2-5-2020

**PROYECTOS INFORMÁTICOS II**

**MEMORIA DEL PROYECTO** EJECUCIÓN Y CONTROL  
 **MONOPOLY**

DANIEL – AYOUB – RAÚL

**ÍNDICE**

[1. Control de documentación 3](#_Toc39936773)

[2. Historial de modificaciones 3](#_Toc39936774)

[3. Fase de ejecución del proyecto 5](#_Toc39936775)

[3.1 Definición del modelo de desarrollo 5](#_Toc39936776)

[3.2 Definición del entorno de trabajo 6](#_Toc39936777)

[3.2.1 Guía de estilo 6](#_Toc39936778)

[3.2.1.1 Identificadores 7](#_Toc39936779)

[3.2.1.2 Archivos fuente 7](#_Toc39936780)

[3.2.1.3 Clases 7](#_Toc39936781)

[3.2.1.4 Métodos 8](#_Toc39936782)

[3.2.1.5 Variables y Constantes 8](#_Toc39936783)

[3.2.1.6 Alineación y espacios en blanco 8](#_Toc39936784)

[3.3 Análisis de requisitos 9](#_Toc39936785)

[3.4 Control de bugs 14](#_Toc39936786)

[3.5 Diagrama de clases 15](#_Toc39936787)

[3.6 Diagrama de componentes 16](#_Toc39936788)

[3.7 Implementación del UML 16](#_Toc39936789)

[3.7.1 Clase Jugador 17](#_Toc39936790)

[3.7.2 Clase Humano 18](#_Toc39936791)

[3.7.3 Clase Dado 18](#_Toc39936792)

[3.7.4 Clase DatosPartida 19](#_Toc39936793)

[3.7.5 Clase CComunidad 20](#_Toc39936794)

[3.7.6 Clase CPublica 21](#_Toc39936795)

[3.7.7 Clase CSuerte 22](#_Toc39936796)

[3.7.8 Clase Calle 23](#_Toc39936797)

[3.7.9 Clase Carcel 25](#_Toc39936798)

[3.7.10 Clase Casilla 26](#_Toc39936799)

[3.7.11 Interfaz Dibujable 27](#_Toc39936800)

[3.7.12 Clase Encarcelar 28](#_Toc39936801)

[3.7.13 Clase abstracta Especial 28](#_Toc39936802)

[3.7.14 Clase Estación 29](#_Toc39936803)

[3.7.15 Clase Ficha 30](#_Toc39936804)

[3.7.16 Clase GUICasilla 31](#_Toc39936805)

[3.7.17 Clase Impuesto 33](#_Toc39936806)

[3.7.18 Clase Lujo 34](#_Toc39936807)

[3.7.19 Clase Parking 35](#_Toc39936808)

[3.7.20 Clase ParrillaFichas 36](#_Toc39936809)

[3.7.21 Clase abstracta Propiedad 37](#_Toc39936810)

[3.7.22 Clase Salida 39](#_Toc39936811)

[3.7.23 Clase SobreCapital 39](#_Toc39936812)

[3.7.24 Clase App 40](#_Toc39936813)

[3.7.25 Clase ConfiguracionPartida 41](#_Toc39936814)

[3.7.26 Clase GUIPartida 42](#_Toc39936815)

[3.7.27 Clase PanelJugador 44](#_Toc39936816)

[3.8 Control de versiones 46](#_Toc39936817)

[4. Fase de control del proyecto 47](#_Toc39936818)

[4.1 Responsabilidades relacionadas con la calidad 48](#_Toc39936819)

[4.2 Procesos de calidad en el proyecto 49](#_Toc39936820)

[4.3 Pruebas de calidad del código fuente: EasyPmd 51](#_Toc39936821)

[4.4 Pruebas código fuente: JUnit 54](#_Toc39936822)

# Control de documentación

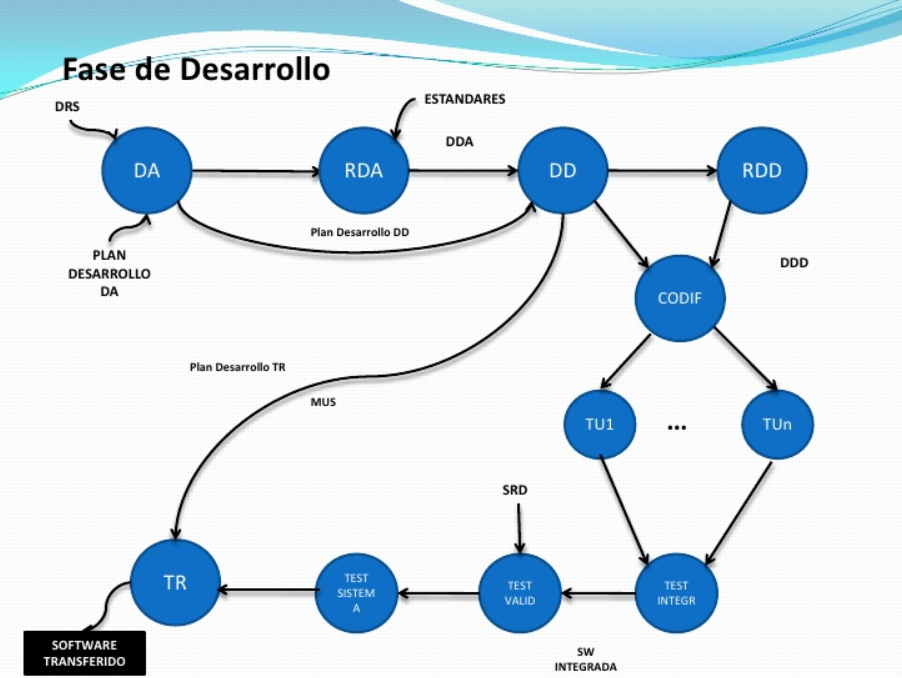
|  |  |  |
| --- | --- | --- |
| **Elaborado por:** | **Revisado por:** | **Aprobado por:** |
| Alumnos de la asignatura (desarrollo) | Alumnos de la asignatura (revisado) | Equipo de la Universidad |
| **Nombre:** | **Nombre:** | **Nombre:** |
| Daniel Ortega  Ayoub Dachour  Raúl García | Daniel Ortega  Ayoub Dachour  Raúl García | [Jordi José](https://opencampus.uols.org/user/view.php?id=1347&course=872) |
| **Fecha:** | **Fecha:** | **Fecha:** |
| 08/05/2020 |  |  |
| **Firma:** | **Firma:** | **Firma:** |
|  |  |  |

# Historial de modificaciones

|  |  |  |  |
| --- | --- | --- | --- |
| **VERSIÓN** | **FECHA DE VIGENCIA** | **MODIFICADO POR:** | **MODIFICACIÓN REALIZADA** |
| 1.0 | 27/03/2020 | Daniel Ortega | Creación de documento |
| 1.1 | 27/03/2020 | Daniel Ortega | -Recuperar casos de uso y requisitos  -Planificación detallada del proyecto |
| 1.2 | 08/04/2020 | Raúl García | - Integrar partes que faltan de la tarea anterior  - Incluir diagrama de clases |
| 1.3 | 09/04/2020 | Raúl García | - Repaso apartado 3.  - Añadir apartado 6 “Fase de control del proyecto” |
| 1.4 | 10/04/2020 | Raúl García | - Ajuste formato  - Añadir contenido apartado 5. Descripción de clases  - Añadir contenido apartado 6 |
| 1.5 | 11/04/2020 | Raúl García | - Añadir contenido apartado 6 |
| 1.6 | 11/04/2020 | Daniel Ortega | - Ajuste formato  - Adaptar fase control y calidad |
| 1.7 | 25/04/2020 | Daniel Ortega | -Se divide documento en DOS. Este queda únicamente para ejecución y control -Añadida columna test en requisitos y casos de uso para controlar si se ha testeado -Añadido apartado bugs y descripción clases humano y todas las de utilidades |
| 1.8 | 27/04/2020 | Ayoub Dahchour | * Añadir los sub-apartados de 3.7.8 hasta 3.7.23 y documentarlos. |
| 1.9 | 27/04/2020 | Raúl García | * Revisión de apartados 3.7.8 hasta 3.7.23 * Revisión ortográfica * Corregir formato de apartados 3.7.8 hasta 3.7.23 |
| 1.10 | 28/04/2020 | Raúl García | * Añadir y documentar los sub-apartados de 3.7.24 al 3.7.27 |
| 1.11 | 02/05/2020 | Raúl García | * Añadir apartado y sub-apartados de la guía de estilo * Actualizar tabla de bugs |
| 1.12 | 03/05/2020 | Raúl García | * Inclusión de ejemplos calidad de código |

# Fase de ejecución del proyecto

La fase de ejecución consiste en la coordinación de personas y otros recursos para llevar a cabo el plan. Es el lugar donde se realiza el trabajo real. El noventa por ciento o más de los esfuerzos del proyecto se gastan durante esta fase, y se completa cuando se cumple la meta del proyecto. La fase de ejecución consta de las siguientes acciones: desarrollo del código, creación de casos de prueba y el establecimiento de la documentación del usuario.

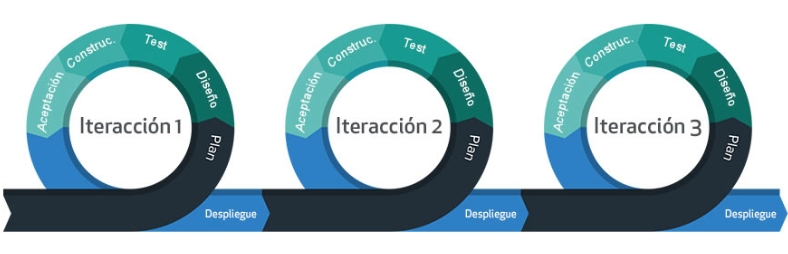


# Definición del modelo de desarrollo

El modelo de desarrollo será de forma ágil, es decir, por Sprint. Algunas de las ventajas de esta metodología son las siguientes:

* Se trabaja con mayor velocidad y eficiencia, ya que se trabaja en entregas parciales del producto.
* Ahorro de tiempo y costes, ya que al ser un desarrollo más eficiente y rápido, el tiempo invertido es menor
* Mejora la motivación e implicación del equipo de desarrollo
* Mejoran la satisfacción del cliente ya que interactúan constantemente con los nuevos desarrollos
* No se desarrollan características innecesarias, solo lo que el cliente necesita y establecido por prioridad
* Alertar de forma rápida tanto de errores o problemas ya que se hace test y validación constante

Esta figura muestra como cada iteración será una entrega parcial y de dicha entrega se realizarán todas las fases correspondientes. En el caso tradicional, el test se realizaría al terminar el proyecto, cosa que no creemos conveniente.



# Definición del entorno de trabajo

Por las garantías que ofrece, hemos seleccionado el IDE **Netbeans**. Como ya se ha comentado en la reunión Kickoff, algunas de las garantías de este IDE son:

**Ventajas de Netbeans**

* Lenguaje Multi-Plataforma: Código escrito en java es leído por un intérprete, por lo que funcionará en cualquier plataforma
* Manejo automático de memoria
* Gratuito
* Reutilización de módulos
* Incluye templates y Wizards
* Instalación y actualización simple

Además, también se utilizará **GitHub** para compartir el código fuente con todo el equipo y con el objetivo de trabajar sobre versiones de software.

# Guía de estilo

En LMC nos preocupamos porque nuestro código sea simple y, por tanto, más fácil de mantener. Para ello seguimos una guía de estilo para lograrlo.

Una guía de estilo es un conjunto de reglas para dar formato a los proyectos de desarrollo software. Seguir normas de estilo agiliza encontrar la esencia de los programas.

Para lograr la legibilidad de un programa es importante considerar aspectos tales como el nombre de los identificadores, escribir el código con cierta alineación y líneas en blanco en lugares apropiados así como realizar una buena documentación.

# Identificadores

Usamos identificadores descriptivos de tal manera que el solo nombre defina el uso que se dará dentro del programa.

Además de eso los escribimos:

* Empezando con mayúscula si se trata del nombre de una clase o interfaz, y empezando cada palabra en identificador con mayúscula. JugadorHumano
* Sólo con mayúsculas si es el nombre de una constante. DIAS\_HABILES
* Empezando con minúscula si es el nombre de cualquier otro identificador. De preferencia el nombre de cualquier método debe ser un verbo en infinitivo y el de todo atributo un sustantivo. primerJugador, asignarDinero().

# Archivos fuente

Cada programa en Java es una colección de uno o más archivos. El programa ejecutable se obtiene compilando estos archivos. En cada archivo especifica su contenido como sigue:

* Los paquetes (instrucción package).
* Los archivos de biblioteca (Instrucciones import).
* Un [comentario](http://hp.fciencias.unam.mx/~alg/normas/estilo.html#inicio) explicando el objetivo del archivo.
* Las clases que defines en ese archivo.

# Clases

Cada clase debe ir precedida por un comentario que explique su objetivo. Normalmente, seguimos la siguiente estructura:

* Estructura de los objetos. Primero las variables y luego las constantes.
* Elementos estáticos.
* Constructores.
* Métodos públicos y privados.
* Métodos estáticos.
* Clases internas.
* Deja una línea en blanco después de cada método.
* Todos los elementos deben estar precedidos por public, private o protected. Las variables deben ser privadas para que queden encapsuladas. Los métodos y las constantes pueden ser privados o públicos, según se requiera.

# Métodos

Todo método excepto main debe empezar con un comentario en formato javadoc.

# Variables y Constantes

* NO se definen más de una variable por línea.
* Las variables, en la medida de lo posible, se encapsularán con el modificador private.
* No se usan números mágicos. Excepto -1, 0, 1 y 2.

