A Pattern Language for Mobile Application Development

Steve Ng
National University of Singapore
steveng.1988@gmail.com

Bimlesh Wadhwa
National University of Singapore
dcsbw@nus.edu.sg

ABSTRACT
The number of developers for the mobile application platform is increasing and there is always a set of inherent design problems such as handling data synchronization or application design that developer will have to tackle in order to create an optimized mobile application. In this paper, we introduce the notion of pattern language for the mobile application development.

1. INTRODUCTION
Mobile application design involves many design considerations such as the user interface of the application, device constraints and the architecture of the application [1]. In this paper, we look at mobile application design in both the front and backend design. Design patterns are enhanced guidelines in software design that follow a problem-solution structure within a given context or situation [2]. It is reusable in many situations as a pattern captures the commonality that exists in design found in many different applications. Particularly, we provide a pattern language initiative following which future developers can reference when designing mobile applications. Pattern language refers to a set of design patterns that are related and can be used together to solve a problem [3]. The contribution of this papers focuses on discovering patterns that address non-functional requirements in mobile application development. These patterns will then be used to guide developers in the design of mobile applications. In doing so, we hope to provide new mobile application developer with a guideline in developing each component.

Most of the design patterns studied for the mobile application design are related to the aesthetic aspect of the application. *Mobile Design Pattern Gallery: UI Patterns for iOS, Android and More* by Niel [4] have a good description on designing the navigation on mobile application. *Welie and Trattenberg* [5] have created a list of interaction patterns in user interfaces. Google has also provided a pattern write-up for developers of Android application on the best practice for interaction [6]. Similarly, Apple has also provided a set of guidelines and principles towards designing for Apple application.

McCormick and Schmidt presented a list of data synchronization patterns in Mobile Application Design [7]. We find the patterns listed in the paper relevant and have included them into our pattern language.

Gu, March and Lee came up with the idea of offloading expensive computational tasks to cloud which will reduce the battery consumption of the mobile application *[8]*. However, the work only presents the concept; developers interested would be required to dig deeper into the possibilities of implementing it, which discourages them often. This leads to the need of formulating a design pattern, which will not only contain the concept of the solution but also implementation details such as the components required.

2. MOBILE APPLICATION PATTERN LANGUAGE (MAPL)

The main intent of MAPL is to serve as an overview of the best practices used in mobile application design for different area of non-functional concerns such as architecture for maintainability or testability, data synchronization for optimizing network usage. In order to provide such an overview, design patterns are grouped into area of concerns that the design pattern is attempting to tackle.

Mobile application developers can refer to this pattern language and look for patterns that can be applied to their current mobile application in an attempt to refactor their mobile application design or solve a problem elegantly. On the other hand, students or anyone who are looking into mobile application development can refer to this document for an understanding of the possible area of concerns for mobile application development and the possible solutions to them. However, readers are assumed to have general idea of what design patterns are and knowledge of basic design patterns such as singleton pattern [9].

By no means is this pattern language comprehensive in covering all possible area of concerns, but this will be a start to inspire other industry mobile application developers to also contribute some of the design patterns that they have applied in their mobile application.

