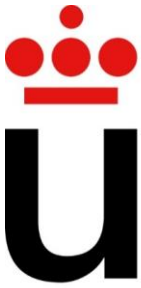


Universidad Rey Juan Carlos

Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2019/2020

Trabajo de Fin de Máster

Refactoring del Juego del Tic-Tac-Toe

Evolución con pruebas y refactoring

Autor: Víctor Sánchez Muñoz

Tutor: Luis Fernández Muñoz

Contenido

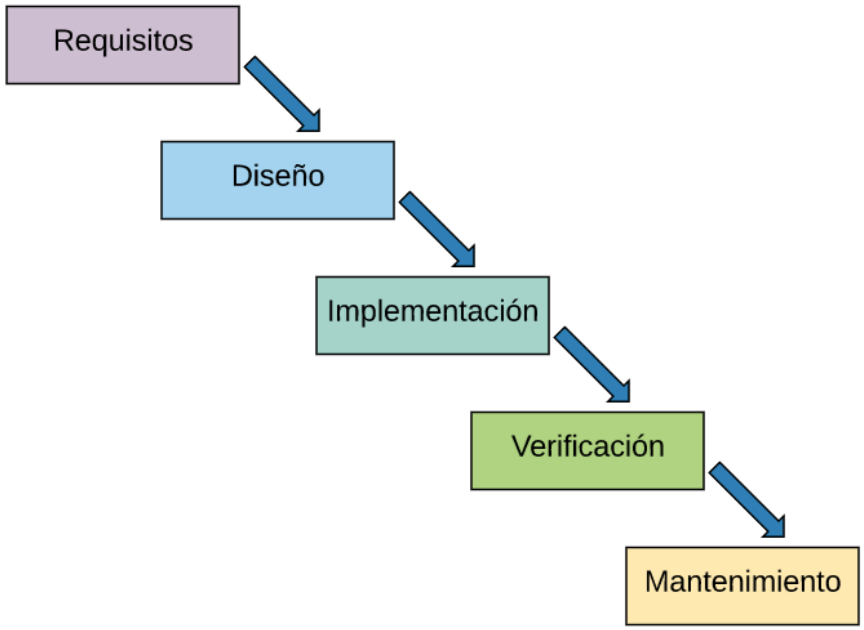
Introducción

Este proyecto se enfoca exclusivamente en las cuatro primeras materias del máster. Entre las que se incluyen: Diseño y calidad software, Patrones y arquitectura software, Pruebas software y Programación extrema.

Evolución del software

En el año 1968 surge el concepto de Ingeniería del software, con el que se intenta conseguir resolver los problemas que se habían acuciado en la crisis del software que se había producido en los pasados años.

En 1970 nace el modelo en cascada, el cual ordena el ciclo de vida del software, esperando cada etapa a que termine la anterior. Este modelo fue revisado posteriormente en 1980, definiéndose como se puede visualizar en la siguiente imagen:



Entre los años 1990 y 2000 se desarrollan importantes modelos relacionados con la mejora de los procesos software (Ideal, TSP…) y normas y estándares de calidad (ISO 9126, ISO 12207…). Adicionalmente, se consolida la orientación a objetos de cara al desarrollo, a partir de la que aparecen muchas metodologías, que acaban dando lugar a la aparición del Lenguaje de Modelado Unificado (UML) y el Proceso Unificado (UP). También surgen diversas técnicas y conocimientos para la construcción y desarrollo de estos sistemas orientados a objetos: patrones, heurísticas, refactorizaciones…

A mediados de la década de 1990 surge la Agilidad. Surge como alternativa a los modelos estructurados y estrictos que aparecieron originados por el uso del modelo en cascada. Con estos nuevos modelos, ágiles e iterativos, se pretende dejar atrás el uso del modelo en cascada.

Entre los años 2000 y 2010 aparecen nuevos procesos ágiles, con una serie de principios, como son:

- **eXtremePrograming (XP)**: retroalimentación rápida, simplicidad asumida, cambios incrementales, aceptar el cambio y trabajo de calidad.
- **Scrum**: control empírico de procesos, auto-organización, colaboración, priorización basada en el valor, time-boxing y desarrollo iterativo.
- **Teoría basada en el valor**
- **MDD**

También surgen métodos híbridos, combinando la adaptabilidad de las metodologías ágiles con una documentación rigurosa.

Del 2010 hasta la actualidad, se han ido afianzando los principios y métodos de años anteriores, así como darle una importancia muy grande a la seguridad.

Objetivo

El objetivo es hacer una aplicación que vaya evolucionando. Para lo que se necesita la asignatura de programación extrema, a la cual se la va añadiendo funcionalidad, a la vez que se realizan distintas refactorizaciones para mejorar el diseño de la misma. Por ello se precisa la asignatura de pruebas y diseño, que, en consecuencia, incluye la de patrones y arquitectura, así como temas de calidad de software. Se desea, además, realizar diversas mediciones comparativas entre distintas versiones para poder examinar como hemos conseguido cambiar el diseño y la estructura de nuestro proyecto.

Requisitos

Modelo del dominio

En el siguiente diagrama se visualiza la relación entre las distintas entidades del modelo y el estado en el que se encuentran cuando se comienza la partida. Cuando se consigue llegar al estado final de la aplicación, con tic-tac-toe en la tercer fila de nuestro tablero, nuestro diagrama cambia, como vemos en la imagen, añadiendo las entidades Piece.

Para poder entender mejor las instrucciones de la aplicación, se ha modelado el siguiente diagrama, donde mediante un flujo, se puede observar la lógica planteada para el modelo de tic-tac-toe.

Casos de uso y prototipo de interfaz

Para exhibir la evolución del software, enseñamos distintos requisitos funcionales y no funcionales que agregamos a las distintas versiones de la aplicación, realizando una evolución con ello. Podemos identificar las siguientes versiones a partir de los requisitos:

1. Básica

Versión básica del juego Tic Tac Toe, con interfaz de texto desde la consola, distribución Standalone y sin persistencia de los datos de la aplicación.

