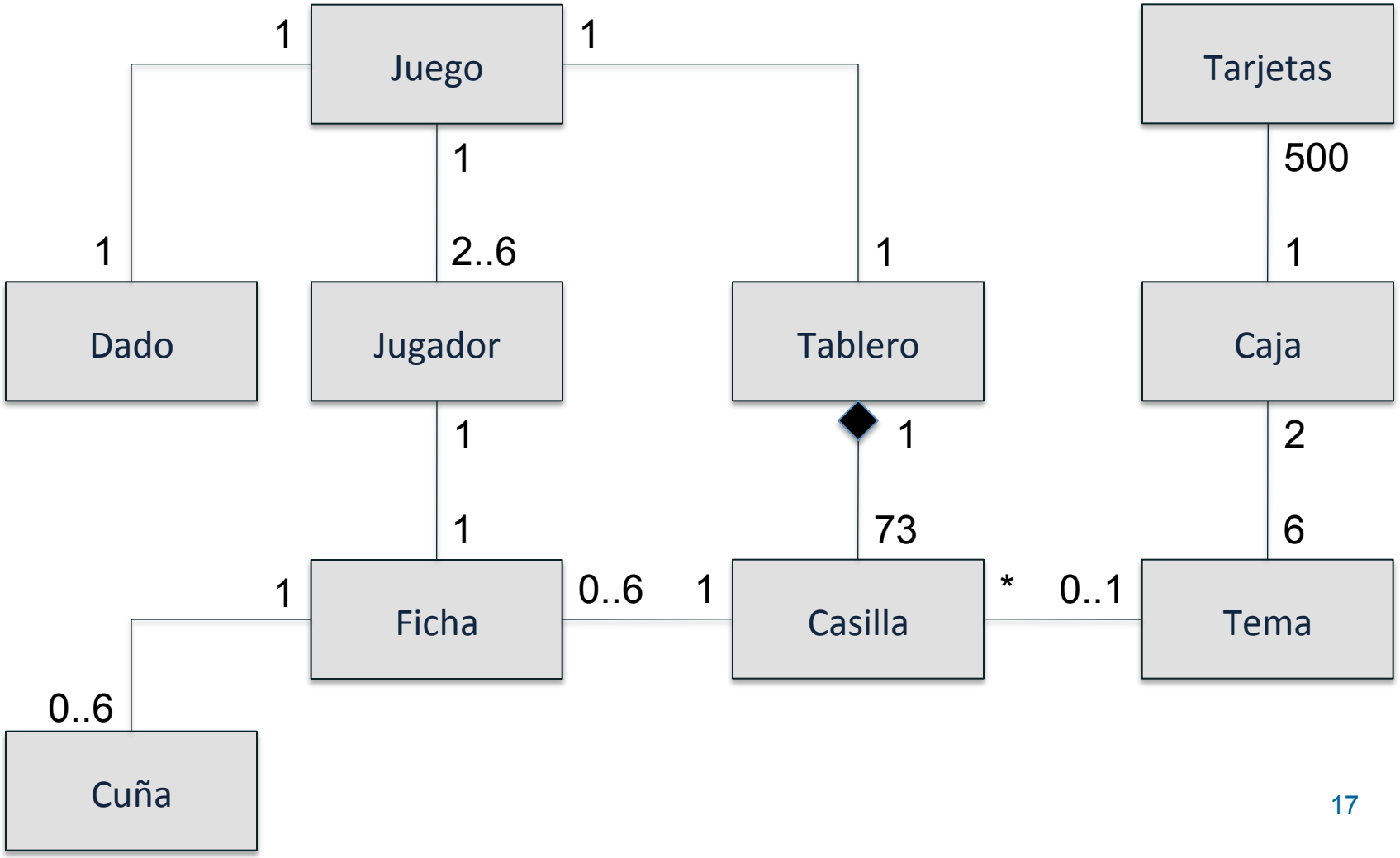
Cohesión es una medida de cuán fuertemente relacionadas y focalizadas son las responsabilidades de una clase, un módulo, un subsistema, etc.