# Alineación y espacios en blanco

La alineación de instrucciones, se puede hacer de manera automática que nos provee el IDE Netbeans.

En cualquier caso los elementos más importantes de ésta son:

* Todos los bloques deben estar alineados de tal manera que sean fácilmente distinguibles. Dentro de un bloque todas las instrucciones van a la misma altura.
* En proposiciones compuestas el paréntesis que abre debe ir en la misma línea que empieza tal proposición. El paréntesis que cierra debe estar sólo en la última línea de la proposición y a la misma altura que la línea de inicio.
* Usa líneas en blanco para separar partes de un método que son lógicamente distintas.

# Análisis de requisitos

A modo de recordatorio, se dejarán todos los requisitos analizados con los stakeholder a tener en cuenta por el grupo de desarrollo en esta fase de ejecución, ya que se deberán aplicar.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nombre** | **Tipo** | **Prioridad** | **Capa** | **Aplicado** | **Test** |
| Programación MVC | No funcional | Alta | Todas |  |  |
| Programación siguiendo buenas prácticas (comentarios, estructura…) | No funcional | Alta | Todas |  |  |
| Soporte de 2 a 8 jugadores | Funcional | Alta | Controlador |  |  |
| Incluir Robots con una IA simplista | Funcional | Baja | Controlador | No se incluye | |
| Deberá soportar importación de partidas mediante ficheros | Funcional | Alta | Modelo |  |  |
| Deberá soportar guardado de partidas mediante ficheros | Funcional | Alta | Modelo |  |  |
| Deberá soportar carga de partidas mediante ficheros | Funcional | Alta | Modelo |  |  |
| El jugador puede crear una partida nueva | Funcional | Alta | Controlador |  |  |
| Se deberá aportar el código fuente del programa | No funcional | Alta | Todas |  |  |
| Consola de texto para visualizar la información de las acciones | Funcional | Alta | Vista |  |  |
| El jugador debe poder mover la ficha | Funcional | Alta | Controlador |  |  |
| El jugador debe poder tirar el dado | Funcional | Alta | Controlador |  |  |
| El jugador debe poder comprar una propiedad | Funcional | Alta | Controlador |  |  |
| El jugador debe poder comprar una propiedad | Funcional | Alta | Controlador |  |  |
| El orden de salida será aleatorio | Funcional | Alta | Controlador |  |  |
| El jugador deberá ver la interacción del juego (Tablero y consola de texto) | Funcional | Alta | Vista |  |  |
| Incluir cartas de suerte y comunidad | Funcional | Alta | Modelo |  |  |
| Jugador podrá visualizar cartas suerte y comunidad | Funcional | Alta | Vista |  |  |
| Sólo se podrá adquirir una propiedad por turno ya sea mediante la compra o la negociación con otro jugador. Las casillas no compradas no saldrán a subasta. | Funcional | Media | Controlador |  |  |
| Se usan dos dados para el juego | Funcional | Alta | Controlador |  |  |
| Un jugador pierde cuando entra en bancarrota y tiene un saldo negativo. | Funcional | Alta | Controlador |  |  |
| Los jugadores recibirán 200€ cada vez que pasen por la casilla de salida mientras queden propiedades sin dueño. | Funcional | Alta | Controlador |  |  |
| Si se sacan dobles se vuelve a tirar, si se sacan dobles 3 veces seguidas el jugador irá a la cárcel. | Funcional | Alta | Controlador |  |  |
| Cuando un jugador pierde, sus propiedades pasan al jugador con quien ha contraído la deuda o a la banca en caso que la deuda sea con ella (en este caso las casillas pasarán a estar en venta y las edificaciones se destruirán). | Funcional | Media | Controlador |  |  |
| Las multas las gestiona la banca creando un fondo de dinero que será obtenido por el primer jugador que caiga en parking gratuito. | Funcional | Media | Controlador |  |  |
| Cuando un jugador va a la cárcel se mueve a la casilla (11). Deberá estar tres turnos y puede tirar dados y salir con dobles o pagar 50€ de multa. Si está tres turnos en la cárcel deberá pagar 50€ de multa y tirar dados. (No se volverá a tirar aunque haya sacado dobles). | Funcional | Media | Controlador |  |  |
| El número total de casas es de 32 y el de hoteles 12. | Funcional | Media | Controlador |  |  |
| Cada jugador empieza con 1500€ | Funcional | Media | Modelo |  |  |

De igual forma, se irá detallando si se ha realizado cada caso de uso en la columna “Aplicado o comentarios”

La siguiente tabla pretende plasmar los casos de uso a desarrollar entrando más en detalle en las características de cada uno de ellos.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Nombre | CU Rel. | Descripción | Trigger | Requisitos Previos | Consecuencias | Excepciones | Aplicado o comentarios | Test |
| 1 | Crear partida |  | Creará la partida con la configuración inicial seleccionada | El usuario selecciona la opción crear partida | La aplicación se ha de abrir y ejecutar correctamente | Se genera una nueva partida |  |  |  |
| 2 | Empezar partida | 1, 14 | Empezará la partida tras crearla o cargarla, en este último caso será desde la última acción realizada.  Si la partida es nueva, el usuario empieza con 1500 euros. | El usuario selecciona la opción empezar partida | La partida ha debido ser creada o cargada | Carga la nueva partida o una ya guardada previamente |  |  |  |
| 3 | Lanzar Dados | 2 | El usuario podrá lanzar los dados, que serán dos y obtendrá un resultado y en función de este podrá avanzar o se desbloquearán varias acciones. | El usuario hace clic en la opción lanzar dados | La partida tiene que estar empezada y todo lo que ello requiere | El usuario avanzará el número de casillas que resulte de tirar los dados. En caso de estar en la cárcel, dobles equivalen a salir de ella. |  |  |  |
| 4 | Mover ficha | 3 | En función del número obtenido en lanzar dados, se deberá mover X casillas la ficha | Al lanzar los dados, se llamará a esta función | El usuario ha debido de tirar los dados | Se moverán X casillas | Está en cárcel no moverá ficha a no ser que haya sacado dobles |  |  |
| 5 | Comprar propiedad | 3,4 | El usuario puede comprar una casa de un color si cae en la casilla correspondiente. El sistema indicará el precio de la propiedad y el usuario elegirá si comprar o no.  El usuario también podrá comprar una propiedad a otro usuario. Por lo que podría hacer una oferta. (relacionado con Caso de uso 8) | El usuario elegirá si comprar propiedad o no. | Se deberá verificar que el usuario tenga saldo para pagar. La partida ha tenido que ser empezada y los dados lanzados. |  | El usuario ya tiene la casa. Por lo que hay que pasar al caso de uso de pagar.  El usuario no tiene saldo |  |  |
| 6 | Elegir carta | 3,4 | El jugador podrá elegir dos cartas, de la suerte y de la comunidad. Sacan Estas cartas cuando se posan sobre las casillas correspondientes y se debe seguir las instrucciones que haya en ellas. | La ficha del jugador cae en una casilla que requiere coger carta | Que haya empezado la partida y el jugador pueda tirar los dados. Significa que no esté en bancarrota ni en la carcel | Se debe tomar una carta y se deberá restar una a la baraja |  |  |  |
| 7 | Pagar Banca | 3,4 | En el caso de caer en una casilla que requiera pagar multa, el jugador tendrá que pagar a la banca un importe X | Caer en casilla que requiera pagar | Que haya empezado la partida y el jugador pueda tirar los dados. Significa que no esté en bancarrota ni en la carcel | Se le restara un importe X al usuario |  |  |  |
| 8 | Oferta a jugador |  | El jugador puede hacer una oferta para comprar una propiedad a otro jugador. También existe la opción de intercambiar casas, por lo que esta oferta puede ser un cambio por otra propiedad o comprar por un importe a introducir | Usar la función de hacer oferta: El jugador deberá introducir la cantidad o la propiedad a intercambiar | Si la oferta es de dinero, verificará el importe del jugador. Si la oferta es de una casa, se deberá tener como mínimo una. Además de que el jugador no puede estar en bancarrota ni en la carcel | Si el jugador B acepta, quitarle la propiedad y/o el importe al jugador A |  |  |  |
| 9 | Vender propiedad |  | El jugador puede ofrecer la venta de su propiedad a otro jugador.  Se deberá introducir el importe por el que se desea vender. | Usar la función de vender propiedad: El jugador deberá seleccionar la propiedad y por cuanto dinero la desea vender | Verificar que la propiedad realmente exista y también que el jugador no esté en la cárcel o bancarrota | Quitarle la propiedad una vez la venta se haya realizado al jugador A y dárselo al jugador B |  |  |  |
| 10 | Ir a carcel | 3 | El jugador irá a la cárcel si cae en dicha casilla. Permanecerá en ella hasta que tirar dados sea dobles o se abone la cantidad de 50 euros | Caer en la casilla cárcel |  |  |  |  |  |
| 11 | Salir carcel | 10,7 | Tirar dados sea dobles, entonces el usuario podrá salir de la cartel en el siguiente turno. También podrá salir si abona 50 euros de multa (pagar a la banca, caso 7) | Tirada de dados dobles o pagar a la banca | El usuario debe estar ya en la cárcel |  |  |  |  |
| 12 | Casilla sailda | 3,4 | Al pasar por la casilla de salida, se le ingresará un importe X al usuario | Cuando se tiren dados y la ficha pase por dicha casilla |  |  |  |  |  |
| 13 | Pagar a jugador | 3,4 | Si el usuario cae en una propiedad de otro jugador. Este le deberá de pagar lo que corresponda | Cuando la ficha caiga en la casilla con una propiedad de otro jugador | La casilla debe tener la propiedad de otro jugador. | El jugador podrá hacer una oferta cambiándole una casa o tendrá que pagar.  La consecuencia mayor es que el jugador entrará en bancarrota y perderá la partida. |  |  |  |
| 14 | Guardar partida | 1,2 | La partida tendrá que estar empezada y el usuario podrá en cualquier momento elegir cuando guardar la partida. Todos los valores se guardarán en ficheros de texto externos que podrán ser cargados posteriormente. | Usuario selecciona la opción guardar partida | La partida debe estar empezada |  |  |  |  |
| 15 | Cargar partida | 13 | El usuario podrá hacer uso de la función cargar partida que obtendrá los valores guardados previamente | El usuario selecciona la opción cargar partida | Debe existir al menos una partida guardada | La partida actual será reemplazada por la nueva |  |  |  |
| 16 | Salir del juego | N/A | El usuario saldrá de la aplicación | El usuario selecciona la opción Salir | Debe estar el juego iniciado | Saldrá del juego completamente |  |  |  |
| 16 | Jugador pierde la partida | 7, 13 | Cuando entra en bancarrota o saldo negativo | Al entrar en bancarrota | Tener un saldo superior a 0 y que pase a ser inferior a 0 |  |  |  |  |

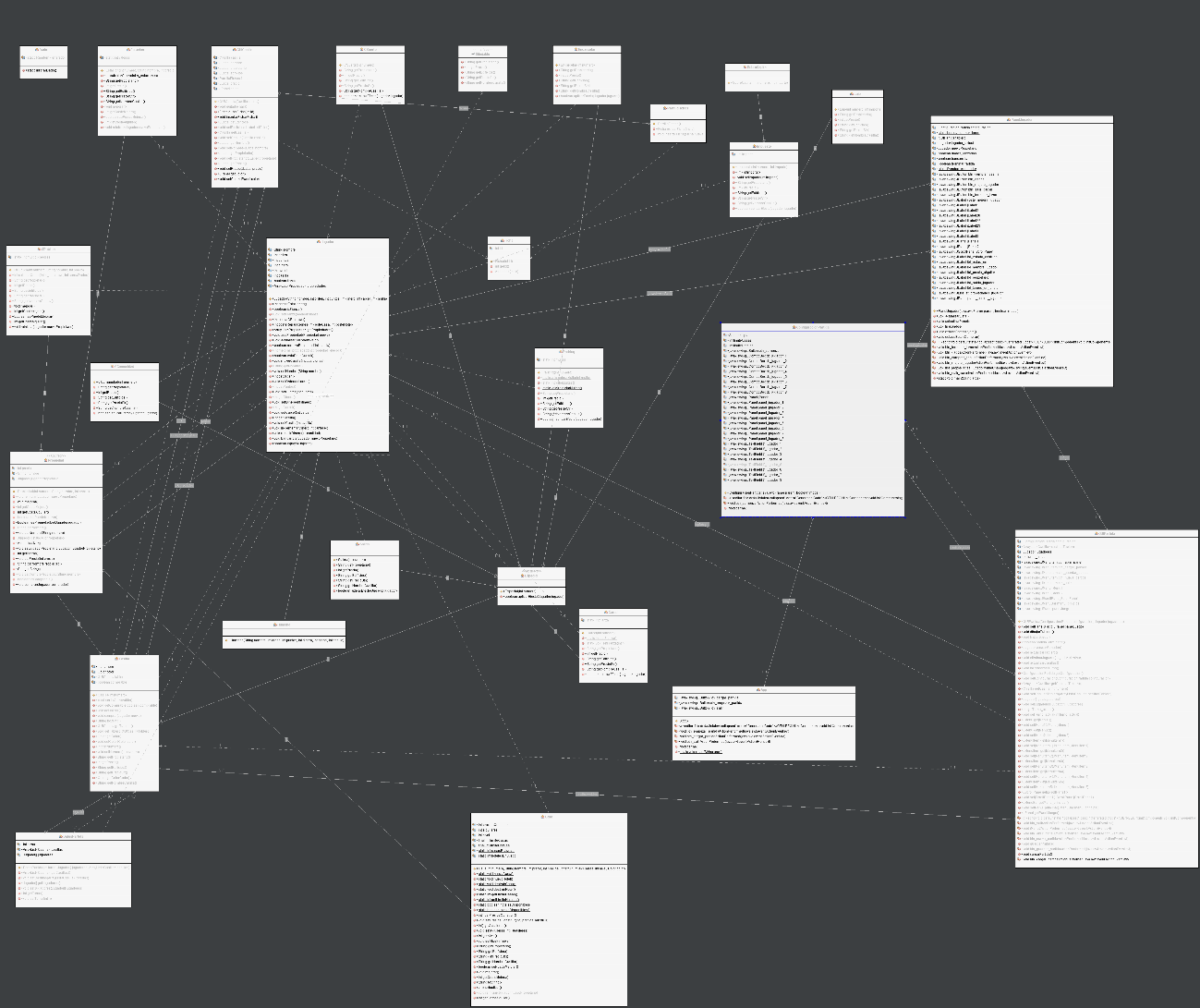
# Control de bugs

A continuación se define una tabla para tener el control de todos los bugs encontrados y disponer de una trazabilidad de si se ha arreglado o no.