2. Gráficos

Se añade la interfaz gráfica al proyecto mediante librerías propias de Java. Con este requisito funcional, conseguimos que el usuario pueda tener una interfaz más agradable que una consola de texto.

3. *UndoRedo*

Se implementa la nueva funcionalidad de undo/redo, con la que el usuario puede deshacer y rehacer los movimientos que se van realizando durante la partida mediante un nuevo menú con una serie de opciones.

4. *ClienteServidor*

Nuevo requisito no funcional para poder realizar la distribución de la aplicación en dos partes separadas y comunicadas. La parte de Cliente para la interfaz propia que usa el usuario y el Servidor dedicado al manejo y gestión de los datos.

5. *Ficheros*

Nueva funcionalidad de cara al usuario, para poder guardar el estado de las partidas en curso y así poder retomarlas en otro momento. Se usan ficheros para el almacenamiento de estos datos y su recuperación.

6. *BaseDatos*

Aparte del almacenamiento con ficheros, se agrega la posibilidad de guardado en base de datos del estado de las partidas. Este requisito se clasificaría como no funcional.

Cada una de estas versiones corresponde a una de las ramas del repositorio github de requerimientos, en el que están documentadas cada una de las versiones.

TDD

Pruebas

Para las pruebas incorporadas en las distintas versiones de la aplicación, se han utilizado diversas tecnologías. La principal de estas tecnologías es **JUnit**, con la que se han podido verificar los distintos valores y resultados esperados. Adicionalmente, se ha utilizado **Mockito**, que nos ha permitido devolver respuestas mediante *mocks* en diversos métodos de la aplicación, para verificar su funcionamiento, como, por ejemplo, en los métodos que requieren de la introducción de datos por parte del usuario. En conjunción a estas tecnologías, se han añadido ***asserts*** al código principal, de cara a poder asegurar que distintas partes del programa están funcionando como deberían, y de esta manera, controlar posibles errores.

Estas pruebas, se han ido modificando, adaptando y ampliando a medida que se han realizado refactorizaciones o añadido de funcionalidades en el código, de cara a las distintas versiones que se han ido desarrollando durante la práctica.

Para el desarrollo se ha utilizado la técnica de **TDD**, con la que, lo que se ha realizado en primer lugar, es el desarrollo de los tests de las distintas clases y métodos, pensados en primer lugar en el diseño. Se han añadido las distintas clases y métodos necesarios para que poder ejecutar los tests implementados comprobando que éstos, en primer lugar,

fallan. A raíz de estos pasos, el siguiente es la implementación de los métodos de las clases, de cara a que las pruebas pasen correctamente. De esta forma, nos podemos asegurar que todos los casos de uso que se han ido definiendo mediante los tests, van a funcionar correctamente en nuestra aplicación.

Refactoring

Durante el proyecto se ha avanzado en el código, tanto añadiendo nueva funcionalidad como realizando refactorizaciones que mejoren la calidad y el diseño, mediante el añadido de patrones de diseño, técnicas y remodelado del código.

Algunos ejemplos de pequeñas refactorizaciones que se han realizado han sido:

- Cuando incluimos el patrón Facade en nuestro código, añadimos la clase Lógica, con la cual desacoplamos todos los controladores de las vistas, las cuales usarán esta clase para comunicarse con los controladores, ya que incluye todos los de nuestra aplicación: StartController, PlayController y ResumeController. Así como la clase Game, con la que se incluyen los modelos también en la Lógica, asociando ésta a cada uno de los controladores.
- Cuando incluimos el patrón DAO, lo que conseguimos es que podamos añadir nuevas tecnologías a nuestra aplicación, de cara a, por ejemplo, para el proyecto de esta práctica, el almacenamiento de los datos para las partidas guardadas. Para conseguir tener este diseño, usamos la interfaz DAO, y extraemos a clases nuevas los métodos propios de los modelos, responsables del guardado y cargado de datos en la aplicación. En nuestro caso, save() y load(). De esta manera conseguimos abstraer esta funcionalidad, pudiendo añadir nuevas formas de almacenamiento (BBDD, ficheros…) sin tener que modificar código, sino solo añadiendo nuevo.

Desarrollo

En esta sección se describen los cambios en el diseño y añadido de funcionalidad que se ha ido realizando en cada una de las versiones desarrolladas durante todo el proyecto.

1.1. domainModel

En esta versión el proyecto se compone solo de modelos, basados en las entidades propias del juego Tic-Tac-Toe que se han identificado para el modelo de dominio, y las relaciones entre éstos. La interfaz es de texto por consola, con la que interactúa el usuario. Podemos ver el diagrama de esta primera versión. Las clases están acopladas a la tecnología de interfaz, tienen la Ley del Cambio Continuo y son de grano grueso ante la llegada de nueva funcionalidad.

2.1. documentView

Se separan las vistas de los modelos, dejando la responsabilidad de interactuar con el usuario a las clases propias del paquete “views”. Los modelos siguen siendo de grano grueso ante la llegada de nueva funcionalidad.

En el diagrama podemos observar que, la clase View se compone de los distintos estados en los que se puede encontrar la vista: StartView(se elige el número de jugadores que va a participar en la partida), PlayView(se realizan los movimientos de las fichas hasta que uno de los jugadores gana la partida) y ResumeView(se decide si continuar con una nueva partida o acabar el juego). Cada una usa la clase Game para realizar la comunicación con los modelos y poder modificarlos.

3.2. dv.withoutFactoryMethod

Añadimos la interfaz gráfica a la interfaz de texto que usábamos hasta ahora. Para ello importamos la librería javax.swing de Java. Las clases de interfaz tienen DRY en vistas de las diferentes tecnologías y seguimos con grano grueso en modelos ante la llegada de nueva funcionalidad.

