SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 16, No. 2, June 2019, 267-287

UDC: 621.39:004.42
004.451.9ANDROID

DOI: https://doi.org/10.2298/SJEE1902267K

An Effective Approach to TV Applications Development – Application Framework

Nemanja Kovačev[1], Veljko Ilkić[1], Dejan Nađ[1], Nikola Vranić[1]

Abstract: Android as an operating system is becoming more and more widespread across digital devices. One of the fields in which Android has gained popularity is TV application development. During our years-long work on this platform, we noticed certain issues with current approaches. The most important issues include a lot of boilerplate code and difficult customization which together increase the overall time necessary for the development and contribute to poor code readability thus making the maintenance more difficult. With that in mind, we decided to take our own approach which includes a framework and a library and which enables us to build high-quality and cost-effective TV applications in terms of fast development and easy maintenance. The approach which will be presented in this paper is more effective than other approaches we took for the development of previous TV applications.

Keywords: Java, Android, MVP, Framework, Library, SDK, Handler, TV application, Backend, Information Bus.

1 Introduction

During the years of development of TV applications, we noticed that existing approaches do not meet our needs in terms of development speed and effectiveness since it is critical to develop an application as quickly as possible without losing quality. Because of the fact that the majority of TV applications have basically the same or very similar code structure and a lot of common features we decided to make a framework library which would facilitate the development process [1].

Framework usage suggests dividing the project into separate modules as the best practice. This way each project has a similar code structure in a way that it contains:

1. App module which is in charge of UI (User Interface) of a complete application including application internal logic and UI look and feel.

[1]RT-RK Institute for Computer Based Systems, Narodnog fronta 23a, Novi Sad, Serbia;
E-mails: nemanja.kovacev@rt-rk.com; veljko.ilkic@rt-rk.com; dejan.nadj@rt-rk.com; nikola.vranic@rt-rk.com

2. SDK (Software Development Kit) module which is in charge of enabling communication with backend server as well as communication with TV middleware software. In general, the SDK module is feeding data to the App module which is in charge of presenting all of that to the end user.
3. Core module which represents the starting point for TV application. It can be regarded as the center of the application which connects the UI on one side with the backend and database APIs (Application Programming Interfaces) on the other side. Fig. 1 shows the connection and data flow between the Core and other components. The Core contains a set of abstractions for all basic and advanced features as well as a set of APIs which are used as a guideline for application development. It is a highly optimized library which introduces a very small memory footprint.

Fig. 1 also shows the main abstractions merged in the Core module [1]. A particular abstraction can be used for different projects and different clients (e.g. BackendHandler provides a set of APIs used by the application module and can be embedded in various backend implementations). This way once developed UI application can be considered as backend agnostic because a different BackendHandler can be added as a plugin to the Core module if needed. This enables us to use the same application and the same logic for different projects and clients because an application always calls the same API abstraction without actually knowing what is in the layer beneath it. The same approach can be applied to TV middleware or player used for decoding TV stream of VoD (Video on Demand) content.

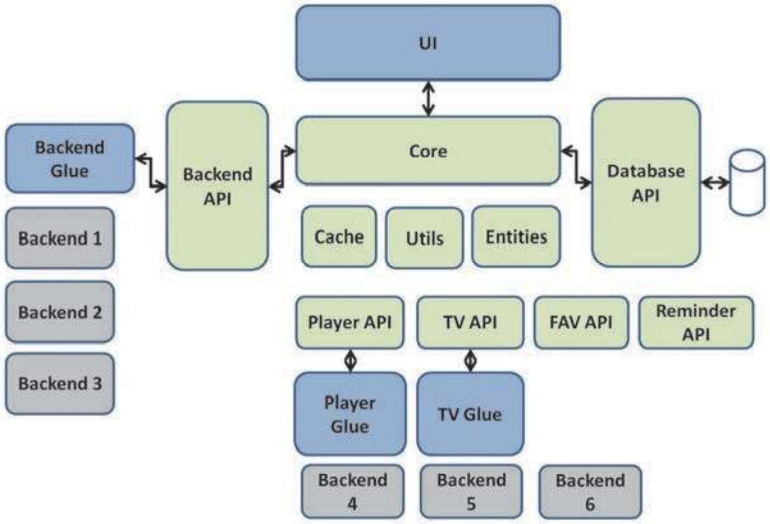


Fig. 1 – *Top level view of the framework.*

We constructed the framework library in a way that the Core and SDK are written in pure Java [2] because we wanted it to be Android agnostic so we can reuse it in multiple projects, and also be platform agnostic which is convenient when we have to develop an application on personal computer or emulator. This is the starting point for the development whereas the UI part is written in Android [3] which provides rich Android UI features (Fig. 2).

A big advantage of this approach is not only the code reusability inside one project but also the ability to easily share feature implementation between different projects that are built this way.

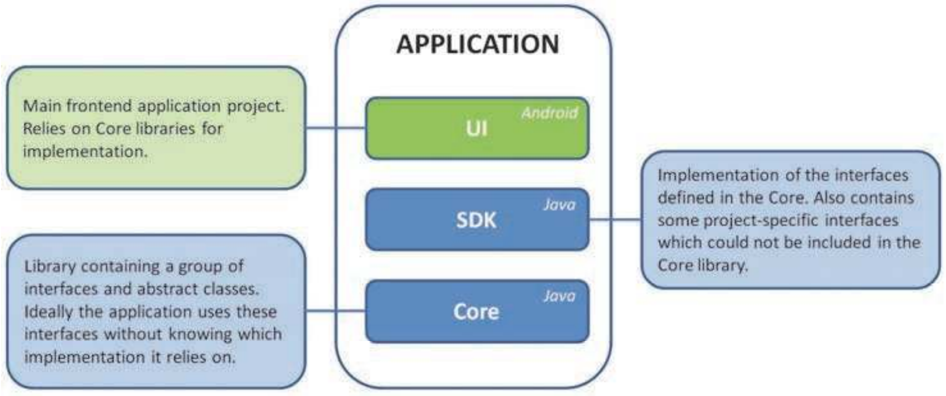

APPLICATION
Main frontend application project. Relies on Core libraries for implementation.
UI
Android
SDK
Java
Core
Java
Library containing a group of interfaces and abstract classes. Ideally the application uses these interfaces without knowing which implementation it relies on.
Implementation of the interfaces defined in the Core. Also contains some project-specific interfaces which could not be included in the Core library.

Fig. 2 – *Application overview.*

2 Design Pattern

The design pattern which turned out to be the best suitable for TV application development is the MVP (Model-View-Presenter) pattern [5] where the model is the application Core, the view is the Android application itself and the presenter is the SDK. The model defines all data entities and data collections and represents the interface which defines data to be displayed, the view defines all graphical components and graphical layers and acts as a passive interface that displays the data whereas the presenter is the "middle-man" between the model and the view which enables the View to communicate with the model.

User events are passed from the view to the presenter which updates the model. The model sends events to the presenter which updates the view as shown in Fig. 3.

