Design Pattern Examples in Telegram

Team Foobar: Xiling Li, Yijia Zhang, Bingfei Zhang

1. Introduction

A software design pattern is a general solution for a kind of repeat occurring problem. It is not a complete solution for the problem, however, it is more a template or a description that can be used under different circumstances. Design pattern is usually an optimized practice for a particular scenario. As design patterns have been researched and used for a long time by experts with trial and error, it is always the easiest and best way for a common problem. Also, design patterns help developers to communicate easier. With a given design pattern, the developers can understand the object instantly.

To learn design patterns, we found 5 design patterns from our project Telegram and analyzed them. In the next part of the report, each of the 5 patterns is introduced. In each part, the explanation of the design pattern is detailed. Then the example of the design pattern in our project and the reason for using the design pattern is discussed follows. At the end of the report, we made a conclusion about how our project benefits from the design pattern.

2. Design patterns

2.1 Observer

The observer design pattern helps with defining a subscription mechanism which alerts subscribers on any changes on observed entity or events. It is usually used when there is a one-to-many relationship. It provides loosely coupled designs and provides the project more flexible with changing.

In Telegram for Android project, a large notification system is implemented and used. The NotificationCenter class is implemented as a singleton, which stores states of all subscribers.

```
public void addObserver(NotificationCenterDelegate observer, int id) {...}

public void removeObserver(NotificationCenterDelegate observer, int id) {...}
```

Add/Remove observer methods and a delegate interface inside Notification Center class

In NotificationCenter, addObserver method, as shown in pic below, adds the specified type of observer to the list of observers, which are essentially subscribers of different types of subscribers. The ArrayList addAfterBroadCast is declared as a private instance variable, while NotificationCenter itself is, again, a singleton.

```
public void addObserver(NotificationCenterDelegate observer, int id) {
    if (BuildVars.DEBUG_VERSION) {
        if (Thread.currentThread() != ApplicationLoader.applicationHandler.getLooper().getThread()) {
            throw new RuntimeException("addObserver allowed only from MAIN thread");
        }
    }
    if (broadcasting != 0) {
        ArrayList<NotificationCenterDelegate> arrayList = addAfterBroadcast.get(id);
        if (arrayList == null) {...}
        arrayList.add(observer);
        return;
    }
    ArrayList<NotificationCenterDelegate> objects = observers.get(id);
    if (objects == null) {...}
    if (objects.contains(observer)) {...}
    objects.add(observer);
}
```

AddObserver Method implementation

Those that need to make use of the publish-subscribe services would implement the interface NotificationCenterDelegate to override the didReceiveNotification method so that they can define their own on-notification behavior.

```
public class DownloadController extends BaseController implements NotificationCenter.NotificationCenterDelegate {
```

```
  @Override
  public void didReceivedNotification(int id, int account, Object... args) {
      if (id == NotificationCenter.fileDidFailToLoad || id == NotificationCenter.httpFileDidFailedLoad) {
          String fileName = (String) args[0];
          Integer canceled = (Integer) args[1];
          listenerInProgress = true;
          ArrayList<WeakReference<FileDownloadProgressListener>> arrayList = loadingFileObservers.get(fileName);
          if (arrayList != null) {
              for (int a = 0, size = arrayList.size(); a < size; a++) {
                  WeakReference<FileDownloadProgressListener> reference = arrayList.get(a);
                  if (reference.get() != null) {...}
              }
              if (canceled != 1) {
                  loadingFileObservers.remove(fileName);
              }
          }
          listenerInProgress = false;
          processLaterArrays();
          checkDownloadFinished(fileName, canceled);
      } else if (id == NotificationCenter.fileDidLoad || id == NotificationCenter.httpFileDidLoad) {...} else if (id ==
```

Example subscriber class and an overridden didReceivedNotification method

Finally, the postNotificationNameInternal method would invoke the didReceivedNotification method on the observers to trigger intended behaviors on subscriber items.

```
@UiThread
public void postNotificationNameInternal(int id, boolean allowDuringAnimation, Object... args) {
    if (BuildVars.DEBUG_VERSION) {...}
    if (!allowDuringAnimation && animationInProgress) {...}
    broadcasting++;
    ArrayList<NotificationCenterDelegate> objects = observers.get(id);
    if (objects != null && !objects.isEmpty()) {
        for (int a = 0; a < objects.size(); a++) {
            NotificationCenterDelegate obj = objects.get(a);
            obj.didReceivedNotification(id, currentAccount, args);
        }
    }
    broadcasting--;
    if (broadcasting == 0) {...}
}
```

Notify()

2.2 Builder

Builder is a design pattern that allows creating different objects of the same type with distinct features and functions step by step when there are various fields and parameters that can be chosen from and customized for special needs. This design pattern alleviates the pressure to perfectly instantiate the target object at the very beginning, and with the step-by-step construction, developers can easily make adjustments in the future when necessary.