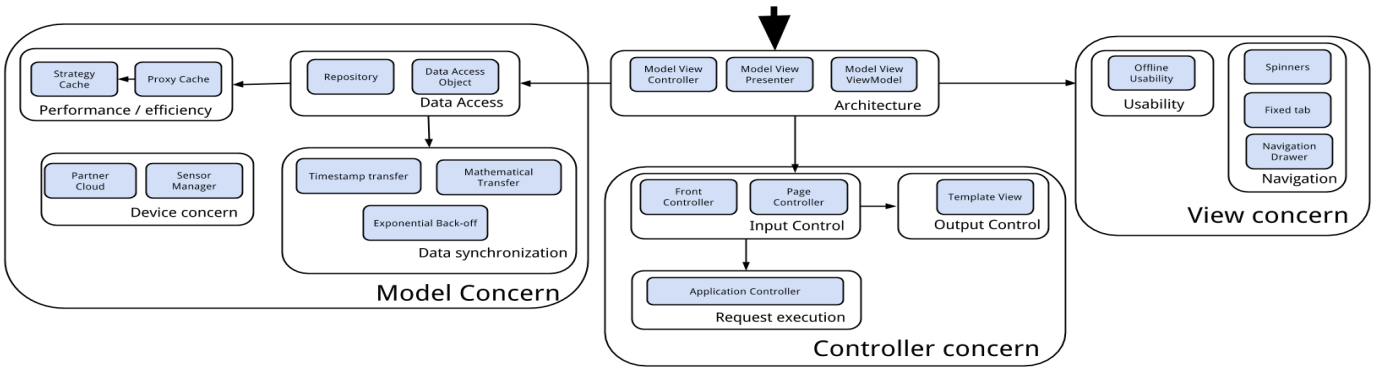

Figure 1: MAPL concerns

As shown in figure 1, the pattern language contains 21 design patterns distributed into 4 sub-domains and they are:

Architecture
This problem area deals with the baseline architecture of your program; they help in structuring your program into general area of concerns (model, views and controllers) to improve maintainability and testability of your mobile application. *Patterns included: Model View Controller, Model View Presenter and Model View ViewModel.*

View concern
This problem area deals with both the user experience and user interfaces concerns. They help in providing an idea of how to design your UI structure or how developer can provide a better overall user experience in their mobile application. *Patterns included: Spinners, Fixed Tabs and Navigation Drawers and Offline Usability*

Controller concerns
This area deals with concerns about requests (HTTP) coming into the mobile application. It includes issues such as refactoring for reusability of cross cutting concerns such as logging or authentication. *Patterns included: Front Controller and Page Controller, Template View and Application Controller*

Model concerns
This problem area covers a huge domain such as concerns in data access from both database and REST API or reduces the battery consumption of the mobile application. This area help developers in understanding the various possible problems in data access or device related concerns and provides possible solutions to them. *Patterns included: Strategy Cache and Proxy Cache, Repository and Data Access Object, Timestamp transfer, Mathematical Transfer, Exponential Back-off, Partner Cloud and Sensor Manager*

3. ARCHITECTURE PATTERNS

The architectural Patterns in this section here are not of much relevance for mobile native developer as the architecture pattern is dictated by the platform. For example in Android, developers have to define views inside the layout folder, and the controllers are the activity class that will render the views and listen for events from the user. However for Android, there is no explicit model class. Each application could have different needs; for example, Instagram would use the model classes for logic on uploading or application of filter on images. On the other end of the application spectrum, a chess application would not require data from external source; hence, the model would contain the algorithm to calculate the next best AI move instead.

However, the patterns in this section are highly relevant for mobile web developer. In this section, we would elaborate on the kind of architecture design pattern that are utilize by different JavaScript frameworks. This section should allow mobile web developers to decide on which framework might be a better choice. However, do note that this should not be the only factor in deciding which frameworks to use. Other factors such as the community support of the framework and documentation should be considered too.

3.1 MVC

Intent
Separate the User Interface code from the application's logic

Forces
A mobile application like any application can evolve rapidly through iteration. Changing any component (user interface, logic or data source) should not affect any other components.

Solution
Separate your application into three key components: model, view and controller.

Structure

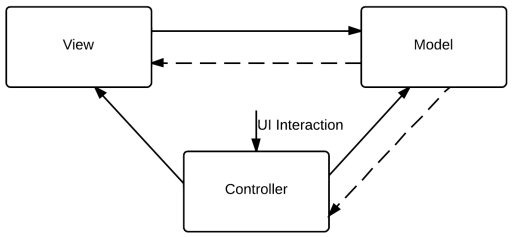

Figure2: MVC Structure

- Model: This component contains the business logic and data access of the application.
- Controller: This component listens for UI interaction events from the user, and handles the request routing.
- View: This component contains the look of the mobile application.

Consequences

Benefits

- Increase maintainability of the system with a clear separation of concern.

Liabilities

- Views are calling the model directly, which may pose a security concern if there are model data that are not supposed to be shown and no security mechanism is in place.

Known uses

Maria.JS framework [10]

Related Works

This pattern is documented in the POSA (Pattern-Oriented Software Architecture) series [2].

3.2 MVVM

Intent

Separate the user interface code from the application's logic

Forces

User interfaces evolve rapidly; changes to the UI should not affect the application logic. Testability of the application is of concern too. You also want to be able to develop in parallel with the UI designer working on the views and the other programmers on the other components. Finally, for a more complex application, they can be multiple views for the same set of data (admin/user) and you would like to reuse the same controller for the set of views.