In this framework, the view is named the Scene and the presenter is named the Scene Manager. The Scene sends information to the Scene Manager via a Scene listener interface and the Scene Manager propagates that information to the model. The model sends the updated data to the Scene Manager which then updates the Scene via *refresh()* method.

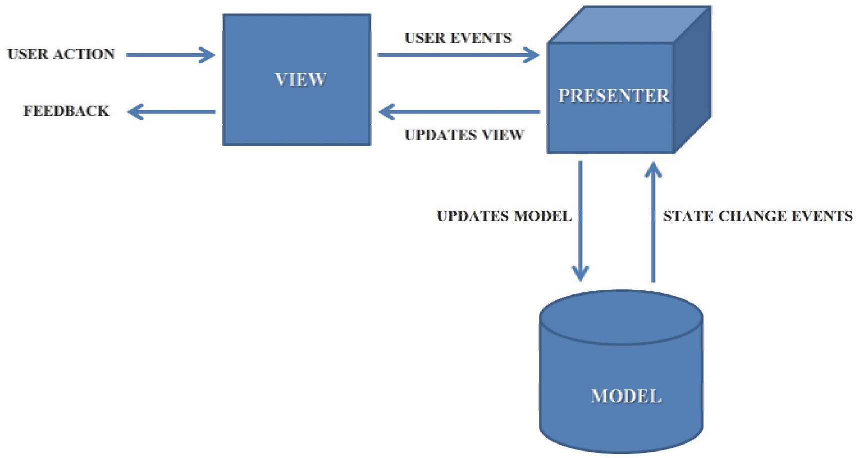


Fig. 3 – *The Model-View-Presenter design pattern.*

A big advantage of using this pattern for development is that we can easily change one module while leaving other modules unchanged. This enables us to quickly apply any change without altering a lot of logic and code. By using this pattern we managed to speed up the development process especially in UI/UX review phase when after several sessions we decide that something in the UI part should be changed or improved.

3 Components of The Framework

The main building blocks inside the framework are Scenes and Scene Managers which mainly communicate via WorldHandler or via Information Bus entity [1]. When we apply this concept into the Android environment it implicates the usage of Fragments and Activities. Inside Android application main instance is called ProjectApplication class. It is the instance holder of the application. It acts as a parent to all available activity objects inside the application, it holds the reference to WorldHandler. In addition to that, the ProjectApplication class is the central point of the entire application and provides ApplicationContext to every scene in the application. The instance of ProjectApplication is in memory as long as the application is active.

3.1 Activities and fragments

3.1.1 Activities

An Activity is a single, focused entity that the user can interact with [3]. The Activity class takes care of creating a window for a user in which the UI is placed. Only one Activity can be active at a time.

Every application must have the MainActivity [6, 7] which has two principal functions: it represents the entry point for the framework and application itself, and also acts as the view entity of the whole system – it draws the view elements on the screen by inflating fragments which are defined in the scenes. The MainActivity handles events sent from the Core module based on screen flow and custom application logic and renders those events on the screen.

The MainActivity is also in charge of forwarding inserted keys to the active scene [1]. Every onKeyDown and onKeyUp event received from the Android system will be forwarded to the active scene by calling its *dispatchKeyEvent()* method. The MainActivity together with the ProjectApplication class represents the central point of every application made by using this framework.

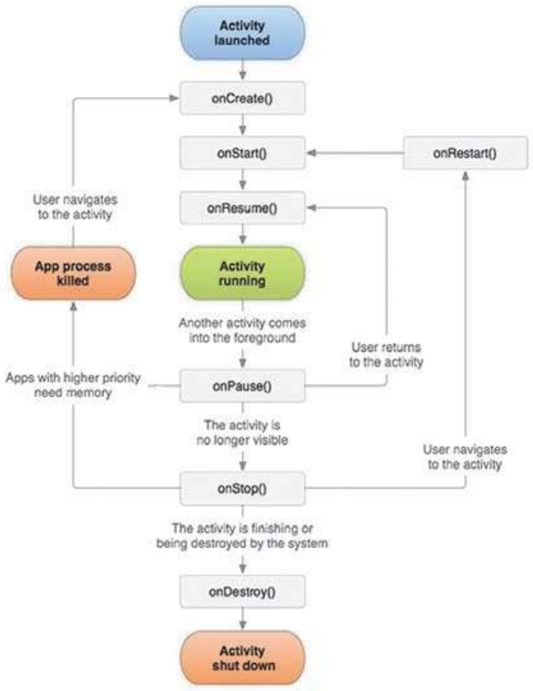

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Fig. 4 – *Activity lifecycle [4].*

Fig 4. shows a complete Activity lifecycle overview [4]. In this framework we use four basic lifecycle methods in the MainActivity:

– *onCreate()* – initializes necessary views and handlers inside coreSDK (e.g. VideoSurface is inflated from the main layout and passed to PlayerHandler from coreSDK so middleware can render video in a separate surface layer);

– *onResume()* – deals with application lifecycle which is important when leaving launcher application and entering other applications (e.g. YouTube);

– *onPause()* – called when a scene gets hidden;

– *onDestroy()* – called when a scene gets destroyed.

In *onResume()* and *onPause()* methods the application prepares its state so it can be easily recovered.

The advantage of using only single Activity inside a complete TV application, no matter how complex it can get, is that complexity handling the playback of TV stream which is the most important feature of TV application is completely reduced. A developer doesn't have to pay attention to Android's lifecycles while handling playback when switching between different scenes. Playback of content is separated from the UI layer which facilitates both development and testing.

3.1.2 Fragments

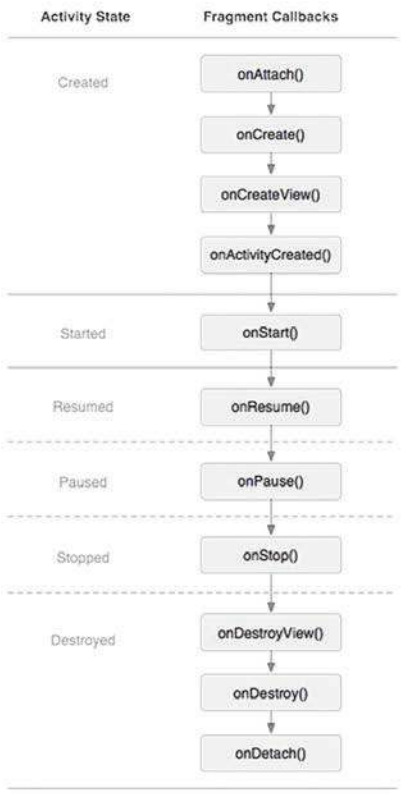


Fig. 5 – *Fragment lifecycle [4].*

Fragments [3] are the passive views which are inflated by the MainActivity. Fragments require a host activity in which they can be displayed and their lifecycle is dependent on the host Activity's lifecycle.

