Eric Alarcon Dolynko 202220287

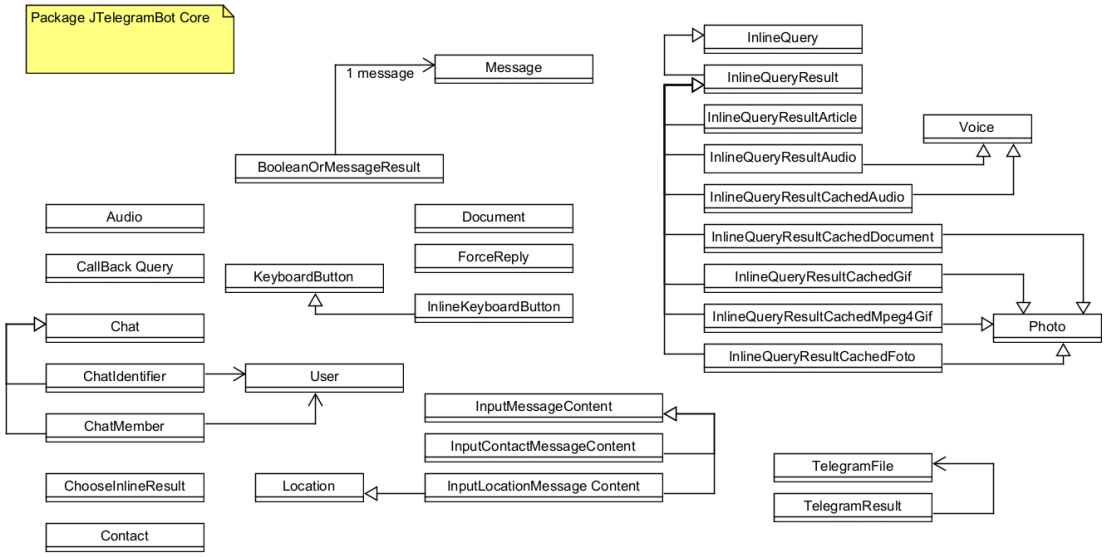
**Taller 5 DPOO**

El patrón que se va a estudiar en este taller va a ser el patrón de Builder Design Pattern, y se va a realizar un análisis del siguiente repositorio: https://github.com/Eng-Fouad/JTelegramBot.git

## 1. Información general del proyecto

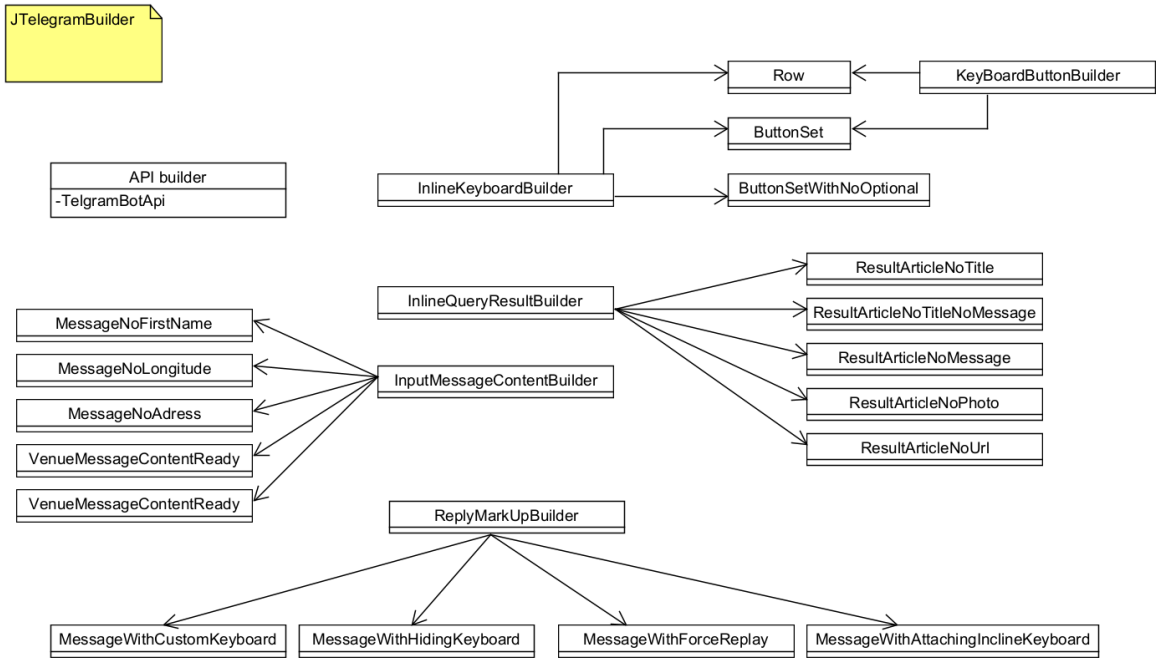
El propósito de este proyecto es de facilitar el uso del API de telegram a través de un API que fue diseñado en el repositorio en java. Es decir que el repositorio busca que el menú de opciones que ofrece el API de telegram sea más corto y efectivo. Cabe recalcar que este proyecto es una librería. Ahora miremos como esta construido el proyecto a través de un diagrama de clases UML:

*Primer Package: Telegram Core*




Podemos observar que en este paquete están los aspectos claves de las funcionalidades del API, es relevante mencionar que en este diagrama de clases no se encuentran todas las funcionalidades que ofrece el repositorio ya que es demasiado extenso para poder agruparlas en un único UML, no obstante se encuentra la estructura general del paquete. No obstante, lo que no se incluye en el UML son diferentes Input Message Content, chats o Inline Query Results que se comportan similar a las demás clases de su mismo tipo.

*Segunda Imagen: Telegram Builder*



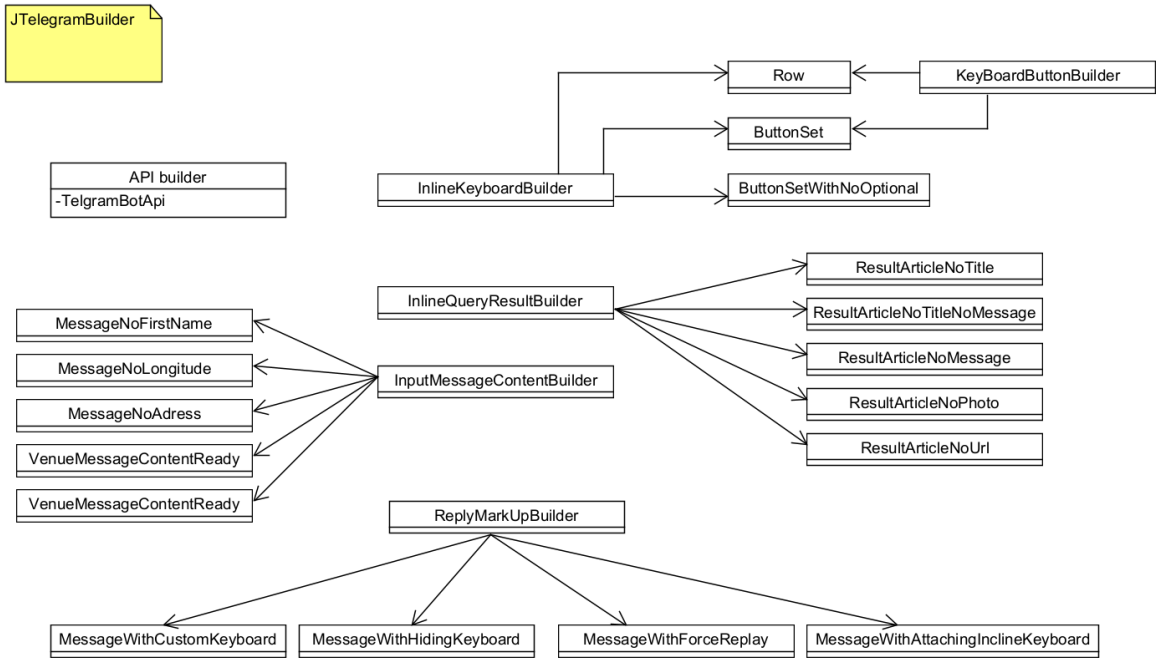

Este paquete se usa exclusivamente para crear las clases que se enunciaron en el primer programa. Es importante que note que las clases que contienen a varias clases son aquellas en las que en el primer diagrama varias clases heredan a ellas, y las demás clases son los hijos de esta clase. Ahora, es vital comprender que en este diagrama de clases no es que “InlineKeyboardBuilder” tenga como atributo otra de las clases, sino que de forma explícitamente tienen el constructor de esas nuevas clases dentro de “InlineKeyboardBuilder” y así para las demás clases que contienen varias clases.

Finalmente con respecto a la estructura general de la librería, hay una clase individual extra que se llama NegativeResponseException.java que simplemente avisa cuando ocurre un error de comunicación entre el computador desde el cual se usa el API y el telegram bot.

Por lo tanto, note que hay un reto grande de acuerdo a las metas de ellos y es que el API de telegram tiene bastantes funcionalidades pero son funcionalidades específicas, por lo que si se quiere que haya una Inline Query Result de un video que no tiene imagen pero si audio, y además está en formato MP3, se tiene que crear una instancia de esa clase en específico, lo cual puede generar que el programa se vuelva poco intuitivo para un usuario que no conoce de todo lo que ofrece la librería.

