

Diseño de Software

Introducción

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

Diseño orientado a objetos

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

- ... **clases de software**,
- ... sus responsabilidades,
- ... y cómo colaboran entre ellas para satisfacer los requisitos

En el proceso de diseño, hay involucrados principios y temas de discusión profundos

Decidir qué métodos van dónde y cómo deben interactuar las clases tiene consecuencias

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

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

Los ejemplos ayudan; la práctica ayuda

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 o relatos de usuario más importantes

Hay artefactos:

- los casos de uso y relatos de usuario definen el comportamiento visible que las clases de software deben proporcionar —las clases son diseñadas para implementar los casos de uso y los relatos

El diseño guiado por responsabilidades (RDD) cubre tres aspectos

Responsabilidades:

- las clases tienen responsabilidades —una abstracción de lo que hacen

Roles:

- las responsabilidades representan el comportamiento o las obligaciones de una clase según su rol en el sistema

Colaboraciones:

- para cumplir sus responsabilidades, muchas veces una clase debe colaborar con otras

RDD nos lleva a mirar un diseño como una *comunidad de objetos que colaboran y tienen responsabilidades*

En RDD, hay dos tipos de responsabilidades:
hacer ...

Responsabilidades de **hacer** de un objeto:

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

En RDD, hay dos tipos de responsabilidades: ... y **saber**

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 los objetos durante el diseño

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
- “elegir estado inicial” 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

Ejemplo de diseño: Una aplicación que permita jugar el juego “The Trivium”

“The Trivium” se juega sobre un tablero de 73 casillas

Los jugadores toman turnos para jugar y cada uno tiene una ficha que señala su posición (la casilla que ocupa) en el tablero

El jugador que tiene el turno lanza un dado

Dependiendo del valor obtenido, el jugador decide a cuál casilla del tablero quiere mover su ficha

Las distintas casillas están asociadas a distintos temas y cuando el jugador mueve su ficha a una casilla, se le hace una pregunta sobre ese tema

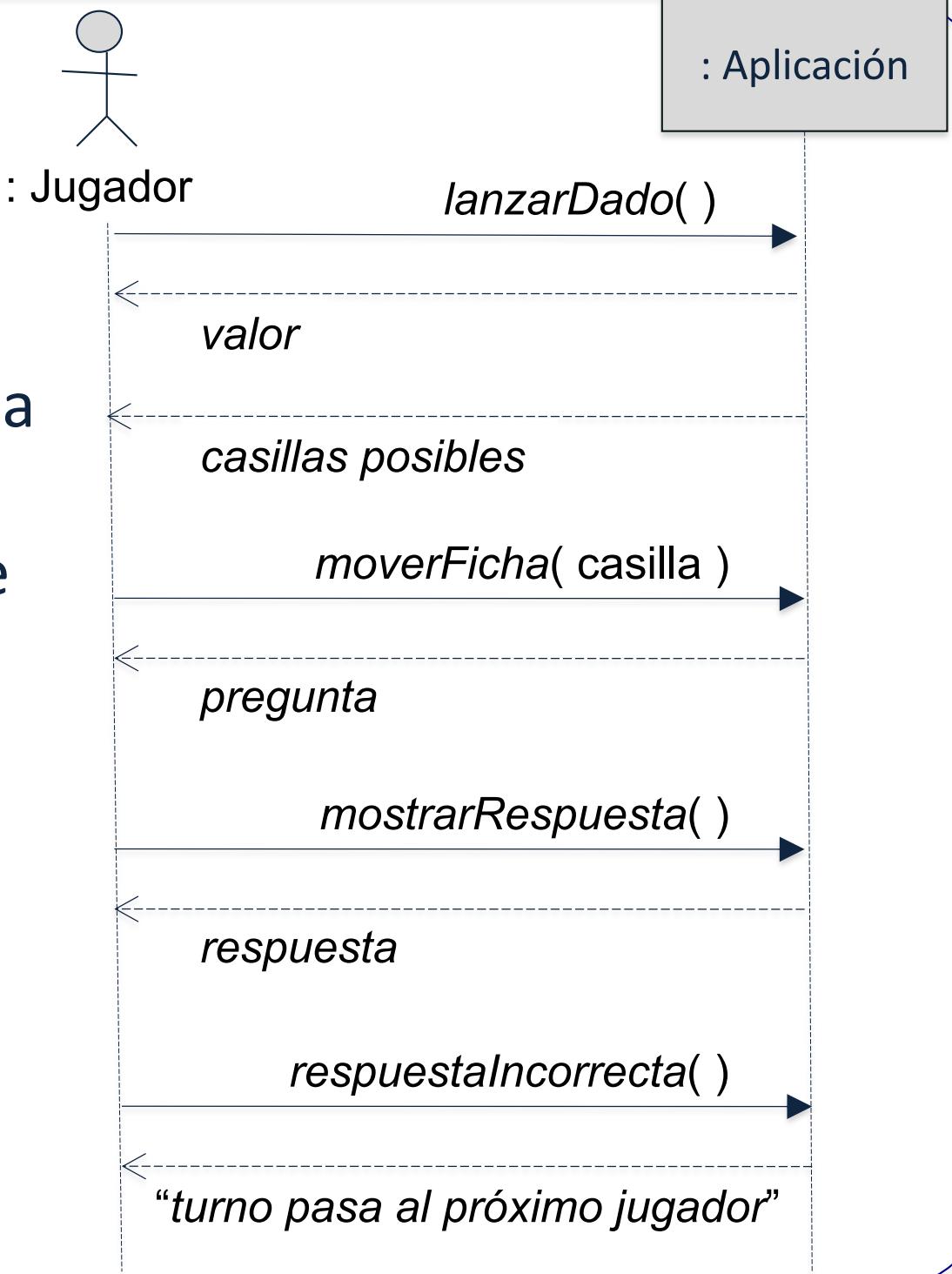
Si el jugador contesta correctamente, sigue jugando; de lo contrario, el turno pasa a otro jugador