The AlertDialog class has a builder that one can set various properties to an alert dialog that is building. Inside the AlertDialog class there is a dedicated Builder class (Line 1046 - 1178) that is used to add information (e.g. titles, messages) or features (e.g. views, buttons) as needed.

```
public static class Builder {

    private AlertDialog alertDialog;

    protected Builder(AlertDialog alert) { alertDialog=alert; }

    public Builder(Context context) { alertDialog = new AlertDialog(context,  progressStyle: 0); }

    public Builder(Context context, int progressViewStyle) {
        alertDialog = new AlertDialog(context, progressViewStyle);
    }
```

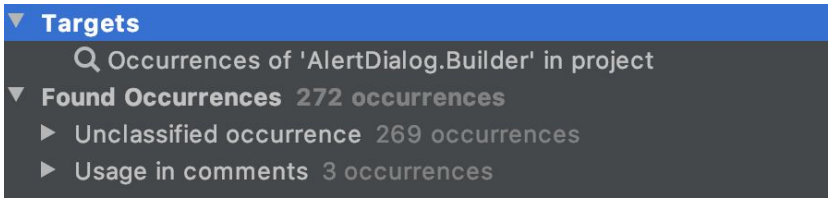
Constructors of AlertDialog

```
public Builder setTitle(CharSequence title) {
    alertDialog.title = title;
    return this;
}

public Builder setSubtitle(CharSequence subtitle) {
    alertDialog.subtitle = subtitle;
    return this;
}
```

```
public Builder setPositiveButton(CharSequence text, final OnClickListener listener) {
    alertDialog.positiveButtonText = text;
    alertDialog.positiveButtonListener = listener;
    return this;
}

public Builder setNegativeButton(CharSequence text, final OnClickListener listener) {
    alertDialog.negativeButtonText = text;
    alertDialog.negativeButtonListener = listener;
    return this;
}
```

Some examples of fields that can be set step by step

With the Builder design pattern the AlertDialog has become a powerful alert that can be used in so many different scenarios. Across the system it can be seen that there are over 200 occurence of the AlertDialog User (with most of them intending to instantiate an alert dialog), from which it can be inferred that there are over 100 different kinds of different usage applied to the AlertDialog.



To name a few examples:

A. During LauncherActivity, if the app was unable to find the current location, it will build an alert dialog with a title, a message and two buttons with one basically doing nothing and the other one have vast logic in it to achieve manually setting location.

```
} else if (id == NotificationCenter.wasUnableToFindCurrentLocation) {
    final HashMap<String, MessageObject> waitingForLocation = (HashMap<String, MessageObject>) args[0];
    AlertDialog.Builder builder = new AlertDialog.Builder( context: this);
    builder.setTitle("Telegram");
    builder.setPositiveButton("OK",  listener: null);
    builder.setNegativeButton("Choose manually", (dialogInterface, i) -> {
        if (mainFragmentsStack.isEmpty()) {
            return;
        }
        BaseFragment lastFragment = mainFragmentsStack.get(mainFragmentsStack.size() - 1);
        if (!AndroidUtilities.isGoogleMapsInstalled(lastFragment)) {
            return;
        }
        LocationActivity fragment = new LocationActivity( type: 0);
        fragment.setDelegate((location, live, notify, scheduleDate) -> {
            for (HashMap.Entry<String, MessageObject> entry : waitingForLocation.entrySet()) {
                MessageObject messageObject = entry.getValue();
                SendMessagesHelper.getInstance(account).sendMessage(location, messageObject.getDialogId(), messa
            }
        });
        presentFragment(fragment);
    });
    builder.setMessage("App was unable to determine your current location");
    if (!mainFragmentsStack.isEmpty()) {
        mainFragmentsStack.get(mainFragmentsStack.size() - 1).showDialog(builder.create());
    }
```