Usemos este principio para evaluar alternativas

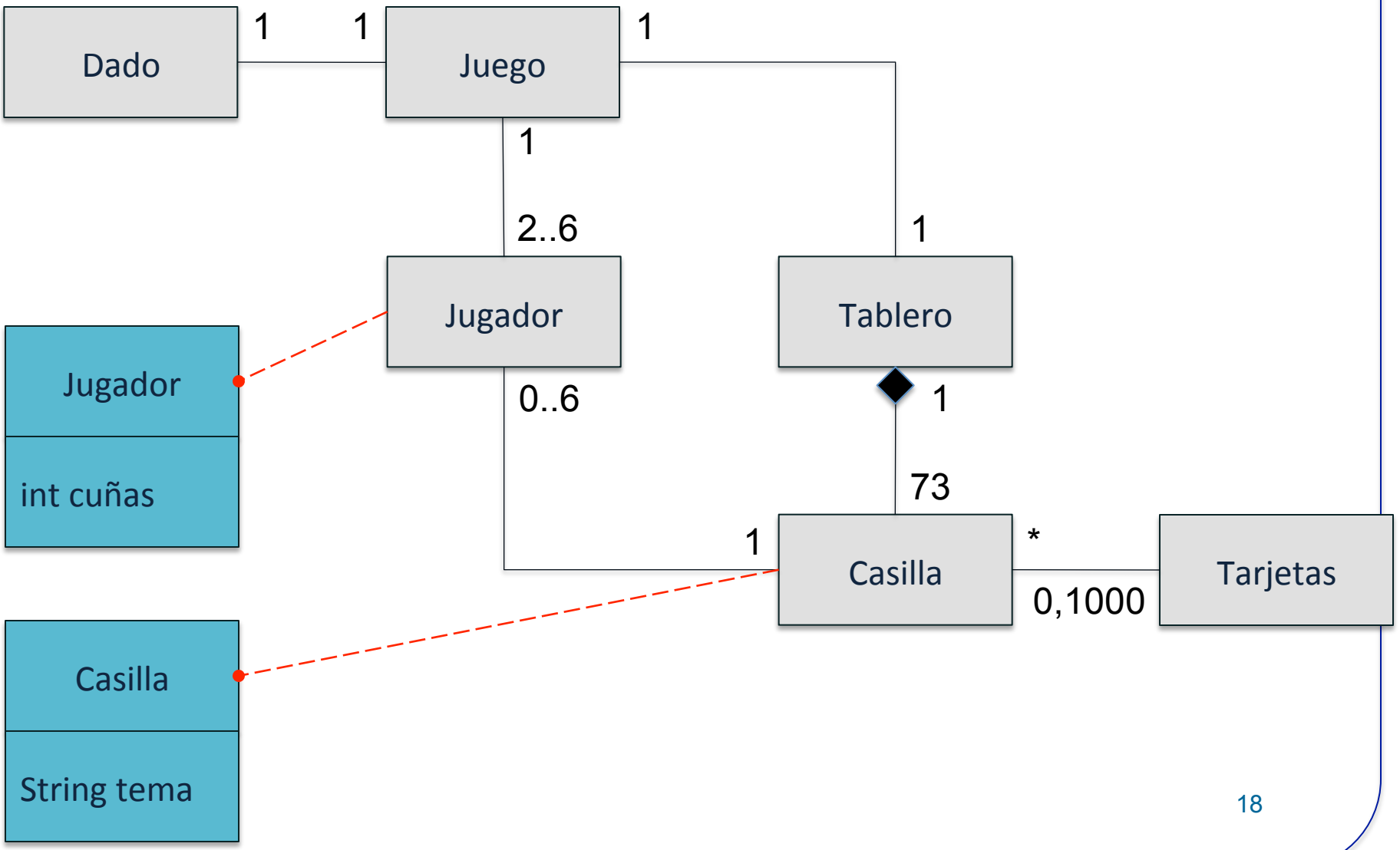
Las clases conceptuales del juego “The Trivium”

- Juego
- Tema
- Tarjeta de preguntas y respuestas
- Caja de tarjetas
- Jugador
- Dado
- Ficha
- Cuña
- Tablero
- Casilla

Modelo de dominio del juego “The Trivium”: resultado del análisis orientado a objetos



Modelo de dominio del juego “The Trivium”, después de convertir clases en atributos



Pasos 1, 2 y 3 del caso de uso *Jugar*

El :Juego recibe (desde el exterior) el mensaje de *lanzar el dado*; ¿a quién se lo pasa?