Each Scene internally binds a fragment reference which can later be rendered on screen [1]. Unlike Activities, there can be multiple Fragments displayed on the screen at the same time. Fragments are handled by FragmentManager. Fig. 5 shows a complete Fragment lifecycle and its relation to Activity lifecycle [4].

3.2 Scenes and Scene Managers

3.2.1 Scenes

A Scene represents the view [5] in the MVP model. It contains fields (id, name, sceneListener, etc.) and methods (*createView()*, *refresh()*, *dispatchKeyEvent()*, etc.) and can be placed and managed in one of the five graphical layers which we defined in this framework - playback, UI, overlay, notification, and global layer (Fig. 6). There is a separate graphical layer dedicated for video and content playback, a separate layer for main scenes as well as separate layers for advanced features like overlay dialogs, notifications and global UI elements [1]. This enables much quicker development and easier debugging since every graphical layer has its own instance. The layers are completely separated in memory and communicate via a dedicated module called WorldHandler.

Since layers have no dependency on each other we can develop several features in parallel. Moreover, if an issue emerges in one layer a developer doesn't have to dive into a whole application logic but only into a dedicated part of the code for that particular graphical layer. This is a good practice since it's much easier to point out which part of the code has issues and needs optimizations and fixing.

The usage of layers basically enables us to cover all standard use cases for different TV applications and designs.

Every project has its abstract core Scene called the ProjectScene. It's a generic abstraction of Scene defined in the AppCore module. All available scenes extend ProjectScene class and introduce customized behavior which is in accordance with the specified design and screen flow. Another distinctive feature is the usage of generics. For creating scenes which have similar look and feel or similar functionality we have developed generic scenes and generic views which can be reused thus reducing development time and debugging time.

Every Scene must follow API (Application Programming Interface) from the ProjectScene which means it has to fill all the abstract methods regarding Scene setup, key handling, disposing resources, etc. The ProjectScene wraps

ProjectSceneFragment view which is placed on TV screen and provides generic key handling. The ProjectSceneFragment is a customized Android fragment which supports custom lifecycle and custom rendering effects (blurring, color correction, etc.) that can be applied if requested by design [1].

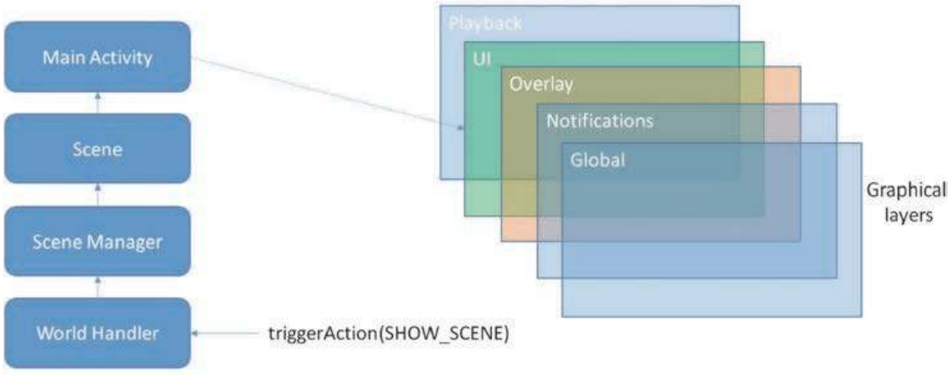


Fig. 6 – *Graphical layers scheme.*

Fig. 7 – *Graphical layers example (Playback layer).*

Fig. 7 shows playback in the Playback graphical layer.

Fig. 8 shows playback in the Playback graphical layer and zap banner showing basic information in the UI layer.

Fig. 9 shows playback in the Playback graphical layer and info banner showing more detailed information in the UI layer.

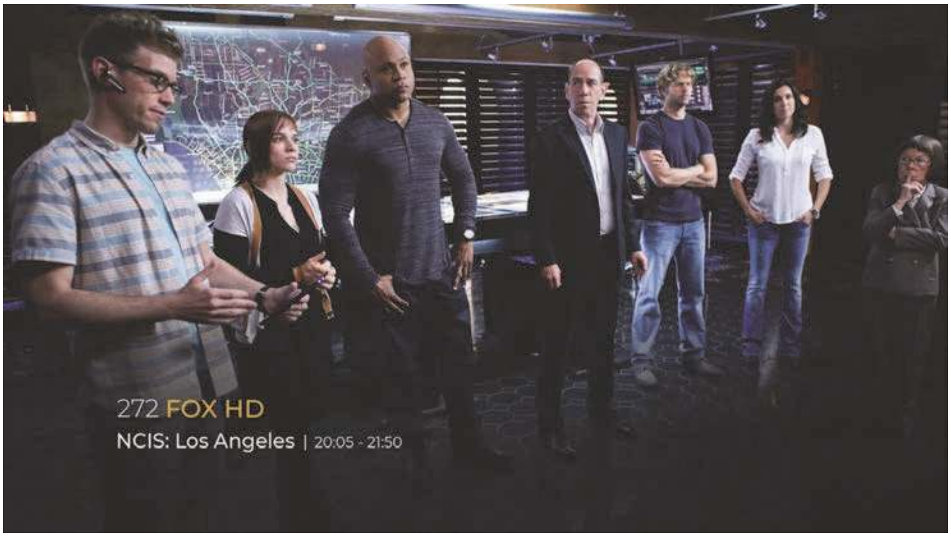
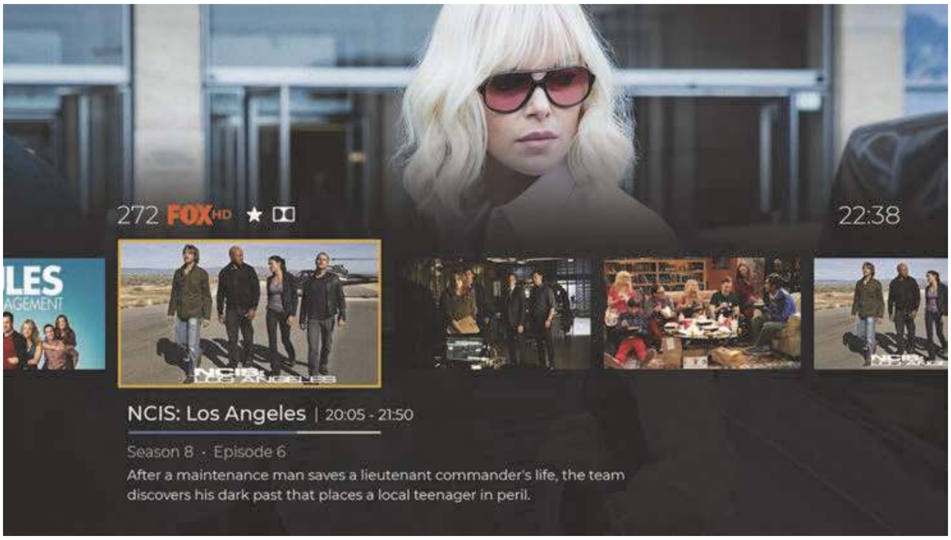


Fig. 8 – *Graphical layers example (Playback and UI layer).*



After a maintenance man saves a lieutenant commander's life, the team discovers his dark past that places a local teenager in peril.