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Fecha** | **Descripción** | **¿Arreglado?** |
| 1 | 21/04/20 | Salta excepción nullpointer al salir del juego. Casualmente después de guardar una partida | Sí |
| 2 | 21/04/20 | Al recuperar partida no respeta el turno del juego guardado | Sí |
| 3 | 22/04/20 | Desaparece a veces alguna de las fichas cuando se cae en la casilla 3 | Single case |
| 4 | 22/04/20 | A veces, al caer en la caja comunidad no saca carta | Sí |
| 5 | 23/04/20 | Al pasar por la casilla de salida, si nos sobra tirada la ficha se queda en la propia casilla salida. En la siguiente tirada de ese jugador se suma la tirada + la sobrante anterior | Sí |
| 6 | 24/04/20 | Cárcel se sale sin sacar dobles y a la primera vez de tirar el dado. | No |
| 7 | 25/04/20 | Opción salir. Revisar | Sí |

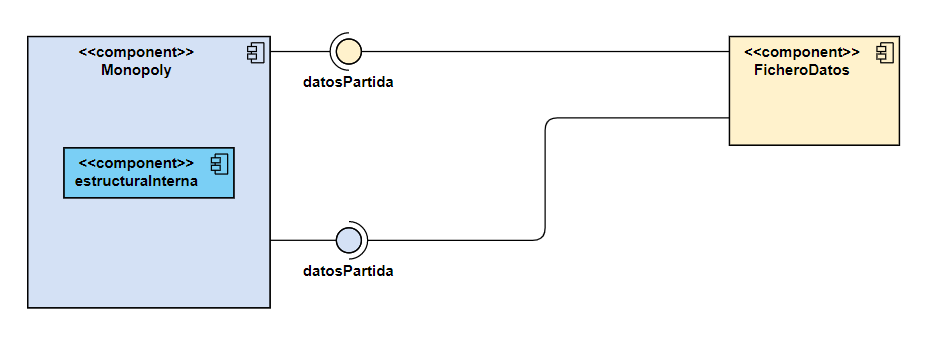
# Diagrama de clases

Se incluye el diagrama de clase que se irá evolucionando a lo largo del proyecto. Algunas clases se eliminarán y otras aparecerán según necesidad. Se ha desarrollado con el pluging de Netbeans EasyUML por la comodidad que conlleva a la hora de llevar el diagrama a código Java.



# Diagrama de componentes

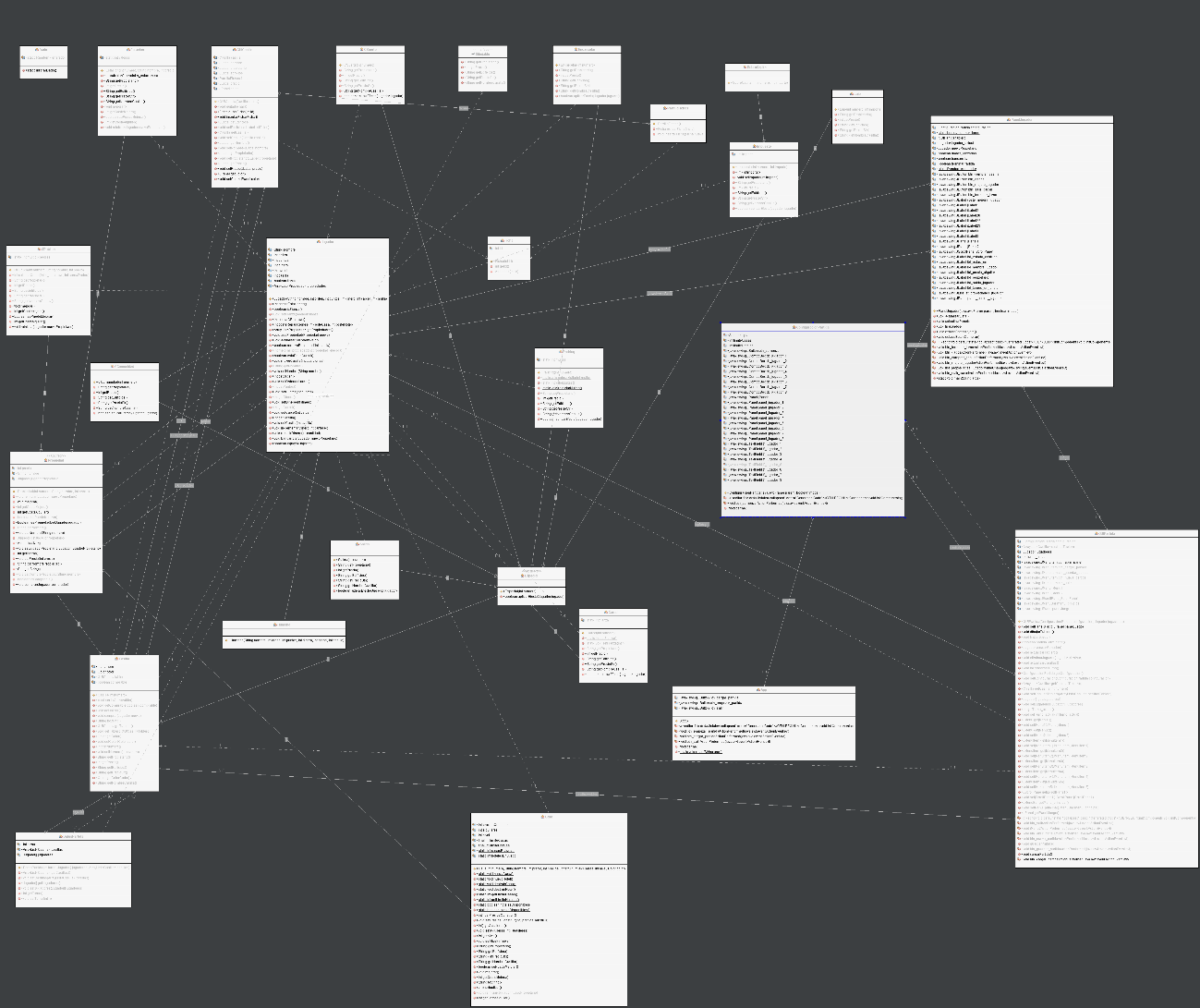
A continuación, adjuntamos el diagrama de componentes. Que aunque es muy básico nos ayudará por si en el futuro queremos ampliar funcionalidades de nuestra aplicación.



# Implementación del UML

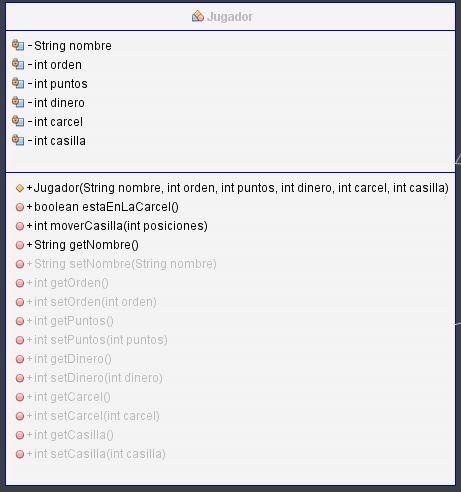
A continuación se ha a explicar de una forma general el diagrama de clases y cuáles de ellas forman la aplicación, para qué existen, sus atributos más importantes y métodos que la integran.

**Diagrama UML**:



# Clase Jugador

La clase jugador, es la clase que representa un jugador con sus atributos y sus métodos. Es una clase genérica por lo que hará de superclase de dos subclases más especializadas.



**Atributos:**

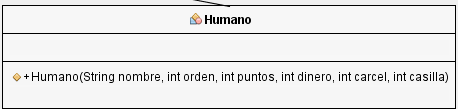
* **nombre**: es el nombre que tendrá el jugador, tanto si es humano como computador.
* **orden**: es el orden de juego que tiene el jugador asignado. Este orden se asigna de una forma aleatoria.
* **puntos**: son los puntos que tiene el jugador en el juego actual.
* **dinero**: el dinero que tiene actualmente el jugador.
* **carcel**: son los turnos que lleva el jugador en la cárcel si está en ella.
* **casilla**: es la casilla en la que está actualmente el jugador.

**Métodos:**

* **jugador**: método constructor con parámetros.
* **estaEnCarcel**: indica si el jugador está o no en la cárcel. Devuelve un boolean con el resultado de la consulta.
* **moverCasilla**: posiciona al jugador en el número de casilla pasado por parámetro. Devuelve un entero con la posición final.
* **Métdodos get/set**: métodos get/set para establecer u obtener el valor de los atributos de la clase.

# Clase Humano

La clase humano extiende de jugador, ya que un humano sería un jugador y otro podría ser, por ejemplo, la IA, que, en este caso está fuera del alcance del proyecto.

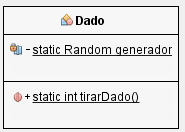


**Métodos:**

* **Humano**: método constructor con parámetros.

# Clase Dado

La clase dado está enfocada al uso de dicha funcionalidad, tirar el dato.



**Atributos:**

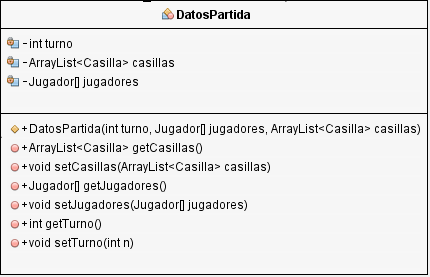
* **generador**: Número aleatorio. Utiliza la función random(). Necesario importar java.util.random

**Métodos:**

* **tirarDado**: Devuelve un número aleatorio que será usado para establecer el valor de la tirada del dado

# Clase DatosPartida

La clase dado está enfocada a establecer y obtener los datos actuales de la partida



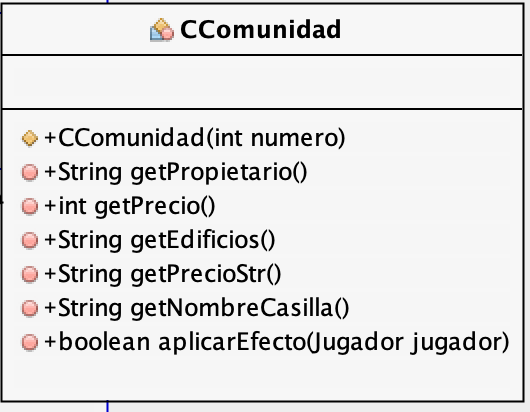
**Atributos:**

* **turno**: Turno del jugador en valor numérico
* **casillas**: Array de la clase casilla que contiene información de las susodichas
* **jugador**: contiene una lista de clases jugadores con todos los jugadores disponibles.

**Métodos:**

* **DatosPartida**: Constructor de la clase el cual recibe el turno, jugadores y casillas
* **getCasillas**: Obtiene la array de casillas con la información de cada una de ellas
* **setCasillas**: Asigna una nueva array de casillas con información de cada una de ellas
* **getJugadores**: Obtener lista de jugadores activo
* **getTurno**: Obtener la variable turno
* **setTurno**: Asignar nuevo turno a la variable

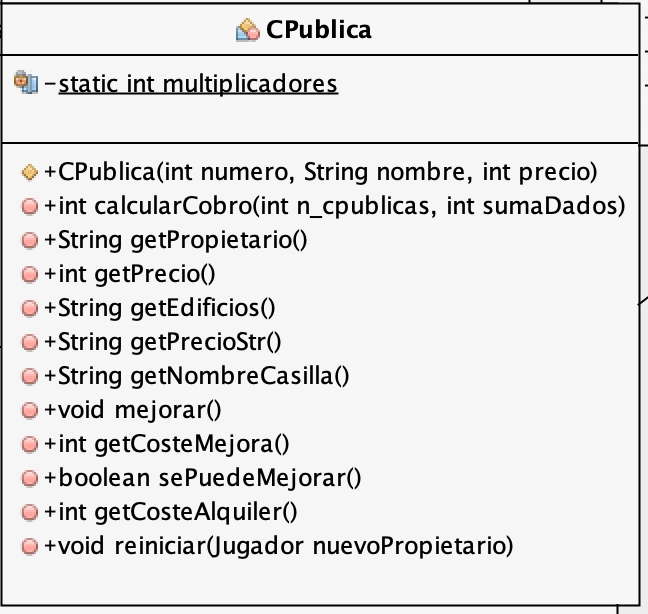
# Clase CComunidad



**Métodos:**

* **CComunidad**: Constructor que recibe el número de la comunidad para obtener sus datos.
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en forma de string.
* **getNombreCasilla**: Obtiene el nombre de la casilla en forma de String.
* **aplicarEfectos**: Devuelve una respuesta booleana para asignar la comunidad al jugador.