4.2. dv.withFactoryMethod

Añadimos una clase abstracta TicTactoe, de la que heredarán todos los tipos de vista, en nuestro caso ConsoleTicTacToe y GraphicsTicTacToe.

De esta manera (véase imagen) conseguimos un comportamiento Open/Close, con el que poder añadir nuevas clases de vista para nuestra interfaz, aun así seguimos con modelos de grano grueso ante la llegada de nueva funcionalidad.

5.2. modelViewPresenter.presentationModel

Se añaden las nuevas clases controladoras que conectan las vistas con los modelos. Las clases controladoras, basadas en los estados propios de la aplicación: StartController, PlayController y ResumeController. Las vistas usan los distintos controladores, los cuáles conocen la clase Game, con la que gestionan los modelos. Podemos visualizar con el diagrama lo comentado. La clase principal y las vistas están acopladas a cada controlador que tenemos actualmente y todo el que implementemos nuevo.

6.2. mvp.pm.withFacade

Se añade la nueva clase Logic en la que se encapsulan controladores y modelos, pero con las vistas con DRY. De esta forma (véase imagen), las vistas solo conocerán a Logic y no a cada uno de los controladores mencionados.

7.2. mvp.pm.withoutDoubleDispatching

Se añade la clase State a Logic de cara a la inversión de control de las vistas, pero violando el Principio de Sustitución de Liskov. Con lo que, como se puede ver en el diagrama, basado en el estado en el que nos encontremos, se usará un controlador u otro y respectivamente, la vista asociada a ese controlador.

8.2. mvp.pm.withDoubleDispatching

Se añade la clase ControllersVisitor con la que conseguimos no depender del tipo de instancia del controlador para conocer la vista a la que está asociada. Sino que se utiliza la clase comentada ControllersVisitor para realizar esta asociación entre el controlador y la vista.

Visualicemos el diagrama de clases para comprenderlo mejor.

9.3. mvp.pm.withComposite

Clase Comando del menú y Controlador Compuesto de ciertos Estados para Open/Close. Se añade la funcionalidad Undo/Redo, con la que poder elegir, mediante el menú, si queremos realizar el movimiento, deshacer el anterior o, en caso de haber deshecho un movimiento, rehacerlo. Para ello se añaden 3 nuevos controladores, cada uno por cada opción del menú en la que nos encontremos: UndoController, RedoController y MovementController. PlayController se compone de cada uno de estos nuevos controladores, ya que en este estado es en el que usaría el menú.

(Véase el diagrama de los controladores)

De cara a la representación del menú y sus opciones en las vistas, se han añadido nuevas clases Command (con las que se representan las distintas opciones) y una Clase PlayMenu que se compone de estos comandos.

(Véase el diagrama de las vistas)

A los modelos le añadimos distintas clases para controlar y almacenar los movimientos que se van realizando a lo largo de la partida, para después poder deshacer y rehacer.

- Memento: almacena uno de los momentos en los que se ha encontrado la partida al realizar un movimiento.

- Registry: almacena un listado de instancias de la clase Memento con todos los movimientos que se han realizado en la partida y el momento en el que se encontraba.

- Session: clase que se compone de State, Game y Registry, y con la que se comunican los controladores para realizar cambios en los modelos.

10.4. mvp.pm.withoutProxy

Añadimos la clase TCPIP con la que conseguimos un Tic-Tac-Toe distribuido en cliente/servidor, pero con los controladores acoplados, poco cohesivos y grano grueso con la nueva tecnología. En el que creamos una clase LogicServer que se encarga de asociar cada una de nuestras clases Dispatcher (cuya funcionalidad es llamar a los controladores mediante la clase AcceptorController, de la que heredan StartController, PlayController y ResumeController) a cada uno de nuestros controladores principales, de cara a obtener información de los modelos o modificarlos. La clase DispatcherPrototype tiene la funcionalidad de, dependiendo del FrameType que obtenga a través de la clase TCPIP, llamar al Dispatcher correspondiente de la aplicación. La relación entre las clases sustituidas la podemos ver en el diagrama.

11.4. mvp.pm.withProxy

Se añaden las nuevas clases Proxy para la funcionalidad Open/Close de cara a nuevas tecnologías de despliegue.

(Véase el diagrama de las clases distribuidas)

Adicionalmente cambiamos los controladores a tipo abstracto y creamos las clases que implementan dichos controladores para comunicarse con los modelos.

(Véase el diagrama de la implementación de los controladores)

12.5. mvp.pm.withoutDAO

Se añaden nuevos controladores (SaveController) y nuevos comandos y menús a las vistas (NewGameCommand, OpenGameCommand, GameSelectCommand y GameSelectMenu) para añadir la funcionalidad de guardado de partidas en ficheros, de cara a reanudarlas en el futuro, que el usuario puede usar a través de la vista SaveView. Los modelos son de grano grueso, con baja cohesión y alto acoplamiento a tecnologías de persistencia de ficheros.

13.5. mvp.pm.withDAO

Para el guardado de datos, se añaden las clases DAO. De esta manera (véase el diagrama) podemos añadir nuevas tecnologías de almacenamiento, ya que abstraemos, mediante la clase DAO, las distintas implementaciones.

14.6. mvp.pm.withoutPrototype

Se añade la nueva tecnología de almacenamiento de las partidas mediante BBDD, que se une al almacenamiento mediante ficheros. Todo ello se realiza a partir de la implementación de los DAOs. Podemos ver el diseño de esta nueva tecnología con patrón DAO en el diagrama. La clase principal está acoplada a las tecnologías del proyecto.

15.6. mvp.pm.withPrototype

Mejoramos la aplicación para que sea open/close en el ámbito de la persistencia del almacenamiento de partidas.

Podemos ver las diferencias con la anterior versión.

