TP1

Semester : Fall 2021

Team Romeo

Chun-An Bau - 2165883

Guilhem Dubois - 1897198

Matyas Jesina - 2165646

Group : 01

LOG8430E - Software Architecture and Advanced Design

Polytechnique Montréal

2021

Table of Contents

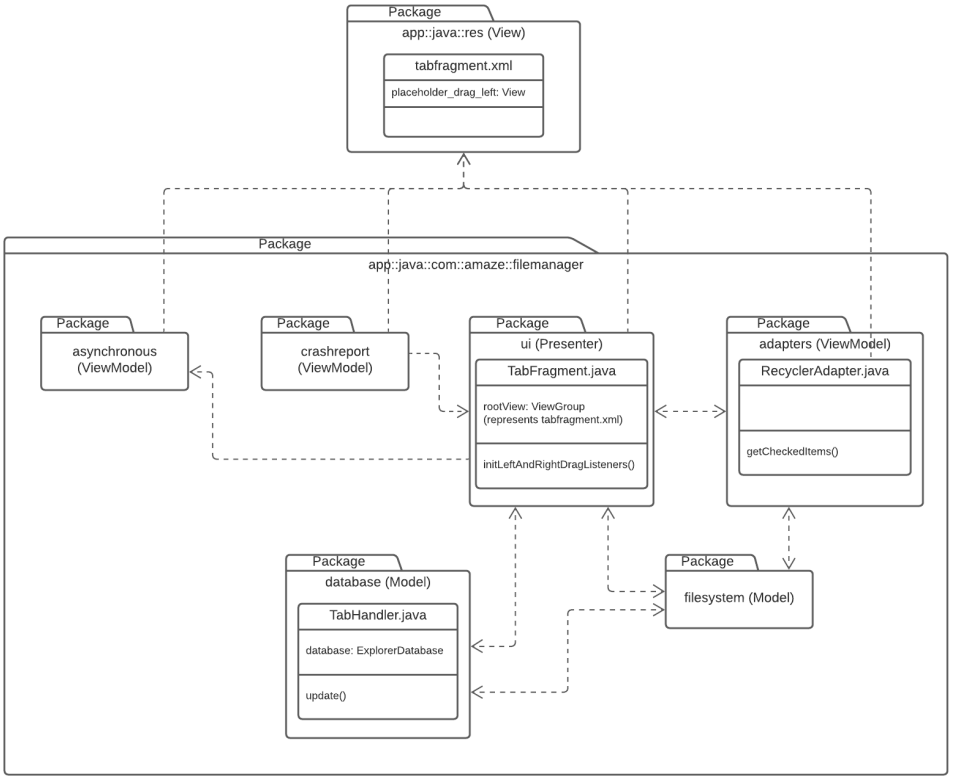
Question 1: Software architecture analysis

Both MVP and MVVM architectures can be found in the system. The system isn't monolithic since the code is separated into modules (multiple files and packages). It's also not a distributed architecture since the entire program is available locally in one system.

As we can see in the UML diagram, the xml files are part of the view since they represent what is shown to the user and the user interacts with those elements. Each view is associated with a class in the ui module, which means the classes in the "ui" package can be seen as Presenter. They handle the events received in the view and react accordingly.

For example, there's a tabfragment.xml file in the **View** (in app/java/res) which displays a layout to the user. There's a class in the ui package called TabFragment, which is directly associated with the tabfragment.xml view. As a **Presenter**, it handles the interaction between the user and this specific view. For example, when the user does a drag interaction on the view, the logic is handled in TabFragment.java, as defined in the method initLeftAndRightDragListeners(). Finally, the presenter also interacts with the "database" and "filesystem" packages, which can be seen as the **Model** since they represent stored data but don't contain logic related to the views. The TabFragment class calls update() from TabHandler (from the database package), which updates the database.

Some views are used in other packages, such as "adapters" and "asynchronous", which can be seen as **ViewModels** since they interact with multiple views. For instance, both the MainFragment and the TabFragment are using the method getCheckedItems() from RecyclerAdapter.