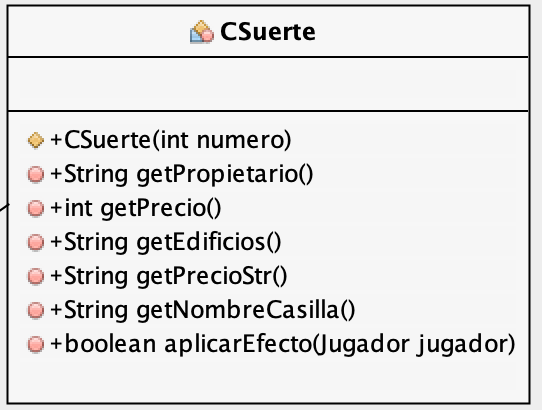
# Clase CPublica



**Métodos:**

* **CPublica**: Constructor que recibe como parámetros el número de la comunidad pública, con su nombre y precio.
* **calcularCobro**: Función que calcula el cobro de la comunidad pública tomando como parámetros el número de las comunidades públicas y la suma de dados.
* **getPropietario**: Obtiene la información del propietario en forma de String.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el nombre del jugador propietario si existe, sino, obtiene el precio.
* **getNombreCasilla**: Obtiene el nombre de la casilla en forma de string.
* **Mejorar**: Al tratarse de comunidad pública, no se puede mejorar
* **getCosteMejora**: Poner el precio a 0
* **sePuedeMejorar**: No hay mejoras
* **getCosteAlquiler**: devuelve lo que cuesta el alquiler de una compañía pública.
* **reiniciar**: Asignar la comunidad pública a un nuevo propietario.

# Clase CSuerte



**Métodos:**

* **CSuerte**: Constructor que recibe como parámetro el número de la suerte.
* **getPropietario**: Obtiene la información del propietario en forma de String.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* **getNombreCasilla**: Devolver la palabra “S U E R T E” en forma de string
* **aplicarEfecto**: Devolver una respuesta booleana en este caso “cierta” al jugador que le ha tocad

# Clase Calle



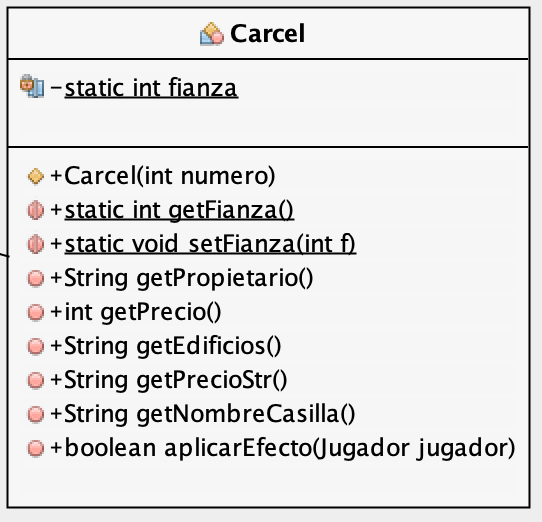
**Atributos:**

* **preciosConstruir**: importe de construcción en valor entero.
* **Alquileres**: importe de alquileres en valor entero
* **nivel**: asignar nivel en valor entero
* **limiteCasas**: establecer el límite de casas en 10
* **limiteHoteles**: establecer el límite de hoteles en 1
* **casasEnJuego**: establecer las casas en juego en principio a 0
* **hotelesEnJuego**: establecer los hoteles en juego en principio a 0

**Métodos:**

* **Calle**: constructor de la clase con parámetros.
* **nuevaCasa**: Método que incrementa en 1 el atributo limiteCasas.
* **nuevoHotel**: Método que incrementa en 1 el atributo limiteHoteles
* **destruirCasa**: Método que decrementa el atributo limiteCasas en uno mientras es más grande de 0.
* **destruirHotel**: Método que decrementa el atributo limiteHoteles en uno mientras es más grande de 0.
* **getLimiteCasas**: Devuelve el valor del atributo limiteCasas.
* **getLimiteHoteles**: Devuelve el valor del atributo limiteHoteles.
* **hotelesDisponibles**: Devuelve el valor de hoteles en juego, siempre que el valor de esos sea más pequeño que el límite de hoteles.
* **casasDisponibles**: Devuelve el valor de casas en juego, siempre que el valor de esas sea más pequeño que el límite de casas.
* **getPreciosConstruir**: Devuelve el valor del atributo preciosConstruir.
* **setPreciosContruir**: Asigna un valor al atributo preciosConstruir.
* **getAlquileres**: Devuelve el valor del atributo alquileres.
* **setAlquileresConstruir**: Asigna un valor al atributo alquileres.
* **getNivel**: Devuelve el valor del atributo nivel.
* **setNivel**: Asigna un valor al atributo nivel.
* **getPropietario**: sobre-escribir el método. Si el jugador propietario es null, printar “sin propietario”, si no, obtener el nombre del propietario.
* **getEdificios**: sobre-escribir el método. Si el nivel es 0, pintar “sin edificar”, si no, si el nivel es menor que 5, printar “casas”, si no, “hoteles”
* **getPrecioStr**: sobre-escribir el método. Si el precio no es null, printar “”, si no, obtener el precio en “€”.
* **getNombreCasilla**: Devuelve el nombre de propiedad.
* **sePuedeMejorar**: Devolverá “true” mientras el nivel es menor que 5.
* **mejorar**: mientras el nivel es menor que 4, es una casa que se puede mejorar, pero si el nivel iguala a 5, es un hotel.
* **getCosteMejora**: Devuelve el precio de mejora del nivel asignado.
* **toString**: pintar el nombre de la casilla seguida por el nivel.
* **actualizar**: actualizar la información de los edificios
* reiniciar: si el nivel es menor de 4, destruir una casa. Si es igual a 5, destruir hotel. Una vez hecho esto, asignar un 0 al atributo nivel.
* **getCosteAlquiler**: Obtenemos el coste del alquiler según el nivel que tenemos.

# Clase Carcel



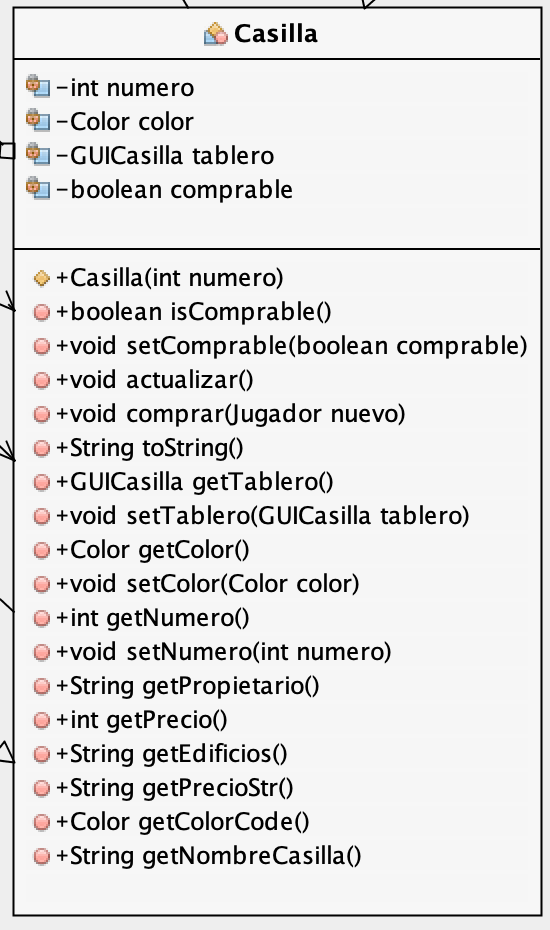
**Atributos:**

* **fianza**: constante que establece cuánto hay que pagar para salir de la cárcel. Según requisitos se deberá pagar 50€

**Métodos:**

* **Carcel**: Constructor que recibe como parámetro el número de la carcel.
* **getFianza**: Establecer el precio de la fianza en 50€
* setFianza: Establece una fianza. Como el atributo es una constante no se podrá asignar más de una vez.
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: retorna 0 ya que este atributo no es necesario para la cárcel. El atributo precio se hereda por una cadena de herencia hasta propiedad.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* **getNombreCasilla**: Devolver la palabra “C A R C E L” ya que se trata de una casilla especial
* **aplicarEfecto**: Devolver una respuesta booleana en este caso “cierta” al jugador que ha caído en la cárcel

# Clase Casilla



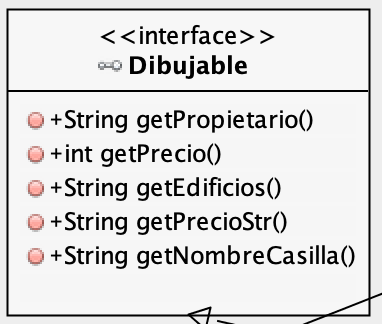
**Atributos:**

* **numero**: declaramos el número en valor entero.
* **color**: creamos clase color sin parámetro
* **tablero**: declaramos el atributo tablero de clase GUIcasilla
* comprable: declaramos un atributo booleano.

**Métodos:**

* **Casilla**: El constructor recibe como parámetro el número e inicializaremos el atributo booleano como “falso” y asignamos un color al atributo color.
* **isComprable**: Método que devuelve el atributo comprable como falso.
* **setComprable**: Método para asignar un valor al atributo comprable.
* **Actualizar**: Método sin cuerpo. Lo hemos implementado en la clase Propiedad.
* **comprar**: Método sin cuerpo. Lo hemos implementado en la clase Propiedad.
* **toString**: Devuelve el número más “-“.
* **getTablero**: Devuelve el valor del objeto tablero.
* **setTablero**: Asigna un valor al objeto.
* **getColor**: Devuelve el valor del objeto color.
* **setColor**: Asigna un valor al objeto.
* **getNumero**: se obtiene el valor del atributo número.
* setNumero: se asigna un valor al atributo número.
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* **getColorCode**: Obtiene el número del color
* **getNombreCasilla**: Devolverá “”

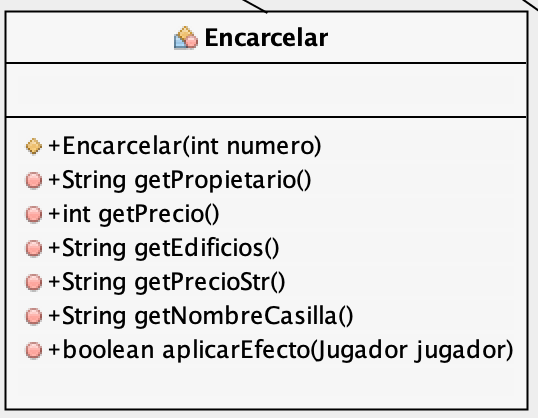
# Interfaz Dibujable



Interfaz que describe cómo se debe comportar las clases que implementes esta interfaz. Las clases que “hereden” esta interfaz, deberán implementar los siguientes métodos:

* getPropietario:
* getPrecio:
* getEdificios:
* getPrecioStr:
* getNombreCasilla:

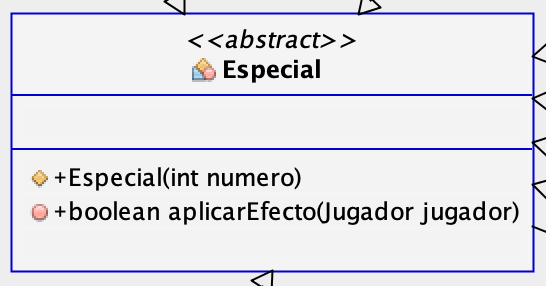
# Clase Encarcelar



**Métodos:**

* **Encarcelar**: Constructor que recibe como parámetro el número de especial para ir a la cárcel.
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* **getNombreCasilla**: Devolver la palabra “VE A LA CARCEL” ya que se trata de una clase que manda el jugador a la cárcel
* **aplicarEfecto**: Devolver una respuesta booleana en este caso “cierta” al jugador que irá a la cárcel

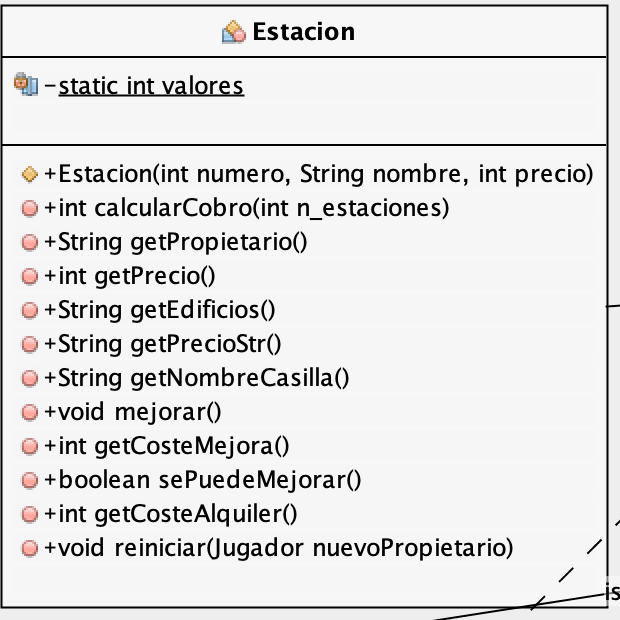
# Clase abstracta Especial



**Métodos:**

* **Especial**: Constructor que recibe como parámetro el número especial, del cual heredan todas las subclases que se extienden de Especial.
* **aplicarEfecto**: Método abstracto que solo se puede usar en una clase abstracta, y no tiene cuerpo. El cuerpo es proporcionado por la subclase (heredada de él).