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 luego) 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 sistema), para el caso de uso *Jugar*



¿Cómo diseñamos la aplicación (por dentro) 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 (o módulos) permanezca alta

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

Usamos este principio para evaluar alternativas

Las clases conceptuales del juego “The Trivium”

Juego

Tema

Tarjeta de preguntas y
respuestas

Caja de tarjetas

Jugador

Dado

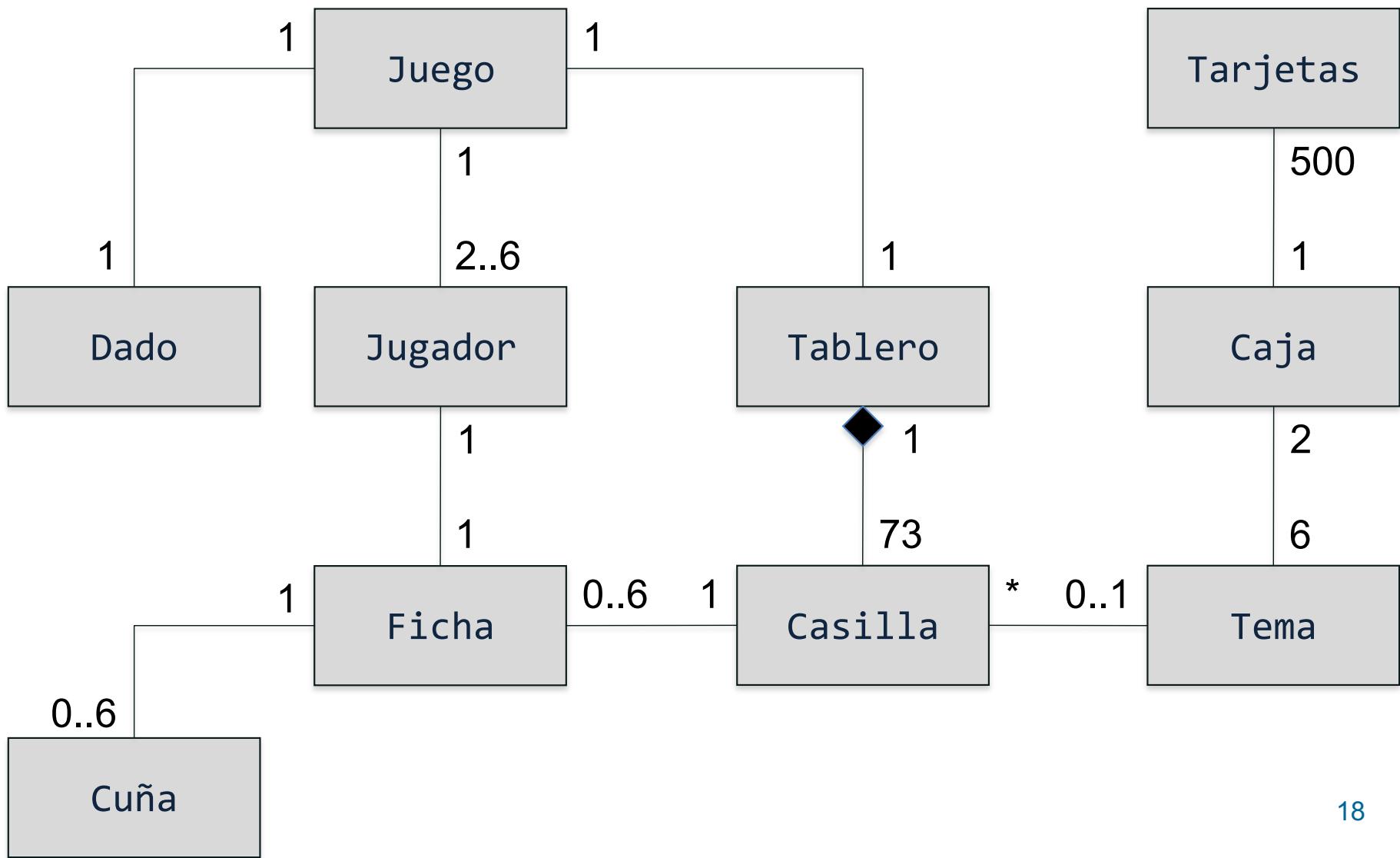
Ficha

Cuña

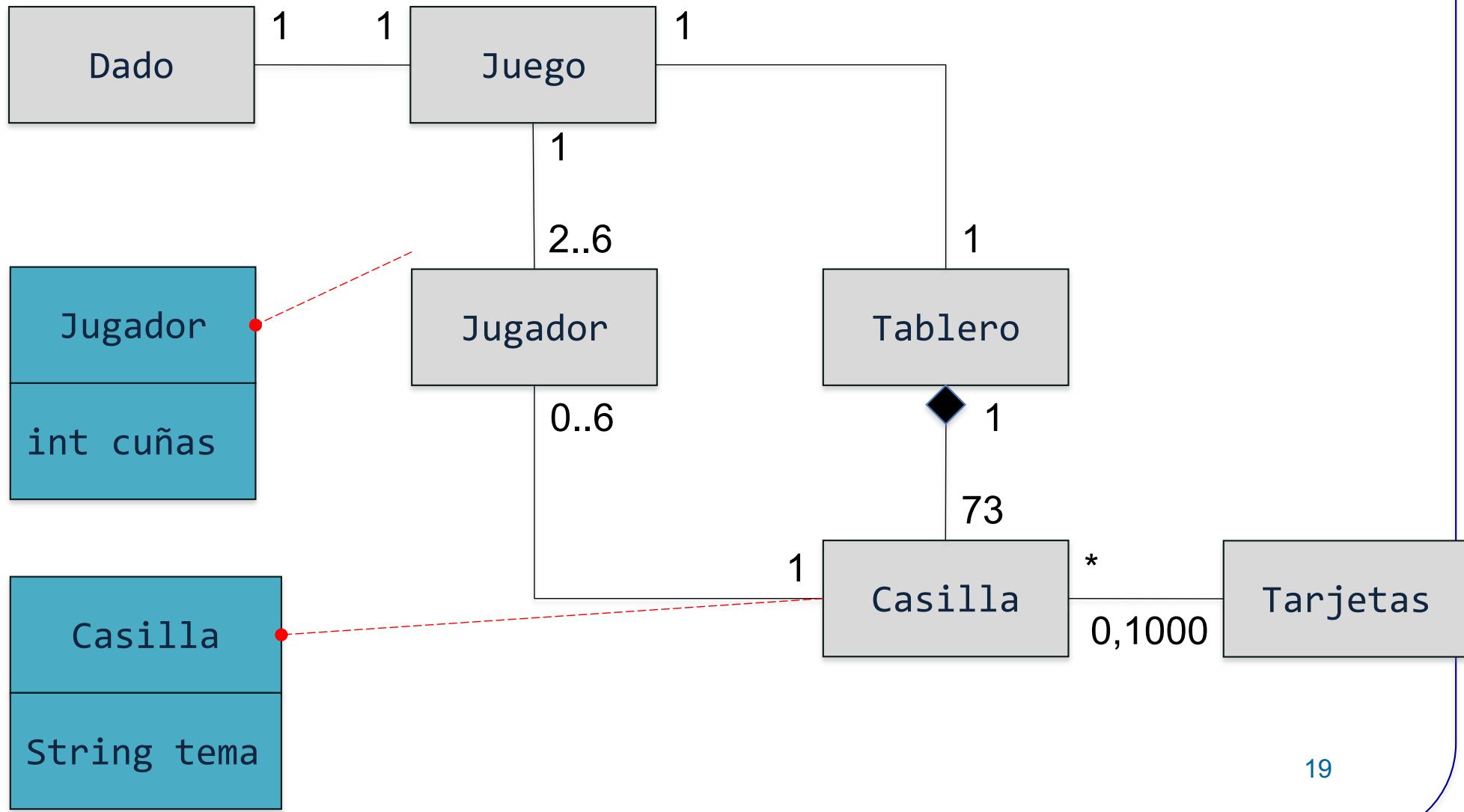
Tablero

Casilla

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



Modelo de dominio del juego “The Trivium”, después de convertir algunas 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 clases?

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 (una responsabilidad de *hacer*)

En este caso, Dado