**2. Información y estructura del fragmento donde aparece el patrón.**

El fragmento en el que aparece el patrón es precisamente en el JTelegramBuilder de la siguiente imágen:




Esto se debe a que el patrón utilizado para solucionar la problemática previamente planteada es el Builder Design Pattern. Antes de profundizar en el patrón, es importante intuir que el package de builder lo que intenta hacer es que solo se tenga que crear una instancia personalizada de las clases padres del código del otro paquete como lo es una Inline Query Result. De tal manera que el usuario no tenga que crear un Inline Query Result específico, sino que lo crea desde el builder y lo va personalizando dependiendo de las necesidades que este tenga.

**3. Descripción del patrón**

El Builder Design Pattern del libro de GoF el cual se utiliza para crear un objeto complejo a través de una serie de pasos. De tal manera que separa al objeto de sus representaciones y permite que se pueda crear diferentes representaciones de un objeto a través del mismo constructor. Lo cual facilita la creación de este tipo de objetos para el usuario. Este patrón suele tener 4 componentes claves:

a) “Director”

Este componente se encarga de organizar la manera y el orden en el que se va a construir el objeto complejo.

b) “Builder”

Este componente suele ser una clase abstracta o una interfaz que define las partes de la construcción del objeto, de tal manera que cada subclase o “concrete builder” haga la implementación necesaria.

c) “Concrete Builder”

Este componente se encarga de la manera de implementar los diferentes aspectos y representaciones que tiene el objeto complejo, es decir es la lógica detrás de los llamados del “Builder”.

d) “Product”

Finalmente el producto es un objeto complejo en sí el cual está siendo construido.

**4. Información aplicada al proyecto**

Ahora para ver que el patrón está presente en el proyecto, basta mencionar que los 4 componentes claves del patrón están implementados en el diseño del repositorio. Por ende, vamos a tomar cualquiera de las clases que contienen varias clases del diagrama número 2, en particular el Inline Query Result Builder. Note que la clase Inline Query Result, que está presente en el primer diagrama corresponde al producto u objeto complejo que se quiere crear. Para lograr esto, se tiene que llamar al “Director” que en este caso es la propia clase del Builder, y el rol de director es el orden establecido de las diferentes subclases que tienen este objeto. Ahora, el Builder se puede notar trivialmente que es el constructor de la clase Inline Query Result Builder y los “Concrete Builder” son las clases que contiene el constructor como Result Article No Title, Result Article No Title No Message, etc… Por lo tanto, sin perder la generalidad de las demás clases se puede ver claramente que se cumple el patrón y está presente en este proyecto.

**5. ¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?**

Tiene sentido haber utilizado este patrón porque el motivo del proyecto era simplificar el API de Telegram y de esta manera a través del Builder Design Pattern, simplemente se puede personalizar los aspectos y representaciones de un objeto complejo desde una sola clase en vez de tener que llamar a varias clases específicas distintas lo que tiene una ventaja de facilidad para el usuario de poder utilizar la API del repositorio de github en contra del bot de Telegram. Por ende, este patrón les ayuda a completar la meta de su librería.

**6. ¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?**

El patrón también trae consigo varias desventajas tales como aumentar la complejidad del código, y repetición del código. Por ejemplo, note que para crear una instancia de un Inline Keyboard, o solo un Keyboard los constructores son similares pero el Inline tiene un método adicional, no obstante, se decide volver a escribir todo un constructor para el Keyboard normal. Por ende, se está escribiendo bastante más código y muchas más clases que pueden ralentizar la eficacia de la librería. Adicionalmente, se puede complicar la creación de objetos simples por las mismas razones que ya expliqué. Finalmente otra desventaja de haber utilizado este patrón es que hay una inflexibilidad ante el cambio de aspectos del objeto, dado que en la implementación actual si se quiere cambiar un detalle de la representación cuando ya se creó el objeto, esto no se puede hacer y se tendría que crear un nuevo objeto o modificar las clases del package Builder para poder modificar aspectos después de la creación.

**7. ¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?**

Otra forma en la que se pudo haber solucionado este caso en particular se pudo también haber utilizado el Factory Method pattern con un par de modificaciones en el código base de la primera imágen. Puesto que el propósito del factory Method pattern es poder tener un objeto y que luego otras

subclases puedan instanciarlo de la manera que lo necesite. Por ende, si el usuario quiere un Inline Query Result de un video, simplemente lo tenga que instanciar, no obstante, esto no sería tan eficiente puesto que igualmente nos encontramos el problema de tener que recurrir a más subclases específicas cada vez que quiera crear un objeto. Es decir el patrón de Builder Design es más eficiente, pero igual el Factory Method puede llegar a solucionarlo también.