Fig. 9 – *Graphical layers example (Playback and UI layer).*

Fig. 10 shows playback in the Playback graphical layer, launcher in the UI layer and volume control in the Global layer.

Fig. 11 shows playback in the Playback graphical layer in the background and notification in the Notification layer.

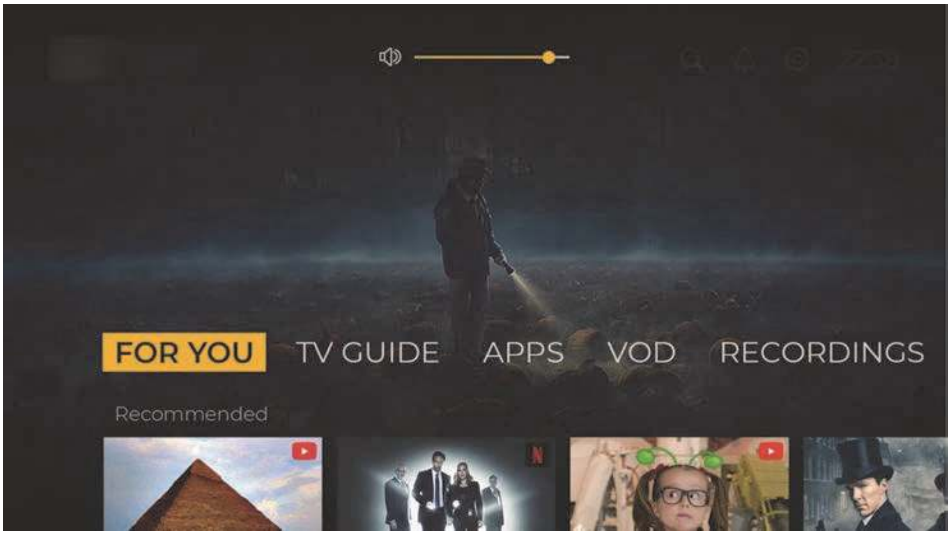


Fig. 10 – *Graphical layers example (Playback, UI and Global layer).*

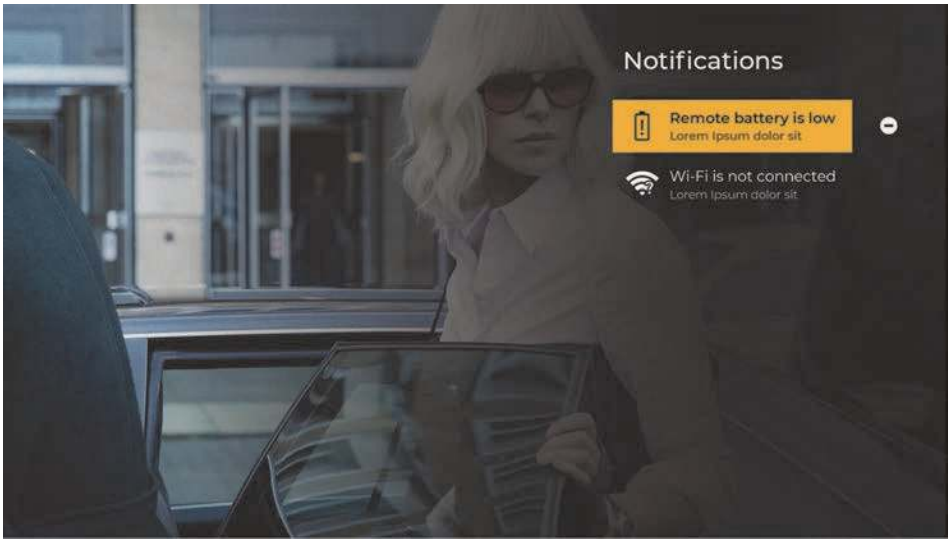
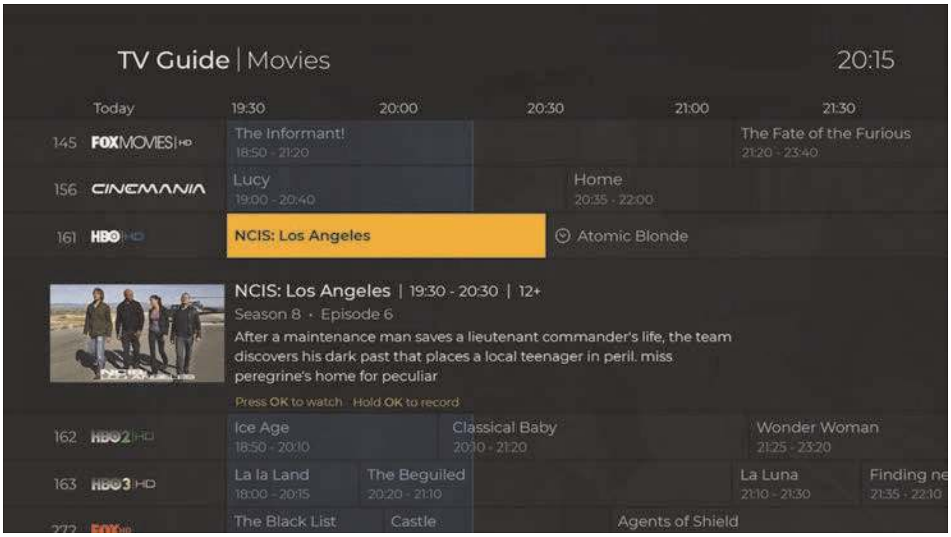


Fig. 11 – *Graphical layers example (Playback and Notification layer).*

Fig. 12 shows playback in the Playback layer and TV guide in the UI layer.

Fig. 13 shows playback in the Playback layer, TV guide in the UI layer and a Dialog scene in the Overlay layer. The UI layer is blurred in the background.