Question 2: Design Patterns

Design Pattern 1: Builder

The builder pattern is a creational design pattern which simplifies the creation of objects by encapsulating the creation and customization of this object. For instance, in ShareTask.java (in the package com.amaze.filemanager.ui.dialogs.share), a builder is used to create a new MaterialDialog window :

```
MaterialDialog.Builder builder = new MaterialDialog.Builder(contextc);
      builder.title(R.string.share);
      builder.theme(appTheme.getMaterialDialogTheme(contextc));
      ShareAdapter shareAdapter = new ShareAdapter(contextc,
targetShareIntents, labels, drawables);
      builder.adapter(shareAdapter, null);
      builder.negativeText(R.string.cancel);
      builder.negativeColor(fab_skin);
      MaterialDialog b = builder.build();
```

We can see in the code example that the builder offers multiple methods to customize the MaterialDialog, such as *title()*, *theme()*, *negativeColor()*, etc. Once all the properties are customized to our needs, the *build()* method is called on the builder to create the MaterialDialog instance. Therefore, using a builder separates the creation logic of the MaterialDialog from the dialog object itself.

The methods used on the builder hide a lot of complexity from the object it's creating. For example, setting the adapter using the builder simply requires to call *adapter()*:

```
builder.adapter(shareAdapter, null);
```

which will run in the builder :

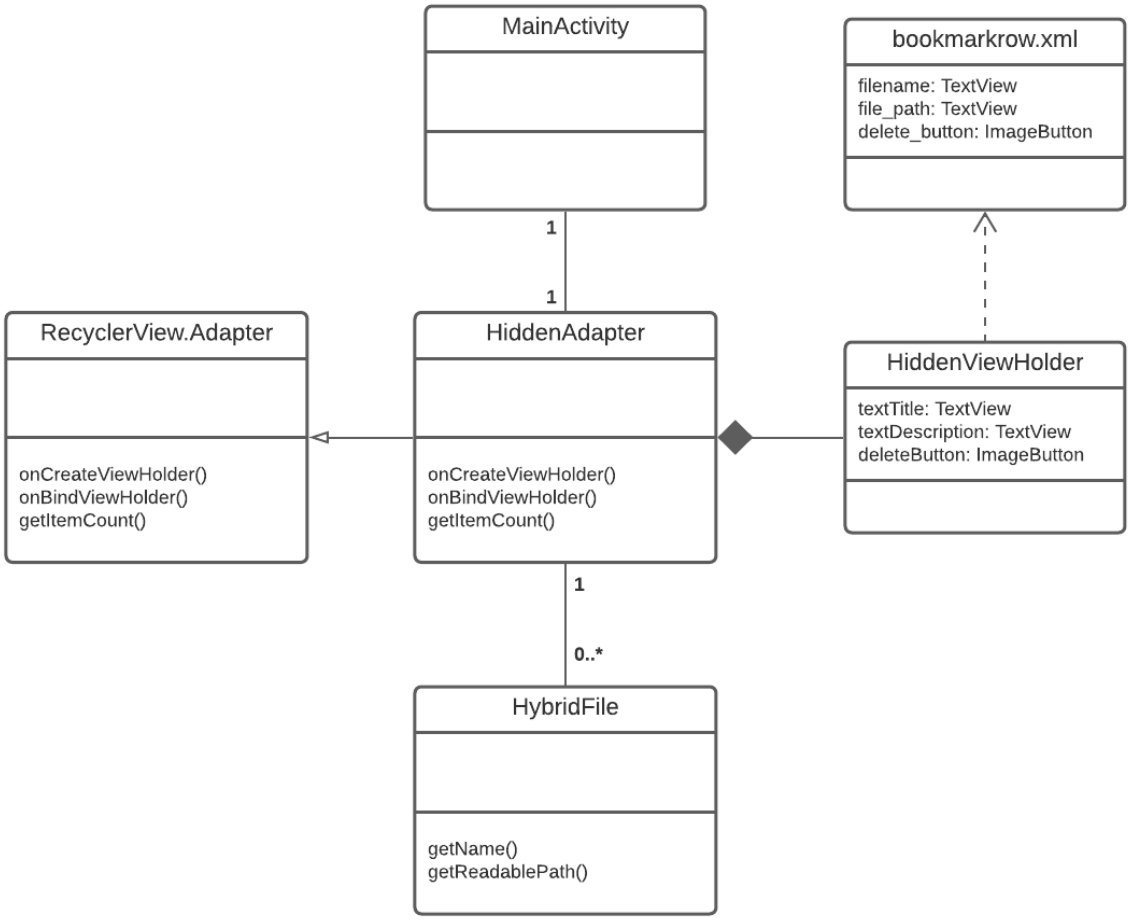
```
public MaterialDialog.Builder adapter(@NonNull Adapter<?> adapter,
@Nullable LayoutManager layoutManager) {
            if (this.customView != null) {
                throw new IllegalStateException("You cannot set
adapter() when you're using a custom view.");
            } else if (layoutManager != null && !(layoutManager
instanceof LinearLayoutManager) && !(layoutManager instanceof
GridLayoutManager)) {
                throw new IllegalStateException("You can currently only
use LinearLayoutManager and GridLayoutManager with this library.");
            } else {
                this.adapter = adapter;
                this.layoutManager = layoutManager;
                return this;
            }
```

```
        }
```