# Clase Estación



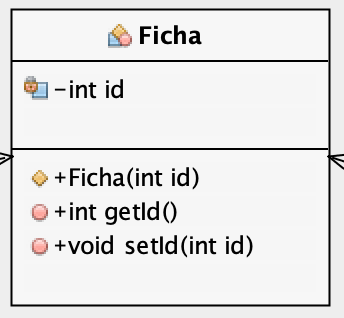
**Atributos:**

* **valores**: declaramos el número en valor entero.

**Métodos:**

* **Estación**: El constructor recibe como parámetro el número especial, el nombre de la estación y su precio.
* **calcularCobro**: Método que calcula el cobro dependiendo de las estaciones que tengamos.
* **getPropietario**: Obtiene la información del propietario en forma de string si existe, si no, imprime “Sin propietario”.
* **getPrecio**: Obtiene el precio de la estación en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena si existe un propietario, si no imprime “”.
* getNombreCasilla: Devuelve el nombre de la propiedad.
* **mejorar**: Al tratarse de estación, no se puede mejorar
* **getCosteMejora**: Método que devuelve un 0.
* **sePuedeMejorar**: Al tratarse de estación, no se puede mejorar
* **getCosteAlquiler**: Método que devuelve un 0.
* **Reiniciar**: Asigna un nuevo propietario a la estación.

# Clase Ficha



**Atributos:**

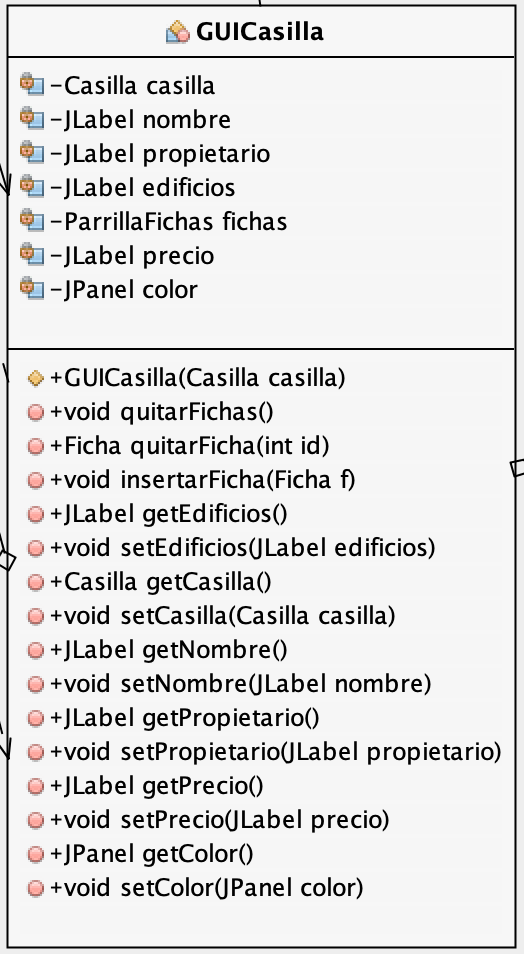
* **id**: id de la ficha en valor entero.

**Métodos:**

* **Ficha**: Constructor que recibe como parámetro el id, y asigna una imagen por cada id.
* **getId**: obtener el id de la ficha en formato entero.
* **setId**: establece el valor de la id.

# Clase GUICasilla

Esta clase hereda de JPanel y “pinta” lo que es una casilla del tablero. Además, implementa la interfaz serializable para gestionar el guardado y carga de partidas.



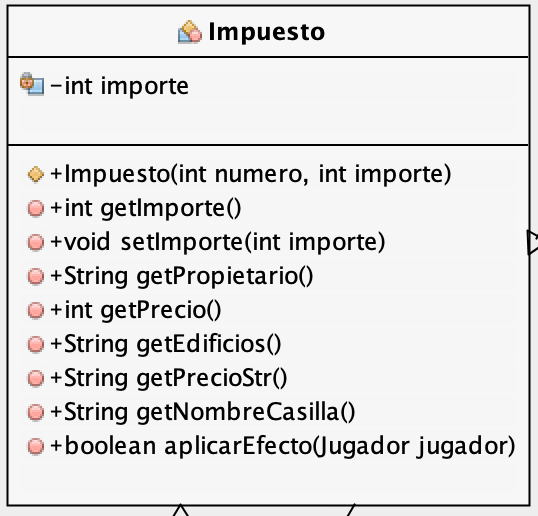
**Atributos:**

* **casilla**: atributo de tipo casilla y de la cual extraeremos los datos necesarios (propietario, precio, etc).
* **nombre**: Nombre que pondremos dentro del JPanel de casilla.
* **propietario**: Propietario que pondremos dentro del JPanel de casilla.
* **edificios**: edificios que pintaremos en la casilla.
* ParrillaFichas fichas: Creamos una lista fichas de la clase ParrillaFichas
* **precio**: Precio que pondremos dentro del JPanel de casilla
* **color**: Color que pondremos dentro del JPanel de casilla

**Métodos:**

* **GUICasilla**: Constructor que recibe como parámetro el atributo casilla y se construye la interfaz gráfica con sus colores y alineamiento.
* **quitarFichas**: Método que descuenta las fichas hasta 8.
* **quitarFichas**: Método que tiene un algoritmo de cómo quitamos las fichas pasando-le un id única.
* insertarFicha: Método que sirve para insertar fichas en la lista “fichas”.
* **getEdificios**: Método que devuelve el nombre del edificio en la interfaz gráfica.
* **setEdificios**: Asignar el nombre mostrado en la interfaz gráfica al atributo edificios.
* **getCasilla**: Método que devuelve la casilla.
* **setCasilla**: Asigna un valor al atributo casilla.
* **getNombre**: Devuelve el nombre de la casilla en la interfaz gráfica.
* **setNombre**: Asigna el nombre de la casilla al atributo nombre
* getPropietario: Devuelve el nombre de propietario en la interfaz gráfica.
* **setPropietario**: Asigna el nombre del propietario al atributo propietario.
* **getPrecio**: Devuelve el importe del precio en la interfaz gráfica.
* setPrecio: Asigna el valor del precio al atributo precio.
* **getColor**: Devuelve el color de la casilla en la interfaz gráfica.
* **setColor**: Asigna un color al atributo color.

# Clase Impuesto



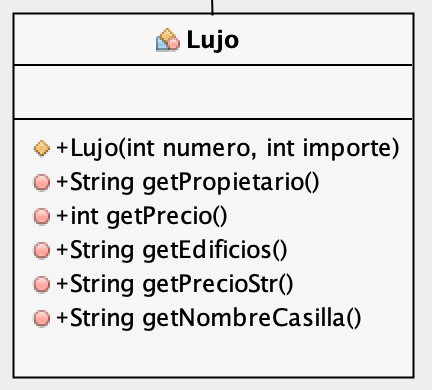
**Atributos:**

* **importe**: importe en valor entero.

**Métodos:**

* Impuesto: Constructor que recibe como parámetro el número de especial para pagar el impuesto y el importe a pagar.
* **getImporte**: Devuelve el valor del importe.
* **setImporte**: establece un importe al atributo importe.
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* **getNombreCasilla**: Devuleve la palabra “INCOME TAX” ya que se trata de una clase que obliga el jugador a pagar
* **aplicarEfecto**: Devolver una respuesta booleana en este caso “pago correcto” si el jugador tiene un saldo más grande que el impuesto a pagar. Si es que sí, se le resta el impuesto de su saldo.

# Clase Lujo



**Métodos:**

* **Lujo**: Constructor que recibe como parámetro el número de especial para pagar el “súper-impuesto” y el importe a pagar heredado de la clase impuesto
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* **getNombreCasilla**: Devolver la palabra “SUPER TAX” ya que se trata de una clase que obliga el jugador a pagar.

# Clase Parking



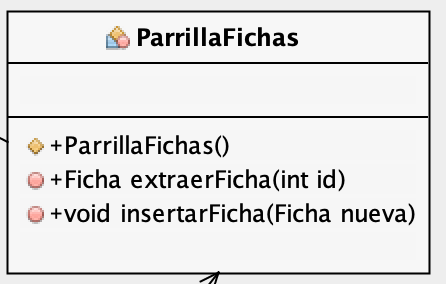
**Atributos:**

* **multas**: importe en valor entero e inicializado a 0.

**Métodos:**

* **parking**: Constructor que recibe como parámetro el número de especial para estar dentro del parking.
* **cobrar**: Asignamos el valor de la multa al atributo multas
* **valorMultas**: Devolver el valor de la multa
* **reiniciarMultas**: Método para reiniciar el valor del atributo multas a 0.
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: Método que devuelve un 0.
* **getEdificios**: Obtiene el nombre del edificio en forma de string. En este caso, el método no devuelve nada “ ”.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* getNombreCasilla: Devolver la palabra “P A R K I N G” ya que el método se encuentra dentro de la clase Parking.
* **aplicarEfecto**: Devolver “cierto” ya que se trata de la clase parking.

# Clase ParrillaFichas

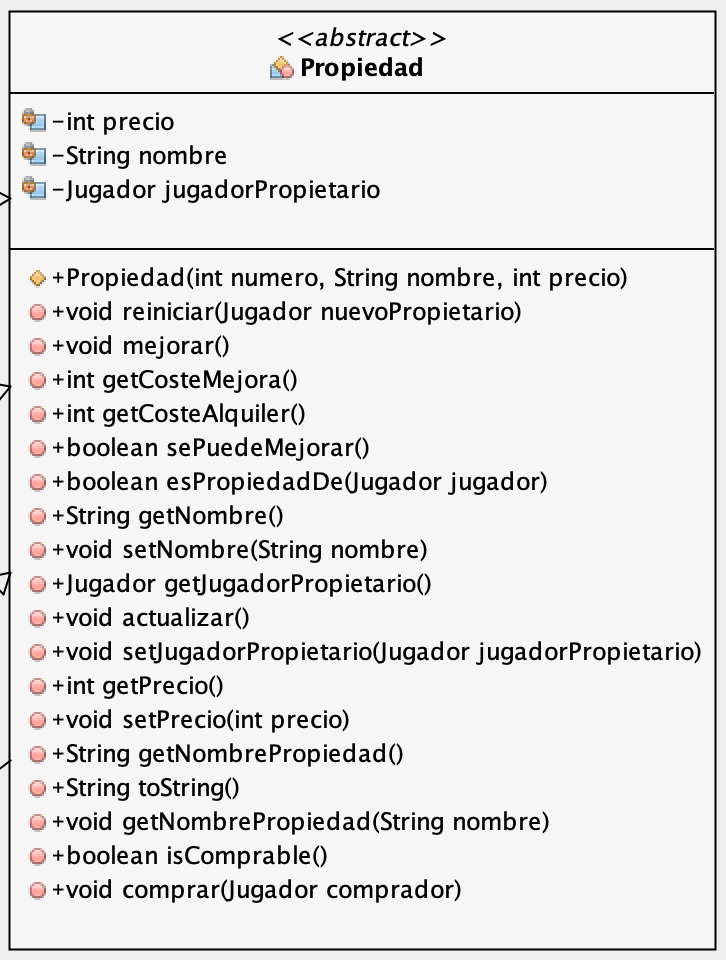


**Métodos:**

* **ParrillaFichas**: El constructor emplea el GridLayout de 2 filas y 4 columnas que vendrían a ser la ficha
* **extraerFicha**: Recorre las fichas que hay en la parrilla y si coincide con la que le pasamos por parámetro la quitamos ejecutando el método remove() heredado de JPanel.
* **insertarFicha**: Inserta ficha al JPanel.

# Clase abstracta Propiedad

Esta clase representa lo que tiene que ser una propiedad y cómo se deben comportar las clases que hereden de ella. Ciertos métodos ya disponen de alguna implementación, pero otros se han declarado como abstractos para que las clases que extiendan de ella implementen su propio comportamiento.



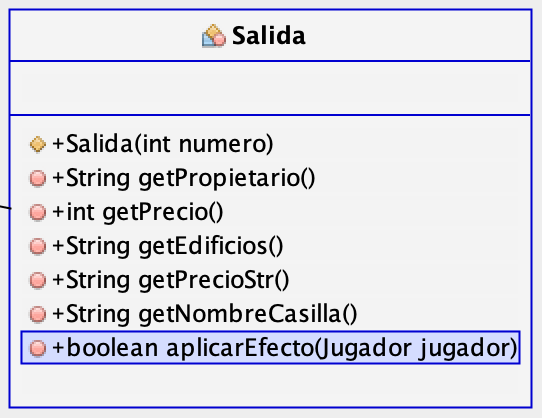
**Atributos:**

* **precio**: precio en valor entero.
* **nombre**: precio en valor entero.
* **jugadorPropietario**: indica mediante el objeto de tipo Jugador quién es el propietario.

**Métodos:**

* **Propiedad**: Constructor que recibe como parámetro el número de Propiedad, nombre y precio
* **reiniciar**: Cuando la propiedad se asigna a un jugador, da igual si es nuevo o es una compra de segunda mano, se reinicia la propiedad para que coja los parámetros del nuevo propietario
* **mejorar**: método sin cuerpo. Se implementa en las clases especializadas de esta clase abstracta.
* **getCosteMejora**: método sin cuerpo. Se implementa en las clases especializadas de esta clase abstracta.
* **getCosteAlquiler**: método sin cuerpo. Se implementa en las clases especializadas de esta clase abstracta.
* **sePuedeMejorar**: método sin cuerpo. Se implementa en las clases especializadas de esta clase abstracta.
* **esPropiedadDe**: Comprueba la propiedad del jugador antes de devolver el resultado booleano.
* **getNombre**: Obtener el nombre de la propiedad
* **setNombre**: Asigna el nombre de la propiedad al atributo Nombre.
* **getJugadorPropietario**: Devuelve el jugador propietario.
* **actualizar**: Actualiza la información del jugador propietario en el tablero, tanto el nombre, como el precio que debe tener unas condiciones especiales como: que la propiedad debe tener un propietario y que el precio debe ser mayor que 0
* **setJugadorPropietario**: Asigna la propiedad al jugador y actualiza la información mediante el método actualizar.
* **getPrecio**: Obtiene el precio de la propiedad.
* **setPrecio**: Asigna el precio a la propiedad y actualiza la información
* **toString**: método para imprimir por pantalla “Sin propietario” cuando no hay ninguno
* **getNombrePropiedad**: método para obtener el nombre de la propiedad
* **isComprable**: Devuelve true cuando la propiedad no pertenece a nadie
* Comprar: Si la propiedad es comprable, debemos tener más dinero de lo que cuata dicha propiedad, y debemos hacer la operación aritmética para restar dinero del jugador. Después de esto, añadimos la propiedad a la lista de propiedades del jugado y actualizamos la información.