- ... aunque podría ser Juego (o cualquier otra), porque “lanzar el dado” no requiere ninguna información especial, es decir, no depende del estado de ningún objeto

Según el modelo de dominio, 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 modelo de dominio, el :Juego puede pedirle directamente al :Jugador su casilla (una responsabilidad de *saber*)

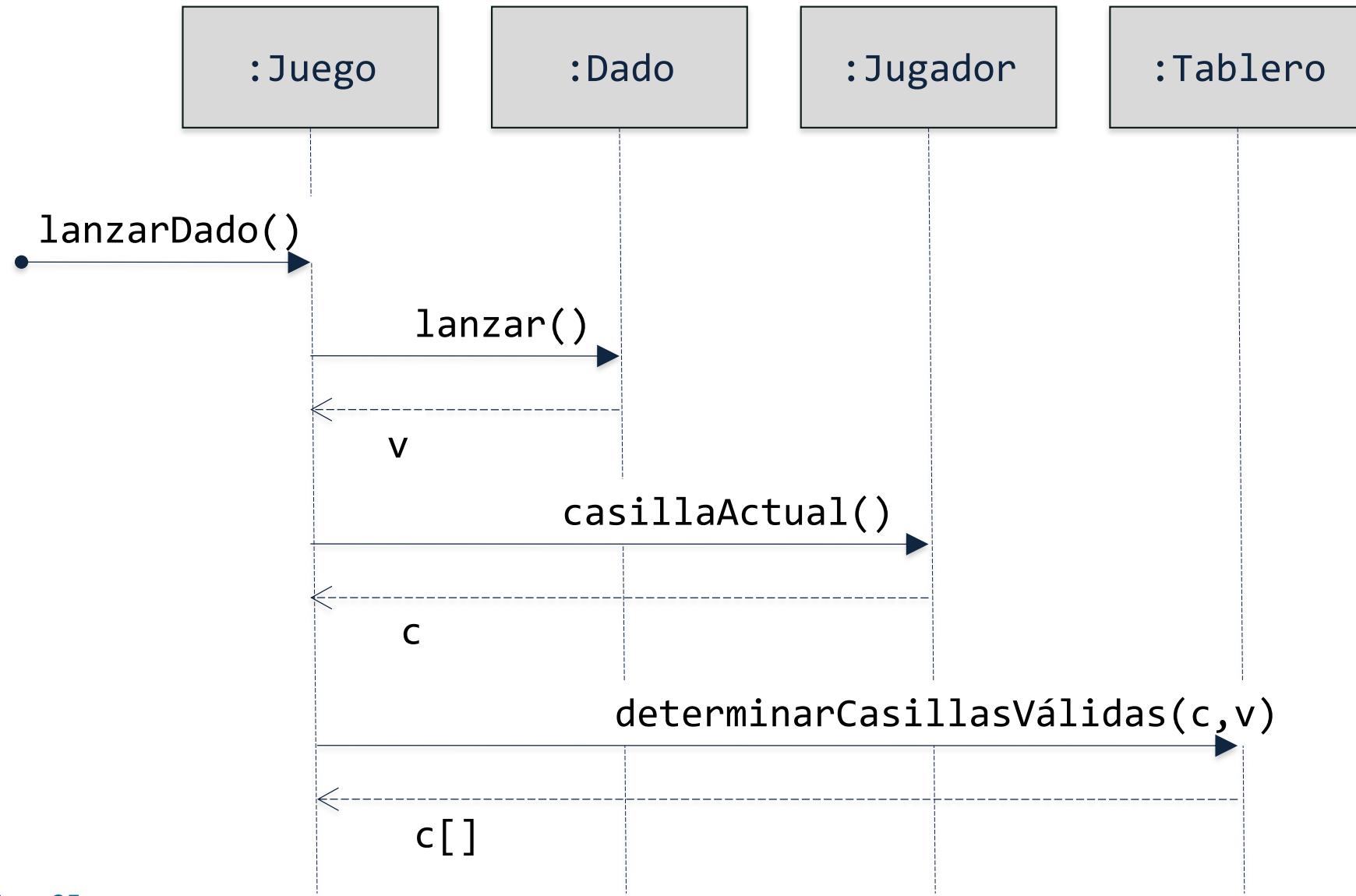
¿Quién determina las casillas a las que el jugador puede moverse?