The MaterialDialog class can then focus on the functionalities of the class, while the MaterialDialog.Builder class focuses on its creation.

Design Pattern 2: Adapter

The adapter pattern is a structural design pattern used to make two incompatible classes work together. In Amaze File Manager, there's a class called *HiddenAdapter* used to show hidden files (represented by the class *HybridFile*) on a RecyclerView:



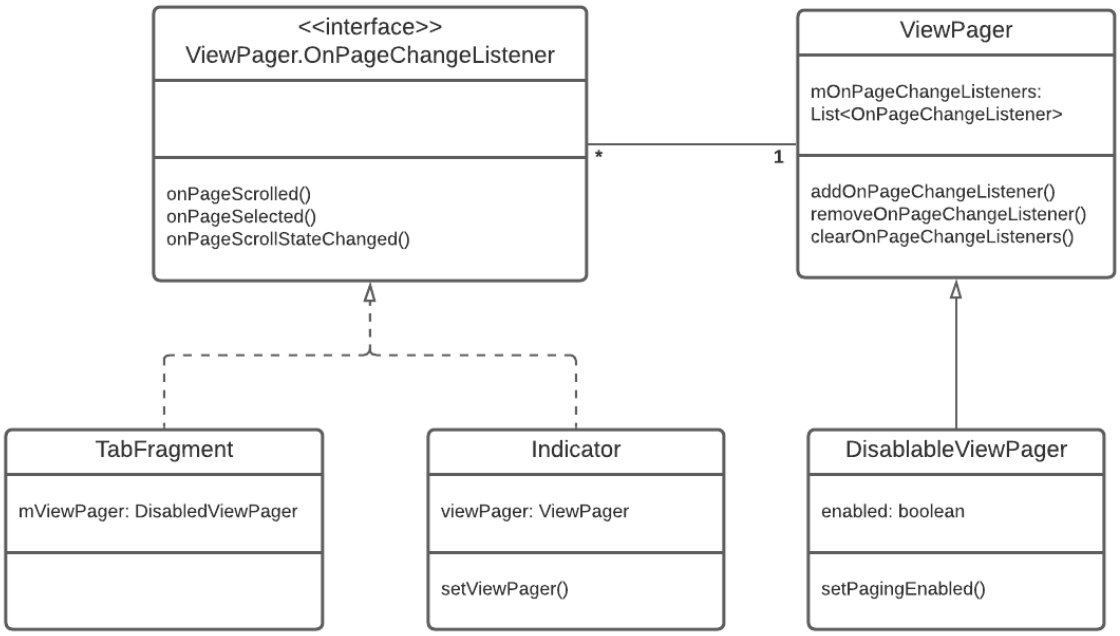
As we can see in the UML diagram, there's a view called "bookmarkrow" which shows elements such as "filename", "file_path" and "delete_button". This data comes from a *HybridFile*. The *HiddenViewHolder* class contains references to the view components, which lets *HiddenAdapter* use those properties to insert the data. When the *MainActivity* displays a dialog with "bookmarkrows", it also creates the adapter which implements method such as *onBindViewHolder()* and *getItemCount()* based on its list of HybridFile.

