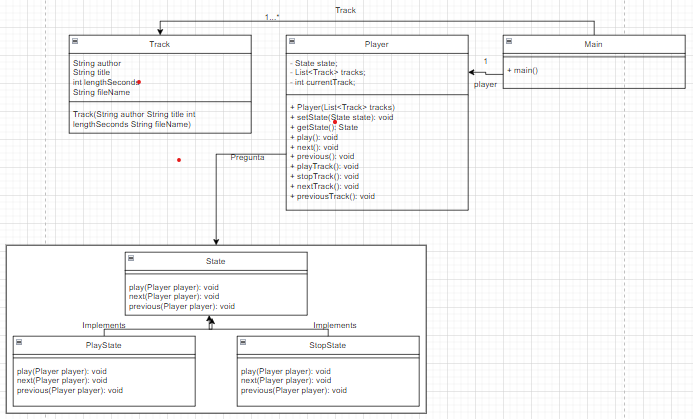
ESTRUCTURA

El proyecto cuenta con 3 clases principales “Main”, “Player” y “Track”. “Main” se encarga de crear una lista de canciones con el formato adecuado y además inicializa un objeto de la clase “Player” se encarga de seleccionar la canción que se está reproduciendo y utilizar los métodos que representan las acciones como adelantar la canción, pausarla, entre otras, dependiendo del estado en el que se encuentre. Por último, la clase “Track” se encarga de crear una canción cada que es llamada.

Ahora bien, para el patrón se usan tres clases secundarias “State”, StopState” y “PlayState”. La primera es el mediador entre las clases principales y los dos estados que se van a utilizar para el reproductor. Las otros dos se encargan de modificar los métodos de “State” de acuerdo con su funcionalidad. Por obvias razones aquí es donde se encuentra el patrón a analizar.



PARA QUE SIRVE

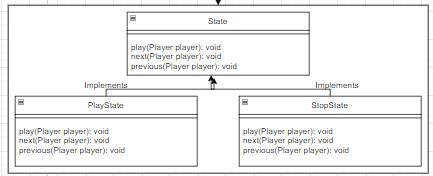
El proyecto trata de crear un reproductor de música con diferentes acciones posibles mientras se reproduce, o no, una canción que cuenta con diferentes acciones (botones) que actúan de una forma distinta dependiendo del estado en el que está.

QUE GRANDES RETOS DE DISEÑO ENFRENTA

El proyecto es muy corto para realmente resolver el problema, es decir, pueden aparecer muchos problemas en la ejecución de la aplicación dado que hay un numero importante de funciones que no se tienen en cuenta, como la doble funcionalidad del botón “previous” no solo para cambiar la canción a la anterior sino para reiniciar la que se está escuchando actualmente.

En adición, no cuenta con un sistema practico para añadir mas canciones al catalogo pues toca hacerlo manualmente desde “Main”, lo que con mas canciones se podría volver un problema. Por otro lado, tampoco hay un sistema para el caso en el que se acaba la lista de canciones, por lo que podría haber un error en ese momento pues el programa se acabaría de improviso.

INFORMACION Y EXPLICACIÓN DEL PATRÓN EN EL CODIGO



Como se puede ver en esta imagen y en el diseño general, el proyecto cuenta con tres clases que pertenecen al patrón. “State” es a la que accede la aplicación en general como se puede observar en el código, sin embargo, esta tiene dos posibilidades y es que puede estar reproduciendo la canción o tenerla pausada. Dependiendo de esto el método que se utilice cambia de funcionalidad, es decir, actúa como un condicional para el valor “state”. Por ejemplo, la acción “play” cuando está en “PlayState” pausa la canción que se este reproduciendo, mientras que cuando está en “StopState” reproduce nuevamente la canción, o esa es la intención.

INFORMACION GENERAL DEL PATRON

El patrón utilizado se llama *STATE* y consiste en utilizar subclases que representan posibles estados en los que puede estar un objeto pensando en que estos estados pueden y van a cambiar entre ellos en el transcurso de la ejecución de la aplicación. Esto es especialmente conveniente cuando se quiere generar diferentes resultados a las mismas acciones si se parte de diferentes estados.

Este patrón es utilizado usualmente cuando se quiere evitar condicionales muy largos o demasiadas líneas de código repetidas en diferentes secciones. Este patrón es especialmente conveniente pues permite a varias clases como estados heredar los mismos métodos y modificarlos a conveniencia para reducir el código y que sea más fácil tanto comprenderlo como implementarlo cuando la extensión del proyecto es muy grande.

QUE VENTAJAS TIENE

Este patrón permite reducir el código significativamente al eliminar condicionales extensos y, en teoría, en este proyecto realiza eso al evitar complicar la lógica de la aplicación transfiriendo esas responsabilidades a las clases que representan los estados.

QUE DESVENTAJAS TIENE

En este caso el proyecto es muy corto como para que el patrón realmente resuelva los problemas que se propone. En general, para proyectos mas complejos y con condicionales muy grandes el patrón es capaz de reducir el código sustancialmente, sin embargo, en este caso considero que complica más la lógica pues los métodos que realmente cambian de un estado a otro son pocos, por lo que no vale la pena usar este patrón pensando en los problemas que resuelve contras los que genera.

DE QUE OTRA FORMA PODRÍA SOLUCIONAR

Para este caso particular yo me devolvería a los condicionales de siempre dado que no serían tan grandes, al contrario, en vez de crear una clase nueva y tener que definir métodos para que realicen diferentes acciones a través de las tres clases de estado se podría usar simplemente un condicional que dependa de una variable llamada “state” que defina el funcionamiento de los botones en ese estado especifico.

URL PROYECTO

<https://github.com/android-code/design-patterns.git>

en la sección de state

Aunque también incluí el código relevante en una carpeta adjunta en este repositorio