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

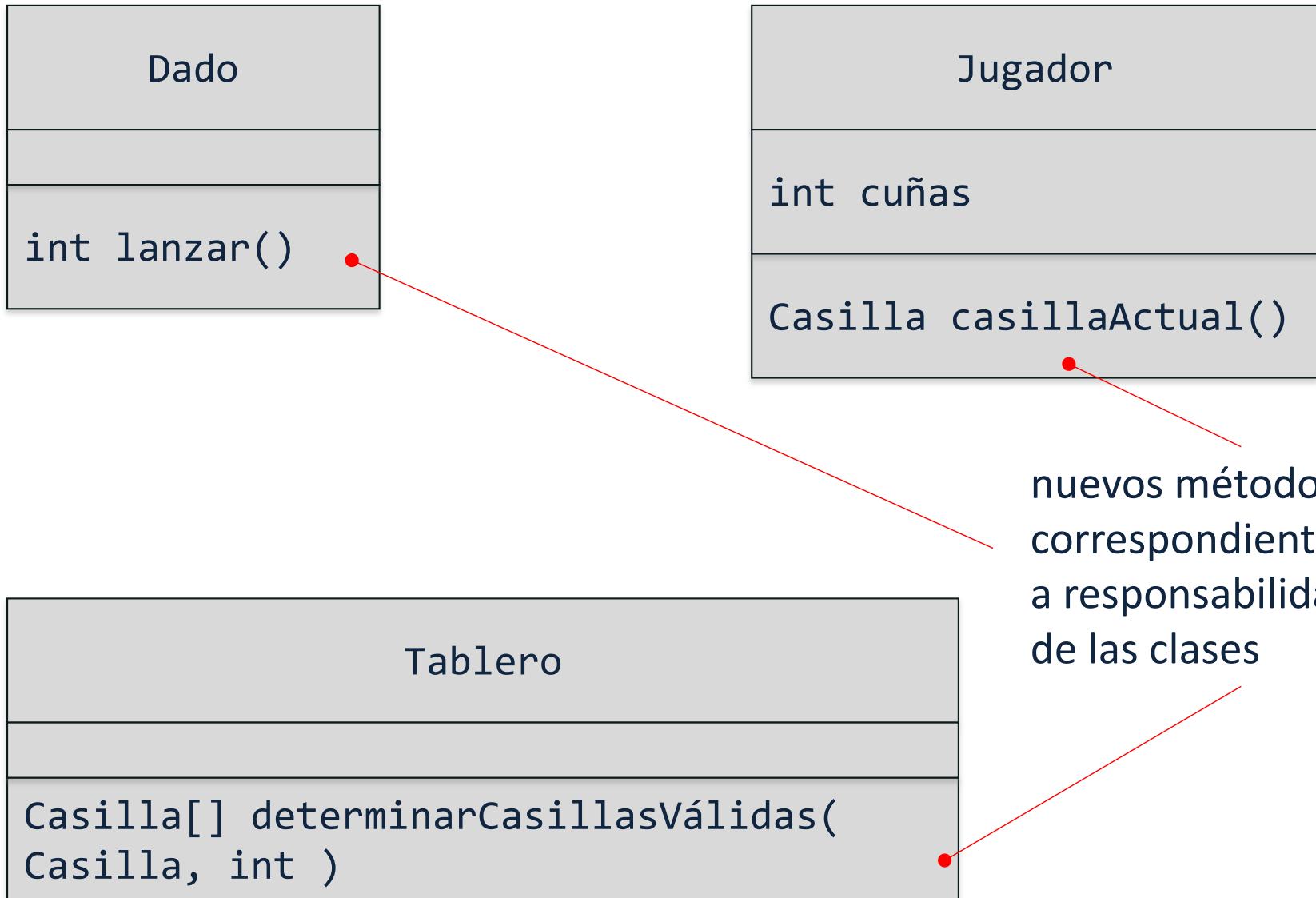
El :Tablero puede determinar las casillas válidas a partir de la casilla actual, para cualquier jugador