(Diagrama de la versión anterior)

(Diagrama de la versión actual)

16.1. mvp.pv

Se cambia la comunicación entre los distintos paquetes. En esta nueva versión los controladores se comunican con las vistas, al contrario que en versiones anteriores, que eran las vistas las que interactuaban con los controladores.

17.1. mvp.sc

Se añade la clase abstracta Command con la que se delega la interacción con la sesión a los distintos comandos, cuando se activan cada uno de ellos.

18.1. mvc

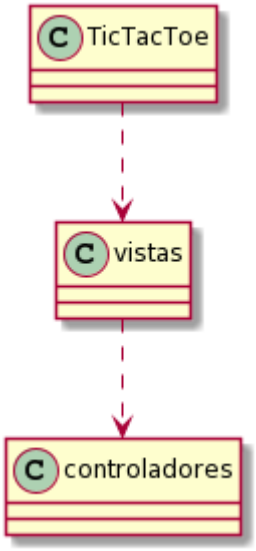
Se añaden los eventos y observadores, para la comunicación entre los paquetes, de cara a los cambios entre los distintos estados de la aplicación.

Grandes refactorings

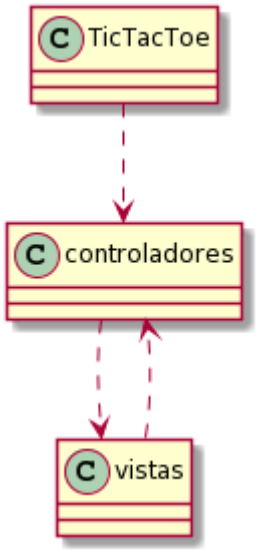
Dentro de las refactorizaciones que se realizan a lo largo de todas las versiones del proyecto, en la versión **mvp.pv** se realiza una refactorización en la que se revoluciona el diseño. En ella cambiamos que las vistas sean las que conozcan a los controladores y los usen, a que esta funcionalidad se produzca al contrario. En esta versión, los controladores conocen a las vistas y las utilizan solo para realizar las tareas propias de la interfaz. De esta forma, las clases con las que interactúa la clase principal son los controladores, en vez de las vistas, como en versiones anteriores.

Para realizar esta refactorización, se ha llevado a cabo a través de una serie de pasos.

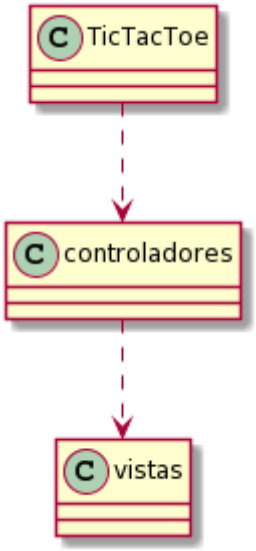
En las versiones anteriores la clase principal interactuaba con las vistas y éstas a su vez con los controladores, para comunicarse con los modelos.



Lo primero que se hace es cambiar los controladores para que sean las clases con las que interactúe la clase principal. Éstos llamarán a su vez a las vistas, las cuales, en este momento, interactuarán de nuevo con los controladores, como se hace en versiones antiguas, con lo que habríamos realizado uno de los cambios necesarios con la aplicación funcionando.



Seguidamente, moveremos toda la lógica de las vistas, que necesitaba de los controladores para comunicarse con los modelos, a los controladores, para que de esta manera las vistas no interactúen con los controladores y éstas solo realicen la comunicación de la interfaz con el usuario. Así, el cambio está completado sin necesidad de realizar todo en un solo paso.

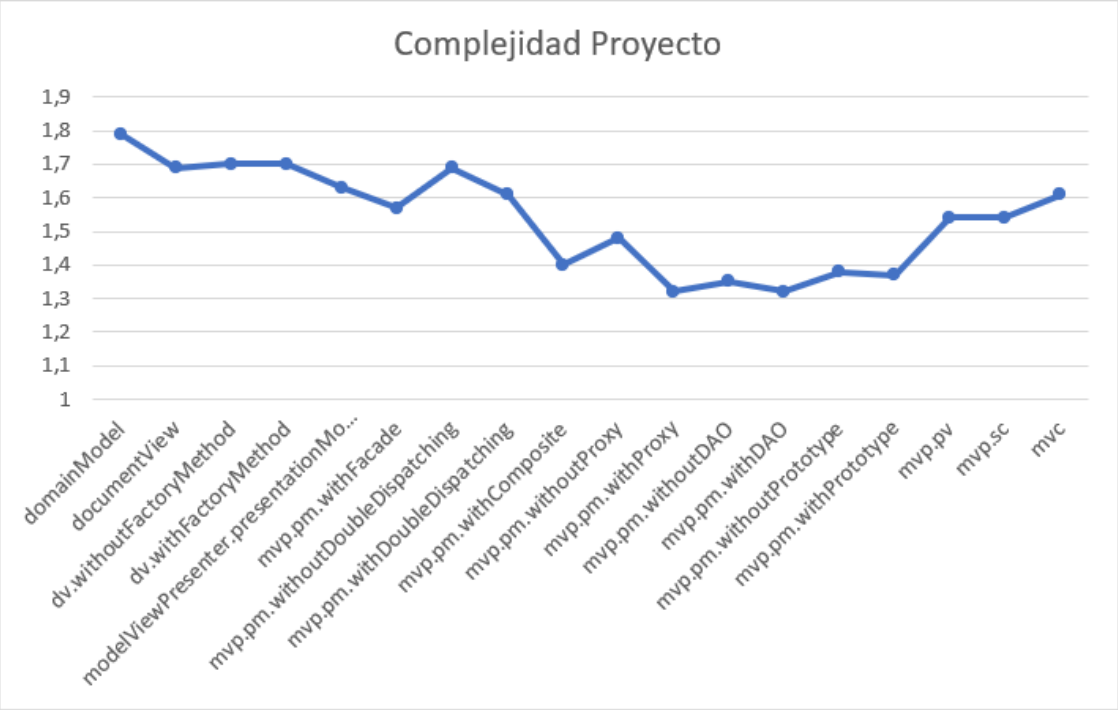


Métricas de calidad

Para obtener las métricas comentadas en esta sección, principalmente se han utilizado dos herramientas: SourceMonitor y VisualJArchitect.