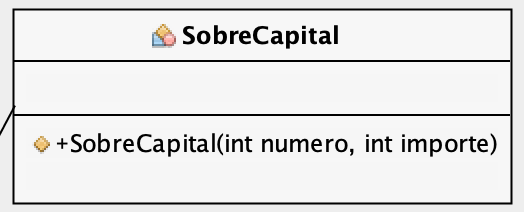
# Clase Salida



**Métodos**:

* **Salida**: Constructor que recibe como parámetro el número de especial para pagar poder salir
* **getPropietario**: Obtiene la información del propietario en forma de string.
* **getPrecio**: Obtiene el precio de la comunidad en forma de un número entero.
* **getEdificios**: Obtiene el nombre del edificio en forma de string.
* **getPrecioStr**: Obtiene el precio en formato cadena.
* **getNombreCasilla**: Devuelve la palabra “S A L I D A” ya que se trata de una clase que obliga el jugador a salir.
* **aplicarEfecto**: Devolver una respuesta booleana en este caso “cierto” al jugador que tendrá que salir

# Clase SobreCapital



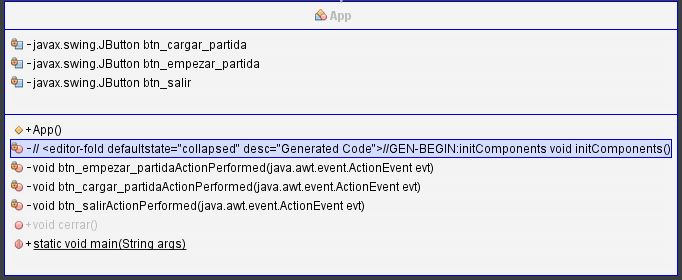
**Métodos:**

* **SobreCapital**: Constructor que recibe como parámetro el número de especial para pagar el “sobre-capital” y el importe a pagar heredado de la clase impuesto.

# Clase App

Clase que es la pantalla de inicio de nuestra aplicación. Es la clase que tiene el método main para hacerla ejecutable.

Gestiona el inicio, carga de partida guardada y salir del juego.



**Atributos:**

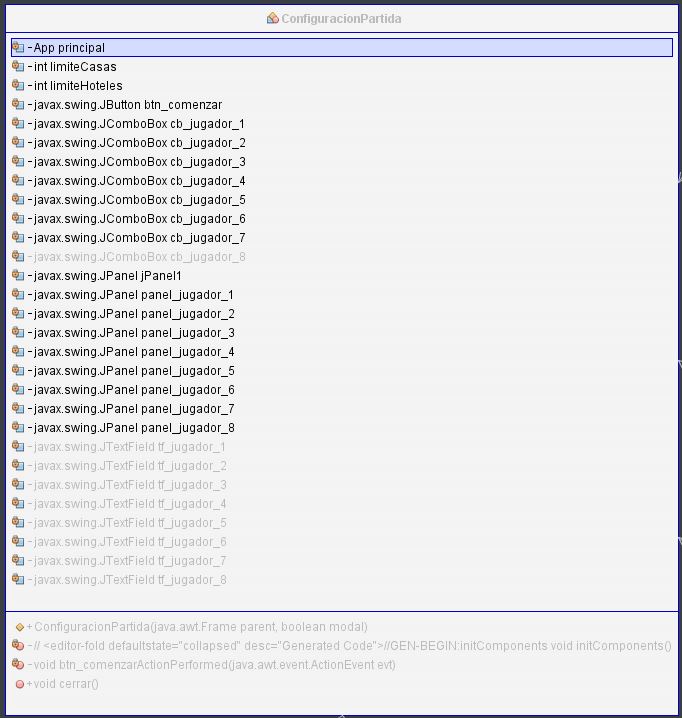
* **btn\_cargar\_partida**: botón de carga de partida y que se muestra en el JFrame.
* **Btn\_empezar\_partida**: botón de carga de partida y que se muestra en el JFrame.
* **btn\_salir**: botón para salir de la aplicación y que se muestra en el JFrame.

**Métodos:**

* **initComponets**: método generado automáticamente por Netbeans cuando añadimos los botones, JFrames, etc. Lo que hace es inicializar todos los componentes por medio del constructor App().
* **btn\_empezar\_partidaActionPerformed**: método que solicita comenzar una partida nueva pasando los parámetros this (objeto App) y valor true a ConfiguracionPartida.
* **btn\_cargar\_partidaActionPerformed**: método que muestra el tablero con los datos de la partida guardados anteriormente mediante el método cargarPartida().
* **btn\_salir\_partidaActionPerformed**: sale de la aplicación por medio del método dispose() de la clase JFrame.
* **main()**: método para “lanzar” la aplicación.

# Clase ConfiguracionPartida

La clase ConfiguracionPartida sirve básicamente para comprobar cuantos jugadores van a intervenir en la partida y “lanzar” el tablero de juego.



**Atributos:**

* Los atributos de esta clase engloban los posibles jugadores de la partida y si son personas o la IA (no implementada).

Se captura el nombre del jugador por medio de su correspondiente JTextField y JComboBox.

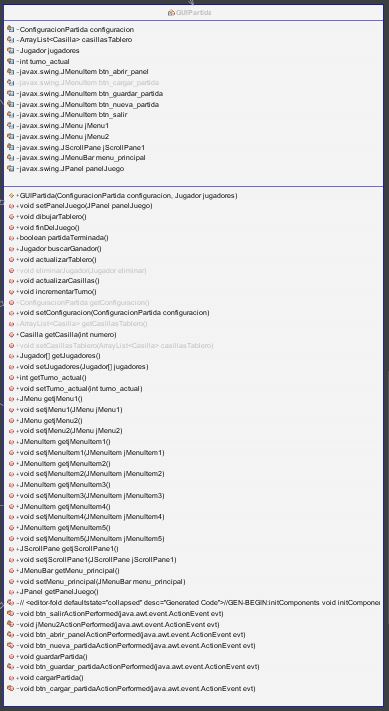
* Además, existe un JButton para comenzar la partida.

**Métodos:**

* **initComponets**: método generado automáticamente por Netbeans y que inicializa los componentes de JDialog
* **btn\_ComenzarActionPerformed**: manda como parámetros el contexto del objeto y un array con los jugadores que intervienen en el jugo.
* **cerrar**: destruye el actual JDialog y muestra el JFrame App de nuevo.

# Clase GUIPartida

Esta clase gestiona toda la parte visual de la partida (tablero) y ciertos puntos de la lógica del juego, como por ejemplo buscar un ganador.



**Atributos:**

* **configuracion**: es la configuración de la partida que se le pasa del JDialog ConfiguracionPartida.
* **casillasTablero**: es un ArrayList con todas las casillas que se van a pintar en el tablero.
* **jugadores**: variable que guarda los jugadores actuales de la partida.
* **turno\_actual**: es el turno que se está jugando en cada momento.
* **btn\_abrir\_panel**: botón para abrir el panel del jugador que tiene el turno de juego.
* **btn\_cargar\_partida**: botón para cargar una partida previamente guardada.
* **btn\_guardar\_partida**: guarda la partida actual en el fichero “partidaGuardada.sav”.
* **btn\_nueva\_partida**: comienza una nueva partida creando un nuevo objeto ConfiguracionPartida y mostrando ese panel para elegir jugadores de nuevo.
* **btn\_salir**: botón para salir del juego.
* **jMenu1**: menú en donde están los botones nueva partida, guardar partida, cargar partida y salir.
* **jMenu2**: menú que contiene el botón de para abrir el panel del jugador actual.
* **jScrollPane1**: es el panel con la barra de scroll para movernos por el tablero.
* **Menú\_principal**: en este menú están jMenu1 y jMenu2.
* **PanelJuego**: es dónde están todas las casillas del tablero.

**Métodos:**

* **GUIPartida**: constructor de la clase con parámetros.
* **setPanelJuego**: establece el JPanel panelJuego.
* **dibujarTablero**: dibuja el tablero y añade las casillas al mismo por medio del método panelJuego.add().
* **finDelJuego**: método que destruye el objeto GUIPartida y muestra de nuevo App para poder elegir de nuevo qué hacer (nueva partida, cargar partida o salir).
* **partidaTerminada**: termina la partida cuando haya sólo un jugador activo.
* **buscarGanador**: revisa si existe un único jugador activo. Si hay alguno, en tal caso, ese jugador es el ganador.
* **actualizarTablero**: actualiza las posiciones de las fichas dentro del tablero.
* **eliminarJugador**: elimina jugador que se le ha etiquetado como eliminar.
* **actualizarCasilla**: recorre todas las casillas del tablero y le aplica el método actualizar del objeto de tipo Casilla.
* **IncrementarTurno**: incrementa el turno hasta 8 (número posibles de jugadores).
* **Métdodos get/set:** métodos get/set para establecer u obtener el valor de los atributos de la clase.

# Clase PanelJugador

Clase gestiona todo lo referente a la jugabilidad del jugador dentro de la GUI. Es una clase que hereda de la clase JDialog.



**Atributos:**

* **configuracion**: es la configuración de la partida que se le pasa del JDialog ConfiguracionPartida.
* **Primer\_turno**: variable que indica si es el primer turno del jugador.
* **tablero**: guarda toda la acción del tablero para que pueda interactuar el jugador.
* **Jugador\_actual**: indica si en ese momento el jugador es el que tiene el turno.
* **nuevoPropietario**: indica que jugador es el nuevo propietario cuando entra en bancarrota.
* **dados\_lanzados**: indica si el jugador ya ha lanzado los dados.
* **bancarrota**: variable que indica si el jugador cuyo turno está en curso entra o no en bancarrota.
* **terminarPartida**: si sólo queda un jugador y la partida ha terminado.
* **generador**: es el generador aleatorio de turno de los jugadores.
* **btn\_comprar\_casilla**: es el botón para ejecutar una compra de la casilla en la que ha caído el jugador.
* **Btn\_dados**: botón para lanzar dados.
* **btn\_mejora\_jugador**: botón para mejorar la propiedad que es propietario el jugador.
* **btn\_salir\_carcel**: botón para salir de la cárcel pagando 50€.
* **btn\_terminar\_turno**: botón para terminar turno del jugador actual.
* **coste\_mejora\_jugador**: muestra en un Jlabel el coste de la mejora de la propiedad.
* **jLabelx**: son las etiquetas auxiliares para mostrar información adicional como puede ser: saldo, turnos en prisión, etc.
* **lbl\_estado\_edificios**: muestra el estado de edificación de las propiedades del jugador.
* **lbl\_estas\_en**: muestra en qué casilla está el jugador actualmente.
* **lbl\_nombre\_jugador**: muestra el nombre del jugador actual configurado al principio de la partida.
* **lbl\_precio\_alquiler**: muestra el precio del alquiler elegida de la lista mediante el método getCosteAlquiler() de la clase Propiedad.
* **lbl\_propietario**: muetra en el panel el propietario, si lo hubiese, de la propiedad elegida.
* **lbl\_saldo\_jugador**: muestra el saldo actual del jugador.
* **lbl\_turnos\_prision**: muestra los turnos que lleva el jugador en la cárcel.
* **List\_propiedades\_jugador**: muestra una lista de las propiedades que tiene el jugador actualmente.
* **Panel\_control\_jugador**: es el contenedor de todos los componentes antes mencionados.

**Métodos:**

* **PanelJugador**: constructor de la clase con parámetros.
* **actualizarLista**: lista las propiedades del jugador actual.
* **actualizarPanel**: actualiza las etiquetas del panel jugador para que se visualice la información del nombre del jugador activo, dinero, turnos en la cárcel y otros detalles.
* **tirarDados**: genera toda la lógica de visualización y de juego en una tirada de dados del jugador actual.
* **estadoBotonMejora**: activa o desactiva según tengamos posibilidad de mejora en la propiedad seleccionada.
* **estadoBotonComprar**: activa o desactiva según tengamos posibilidad de compra en la casilla que hemos caído.
* **btn\_terminar\_turnoActionPerformed**: cuando pulsamos botón de terminar turno corre el turno al siguiente jugador.
* **btn\_dadosActionPerformed**: básicamente ejecuta método tirarDados().
* **btn\_comprar\_casillaActionPerformed**: comprueba si la casilla actual es una propiedad y está libre. En tal caso activa el botón de comprar.
* **btn\_mejora\_jugadorActionPerformed**: comprueba si la casilla actual se puede mejorar. En tal caso, mejora descontando el dinero de la mejora al jugador.
* **list\_propiedades\_jugadorValueChanged**: actualiza en la lista las mejoras en las propiedades.
* **btn\_salir\_carcel**: activa la salida de la cárcel si tenemos la carta “salir de cárcel” o pagamos 50€. Si no, nos indicará que no tenemos suficiente dinero para salir.