Solution

Separate your application into three key components: model, view and ViewModel.

Structure

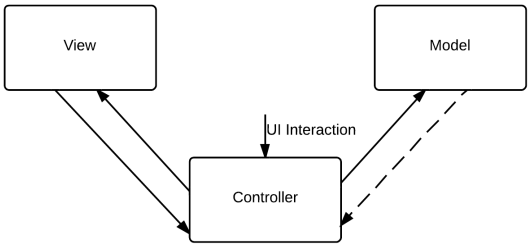

Figure 3: MVVM Structure

- Model: This component contains the business logic and data access of the application.
- ViewModel: contains the values and method that the view can call
- View: utilize data-bind to get both values and return UI events

Consequences

Benefits

- Improve overall maintainability of the code
- Increases testability of the application than MVC as view is only coupled to ViewModel
- Support parallel development of both components

Liabilities

- Data-binding can be complicated to understand initially

Known uses

The following three javascript framework implements this pattern: KendoUI [11], Knockout.JS [12], angularJS [13]. A developer has also created a framework to implement such pattern in Android [14].

3.3 MVP (Passive View)

Intent

Separate the User Interface code from the application's logic (Similar to MVVM)

Forces

User interfaces evolve rapidly; changes to the UI should not affect the application logic. Testability of the application is of concern too. You also want to be able to develop in parallel with the UI designer working on the views and the other programmers on the other components. Finally, for a more complex application, they can be multiple views for the same set of data (admin/user) and you would like to reuse the same controller for the set of views.

Note: This is similar to MVVM. MVP Passive View is another alternative way of implementation.

Solution
Separate your application into three key components: model, view and Presenter.

Structure

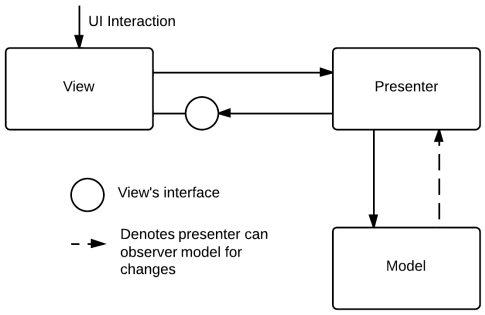

Figure 4: MVP Structure

- Model: This component contains the business logic and data access of the application.
- Presenter: Has method that Views can call when UI event is triggered, update views through an interface provided by view
- View: call presenter method when UI event is triggered.

Consequences

Benefits

- Improve overall maintainability of the code
- Increases even more testability of the application (highest among the 3 architecture pattern)
- Support parallel development of both components (views and presenter)

Liabilities

- More effort is required than the other 2 pattern above due to an interface to update view is required

Known uses
There is no mobile web JavaScript framework that supports this pattern off the shelf currently. This pattern is used more often for Google Web Toolkit project: particularly GWTP. [15]

Related Works
Martin fowler has documented this pattern for the enterprise domain [16].

4. MODELS CONCERNS

Model concerns are highly relevant regardless of mobile web or native mobile application; in both type of mobile applications, there will definitely be business logic and most of the time there will be some form of CRUD (create, read, update and delete) operations as too. In this section, we would elaborate the kind of design pattern developers can use in their models. For example, to separate business and data access logic, a combination of repository and data access object design pattern can be used. Also, to improve performance of your mobile application, adding proxy cache will help too.

As shown below are the design patterns in this category and how the patterns can be linked together. Some patterns are not included in this paper as extensive documentation can be found elsewhere. Repository pattern can be found from Microsoft Developer Network documentation [17]. Data Access Object can be found from Oracle's documentation [18]. *McCormick and Schmidt of Data Synchronization Pattern in Mobile Application Design* documented both timestamp transfer and mathematical transfer [7].