Una vez que esté el nuevo valor del dado, el :Juego tiene que mostrarlo,

… y también tiene que mostrar las casillas a las que el jugador puede moverse:

- estas dependen del valor del dado y de la casilla que el jugador ocupa actualmente
- ¿de dónde obtenemos la casilla que el jugador ocupa actualmente?
- ¿quién determina las casillas a las que el jugador puede moverse?

El principio de diseño ***Experto en Información***

Problema: ¿Cuál es un principio general de asignación de responsabilidades a objetos?

Solución: Asignemos una responsabilidad a la clase que tiene la información necesaria para cumplirla.

¿A quién enviamos el mensaje de *lanzar el dado*?

(La pregunta quiere decir, ¿en qué clase ponemos el método que simula el lanzamiento del dado?)

De acuerdo con el principio *Experto en Información* …

… a la clase que tiene la información necesaria para lanzar el dado

En este caso, Dado

- … aunque podría ser Juego (o cualquiera otra), porque "lanzar el dado" no requiere ninguna información especial, es decir, no depende del estado de ningún objeto

Según el diagrama de clases, el :Juego puede pedirle directamente al :Dado que "se lance"

¿De dónde obtenemos la casilla que el jugador ocupa actualmente?

(La pregunta quiere decir, ¿en qué clase ponemos el método que devuelve la casilla ocupada actualmente por el jugador?)

De acuerdo con el principio *Experto en Información* …

el :Jugador sabe en qué casilla está

Según el diagrama de clases, el :Juego puede pedirle directamente al :Jugador su casilla

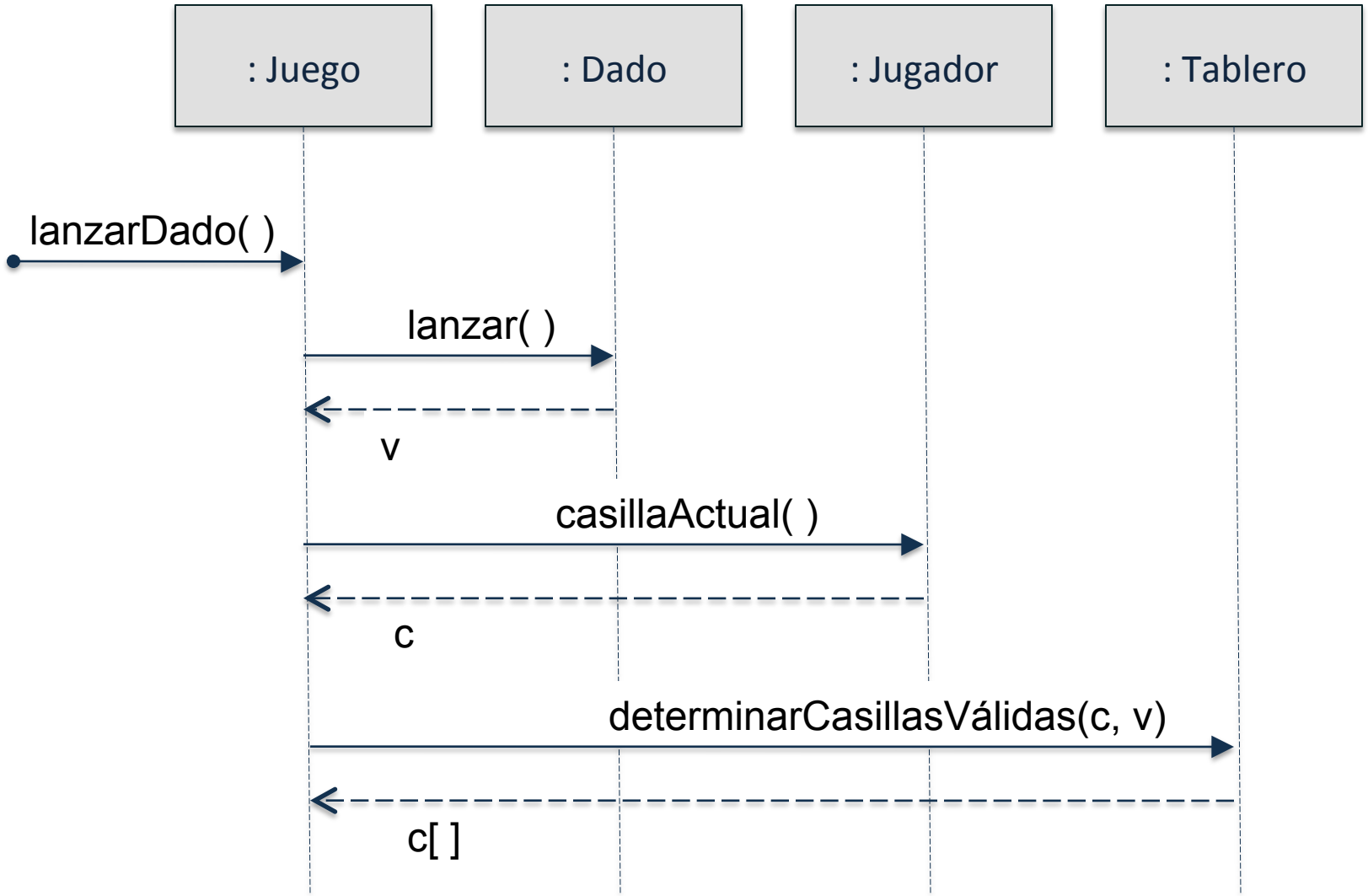
¿Quién determina las casillas a las que el jugador puede moverse?