TV Guide | Movies 20:15

Today 19:30 20:00 20:30 21:00 21:30

145 FOXMOVIES HD — The Informant! 18:50 - 21:20 — The Fate of the Furious 21:20 - 23:40

156 CINEMANIA — Lucy 19:00 - 20:40 — Home 20:35 - 22:00

161 HBO HD — NCIS: Los Angeles — Atomic Blonde

NCIS: Los Angeles | 19:30 - 20:30 | 12+

Season 8 · Episode 6

After a maintenance man saves a lieutenant commander's life, the team discovers his dark past that places a local teenager in peril. miss peregrine's home for peculiar

Press OK to watch Hold OK to record

162 HBO2 HD — Ice Age 18:50 - 20:10 — Classical Baby 20:10 - 21:20 — Wonder Woman 21:25 - 23:20

163 HBO3 HD — La la Land 18:00 - 20:15 — The Beguiled 20:20 - 21:10 — La Luna 21:10 - 21:30 — Finding ne 21:35 - 22:10

272 FOX HD — The Black List — Castle — Agents of Shield

Fig. 12 – *Graphical layers example (Playback and UI layer).*

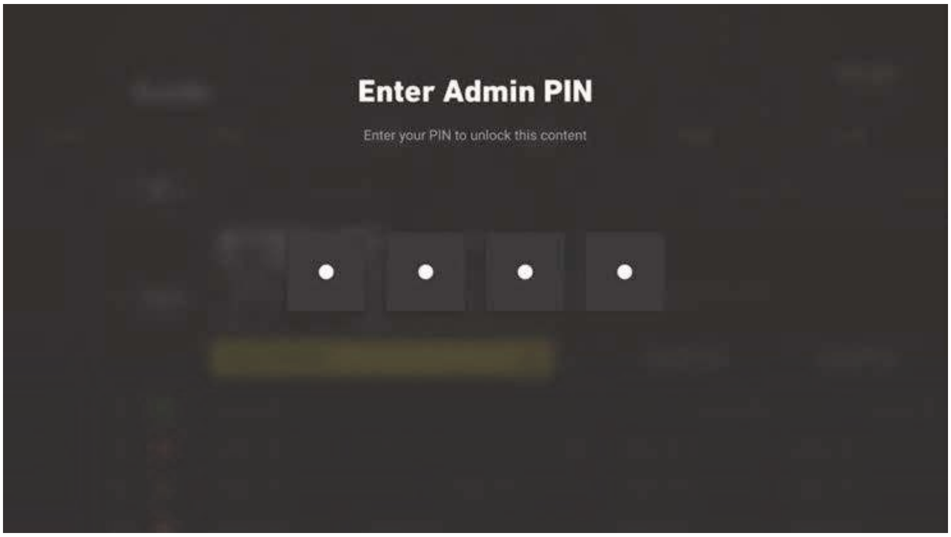


Fig. 13 – *Graphical layers example (Playback, UI and Overlay layer).*

3.2.2 Managers

Scene Managers are in charge of Scene creation and Scene control – a Scene Manager can show a Scene, show the Scene as an overlay, hide it or destroy it by calling Manager's *triggerAction()* method [1].

ProjectSceneManager is a custom implementation of SceneManager class defined in AppCore framework library. It defines abstract methods which have to be implemented by all Scene Managers registered inside an application. It implements generic logic for data sharing, back event handling, data loading, etc.

All available Scene Managers attached to available screens extend this class and introduce custom implementation and behavior based on specified screen flow. Every Manager has to follow the defined pattern for event handling and data sharing between different Scene Managers.

Data sharing between Scene Managers is implemented via ProjectSceneData object. Every Scene Manager defines its own custom ProjectSceneData object. This object is used as a communication token between different Managers. It is used for passing data between the Scenes, for initialization of the upcoming Scene, for storing the ID from the previous Scene and automatic back handling in most use cases.

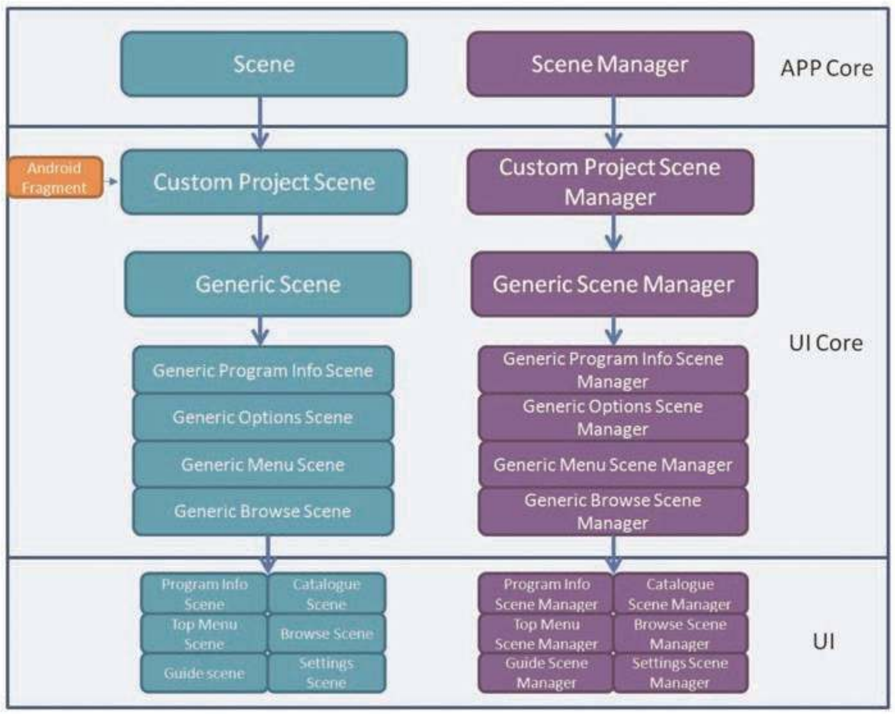


Fig. 14 – *Scene abstraction and hierarchy.*

All Scenes and all Managers are developed in the same code style which makes them easy to update, maintain and debug. As mentioned earlier, since we noticed that in most of the projects some Scenes are similar or contain some similar parts which can be reused we introduced generic Scenes and generic Scene Managers to serve as wrappers for similar Scenes (Fig. 14). The generalization greatly reduces the amount of boilerplate code and makes development and debugging process a lot easier and less time-consuming.

The way we treat Scenes and Scene Managers we also treat views. Since a lot of views are reused in different Scenes but also possess custom features not available in standard Android view toolkit we introduced generic views as an internal library which is developed based on the design specification. That enables us to reuse most of the views throughout an application.

3.3 SDK class

This is a Singleton class [5] which contains instances of all handlers. Handler methods are asynchronous and data is retrieved through the *AsyncDataReceive* callback. In order to obtain the needed data, a Scene will call its Scene Manager via its Scene listener. The Manager will then directly ask SDK for data via the appropriate handler and the data will be received by the *AsyncDataReceive* callback. Main handlers are listed in Fig. 15 [1].