Therefore, the data class (*HybridFile*) contains no view logic and the view (*bookmarkrow*) only contains what it needs to show, without needing to know how *HybridFile* is structured. The *HiddenAdapter* class is the middlemen between the two:

```
public class HiddenAdapter extends RecyclerView.Adapter<HiddenViewHolder> {
...
private ArrayList<HybridFile> hiddenFiles;
...
@Override
  @NonNull
  public HiddenViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int
viewType) {
    LayoutInflater mInflater =
        (LayoutInflater)
context.getSystemService(Activity.LAYOUT_INFLATER_SERVICE);
    View view = mInflater.inflate(R.layout.bookmarkrow, parent, false);

    return new HiddenViewHolder(view);
  }

  @Override
  @SuppressWarnings("unchecked") // suppress varargs warnings
  public void onBindViewHolder(HiddenViewHolder holder, int position) {
    HybridFile file = hiddenFiles.get(position);

    holder.textTitle.setText(file.getName(context));
    holder.textDescription.setText(file.getReadablePath(file.getPath()));
...
```

Design Pattern 3: Observer

The observer pattern is a behavioural design pattern where a subject notifies its observers when its state changes. This can be found in Amaze File Manager, where some components need to know when a page shown is changed or scrolled :



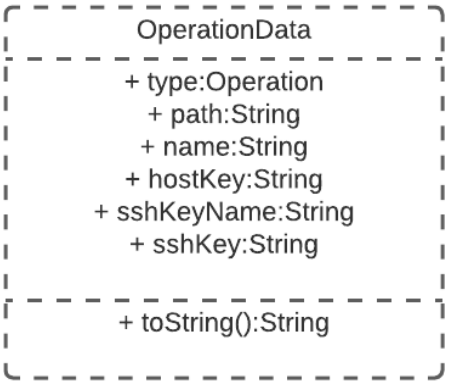
In this diagram, *ViewPager* is the base subject, which holds the state of the page. The *Indicator* class needs to know when the page changed in order to react to it, such as by moving the position of the indicator when needed. In this case, the indicator is the observer of the subject. To observe it, it implements the interface *ViewPager.OnPageChangeListener*, which holds methods such as *onPageSelected()*. When the indicator is created, it starts listening to the *ViewPager* by calling *addOnPageChangeListener(this)*. Therefore, the *ViewPager* keeps the listener in a list and will call its appropriate method when needed. For example, when the page changes, *ViewPager* will call the method *onPageSelected()* on all its listeners. The *Indicator* class that has implemented this method will be able to react to the new page selected.

This pattern is useful when needing to react to events, since it's able to add multiple listeners to the same event dynamically. For instance, in the same example, *TabFragment* and *Indicator* both listen to the *ViewPager* events, without making the objects tightly coupled. There could be more listeners added or removed during runtime depending on the user interaction with the app, since the listeners are added dynamically.

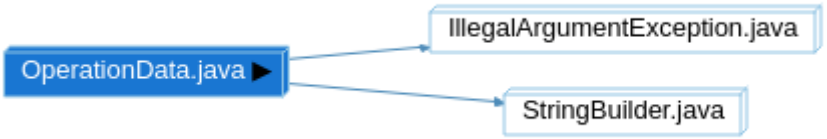
Question 3: SOLID Design Principles

SOLID Principle 1: SRP

The instance OperationData guarantees SRP. The instance stores lots of information related to an operation and provides only one method to export the information in the form of string, which means that the instance is responsible for only one task.