(La pregunta quiere decir, ¿en qué clase ponemos el método que, a partir de la casilla actual y del valor del dado, determina las nuevas casillas posibles?)

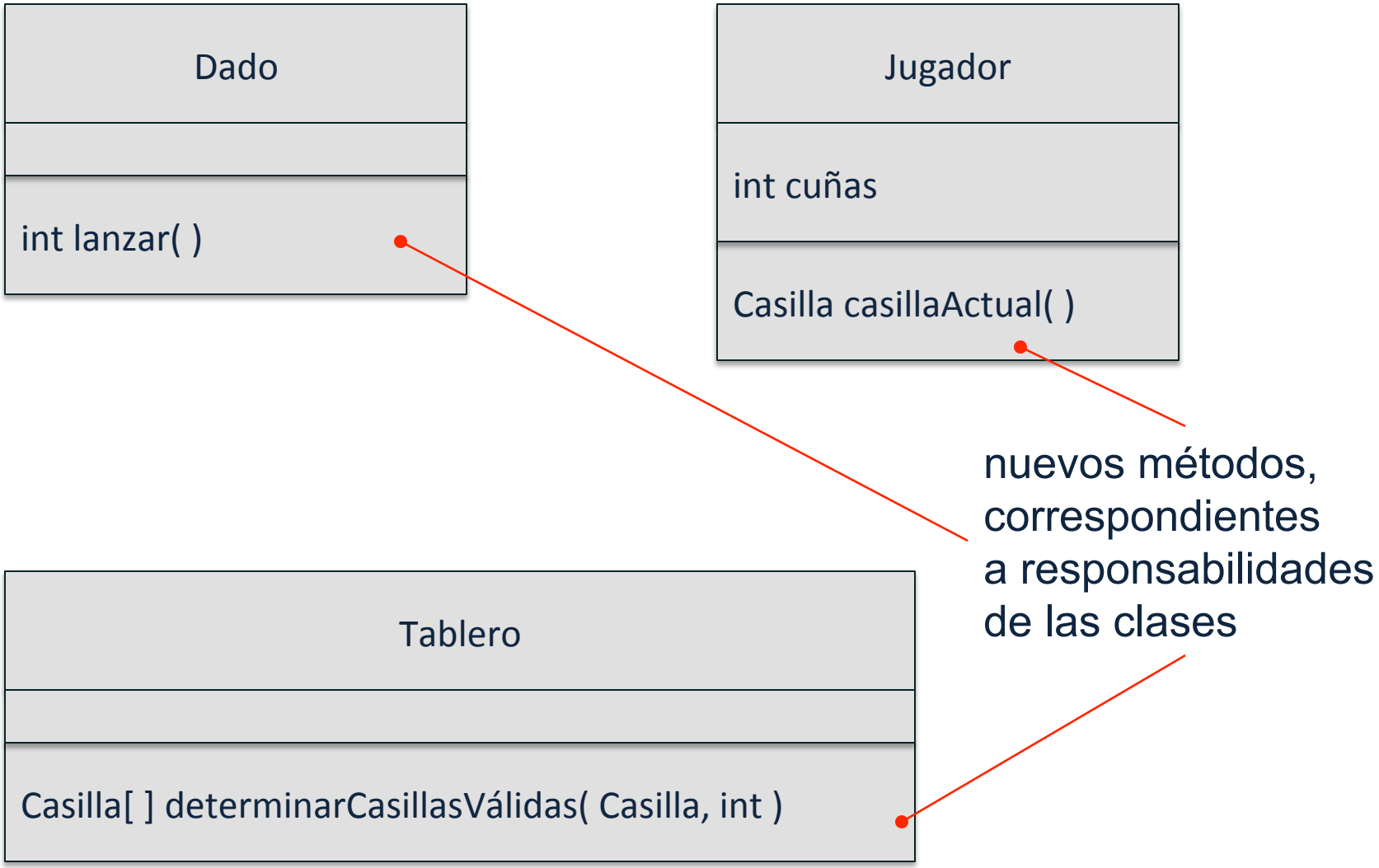
El :Tablero puede determinar las casillas válidas a partir de la casilla actual, para cualquier jugador.