Proyecto

Complejidad



Revisando la gráfica de complejidad de cada una de las versiones que se han ido incluyendo en el proyecto, podemos ver que la tendencia del proyecto ha sido a mejorar la complejidad media. Podemos visualizar como en nuestra primera versión “domainModel” tenemos una complejidad de 1,79 y en la última versión, antes de cambiar a la estructura “mvp.pv”, acabamos con 1,37. La tendencia es muy positiva.

Otro tema que se puede observar, es que en las versiones anteriores a añadir una técnica o patrón (mvp.pm.wihoutDoubleDispatching, mvp.pm.withProxy…) los valores de complejidad dibujan “picos”, visualizando que es necesaria alguna acción sobre el código para bajar esa complejidad. Con lo que podemos certificar la importancia de la inclusión de estas técnicas y/o patrones, con las que conseguimos mejorar nuestro proyecto en cuanto a complejidad.

Entre las versiones mvp.pm.withDoubleDispatching y mvp.pm.Composite, la complejidad baja considerablemente. Lo que se debe a que, en esa transición, se elimina toda la parte de interfaz gráfica.

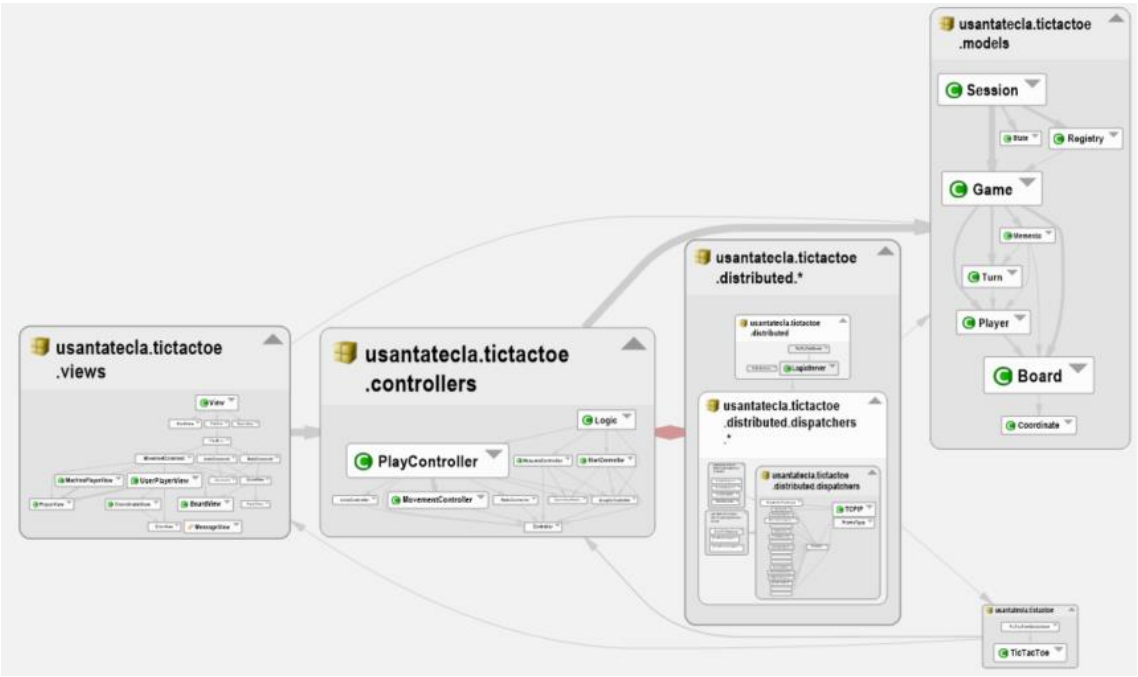
A partir de la versión mvp.pv, vemos como la gráfica aumenta, ya que estas versiones no poseen de los patrones que habíamos añadido en las anteriores.

Paquete

Acoplamiento

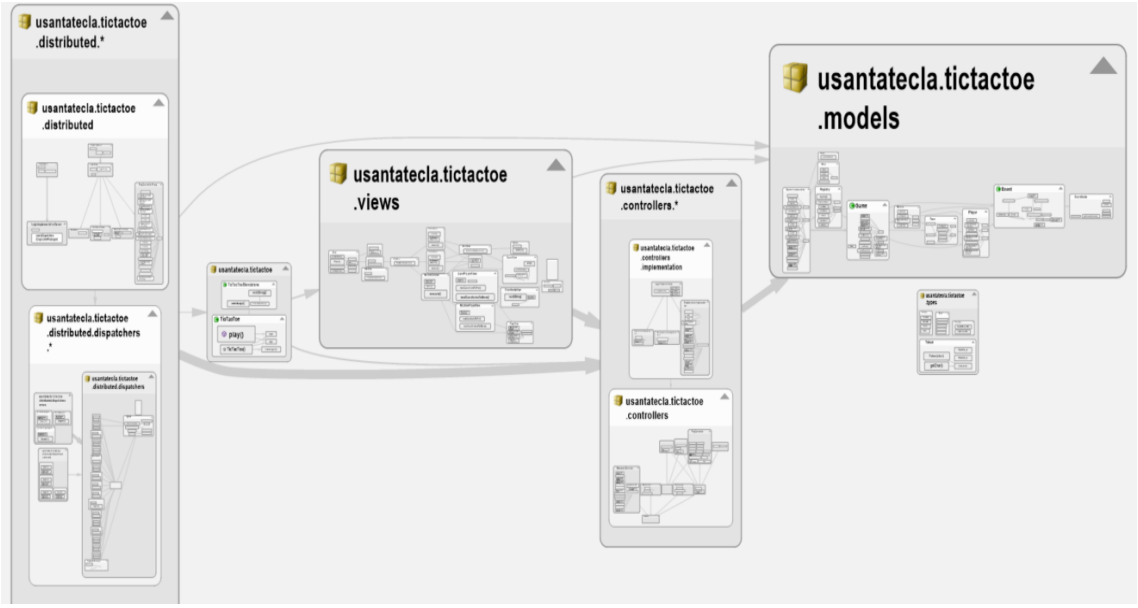
En las versiones mvp.pm.withoutProxy y mvp.pm.withProxy, podemos encontrar un buen ejemplo de reducción del acoplamiento, en el que además, se consigue eliminar una dependencia cíclica.