Instances that achieve SRP should be high cohesion and low coupling. As depicted in the figure, the instance only uses two external instances, which implies its low coupling. In the code snippet, we can see that the method toString uses most of the private members in the OperationData, which shows its high cohesion.




```
public String toString() {
    StringBuilder sb =
        new StringBuilder("OperationData type=[")
            .append(type)
            .append("],path=[")
            .append(path)
            .append("]");

    if (!TextUtils.isEmpty(hostKey)) sb.append(",hostKey=[").append(hostKey).append(']');

    if (!TextUtils.isEmpty(sshKeyName))
      sb.append(",sshKeyName=[").append(sshKeyName).append("],sshKey=[redacted]");

    return sb.toString();
}
```

SOLID Principle 2: DIP

The instances that guarantee DIP can complete the operations using abstract methods but without knowing what actually happens beyond. The instance *database/UtilsHandler.java* is done with this principle. As shown in the code section, none of the database operations are specific to a particular technology, like SQL, but the operations can still work properly.

```
public void saveToDatabase(OperationData operationData) {
    switch (operationData.type) {
      case HIDDEN:
        utilitiesDatabase
            .hiddenEntryDao()
            .insert(new Hidden(operationData.path))
            .subscribeOn(Schedulers.io())
            .subscribe();
        break;
      case HISTORY:
        utilitiesDatabase
            .historyEntryDao()
            .insert(new Hidden(operationData.path))
            .subscribeOn(Schedulers.io())
            .subscribe();
        break;
      case LIST:
        ...
      case GRID:
        ...
      case BOOKMARKS:
        ...
      case SMB:
        ...
      case SFTP:
        ...
      default:
        throw new IllegalStateException("Unidentified operation!");
    }
  }
```

Variable *utilitiesDatabase* is set while constructing the instance.

```
private final UtilitiesDatabase utilitiesDatabase;
public UtilsHandler(@NonNull Context context, @NonNull UtilitiesDatabase
    utilitiesDatabase) {
    this.context = context;
    this.utilitiesDatabase = utilitiesDatabase;
}
```

And the value is decided while calling the function *initialize* of instance *UtilitiesDatabase*, which is the concrete technologies located.

```
utilitiesDatabase = UtilitiesDatabase.initialize(this);
utilsHandler = new UtilsHandler(this, utilitiesDatabase);
```

SOLID Principle 3: OCP

The principle aims to ease the project to extend without modifying the existing code sections. *View.OnClickListener* (https://developer.android.com/reference/android/view/View.OnClickListener) is an interface with an abstract method *onClick,* which is being used widely to handle the click events on specific views. We found several implementations of the method *onClick* for distinct purposes, as shown in the code clips below.

```
public class AboutActivity extends ThemedActivity implements
View.OnClickListener {
 ...

 @Override
 public void onClick(View v) {
   switch (v.getId()) {
     ...
   }
 }
 ...
}
```

```
public class TextEditorActivity extends ThemedActivity
   implements TextWatcher, View.OnClickListener {

 @Override
 public void onClick(View v) {
   final TextEditorActivityViewModel viewModel =
       new ViewModelProvider(this).get(TextEditorActivityViewModel.class);

   switch (v.getId()) {
     ...
   }
 }
}
```

```
public class CloudSheetFragment extends BottomSheetDialogFragment
implements View.OnClickListener {
  ...
  @Override
  public void onClick(View v) {

    Log.d(TAG_FRAGMENT, "Clicked: " + v.getId());

    switch (v.getId()) {
      ...
    }
   }
}
```

This interface makes the project able to register different handlers to deal with distinct events. Also, following Open Closed Principle increases the project's extensibility. If new events are required to be handled, the goal can be achieved by adding new instances, which implement *OnClickListener*, without modifying the existing part.