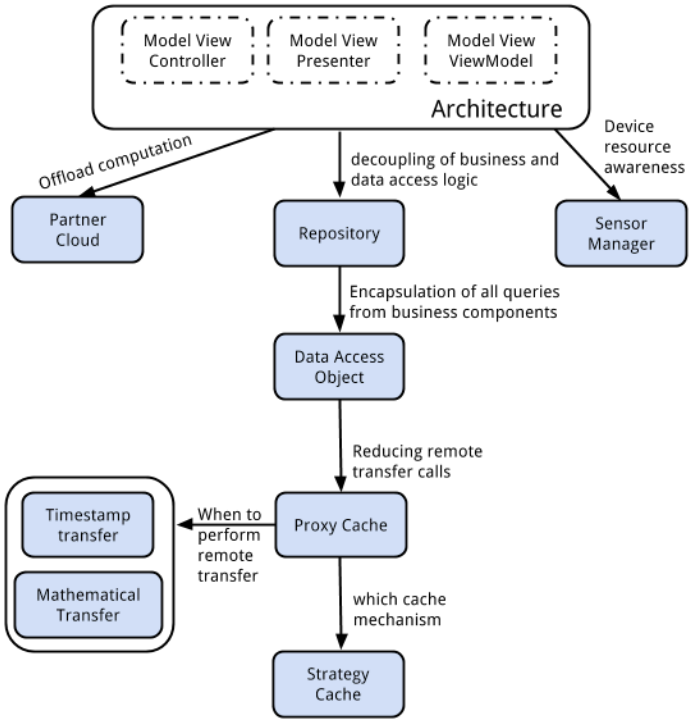

Figure 5: Model concerns

4.1 Proxy Cache

Intent
Re-use data instead of pulling the same dataset when the application requires data.

Forces
Some data does not change at all such as historical stock price, pulling the data each time the user request on it wastes bandwidth.

Solution
Cache data that will not change into device's cache

Structure

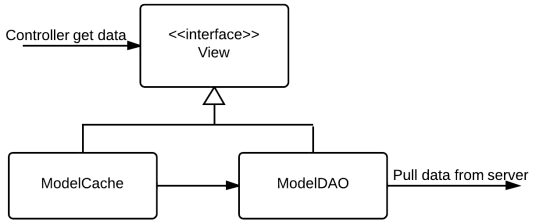

Figure 6: Proxy Cache Structure

- *Interface for Model*: setter and getter method that will be implemented by both ModelDAO and ModelCache
- *ModelCache*: Implement the method stated for the model.
- *ModelDAO*: Data access object, queries from the data source. (refer to data access object pattern)

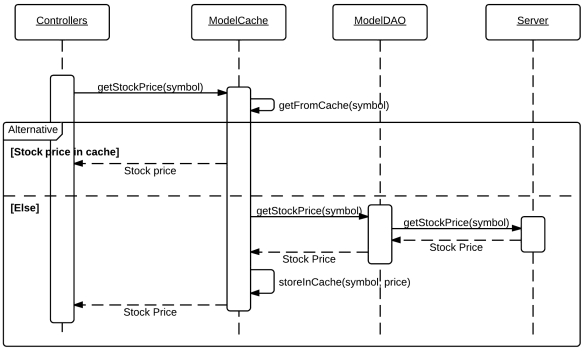

Figure 7: Proxy Cache Interaction

Consequences

Benefits

- Reduces network calls which saves mobile devices bandwidth
- Increase performance when the data is already in cache

Liabilities

- Developers have to understand the different ways to cache and also the cons of each caching strategy. For example caching in-memory will consume the memory space of the mobile device and caching in local storage occupies disk storage.

Known uses

Android's Volley networking library utilized some form of this pattern for caching when making network request [19].

4.2 Strategy Cache

Intent

Common location for caching mechanism

Forces

There can be multiple ways to cache data (in plaintext, in local database or even in memory). Directly accessing such caching mechanism will result in duplication of code.

Solution

Have a default cache interface that is implemented by each caching method.

Structure

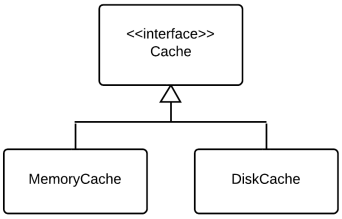

Figure 8: Strategy Cache Structure

- *Interface for Cache*: methods that all the cache should implement such as storeInCache and getFromCache
- *MemoryCache*: Implementation of caching in memory
- *DiskCache*: Implementation of caching in disk

Consequences

Benefits

- Reduces duplication code
- Increased flexibility to add or edit cache mechanism

4.3 Exponential back-off

Intent

When retrying failed data synchronization, instead of retrying immediately, retry at after a delay.