En el siguiente gráfico, vemos que tenemos una dependencia cíclica entre el paquete controllers y distributed.

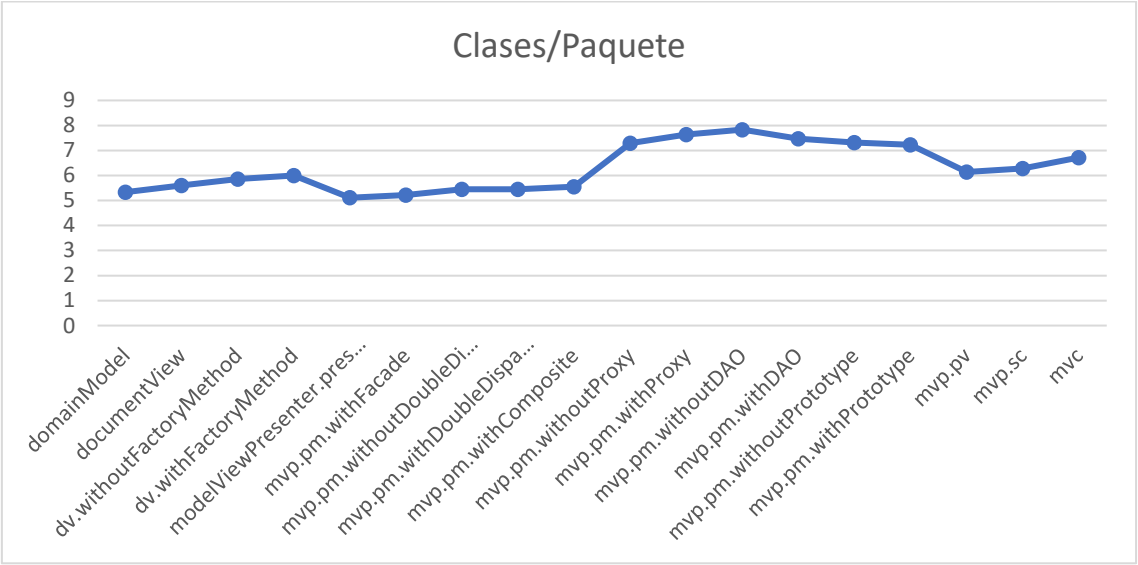


Esto es debido a que hay clases que heredan desde el paquete distributed a clases del paquete controllers, y en el paquete controllers se usa la clase TCPIP, para agregar la lógica distribuida a la standalone.

Con el patrón Proxy, vamos a eliminar esta dependencia cíclica, y bajar el acoplamiento del paquete controllers. Ya que añadimos abstracción a nuestras clases, con lo que conseguimos separar la implementación standalone de la distribuida, y con ello no necesitamos usar TCPIP en los controladores, eliminando esa dependencia. Podemos apreciarlo en la siguiente imagen.



Tamaño



Podemos observar como a partir de la versión mvp.pm.withoutProxy, el tamaño medio de los paquetes aumenta y, más o menos, se mantiene hasta la versión mvp.pm.withPrototype. Esto es debido a que se añaden las clases Dispatchers, que son muy numerosas y se almacenan en el paquete dispatchers.

Clase

Complejidad

Uno de los ejemplos que podemos observar de cómo se ha mejorado la complejidad en algunas clases, es cuando aplicamos Double Dispatching. Con ello conseguimos bajar la complejidad de la clase GraphicsView de 5 a 2 y ConsoleView de 4 a 1, añadiendo ControllersVisitor.

Otro ejemplo que podemos revisar es el caso de añadir el patrón Proxy, donde conseguimos que, clases controladoras, como PlayController, cambie de tener una complejidad de 43, a conseguir que su implementación en la clase PlayControllerImplementation sea de 21, ya que separamos la implementación distribuida y standalone del controlador, que en la versión anterior se realizaba mediante lógica "if else".

Cohesión

Para la medición de la cohesión, mediremos la falta de cohesión de las clases, que tomaremos como ejemplo, mediante la siguiente fórmula:

LCOM = 1 – (sum(MF)/M*F)

Donde cada uno de los datos se definen de la siguiente manera:

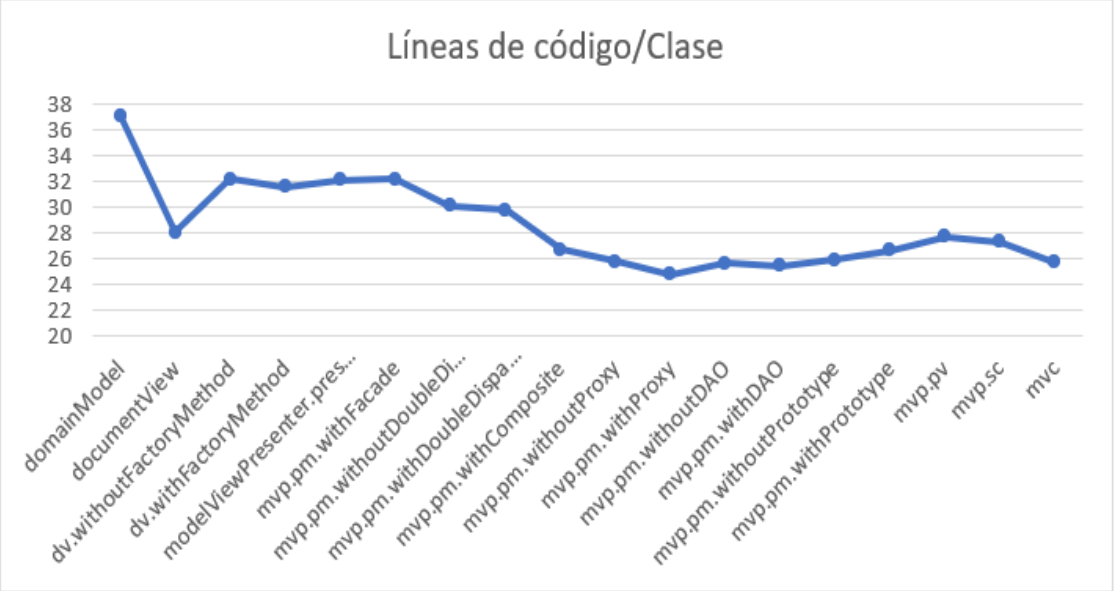
- **M** es el número de métodos en la clase (se cuentan tanto los métodos estáticos como los de instancia, incluye también los constructores, los que obtienen/establecen propiedades, los que añaden/quitan eventos)
- **F** es el número de campos de instancia en la clase.
- **MF** es el número de métodos de la clase que acceden a un campo de instancia determinado.
- **Sum(MF)** es la suma de MF sobre todos los campos de instancia de la clase.

Podemos ver una diferencia de cohesión entre las versiones en las que se realiza la separación de las vistas de los modelos. En la versión domainModel, la falta de cohesión que obtenemos en la clase Board es de 0,62, mientras que al realizar la separación de las vistas en documentView es de 0,54, con lo que conseguimos una mayor cohesión de nuestra clase.

Acoplamiento

Un ejemplo en el que se ha podido bajar el acoplamiento eferente de una clase, lo podemos observar al avanzar a la versión documentView. En la versión anterior, domainModel, podemos observar que la clase TicTacToe depende de una gran cantidad de clases, hasta 7, de las cuales se compone y usa. Como podemos visualizar en el siguiente gráfico. Mediante el progreso a documentView, y la inclusión de las vistas, cambiamos totalmente el acoplamiento, repartiéndolo, y consiguiendo bajarlo en la clase TicTacToe a que solo dependa de 2 clases, como visualizamos en la imagen. Esto se consigue, a parte de con la inclusión de las vistas con la clase View, con lo que conseguimos que no dependa de WithConsoleView, con el añadido de la clase Game, la cual tiene la función de gestionar los modelos.

Tamaño

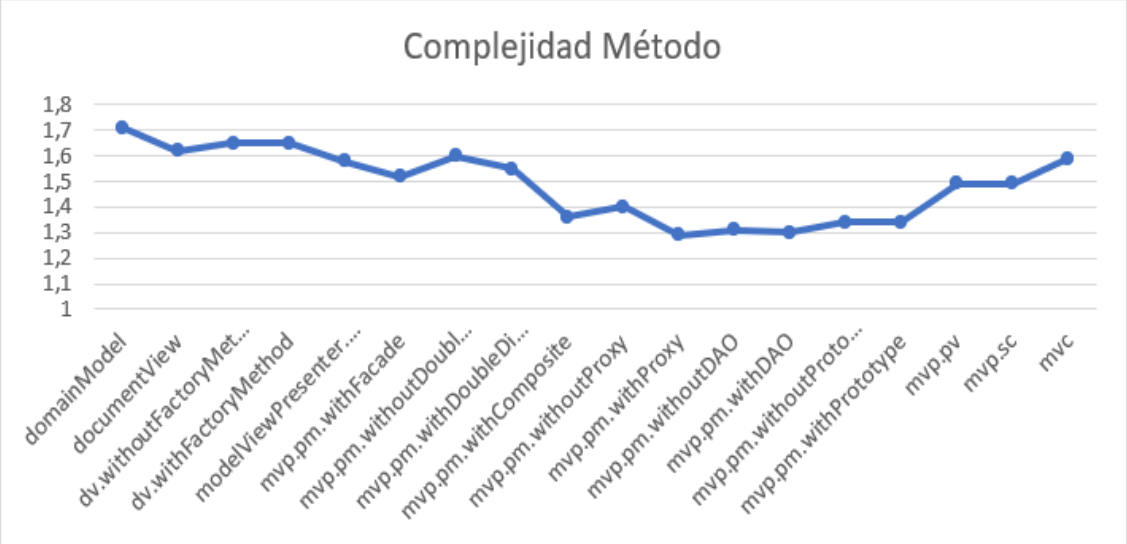


En la gráfica se aprecia que, desde la primera versión, el número de líneas que contienen las clases, han ido disminuyendo. La tendencia es a la baja.

En la versión documentView, se puede ver que, al separar las vistas de los modelos, conseguimos que las clases sean más pequeñas y se distribuya la lógica, bajando el tamaño. En las versiones posteriores, el tamaño aumenta, debido a la inclusión de las clases propias de la interfaz gráfica, las cuales tienen un tamaño mayor. Observamos ya en la versión mvp.pm.withComposite, que la gráfica baja considerablemente, debido a la eliminación de esta parte gráfica.