Question 4: Violation of SOLID Design Principles

Violation 1: SRP in Drawer.java

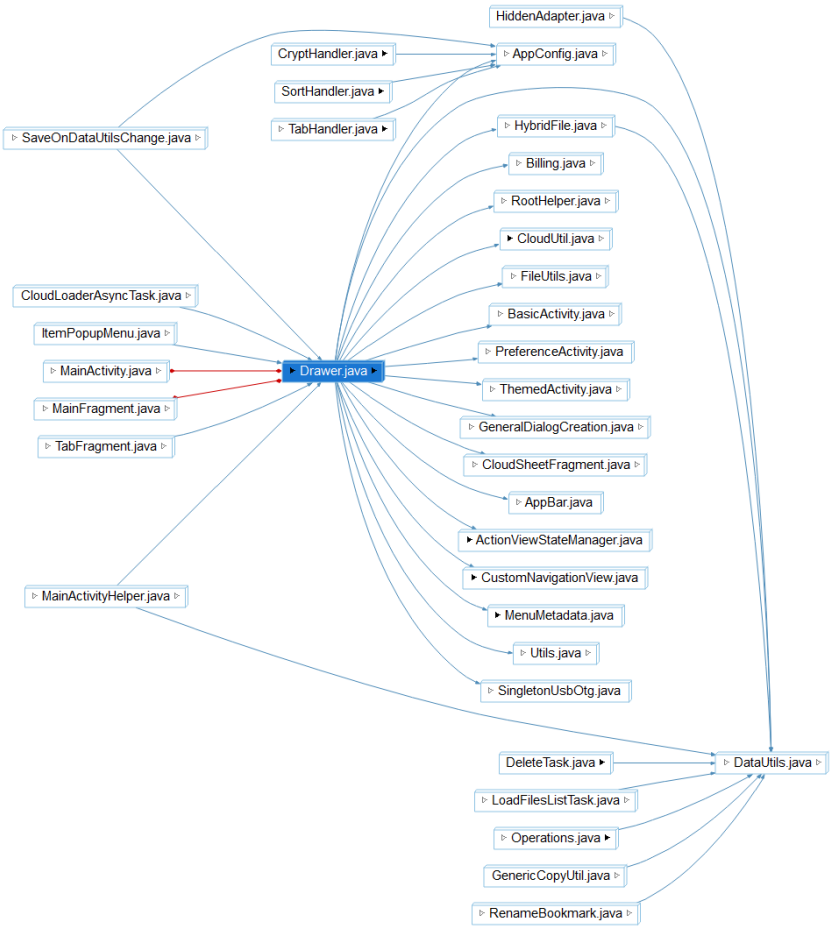
The Drawer class acts as a large and complex view. It contains private instances of many other classes, such as Resources, DataUtils, MainActivity, Billing, FragmentTransaction and so on, together with various enums. This violates the Single Responsibility Principle. MetricsTree reports this Responses For a Class value as Extreme (192), the standard range is [0..45).

```
public class Drawer implements NavigationView.OnNavigationItemSelectedListener {

  public static final int image_selector_request_code = 31;

  public static final int STORAGES_GROUP = 0,
      SERVERS_GROUP = 1,
      CLOUDS_GROUP = 2,
      FOLDERS_GROUP = 3,
      QUICKACCESSES_GROUP = 4,
      LASTGROUP = 5;
  public static final int[] GROUPS = {
    STORAGES_GROUP, SERVERS_GROUP, CLOUDS_GROUP, FOLDERS_GROUP, QUICKACCESSES_GROUP, LASTGROUP
  };

  private MainActivity mainActivity;
  private Resources resources;
  private DataUtils dataUtils;

  private ActionViewStateManager actionViewStateManager;
  private volatile int phoneStorageCount =
      0; // number of storage available (internal/external/otg etc)
  private boolean isDrawerLocked = false;
  private FragmentTransaction pending_fragmentTransaction;
  private String pendingPath;
  private ImageLoader mImageLoader;
  private String firstPath = null, secondPath = null;

  private DrawerLayout mDrawerLayout;
  private ActionBarDrawerToggle mDrawerToggle;
  private CustomNavigationView navView;
  private RelativeLayout drawerHeaderParent;
  private View drawerHeaderLayout, drawerHeaderView;
```