Context

A mobile application is required to make a lot of data synchronization calls, however at times they might fail but the data still must be sent to the remote server. Hence the mobile phone retries, however the mobile phone might be in a subway, constant immediate retry of data synchronization is fruitless and will drain the battery instead.

Forces

Battery usage **–** The battery consumption of a mobile application should be as minimal as possible.

Usability – A sudden disrupt of connectivity should not disrupt the usability of the application.

Solution

Implement a retry mechanism that retry at a fixed interval and if it fails again, retry at an even later interval.

Structure
There is no structure for this pattern as it is a timer that increases after each retry.

Consequences
Benefits
- Prevents unnecessary network calls, which reduce battery consumption of the application.

4.4 Partner Cloud

Intent
Reduces the battery consumption and improve the execution speed of the application by offloading computational intensive tasks on cloud.

Context
Mobile devices have limited computational power and running algorithms which take a considerable amount of time not only cause the user to wait if they are dependent on the algorithm which result in bad usability, it also drains the battery life.

Forces
Battery usage **–** The battery consumption of a mobile application should be as minimal as possible.

Solution
Instead of using the mobile phone to compute all the tasks required of the mobile application, partition the application with the computationally intensively tasks to the cloud.

Structure

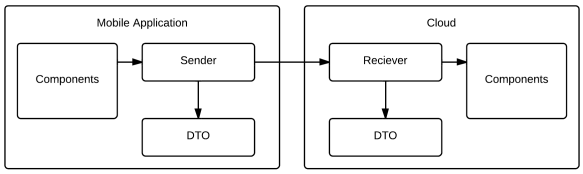

Figure 9 : Partner Cloud Structure

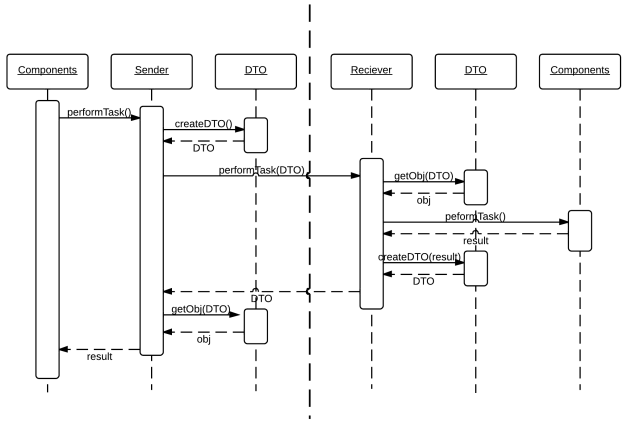

Figure 10: Partner Cloud Interaction

Consequences
Benefits
- Computations that wasn't possible due to the processing power of the device is no longer a concern.
- Battery life/ availability of the application is improved – Mobile resources are less heavily taxed as computationally intensive tasks are offloaded to cloud.
- Response time of the application increases – cloud will compute the tasks at a much faster speed as compared to mobile device.

Liabilities
- Low network bandwidth might results in an even slower execution time than computing locally on the mobile.
- Unstable network might result in not receiving the cloud's reply, which will result in an even slower execution time.
- Offline usability – the mobile phone should still be able to compute the intensive algorithm by itself if there is not network available.
- Security concerns – since data are transmitted over the network, there is a chance of data being eavesdropped.
- If overused, it might create a heavy load on the server.

Known uses
No examples of industry mobile application that utilize this pattern are available. A possible use case is to offload calculation of the next move by an AI in a chess game to the cloud.

Related patterns
Data Transfer Object Pattern

Related works
CloneCloud [20] is a system in place that also offload computationally intensive task to the cloud. It automatically and seamlessly off-loads part of the execution of mobile applications from mobile devices onto the device clones in a computational cloud. The concept is worth a try if the offloaded task requires the state of the Android app, however the complexity of this is much higher as well.

μCloud [21] shares the same key concept of separation of components (either in Android or in Cloud) and having the components call one another. However, they didn't take into the account of the possibility that the algorithm might require information such as the current state of the Android application. Also, communication protocol component was not present as well.

4.5 Sensor Manager

Intent
Make device optimal decision based on the current application's connectivity and battery state.