WorldHandler	DatabaseHandler
AppHandler	AccountHandler
BackendHandler	ChannelsHandler
DisplayHandler	EpgHandler
PlayerHandler	NotificationHandler
VolumeHandler	PrefsHandler
TvHandler	ReminderHandler
VodHandler	TimeHandler
DeviceHandler	SearchHandler
F avoritesHandler	UpdateHandler
ParentalControlHandler	GuideHandler
RegionHandler	ConfignrationHandler
PaymentHandler	LanguageHandler
PackagesHandler	ProfilesHandler
Category Handler	TrialHandler
ItemlnfoHandler	BluetoothHandler

Fig. 15 - *List of main handler classes.*

The most common handler classes include:

1. WorldHandler – contains instances of registered Scene Managers and defines all available Scenes and screens inside an application. Every Scene Manager must be registered inside the WorldHandler in order to be recognized by the AppCore framework. The WorldHandler provides access to a Scene Manager that belongs to some other Scene. It contains *show()*, *hide()* and *destroy()* methods which are used for handling Scenes. It is in charge of communication between Scenes. It also handles user interaction, propagates user events to the active Scene and takes charge of memory consumption management – it clears memory by destroying Scenes which are no longer needed on screen. It also provides additional functionalities like saving Scene history so we can easily handle Scene back stacking and navigation without adding extra code logic.
2. AppHandler – enables the switching between different Scenes. It takes care of which Scene is displayed (active). When a Scene Manager's action is triggered the Manager will delegate an event to the corresponding Scene (depending on triggered action type). Once the AppHandler is notified that some Scene action is triggered it will submit an appropriate InformationBus event and ensure Scene switching.
3. BackendHandler – embeds support for login and communication with a backend server via HTTP protocol. It can be easily customized with plugins based on client's requirements. BackendHandler should implement all features stated in customer specification and support communication with backend servers. Fig. 16 shows BackendHandler communication diagram. Based on requirements a developer can add different APIs that need to be implemented (e.g. Favorites, Recommendation, Profiles, etc.).
4. DisplayHandler – manages display resolution, screen position, aspect ratio, color setup, etc.
5. PlayerHandler – sets the playback view, manages currently selected channel, getting channel list from backend, channel switching, etc.
6. VolumeHandler – manages volume and audio parameters.
7. TvHandler – manages active channel, previous active channel, next channel, channel switching, etc.
8. VodHandler – manages video on demand items.
9. DeviceHandler – manages devices (e.g. Bluetooth connections, USB connections), gets device status, lists available devices, adds, removes, updates devices, etc.

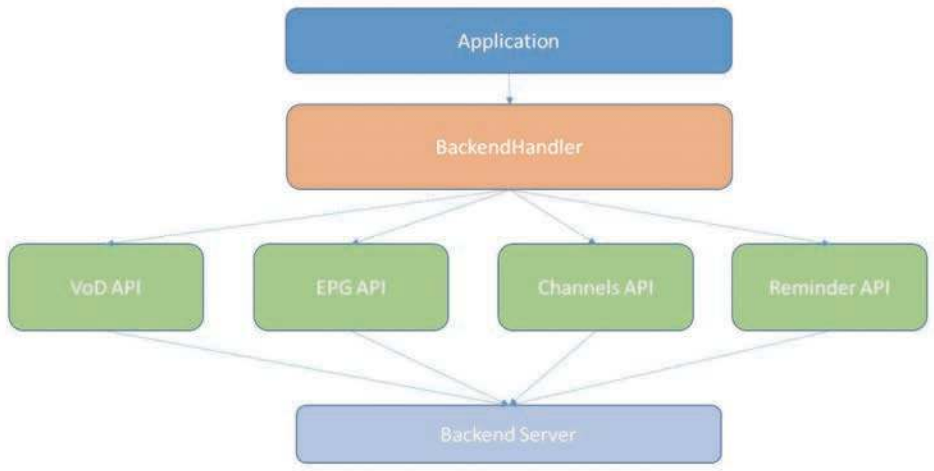


Fig. 16 – *BackendHandler communication.*

Fig. 17 shows an example of PlayerHandler implementation. When a user changes the channel, a signal is sent via UI and Core to the PlayerHandler which starts a backend service.

When a channel is changed the UI is updated via the appropriate callbacks (onZapCallback, onVideoPlayed, onChannelChanged).

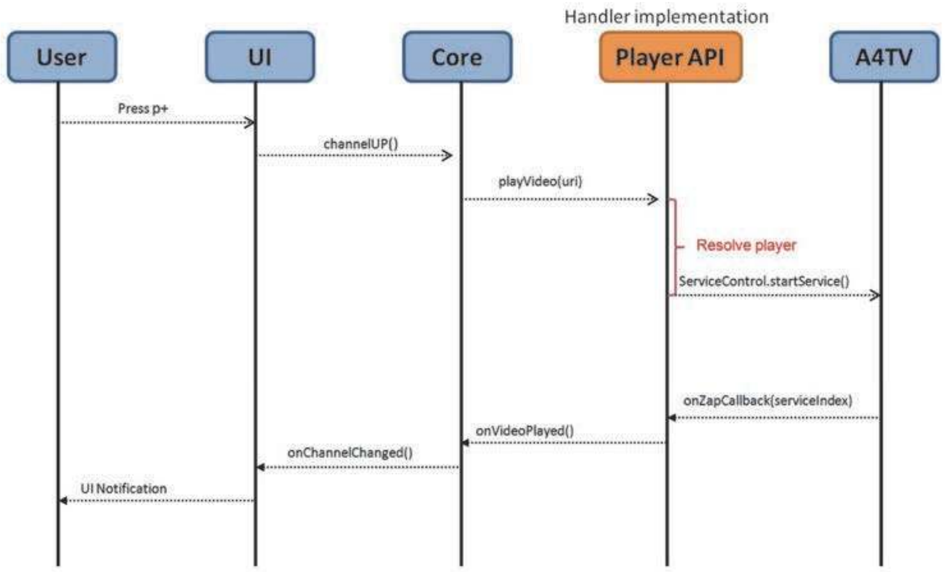


Fig. 17 – *PlayerHandler implementation example.*

3.4 Information Bus

Information Bus is a part of the Observer design pattern [8]. It is in charge of sending and receiving events between different parts of the application. Every event has an ID and a data object and can be submitted and received. Different modules of the application can be registered as event broadcasters and event receivers/listeners. Once the AppHandler is notified that some Scene action is triggered it will submit an appropriate Information Bus event.