The whole class is almost 1000 lines long and contains other coding style violations, such as old commented code or unused fields.

```
172     // set width of drawer in portrait to follow material guidelines
173     /*if(!Utils.isDeviceInLandScape(mainActivity)){
174         setNavViewDimension(navView);
175     }*/
```

We can also see various responsibilities on a call graph. Fixing this would require rewriting a significant part of the application, which is outside the scope of this project.



Violation 2: OCP in CustomFileObserver.java

CustomFileObserver's *onEvent* method processes incoming file system events using a switch statement with the event's code. The switch statement is repetitive and needs to be modified every time another event is added to the application. This violates the Open Closed Principle. We can see the switch statement below (the method continues after the statement). Alternative solutions should be considered, such as taking advantage of polymorphism.

```
public void onEvent(int event, String path) {
  if (event == IN_IGNORED) {
    wasStopped = true;
    return;
  }

  long deltaTime = Calendar.getInstance().getTimeInMillis() - lastMessagedTime;

  switch (event) {
    case CREATE:
    case MOVED_TO:
      pathsAdded.add(path);
      break;
    case DELETE:
    case MOVED_FROM:
      pathsRemoved.add(path);
      break;
    case DELETE_SELF:
    case MOVE_SELF:
      handler.obtainMessage(GOBACK).sendToTarget();
      return;
  }
```

We can create an abstract parent class for file system event and child classes for specific actions. New methods can be added to *CustomFileObserver* to act upon these events, which removes the need of switch statements.
Since Java doesn't support double dispatch, we will use the Visitor pattern to implement handling of specific events. The visitor pattern consists of two main parts: Visitor and *Visitable*. The visitor interface was implemented in *CustomFileObserver* and can be seen below:

```
public interface FilesystemEventVisitor {
    void visit(FilesystemCreateEvent event);
    void visit(FilesystemDeleteEvent event);
    void visit(FilesystemDeleteSelfEvent event);
    void visit(FilesystemIgnoredEvent event);
    void visit(FilesystemMovedFromEvent event);
    void visit(FilesystemMovedToEvent event);
    void visit(FilesystemMoveSelfEvent event);
}
```

Visitable interface was implemented in individual events. Even though the code is exactly the same in all cases, it cannot be placed into the parent event class as this would not work.

```
public interface Visitable<V> {
    void accept(V visitor);
}
```

```java
public abstract class FilesystemEvent implements Visitable<FilesystemEventVisitor> {
    private final String Path;

    public FilesystemEvent(String path) { Path = path; }

    public String getPath() { return Path; }
}
```

```java
public class FilesystemCreateEvent extends FilesystemEvent {
    public FilesystemCreateEvent(String path) { super(path); }

    @Override
    public void accept(FilesystemEventVisitor visitor) {
        visitor.visit( event: this);
    }
}
```

Events were created for every action from the initial switch. Since they all have the same implementation, only *FilesystemCreateEvent* is displayed in this document.
This enabled us to remove the switch statement and replace it with a single call, which leads to a specific *visit* method being called. Some switch branches caused the method to set global variables and return early - this was solved by using another global variable (*continueEventProcessing*) to indicate whether to continue execution after visit. The result can be seen below (not all *visit* methods are displayed):

CustomFileObserver:

```java
@Override
public void visit(FilesystemCreateEvent event) { pathsAdded.add(event.getPath()); }

@Override
public void visit(FilesystemDeleteEvent event) { pathsRemoved.add(path); }

@Override
public void visit(FilesystemDeleteSelfEvent event) {
  handler.obtainMessage(GOBACK).sendToTarget();
  continueEventProcessing = false;
}

@Override
public void visit(FilesystemIgnoredEvent event) {
  wasStopped = true;
  continueEventProcessing = false;
}
```

```java
public void onEventReceived(FilesystemEvent event) {
  continueEventProcessing = true;
  long deltaTime = Calendar.getInstance().getTimeInMillis() - lastMessagedTime;

  event.accept( visitor: this);

  if (!continueEventProcessing) return;
```