B. If Telegram finds that you have many unsynced contacts, it will use an AlertDialog that asks you to decide what to do next. Apart from the message and button functions are different from the first example, it also has other different features: It has a BackButtonListener, and you cannot cancel the alert when touched outside the dialog.

```
AlertDialog.Builder builder = new AlertDialog.Builder( context: LaunchActivity.this);
builder.setTitle("Update contacts?");
builder.setMessage("Telegram has detected many unsynced contacts, would you...");
builder.setPositiveButton("OK", (dialogInterface, i) -> ContactsController.getInstance(account).syncPhoneBookByAlert(contactHashMap, first, schedule,  cancel: false));
builder.setNegativeButton("Cancel", (dialog, which) -> ContactsController.getInstance(account).syncPhoneBookByAlert(contactHashMap, first, schedule,  cancel: true));
builder.setOnBackButtonListener((dialogInterface, i) -> ContactsController.getInstance(account).syncPhoneBookByAlert(contactHashMap, first, schedule,  cancel: true));
AlertDialog dialog = builder.create();
fragment.showDialog(dialog);
dialog.setCanceledOnTouchOutside(false);
```

From the two examples above, it can be seen that there are different unique needs on every AlertDialog, but how they are built and their general actions can fall into categories. Developers can just combine the categories needed to construct a dedicated alert dialog. Therefore, using a builder pattern for AlertDialog is perfect for constructing these different alert dialogs step by step.

2.3 Memento

The Memento design pattern is used when there is a need to restore the previous state of an object. This design pattern stores related data outside of the object so that the object can be returned to the saved state later. This pattern provides easy restoration of the application status.

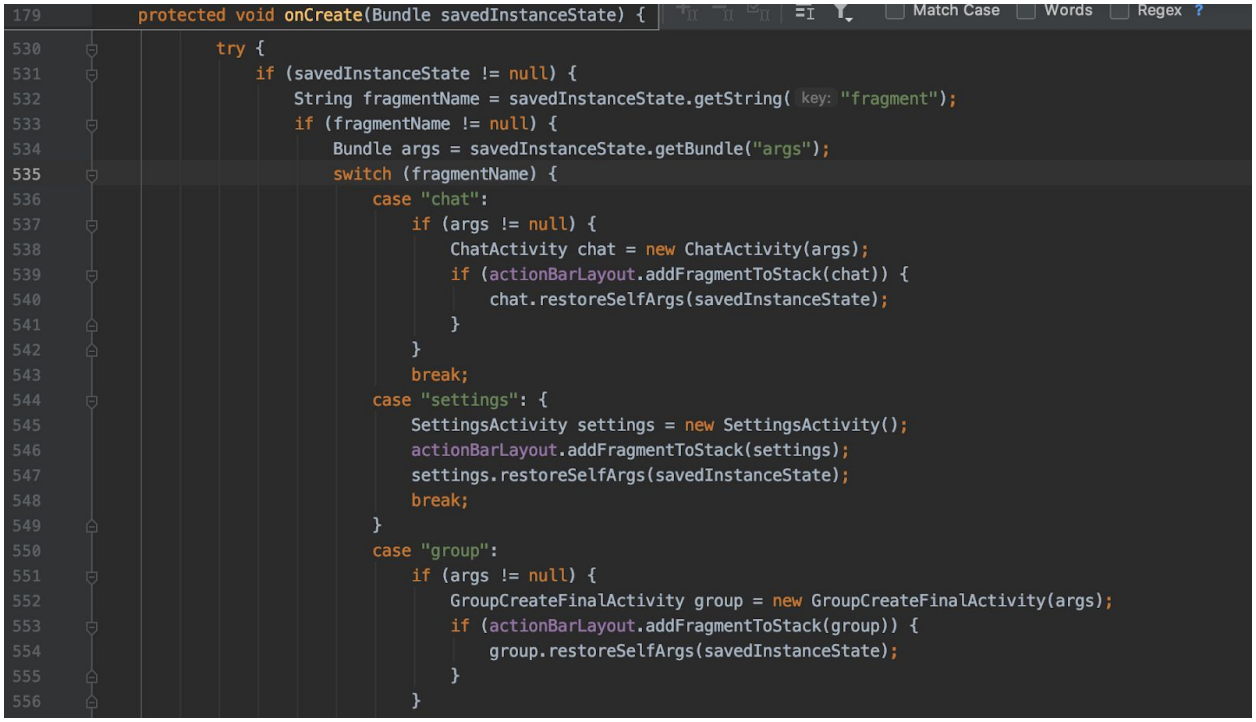
During the usage of the android application, the status of the application would be always stored when the application goes to the background or not on focus. The memento can help restore the previous of the application when the application is back to focus.

For example, in the launcherActivity, when Telegram goes to the background, the activity is stopped and Memento design pattern is used to store current states of the activity using a Bundle to store the last state of the view.