Método

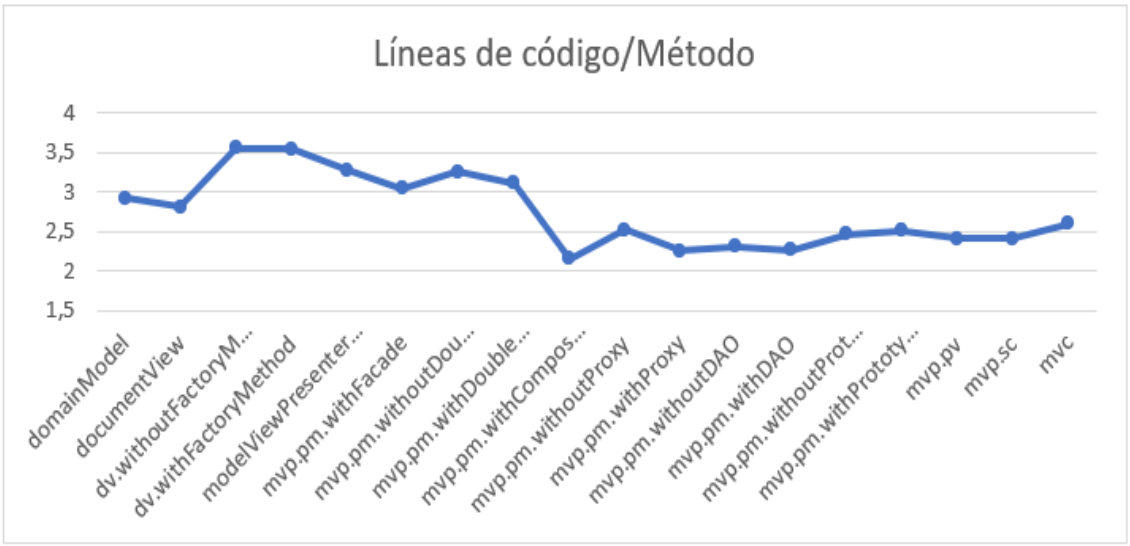
Complejidad



Se puede observar que la tendencia del progreso en las versiones es a la disminución de la complejidad de los métodos. Sobre todo, se aprecia que, cada vez que añadimos un nuevo patrón, lo normal es que consigamos bajar la complejidad.

Las versiones a partir de mvp.pv tienen una lógica distinta y no usan los patrones implementados anteriormente, por lo que cambia la tendencia comentada.

Tamaño



Se puede apreciar como la gráfica aumenta y disminuye con las versiones que contienen interfaz gráfica, ya que para implementar la visualización con las librerías de java se usan métodos con varias líneas.

Conclusiones y líneas futuras

Con esta práctica se ha conseguido el objetivo que se quería desde un principio: aplicar distintas técnicas y patrones a una aplicación para mejorar y cambiar su diseño, mientras se va añadiendo nueva funcionalidad y pruebas. Todo esto para, adicionalmente, poder analizar cómo cambia nuestra aplicación mediante métricas y diagramas. De esta forma, lograr demostrar la importancia de un buen diseño. Que la refactorización es necesaria, para obtener un código ordenado y con mayor sentido. Y que, para ello, las pruebas son necesarias, ya que nos permiten realizar estos cambios con mayor seguridad.

¿Qué se ha conseguido?

Se ha conseguido un trabajo que ha llegado hasta las 2938 líneas (sin contar los tests), 116 clases (todo esto en la versión más grande del proyecto, habiéndose desarrollado 18 versiones) y en el que se ha ido consiguiendo mejorar su estructura, mediante refactorizaciones y el añadido de nuevos patrones. También se ha realizado el añadido de funcionalidad nueva en algunas versiones, tanto funcional como no funcional. Partimos de un código donde no había ningún reparto de responsabilidad, sin abstracción, con una alta complejidad…Y mediante el avance de las versiones, vamos consiguiendo, aparte de añadir nueva funcionalidad, ordenar nuestro código, repartir la responsabilidad, bajar la complejidad, el acoplamiento etc.

En adición al código, se ha logrado ir documentando cada una de las versiones de la aplicación mediante AsciiDoc y la inclusión de diagramas UML, ejemplos de funcionalidad de las versiones y características de cada una.

Evolución del software

El proyecto se comenzó con un programa que se componía de 15 clases y 555 líneas, sin tener una estructura con clases controladoras y vistas separadas de la lógica de los modelos. Una aplicación que no tenía la posibilidad de ejecución de forma distribuida, en la que no se realizaba almacenamiento de información en ficheros o BBDD, de cara a recuperarla en un futuro, y sin la funcionalidad, de cara al usuario, de deshacer y rehacer las acciones que se realicen en el programa. Todo esto en conjunción con los distintos patrones y reestructuraciones que se han conseguido incluir (patrón DAO, Proxy…). Se ha llegado a evolucionar un proyecto, durante 18 versiones, hasta conseguir todo lo mencionado anteriormente, consiguiendo mejoras en métricas como acoplamiento, cohesión, complejidad… Sin olvidarnos de la importancia de la nueva funcionalidad, de cara al usuario. El proyecto empezó muy simple y se ha ido evolucionando hasta conseguir un proyecto mucho más complejo, añadiendo adicionalmente, documentación y pruebas para cada una de las versiones.

Líneas futuras

En cuanto a la evolución del proyecto, para próximas adaptaciones y mejoras, se podrían añadir los siguientes puntos:

- Cambiar la aplicación para que funcione con el framework Angular, de cara a añadir una interfaz web con la que obtenemos muchas posibilidades. El movimiento de las fichas se podría realizar mediante drag & drop.
- Conectar la aplicación a un backend desarrollado con node.js y desplegado en AWS mediante Lambdas. Almacenando datos en BBDD DynamoDB. Como añadido, se podrían crear usuarios, para el almacenamiento de los resultados de partidas ganadas, y así agregar una clasificación.
- Como necesidad de despliegue de la aplicación y de ejecución de las pruebas de una manera automática, se puede usar github actions. En caso de desplegar la aplicación, habiendo sido adaptada a Angular, se puede desplegar en un S3 con un CloudFront, mediante AWS.