# Control de versiones

Como hemos comentado en el apartado 4.11 (Definición del entorno de trabajo), el control de versiones y trabajo conjunto se realizará con el entorno VCS Github. Los motivos que hayamos elegido este SW es que tenemos relativamente bastante experiencia y es fácil de usar por todos los miembros del equipo de desarrollo.

La estructura de funcionamiento a alto nivel de esta aplicación se explica en la siguiente imagen:



Además del proyecto en Netbeans, hemos incluido el borrador de la memoria del proyecto (este mismo documento) y el cronograma con los hitos del proyecto. De esta forma, podemos tener toda la documentación sincronizada para todos los miembros del equipo.

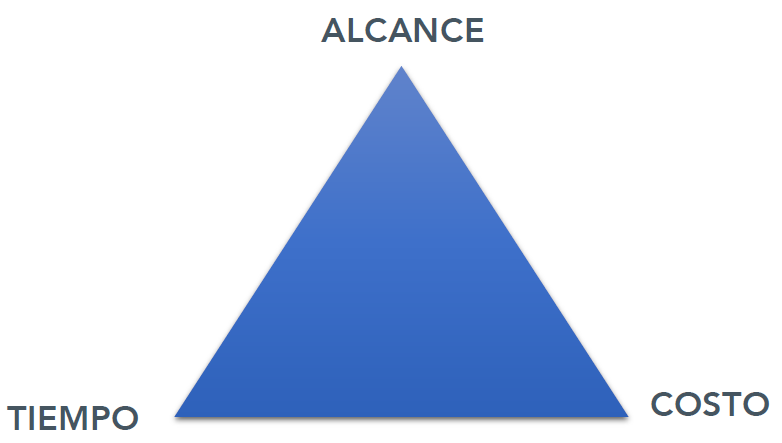
Cada desarrollador ha instalado el programa Github en su equipo local, de esta forma se crea el entorno local de trabajo. Seguidamente un desarrollador ha creado en la plataforma web la estructura del proyecto en Netbeans y documentación en el servidor de Github, esta estructura tendrá una dirección URL de acceso y todos los desarrolladores han clonado en su dispositivo local con el objetivo de vincular la carpeta local con el servidor común, para nuestro trabajo la carpeta en el servidor se accede desde la siguiente URL:

<https://github.com/raulgarcia03004/Monolopy.git>

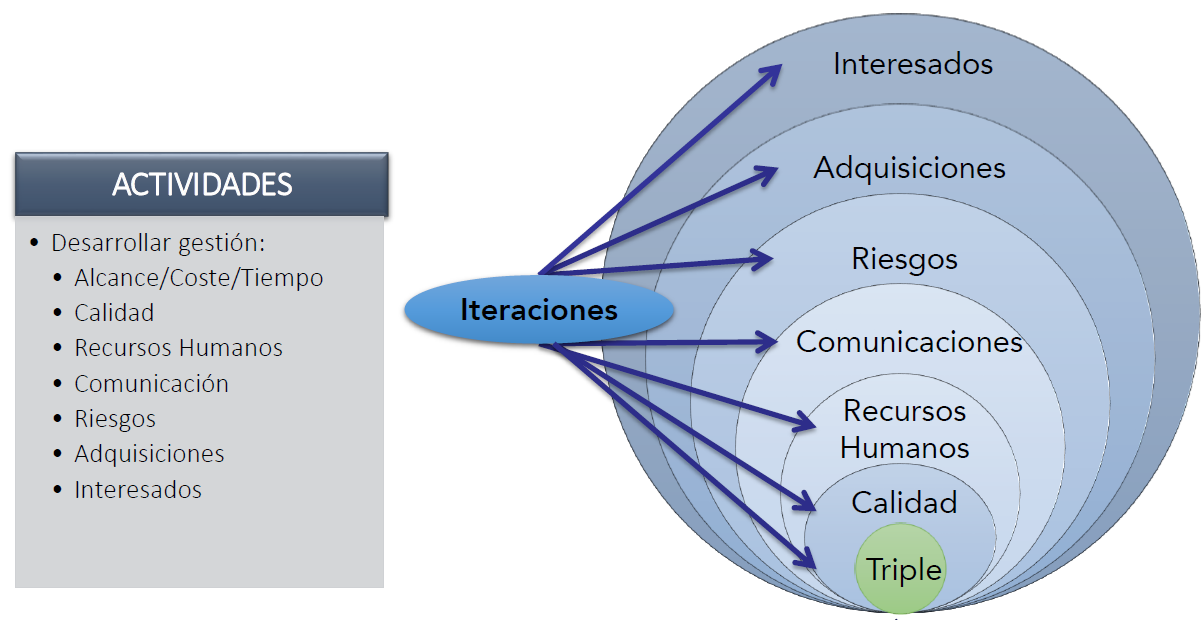
# Fase de control del proyecto

En esta fase se incluye los procesos y actividades que permite asegurar que el proyecto satisfará las necesidades por las cuales fue iniciado.

Siempre nos apoyaremos en la triple restricción: Alcance, Costo y Tiempo para equilibrar la balanza entra lo que espera nuestro cliente “Inv2AS” versus EL tiempo y costo estimado en la fase de planificación. Esta triple restricción, en su estado óptimo, tendría el aspecto de un triángulo equilátero como el de la figura siguiente:



Nosotros nos basaremos en todo momento en esta triple restricción para alcanzar los objetivos del proyecto. No obstante existe un nuevo paradigma en el cual nos apoyaremos en alguna ocasión. Este modelo se denomina “Triple restricción ampliada”



# Responsabilidades relacionadas con la calidad

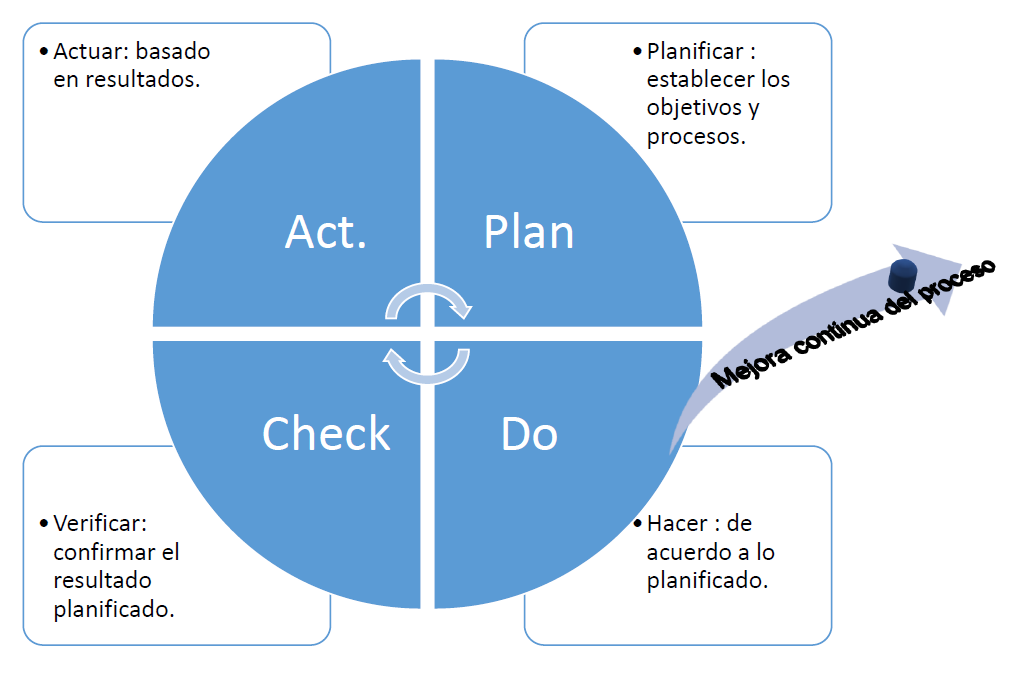
Si bien la calidad de cualquier producto de software depende de todas las personas involucradas durante el proceso de creación del mismo, la mayor responsabilidad sobre la calidad final del producto debería ser garantizada por un profesional del software testing. Dicho profesional buscará apoyar con su metodología, técnicas y herramientas, la calidad del producto mediante actividades de pruebas bien definidas, para proporcionar la confianza adecuada de que un producto o servicio cumplirá las expectativas y necesidades de los usuarios en términos de efectividad, eficiencia, usabilidad y satisfacción. De todas formas en “LMC”

* **Senior Manager**: responsable de la calidad en la empresa.
* **Project Manager**: último responsable de la calidad del proyecto.
* **Equipo del Proyecto**: responsables de su propio trabajo.
* **Responsable de calidad**: responsable de certificar que se están cumpliendo los requisitos del proyecto. No puede depender del PM

# Procesos de calidad en el proyecto

|  |  |
| --- | --- |
|  |  |
| **Planificación de calidad**:   * Encontrar estándares de calidad para el producto y la gestión del proyecto. * Crear estándares específicos para el proyecto * Determinar el trabajo a desarrollar * Determinar cómo se medirán los productos y actividades * Balancear las necesidades de calidad con el alcance, coste , tiempo y satisfacción |
| **Realizar Aseguramiento de Calidad**:   * Auditar los resultados obtenidos * Determinar si el proyecto cumple con las políticas y procesos de la organización * Identificar las oportunidades de mejora * Recomendar cambios y acciones correctoras |
| **Realizar Control de Calidad**:   * Medir los resultados según los estándares |

Para el control de calidad, LMC se basa en la mejora continua o ciclo PDCA. El Ciclo PDCA constituye una estrategia de mejora continua de la calidad en cuatro pasos, también se lo denomina espiral de mejora continua y es muy utilizado por los diversos sistemas utilizados en las organizaciones para gestionar aspectos tales como calidad (ISO 9000), medio ambiente (ISO 14000), salud y seguridad ocupacional (OHSAS 18000), o inocuidad alimentaria (ISO 22000).



Para nuestro proyecto, nos ayudará a revisar qué se está haciendo, si se está haciendo bien y si podemos mejorarlo, tanto para este proyecto (Monopoly Inv2AS) como para futuros proyectos que puedan surgir.

Para llevar a cabo este proceso, LMC implanta unas medidas y actividades concretas para el desarrollo del ciclo PDCA. Estas medidas son:

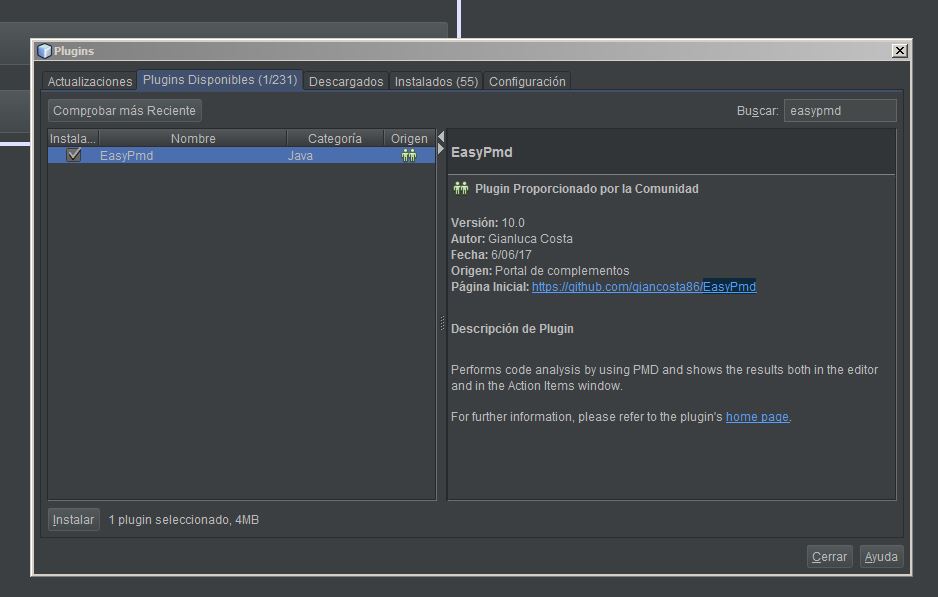
* Asignar un **líder** del proceso: en nuestro caso es la misma persona que desarrolla el rol de PM.
* Revisar los **niveles** de calidad del proyecto: cada vez que se va a cumplir un hito, en LMC revisamos (auditamos) si se ha cumplido al 100% las metas de dicho hito.
* Revisar los posibles valores de **riesgo** del proyecto: en estas auditorías intentamos identificar riesgos que se puedan dar.
* Establecer medidas de **corrección**: estableceremos medidas de corrección para los riesgos identificados.
* Analizar la **eliminación** de aquellos indicadores que ya no se consideren necesarios.

# Pruebas de calidad del código fuente: EasyPmd

Según avanzamos en la implementación del código fuente, en paralelo, vamos testeando la calidad del código generado. Para ello, nos ayudamos de EasyPmd.

EasyPmd es un plugin que se integra en el IDE Netbeans y que nos ayuda a testear la calidad del código de una forma fácil e integrada con nuestro proyecto.

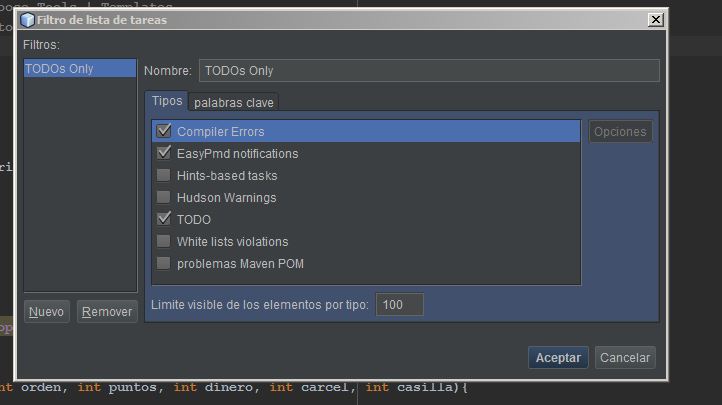
Para activar EasyPmd sólo tenemos que instalar el plugin que tiene el mismo nombre.