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    try {
        super.onSaveInstanceState(outState);
        BaseFragment lastFragment = null;
        if (AndroidUtilities.isTablet()) {
            if (!layersActionBarLayout.fragmentsStack.isEmpty()) {
                lastFragment = layersActionBarLayout.fragmentsStack.get(layersActionBarLayout.fragmentsStack.size() - 1);
            } else if (!rightActionBarLayout.fragmentsStack.isEmpty()) {
                lastFragment = rightActionBarLayout.fragmentsStack.get(rightActionBarLayout.fragmentsStack.size() - 1);
            } else if (!actionBarLayout.fragmentsStack.isEmpty()) {
                lastFragment = actionBarLayout.fragmentsStack.get(actionBarLayout.fragmentsStack.size() - 1);
            }
        } else {
            if (!actionBarLayout.fragmentsStack.isEmpty()) {
                lastFragment = actionBarLayout.fragmentsStack.get(actionBarLayout.fragmentsStack.size() - 1);
            }
        }

        if (lastFragment != null) {
            Bundle args = lastFragment.getArguments();
            if (lastFragment instanceof ChatActivity && args != null) {
                outState.putBundle("args", args);
                outState.putString("fragment", "chat");
            } else if (lastFragment instanceof SettingsActivity) {
                outState.putString("fragment", "settings");
            } else if (lastFragment instanceof GroupCreateFinalActivity && args != null) {
                outState.putBundle("args", args);
                outState.putString("fragment", "group");
            } else if (lastFragment instanceof WallpapersListActivity) {
                outState.putString("fragment", "wallpapers");
            } else if (lastFragment instanceof ProfileActivity && ((ProfileActivity) lastFragment).isChat() && args != null) {
                outState.putBundle("args", args);
                outState.putString("fragment", "chat_profile");
            } else if (lastFragment instanceof ChannelCreateActivity && args != null && args.getInt( key: "step") == 0) {
                outState.putBundle("args", args);
```

Saving state

And later while relaunching the app and restoring the previous state is expected, the launcher will launch based on the last state saved in the bundle and restore accordingly. The memento pattern helped to finish this process quickly, and the encapsulation of the data is kept.



Restoring state

2.4 Singleton

Singleton pattern of design is used when the system needs to ensure that a certain class can have only one created instance at a time. With this class being designed so that no more than one object instantiation is allowed, all components of the system that uses it shares the only instance that is created. Thus, a singleton is essentially global to the software system. To ensure the effectiveness of the singleton design, such classes usually block access to its default constructor and instead provide a static creation method that verifies its static states. They should also be thread-safe, as they provide global access with state.

ImageLoader class is in charge of all the functions that deals with receiving, loading and caching an image. Although its constructor is not declared as private, it provides a static creator method “getInstance()”, which checks on whether a static volatile instance of ImageLoader has been instantiated or not. If not, it creates a class instance in a synchronized block. As long as the public constructor is not used anywhere else in the system, the singleton design stands and is safe and sound. Quickly checking on the usages of the public constructor, it can be found that it is only used by the instance creator, which proves that the class exists as a singleton.

```
private static volatile ImageLoader Instance = null;