Según el diagrama de clases, el :Juego puede pedirle directamente al :Tablero que determine las nuevas casillas válidas

Diagrama de secuencia para los pasos 1 al 3 del caso de uso *Jugar*



El diagrama de secuencia anterior tiene implicaciones para algunas clases



Pasos 4 y 5 del caso de uso *Jugar*

El :Juego recibe (desde el exterior) la nueva posición de la ficha del jugador; ¿a quién se la pasa para identificar a la casilla propiamente tal?

Una vez que la nueva casilla está identificada, el :Juego tiene que hacerle una pregunta al jugador; ¿cómo la obtiene?

Finalmente, y aunque no aparece en el caso de uso *Jugar*, no todas las casillas se comportan igual

¿A quién enviamos el mensaje de *obtener casilla*?

La distribución de las casillas en el tablero es una propiedad (atributo) del :Tablero

Así, según el principio *Experto en Información*,

… el :Tablero es el objeto más apropiado para identificar la casilla elegida por el jugador

¿Cómo obtiene el :Juego la pregunta que tiene que hacerle al jugador?

Las casillas tienen acceso a las tarjetas con las preguntas;

… pero :Juego no tiene acceso (directo) a las casillas

Tenemos dos posibilidades

Una posibilidad:

- conectar Juego con Casilla en el diagrama de clases,
- … luego, hacer que el :Tablero devuelva la :Casilla al :Juego,
- … finalmente, hacer que el :Juego le pida la pregunta a la :Casilla

Otra posibilidad:

- hacer que el :Tablero devuelva la casilla al :Juego,
- … luego, hacer que el :Juego informe al :Jugador de su nueva :Casilla,
- … finalmente, hacer que el :Jugador le pida la pregunta a la :Casilla

El principio de diseño ***Bajo Acoplamiento***

Problema: ¿Cómo conseguir una baja dependencia, un bajo impacto al cambio y una mayor reusabilidad?

Es decir, ¿cómo favorecemos la *modularidad*?

Solución: Asignemos una responsabilidad de manera que el acoplamiento entre clases permanezca bajo.

Usemos este principio para evaluar alternativas (similarmente a *Alta Cohesión*).