As we can see, the method was significantly shortened and explicit type checking was removed. If a new filesystem event is added, it can simply be entered into the *FilesystemEventVisitor* interface, which will force developers to integrate it to all classes. Calling conventions remained without major changes and only required a simple modification, as can be seen below:

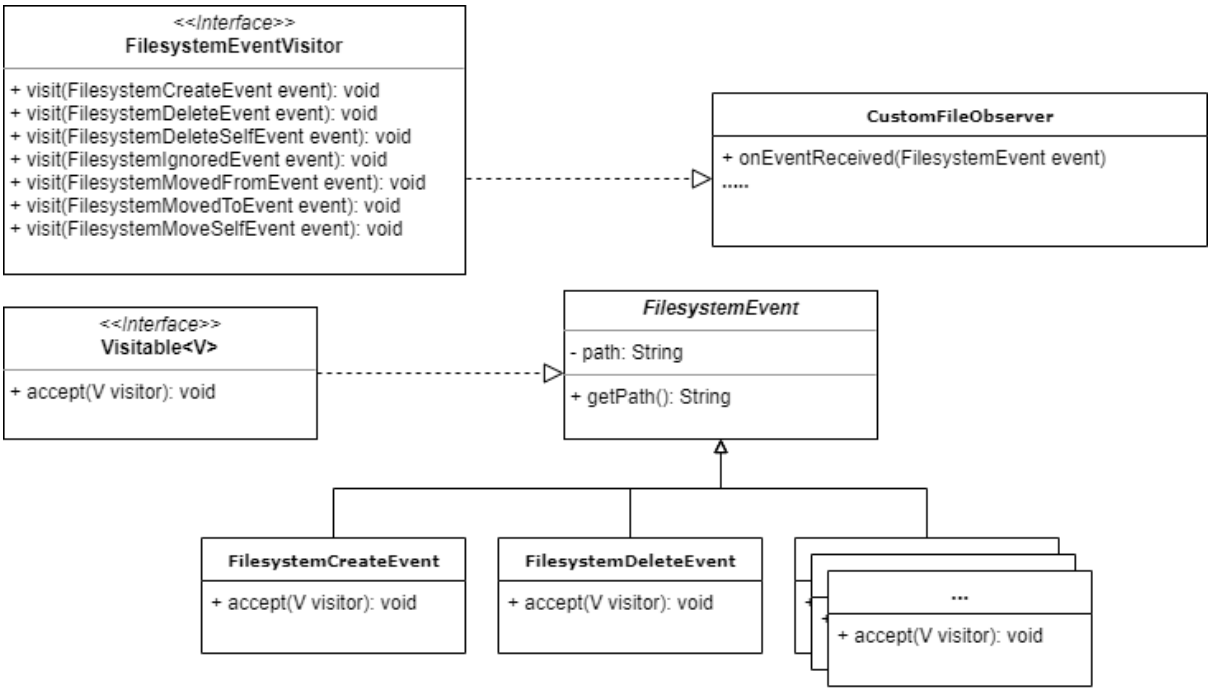
Before change:

```java
for (String s : compare(files, newFiles)) {
  cfileObserver.onEvent(DELETE, s);
}
```

After change:

```java
for (String s : compare(files, newFiles)) {
  cfileObserver.onEventReceived(new FilesystemDeleteEvent(s));
}
```

This refactoring will improve maintainability extensibility and helps reduce method lengths. The corresponding UML diagram after change can be seen below:



Resources

Android Studio
MetricsTree (https://plugins.jetbrains.com/plugin/13959-metricstree)
Understand (https://www.scitools.com/)

Moodle Slides
SourceMaking (https://sourcemaking.com/design_patterns)
Android Documentation (https://developer.android.com/docs)