Context
Mobile devices have limited battery life, and consumer typically has a limited data usage on the mobile imposed by the telecoms. Also, Android can display each individual application's data and

battery usage. If a mobile device has high data or battery usage, user will be inclined to uninstall them.

Forces

Network usage – Mobile application have to be prudent in network usage when the user is on 3G or 4G.

Battery usage –Battery consumption of mobile application should be as minimal as possible.

Usability – Mobile application has to be usable still.

Solution

Whenever performing a data synchronization task that might potentially drain battery or consume large amount of data (determined by developer), take into consideration of the user's current battery and connectivity status.

Structure

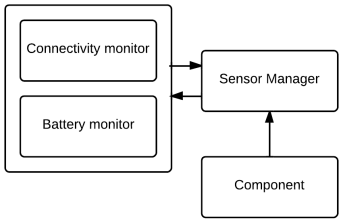

Figure 11 : Sensor Manager Structure

- *Battery Monitor* – this component role is to provide the battery status of the mobile phone *Connectivity Monitor* – this component role is to provide the connectivity status of the mobile phone
- *Component* – this set of components are the main logic of the mobile application, it will ask the sensor manager the best way to perform each data synchronization task.
- *Sensor Manager* – Based on the input from connectivity and battery monitor, this component will make the decision for the best way for each data synchronization task.

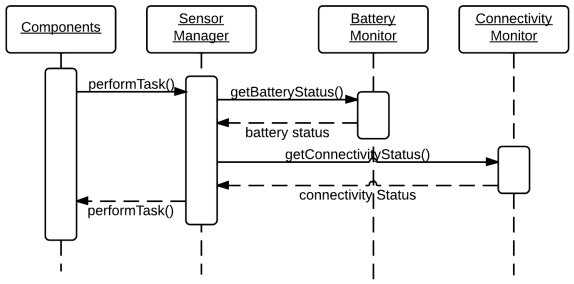

Figure 12 : Sensor Manager Interaction

The above shows the general interactions between the components. When the component wishes to perform a data synchronization task, it will ask the sensor manager (with the performTask() method). Next, the sensor manager will based on the current connectivity and battery state, it will reply the best way to perform a task.

For example:

1) Streaming video -- if the connectivity is slow, sensor manager will inform the component to stream low quality
2) Sending big files – if the user is on 3G, sensor manager will inform component not to send now but to only send when it's connected to wifi. (refer to Offline Usability pattern for a way to implement that sends automatically when the user connects to wifi). Also if there is low battery, the sensor manager can advise only to send the file when the device is charging or reach a satisfactory battery level

Consequences

Benefits

- Mobile data usage is improved – tasks that consumed huge data plan can be performed only when there's Wi-Fi.
- Battery consumption improves.

Liabilities

- Usability might suffer if the interaction is not handled correctly – For example even though the sensor manager might advise not to send the file, the final say should be given to the user.

Known uses

Funf, is an open sensing framework that utilize a form of this pattern with *SensorManager* as *FunfManager* and *monitor* as *probe [22]*.

This pattern is also common in video streaming application and Android Play store. As shown in figure 13 is a video streaming application which keep tracks of the user's mobile connectivity status and selects video quality accordingly for streaming. Figure 14 shows Android Play store informing user that a download task will potentially cause data usage charge or delay if the user wants to download when the device is not on wifi.

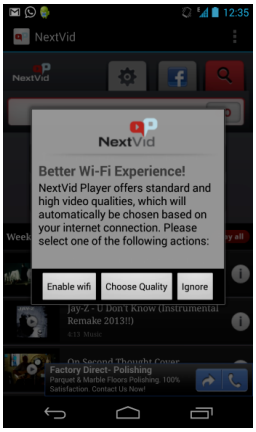

Figure 13: NextVid App

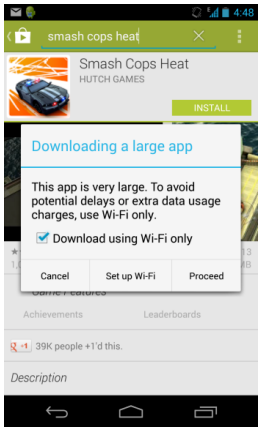

Figure 14: Android Play Store

Related patterns
Observer Pattern
Offline Usability

Related works
Reto Mier, a tech lead at Google discussed a similar form of this pattern [23].