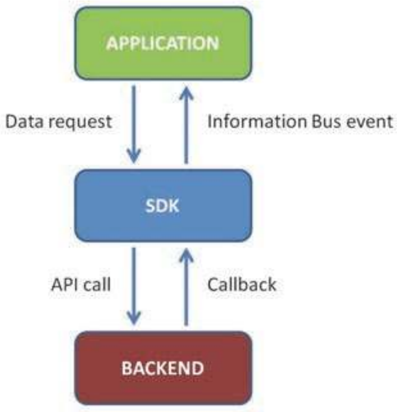


Fig. 18 - *Communication between application and backend.*

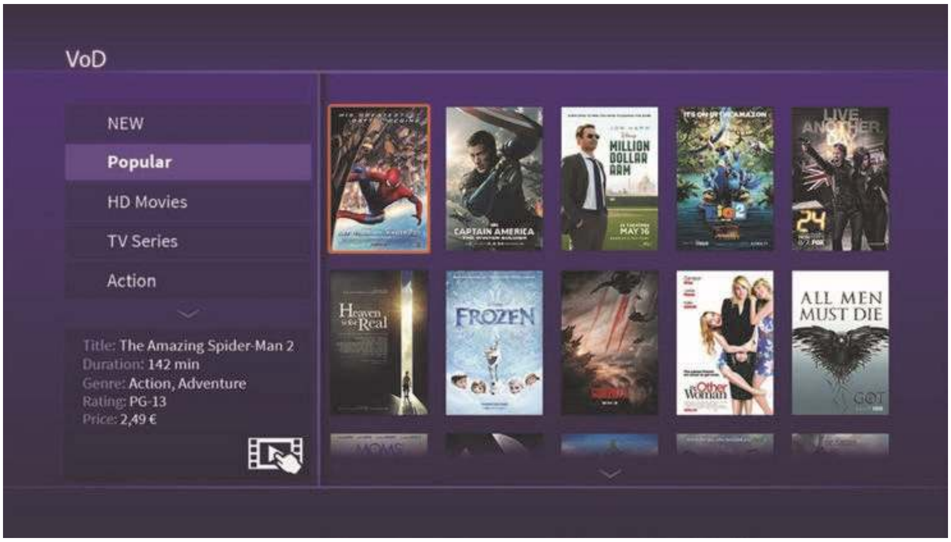
The Information Bus conveys the events (show, hide, destroy) from the Core's AppHandler to the MainActivity which is registered as the listener of these events [1]. The MainActivity then performs the corresponding actions on the Scene views so that the Scene switching mechanism can be implemented. The communication flow between different modules which is performed using Information Bus is shown in Fig. 18.

4 The Usual Scenario

4.1 Scene creation

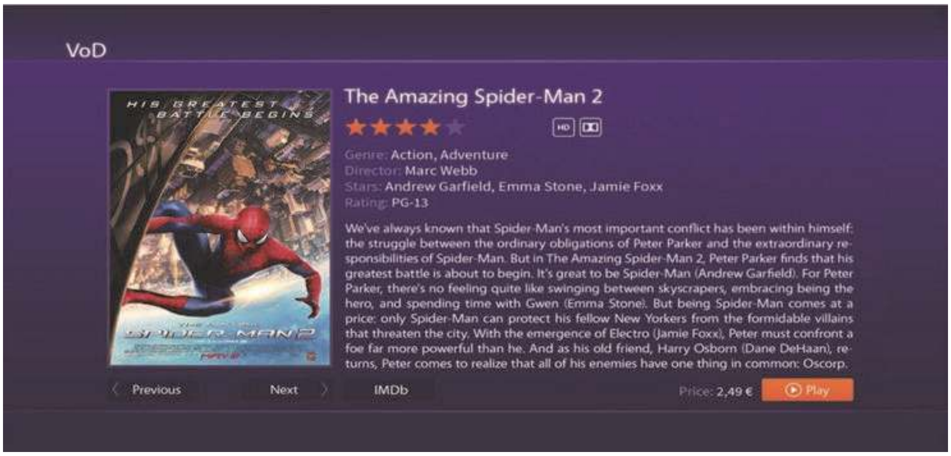
1. Create a Scene class and a layout XML file. The Scene extends the AppCore Scene. In the Scene itself override the *createView()* method and inflate the layout.
2. Create a Scene listener interface. The listener interface contains all of the methods needed by the Scene. It is used for sending requests to the Scene Manager.
3. Create a Scene Manager class. The Scene Manager extends the AppCore Scene Manager. Override the *createScene()* method and create the Scene object. At the end of the *createScene()* method call the parent *setScene()* method in order to save the Scene instance.

Fig. 19 shows an example of a simple VoD (Video on Demand) scene. The menu is on the left side and the list of movies is on the right side. Movie posters are retrieved via the ImageLoader class which enables the smooth loading of multiple images (Fig. 21). When an item is clicked a VoD details scene is shown (Fig. 20).



Title: The Amazing Spider-Man 2
Duration: 142 min
Genre: Action, Adventure
Rating: PG-13
Price: 2,49 €

Fig. 19 – *VoD scene.*



The Amazing Spider-Man 2

Genre: Action, Adventure
Director: Marc Webb
Stars: Andrew Garfield, Emma Stone, Jamie Foxx
Rating: PG-13

We've always known that Spider-Man's most important conflict has been within himself: the struggle between the ordinary obligations of Peter Parker and the extraordinary responsibilities of Spider-Man. But in The Amazing Spider-Man 2, Peter Parker finds that his greatest battle is about to begin. It's great to be Spider-Man (Andrew Garfield). For Peter Parker, there's no feeling quite like swinging between skyscrapers, embracing being the hero, and spending time with Gwen (Emma Stone). But being Spider-Man comes at a price: only Spider-Man can protect his fellow New Yorkers from the formidable villains that threaten the city. With the emergence of Electro (Jamie Foxx), Peter must confront a foe far more powerful than he. And as his old friend, Harry Osborn (Dane DeHaan), returns, Peter comes to realize that all of his enemies have one thing in common: Oscorp.

Price: 2,49 €

Fig. 20 – *VoD details scene.*

4.2 Scene destruction

1. A Scene Manager triggers the destroy action and calls the Scene's *destroy()* method.
2. The Scene notifies the AppHandler that it is destroyed.
3. The AppHandler submits the Scene's destroy event.
4. The destroy event is received by the MainActivity which then destroys the Scene itself.

5 Optimizations

5.1 Global System Keys

To support this feature we had to introduce a custom BroadcastReceiver available inside Android SDK to be able to handle and detect system keys from the remote controller. Inside GlobalKeyReceiver we have implemented custom logic and supported custom application lifecycle in order to manipulate application state without losing any available information.

5.2 Util classes

These classes contain additional logic that facilitates application development. The classes include:

1. FontCache/TypeFace provider – provides optimization and additional logic to speed up and support usage of different fonts in the application. It is a dedicated module for loading and handling custom typefaces on Android. It's implemented as a custom cache for Android fonts. Android OS is not optimized for Typeface updating in runtime because it requires a lot of CPU power in order to load a large bitmap defined in a font file that contains all letters and glyphs. Android OS loads a single bitmap image of each glyph. If many views must have different font styles in case of selected and non-selected state, it requires a lot of time for OS to render the proper font and to update it on screen. This class facilitates the process by reducing memory consumption and CPU overload when changing fonts at runtime.
2. UnitConverter – enables support for multiple resolutions.
3. KeyMapper – provides emulation and mapping different keys from keyboard and remote controller enabling us to support Emulator mode in the development process.
4. Image loader – a customizable module that can be easily adapted to the requirements and specifics of the project. If needed, the embedded image cache can be replaced by custom implementation or by some open source image loader libraries. The image loading process is shown in Fig. 21.

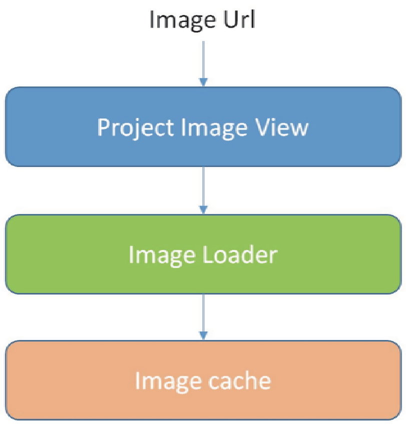


Fig. 21 – *Image loading.*

5.3 Scene and Fragment handling

Scene entities have custom implementation optimized for the Android platform. Scenes are implemented as separated Android fragments which are managed via custom Android's FragmentManager [1]. FragmentManager is in charge of disposing fragments that are not needed in memory any more. Each custom fragment has a built-in logic for handing its own lifecycle so when a fragment detects action for hiding or disposing it handles its own resources without the need for developer's intervention. This significantly reduces the amount of code.