Las alternativas anteriores tienen distinto efecto sobre el acoplamiento entre clases

Primera posibilidad:

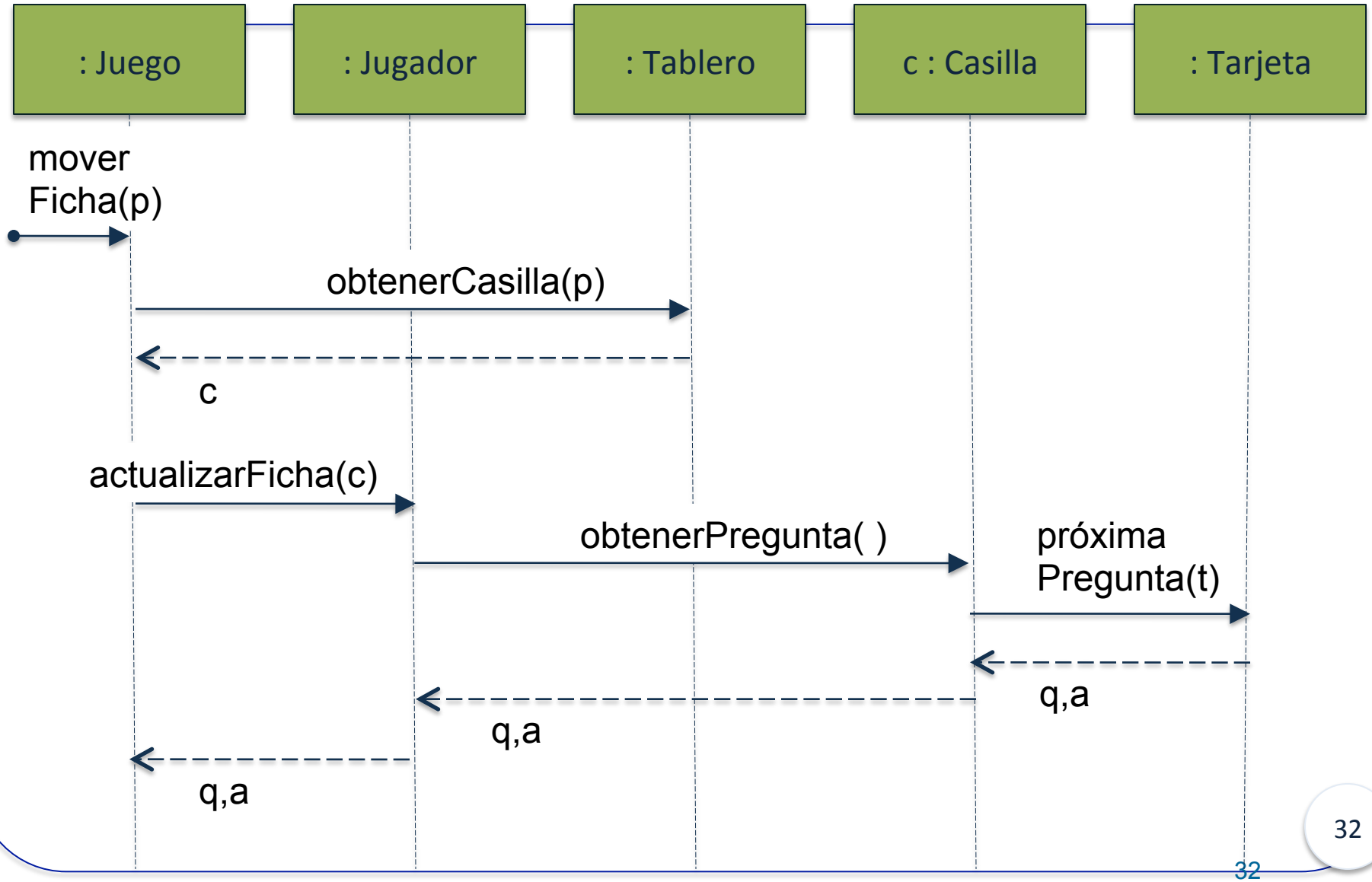
- Aumenta el acoplamiento entre las clases, al conectar Juego con Casilla

Segunda posibilidad:

- No aumenta el acoplamiento, ya que no requiere nuevas conexiones entre clases

En consecuencia, según el principio *Bajo Acoplamiento*, preferimos la segunda alternativa.

Diagrama de secuencia para los pasos 4 y 5 del caso de uso *Jugar*



32

El diagrama de secuencia anterior tiene implicaciones para algunas clases