Según el modelo de dominio, el :Juego puede pedirle directamente al :Tablero que determine las nuevas casillas válidas (una responsabilidad de *saber*)

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 módulo más apropiado para identificar la casilla elegida por el jugador (una responsabilidad de *saber*)

¿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 modelo de dominio,
- ... 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:

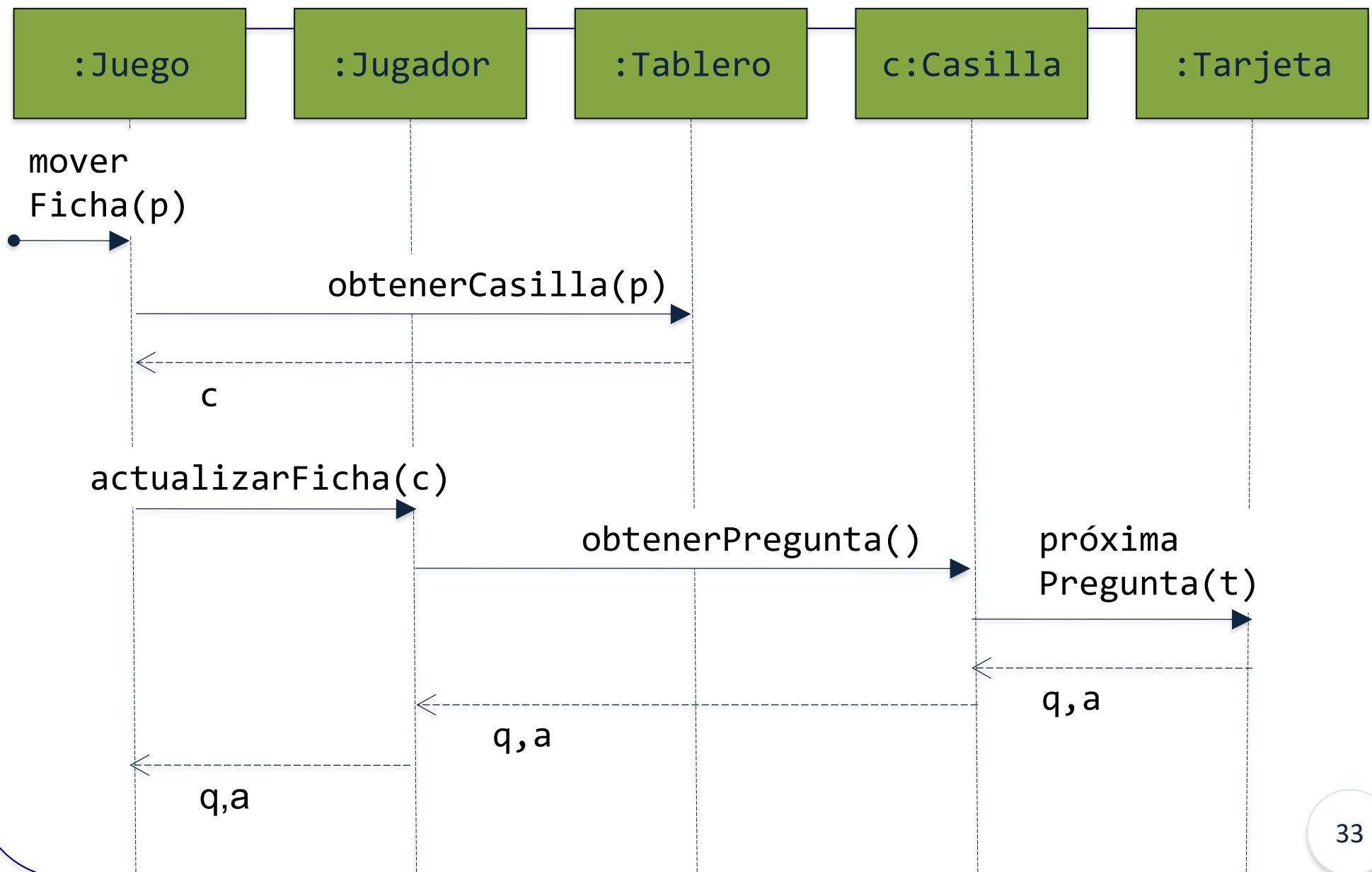
- aumenta el acoplamiento entre las clases, al conectar Juego con Casilla