FragmentManager supports adding, removing, replacing, hiding and disposing fragments and all standard Android operations upon fragments like custom entrance and removal animations. WorldHandler takes care of the hierarchy tree and provides information of currently and previously used scenes which helps developers to detect memory leaks and prevent issues in screen flows.

6 Measurement

The TV application development is a relatively new field of software development and not many developers are dealing with the problems described in this paper so there is not much currently available literature. Since our team is one of the pioneers in this field the measurement was based on our considerable experience in constructing TV applications [1].

Three projects with the same set of features developed by our team can be seen in Fig. 22. A different approach was used for each of these projects – using no framework whatsoever, using our framework and using Android's Leanback library. The approach with no framework is the hardest and slowest because there is no starting point and the development has to be done from scratch. The approach with Leanback library has its advantages in terms of templates and predefined features but it is hard to customize. The approach with our framework is the fastest since we have a good starting point – a basic TV application with basic features (Channel list, Channel zapper, TV guide, VoD, etc.) which can easily be customized.

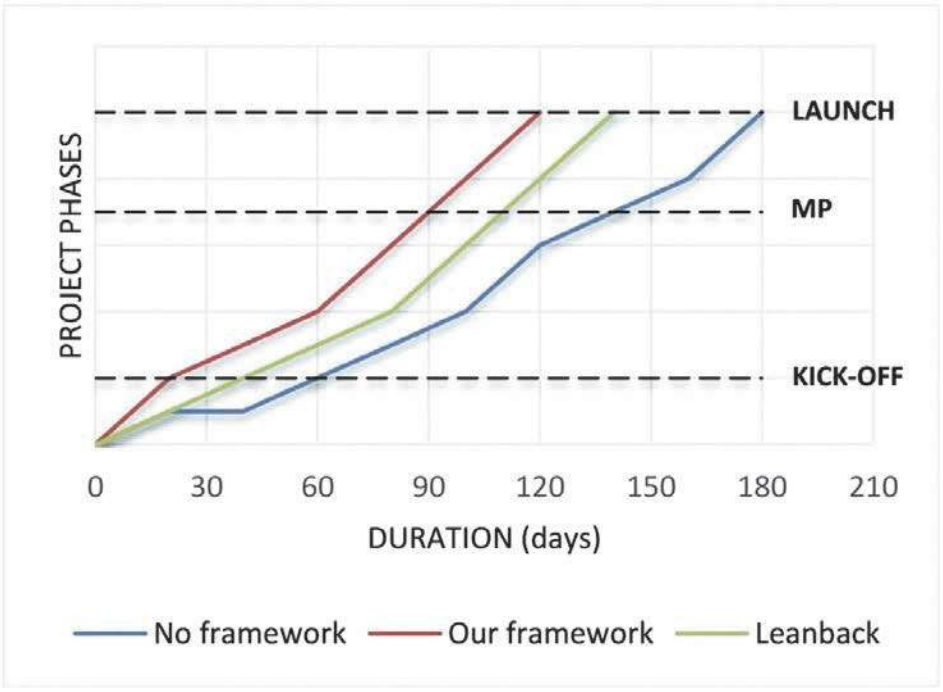


Fig. 22 – *Measured project timelines.*

7 Conclusion

In this paper, we have presented our approach to TV application development. Our experience shows that the usage of this framework library considerably reduces the amount of code in the application, makes the code more readable and easier to maintain and also shortens the time needed for coding thus making TV application development significantly more effective [1]. By utilizing this framework we managed to reduce time and effort for TV application development. The possible downside of this approach is that it can be complicated during the first few weeks of development for programmers who haven’t had any previous experience with TV application development but that gets compensated in the later phases of the development. The other possible downside is that if a programmer uses generalization uncritically it may cause code regressions. Although this method is more efficient in comparison to other methods of TV application development [9 – 13] there is still room for further improvements such as the introduction of more common features and sub-library modules for TV-centric applications. Other improvements include data and performance optimizations as well as support for different Android versions and screen resolutions within the framework by utilizing dynamic UI support.

8 Acknowledgment

This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, under grant number: III_044009_2.

9 References

[1] N. Kovačev, V. Ilkić, D. Nađ, N. Vranić: Framework Library With Guidelines for Effective TV Application Development, Proceedings of the 5th International Conference on Electrical, Electronic and Computing Engineering, IcETRAN 2018, Palić, Serbia, June 2018, pp. 1160 –1163.

[2] Official Java website, https://www.oracle.com/java/

[3] Official Android website, https://www.android.com

[4] Official website for Android application developers, https://developer.android.com/

[5] K. Mew: Android Design Patterns and Best Practice, Packt Publishing Ltd, Birmingham, UK, 2016.

[6] J. Horton: Android Programming for Beginners, Packt Publishing Ltd, Birmingham, UK, 2015.

[7] B. Phillips, C. Stewart, B. Hardy, K. Marsicano: Android Programing: The Big Nerd Ranch Guide, 2nd Edition, Big Nerd Ranch Guides, Atlanta, USA, 2015.

[8] H. Schildt: Java: The Complete Reference, 9th Edition, McGraw-Hill Education, New York, USA, 2014.

[9] E. G. Lima, R. de Andrade Lira Rabelo: An Architectural Model for Communication Between the iDTV and Mobile Devices, Proceedings of the 2015 International Conference on Computing, Networking and Communications (ICNC), Garden Grove, USA, February 2015, pp. 1102 – 1105.

[10] Y. Wahyu, F. Oktafiani, Y. P. Saputera: Development of Set Top Box (STB) for DVB-T2 Standard Television Based on Android, Proceedings of the 2014 8th International Conference on Telecommunication Systems Services and Applications (TSSA), Kuta, Indonesia, October 2014, pp. 1 – 4.

[11] Z. Xu, L. Yang, S. Cao: Design and Implementation of Mobile Lightweight TV Media System Based on Android, 7th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, August 2016, pp. 730 – 733.

[12] M. Milanović, B. Pavlović, I. Petrović, T. Maruna: One Implementation of UI TV Application on Android STB, 21st Telecommunications Forum Telfor (TELFOR), Belgrade, Serbia, November 2013, pp. 724 – 726.

[13] S. Pravin, R. BalaKrishnan: Set Top Box System with Android Support Using Embedded Linux Operating System, Proceedings of the IEEE International Conference on Advances in Engineering, Science and Management (ICAESM-2012), Nagapattinam, India, March 2012, pp. 474 – 478.