public static ImageLoader getInstance() {
    ImageLoader localInstance = Instance;
    if (localInstance == null) {
        synchronized (ImageLoader.class) {
            localInstance = Instance;
            if (localInstance == null) {
                Instance = localInstance = new ImageLoader();
            }
        }
    }
    return localInstance;
```

The creator method for ImageLoader class

To understand why the class adopts the singleton pattern, the system wise usage of the getInstance() method is checked. The finder shows *96* usages scattering everywhere in the system. This means that, when multiple components want to invoke ImageLoader's non-static methods at the same time, many identical ImageLoader objects have to be created in the memory, which is apparently inefficient. While the instance is thread-safe, one should certainly utilize the static global access to it. Also, since some of the methods entails reading/writing from/to memory (caching), having a single volatile instance helps controlling concurrent access, making the components' behaviors more predictable.

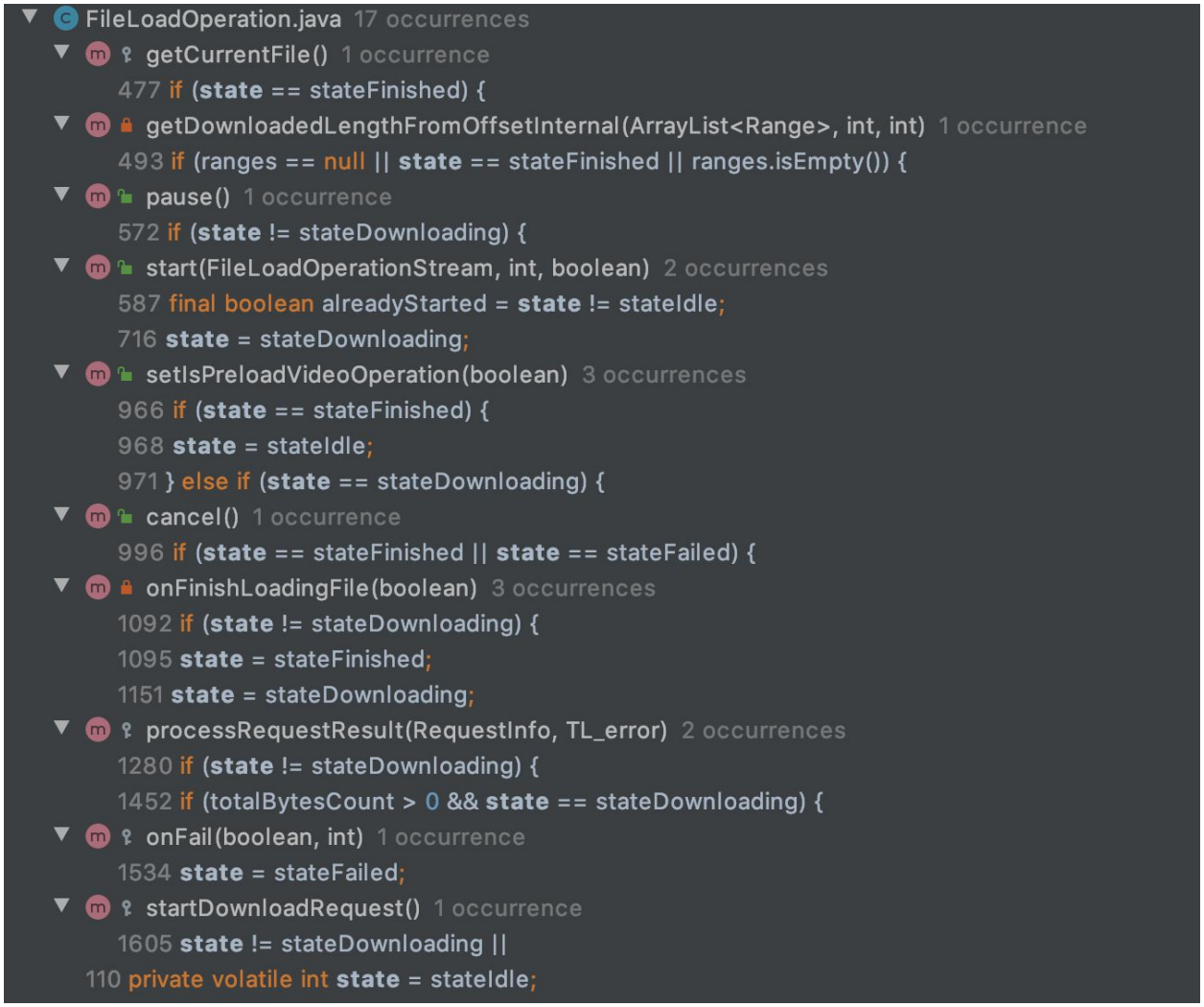
2.5 State

State is a behavioral design pattern that lets an object change its behavior when its internal state changes. Systems with this property can utilize its advantage while designing behavior of the class by identifying different actions with different states rather than trying to fit in all situations in one single method. This will make the code easy to write, understand, test and maintain.

In Telegram repo, the FileLoadOperation class utilize this advantage as there are certain states during downloading a file, and partitions these loading states into 4 categories:

```
private final static int stateIdle = 0;
private final static int stateDownloading = 1;
private final static int stateFailed = 2;
private final static int stateFinished = 3;
```

From the screenshot below it can be seen that state is taking an active role in identifying method operations:



For a simple example here, below is a method that is intended to pause the loading process. and its actions are clearly partitioned by the states defined above. If the state is not downloading, then there is nothing to be done, if the state is downloading, then it will pause the loading process.

```
public void pause() {
    if (state != stateDownloading) {
        return;
    }
    paused = true;
}
```

It can be seen that the downloading process is easy to identify different states and the states take an important role in determining actions of many methods. Therefore, it is an ideal scenario to use the State design pattern in FileLoadOperation class.

3. Conclusion

From the cases discussed above, it can be concluded that the design pattern helped developers solve problems more easily. With one design pattern, multiple similar problems are solved, the efficiency of the development is improved. Also, with design patterns, the code is more understandable, which can help other developers working on the source code. In conclusion, the design pattern is a good practice for software development.