Segunda posibilidad:

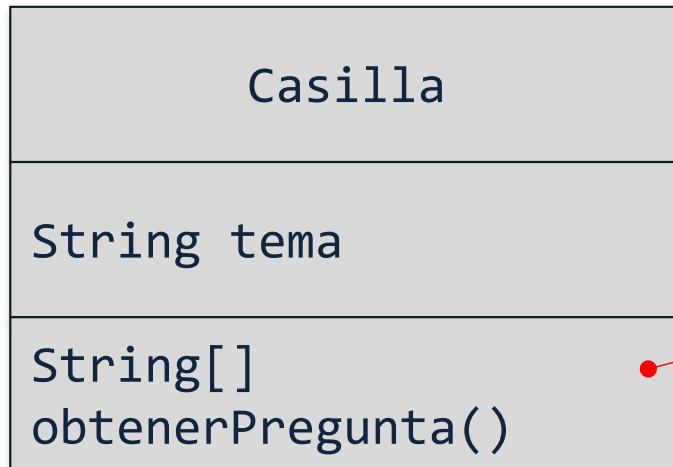
- no aumenta el acoplamiento, ya que no requiere nuevas conexiones entre clases

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

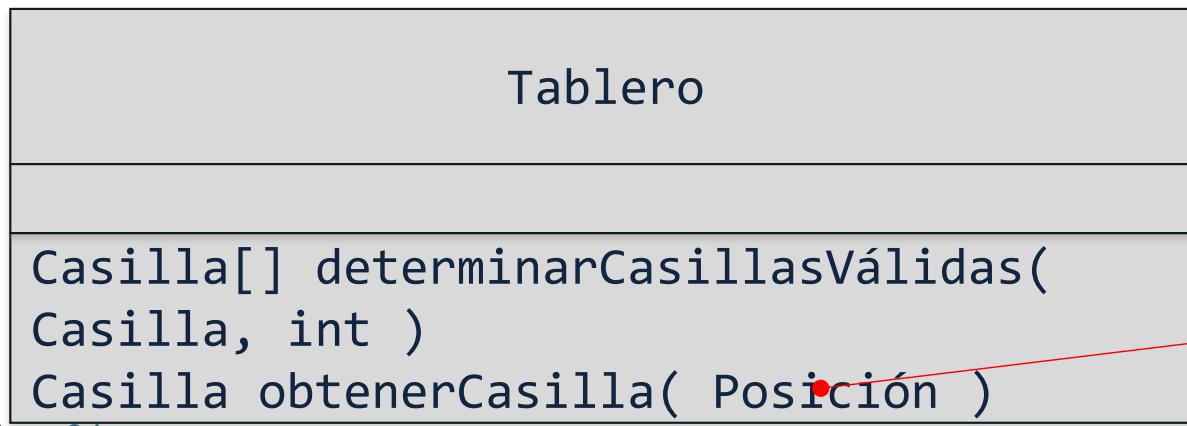
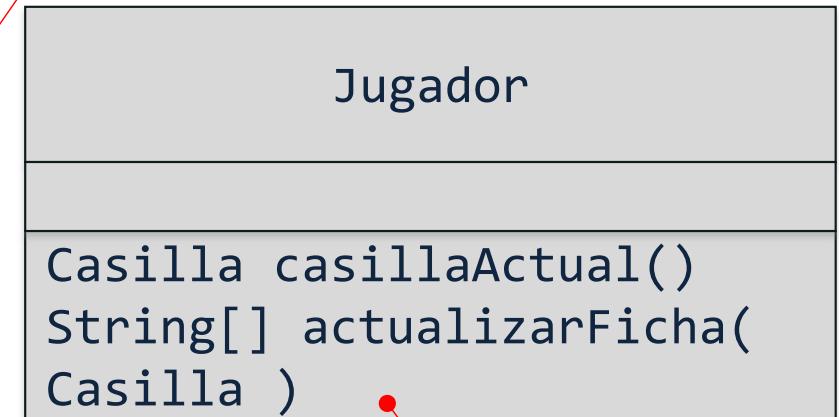
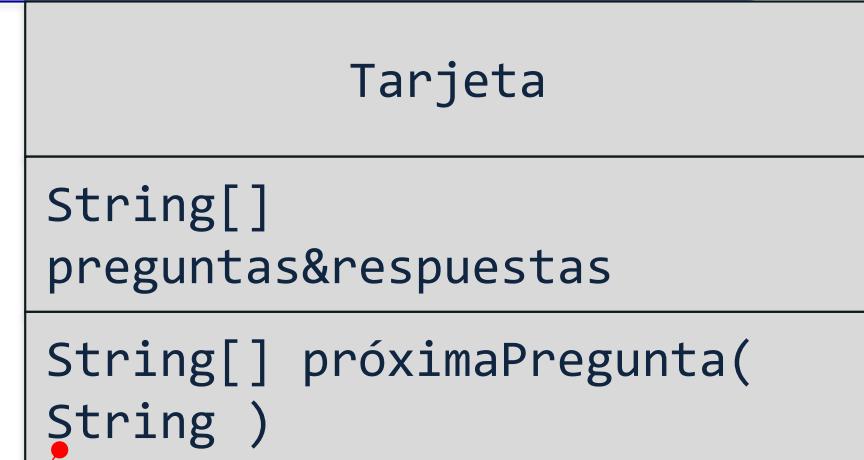
Diagrama de secuencia para los pasos 4 y 5 del caso de uso *Jugar*



El diagrama de secuencia anterior tiene implicaciones para algunas clases



nuevos
métodos



nuevos
métodos

En resumen, ¿qué hicimos?

Primero, a partir de la descripción del juego, construimos
... un **modelo de dominio**, esto es, un diagrama de clases UML que
representa los elementos o conceptos principales del juego y sus
relaciones principales

Segundo, a partir de un caso de uso y del modelo de dominio, fuimos construyendo (más o menos en paralelo)

... un diagrama de clases UML que representa **las clases principales de la aplicación**, incluyendo algunos de sus métodos,

... y diagramas de secuencia UML que representan **las interacciones entre instancias (objetos) de estas clases**

Tercero, en el proceso de diseño, asignamos responsabilidades a las clases de software aplicando tres principios de diseño:

Procuremos que nuestras clases sean cohesivas:

- una clase cohesiva hace una cosa muy bien y no trata de hacer o ser algo más

Procuremos que nuestras clases tengan poco acoplamiento entre ellas:

- los cambios a una clase no requieren que uno haga un montón de cambios a otras clases

Asignemos una responsabilidad a la clase que tiene la información necesaria para poder cumplirla

¿Qué es software de gran calidad?

El programador *customer-friendly*:

- “siempre hace lo que el cliente quiere que haga; incluso si el cliente piensa en formas nuevas de usarlo, no se cae ni produce resultados inesperados”

El programador orientado a objetos:

- “código que es orientado a objetos; no hay código duplicado y cada objeto controla su propio comportamiento; es fácil de extender, porque el diseño es sólido y flexible”

El programador diseñador:

- “cuando usamos principios y patrones de diseño probados; los objetos están débilmente acoplados y el código está abierto para extensiones pero cerrado para cambios; el código es más reusable”

Software de gran calidad en tres pasos

1. Asegurémonos de que el software hace lo que el cliente quiere que haga
2. Apliquemos principios básicos de orientación a objetos para agregar flexibilidad
3. Esforcémonos por hacer un diseño mantenible y reusable, aplicando patrones de diseño