Plugin EasyPmd

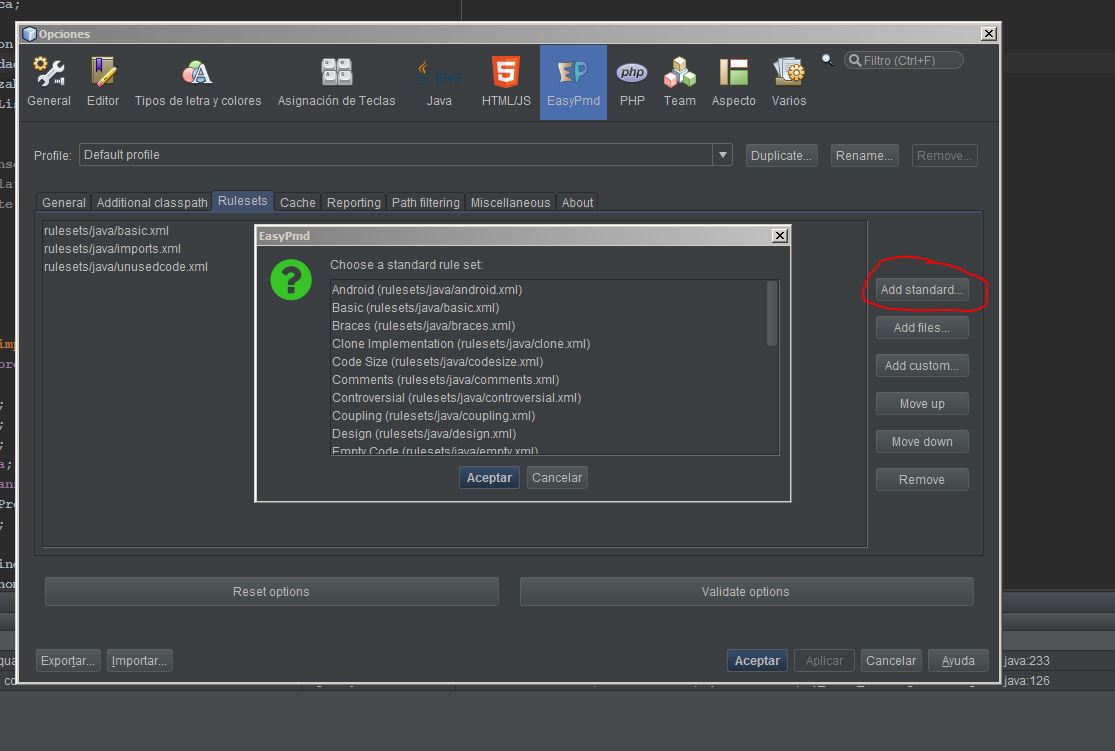
El proceso es como cualquier plugin. Seleccionar, instalar, aceptar términos, instalar dependencias (si las hubiese) y reiniciar el IDE.

Luego, activamos los filtros que nos interesen para que el plugin los encuentre y nos avise de estas anomalías.



Activar filtro EasyPmd

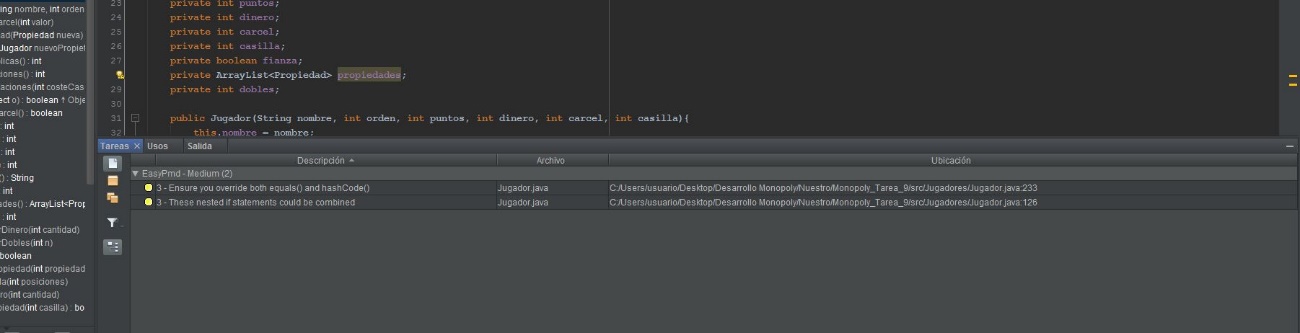
Añadimos las librerías estándar de filtros. Si queremos, podemos añadir ficheros externos con nuevos filtros.



Activar librería de filtros estándar

A continuación mostraremos varias correcciones que hemos hecho del código una vez ejecutado EasyPmd en todo nuestro proyecto.

**Clase Jugador**

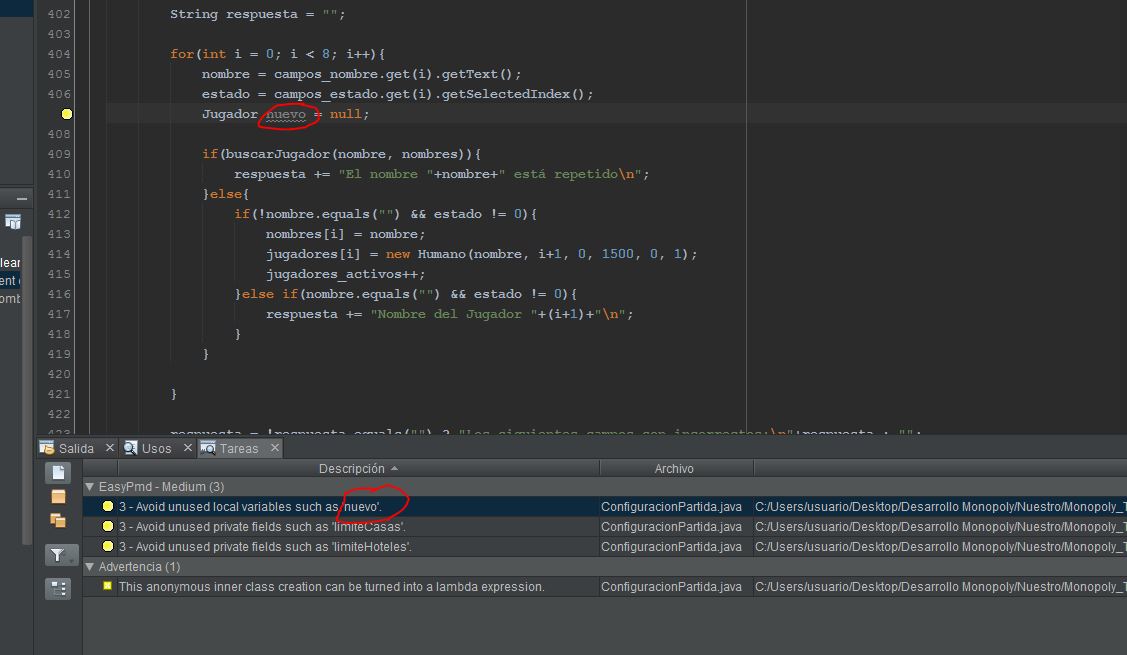


Clase Jugador

En esta clase hemos encontrado dos problemas de nivel médio (no demasiado importantes)

* Problemas al sobreescribir los métodos equals() y hashCode()
* Se pueden simplificar condiciones if combinando sus expresiones.

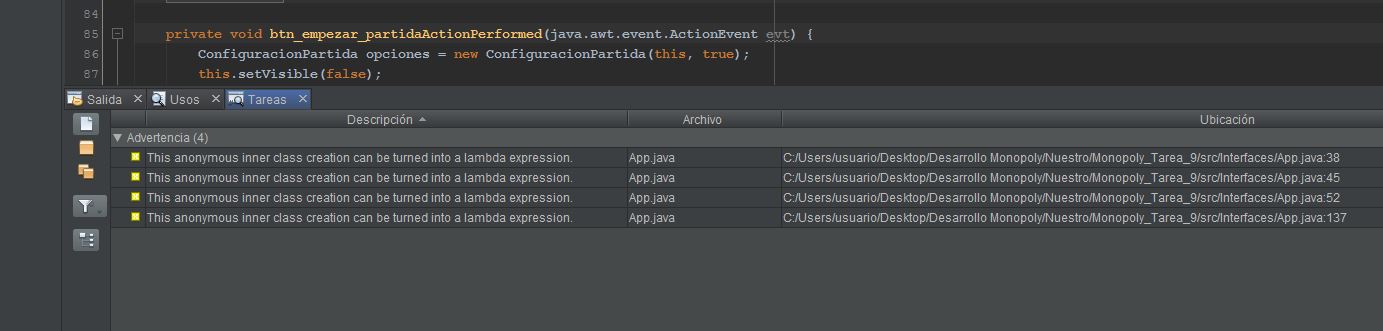
**Clase ConfiguracionPartida**



Clase COnfiguracionPartida

Principalmente, en esta clase hemos declarado varios atributos que luego no usamos. Si nos fijamos bien podemos ver que Netbeans también nos advierte directamente en el editor de código.

**Clase App**



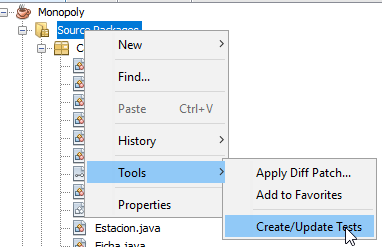
Clase App

En la clase App hemos resuelto problemas de código lambda. El código lambda o función lambda es una subrutina definida que no está enlazada a un identificador.

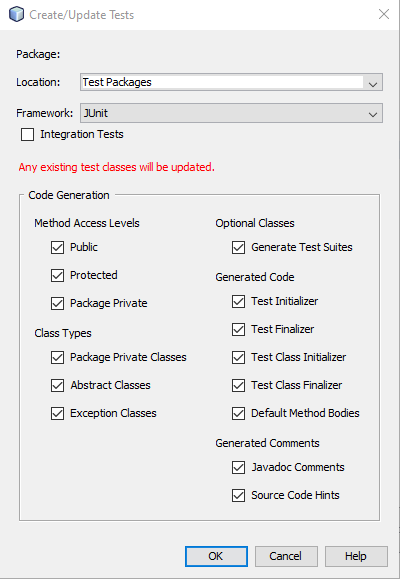
# Pruebas código fuente: JUnit

Unit es un framework java que permite la realización de la ejecución de clases de manera controlada con el fin de que el resultado por el que se han desarrollado se cumple y es el esperado, además de controlar algún error que pueda suceder.

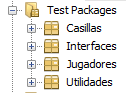
Para realizar las pruebas, hemos creado mediante esta funcionalidad todas las clases de test, las cuales, hemos tenido que adaptar para comprobar cada función o método.



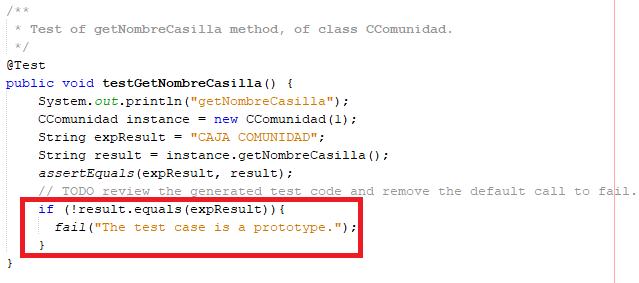
Mediante esta funcionalidad, crearemos el paquete de clases a testear de toda la aplicación:



A partir de aquí se ha creado nuestro paquete de pruebas.

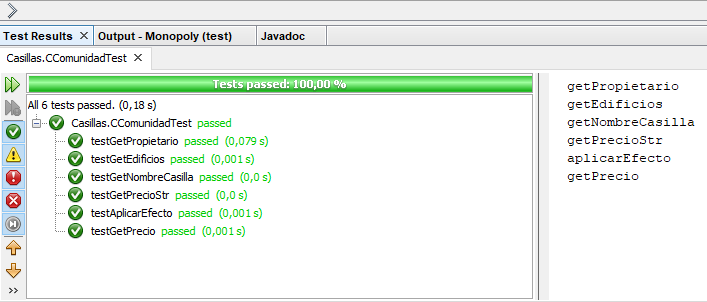


A continuación se dejan algunas clases testeadas con JUnit a modo de ejemplo. Siempre se tienen que adaptar los métodos instanciando los objetos correspondientes y haciendo una validación en cada uno de ellos. Por defecto JUnit no hace la validación final, por lo que siempre hay que incluir un *if* para que nos diga si el resultado es el correcto o no tal como este:

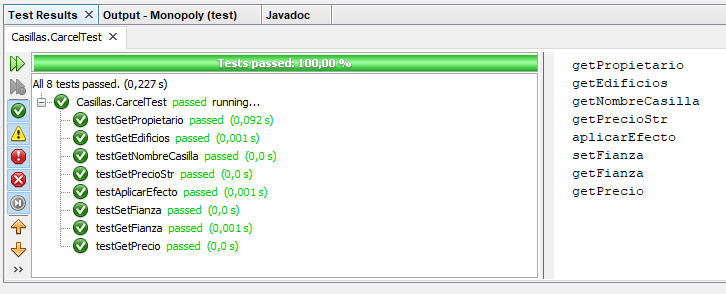


A continuación plasmamos a modo de ejemplo alguna clase analizada por JUnit:

**CComunidad.java**



**Carcel.java**



**Ficha.java**