5. CONTROLLERS CONCERNS

Controllers deals with handling user's interaction at the UI and calling the appropriate model if there is any CRUD to be performed. Figure 15 shows the design patterns that can deals with the possible concerns found in the controller. Both mobile web and mobile native developers will find these patterns very useful. All the patterns listed in this category are documented in the POSA series. [24]

6. VIEWS CONCERNS

Views concerns deals with structuring of the user interface or providing a good user experience for the user. As shown below are the design patterns in this category. Views pattern are very well documented for mobile applications in the different literatures, hence, mobile developers who are interested to know more can search over the net for mobile application UI design pattern.

The three navigation patterns in Figure 16 are documented in Android's developer documentation [25][26][27].

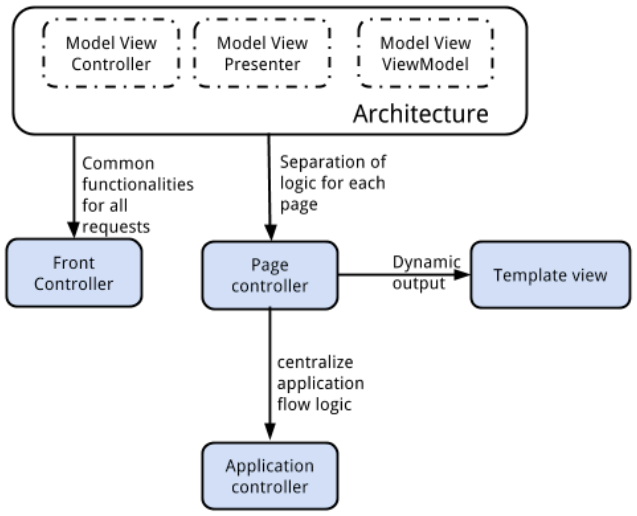

Figure 15: Controller concerns

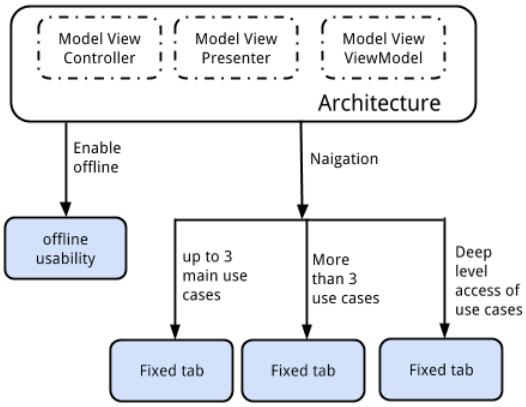

Figure 16: View concerns

6.1 Offline usability

Intent
Allow user to perform data synchronization tasks even when they are offline.

Context
Mobile devices stays connected to the Internet most of the time. However there are cases when the connectivity can be disrupted (in the subway or when the network is congested). In the times when there's no connectivity, user would still like to perform data synchronization task such as sending email or sending a message.

Forces
Usability –the usability of the application should not be affected by disruption of connectivity.

Solution
Whenever user does not have a connection and perform a data synchronization task, add the task to an queue and perform it when the user is back online.

Structure

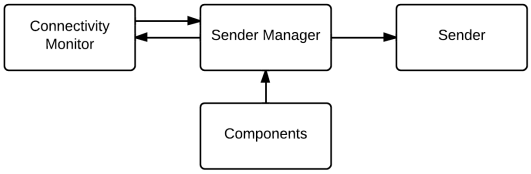

Figure 17: Offline Usability Structure

- *Connectivity Monitor* – this component role is to provide the connectivity status of the mobile phone
- *Sender Manager* – This component manages the sending of message. When it receive a message, it will check if there is connectivity and if there isn't, add the message to a queue and listen for publish event from the monitors and ask the sender to send the message from the queue as soon as connectivity is restored. If there is a connection, ask the sender to send immediately.
- *Component* – this set of components are the main logic of the mobile application.
- *Sender* – this component provides the method on communicating with the cloud's receiver.

Figure 18 shows the general interactions between the components. When the component wishes to perform a data synchronization task, it will pass the message to the sender manager to perform the task. The sender manager will then check for the connectivity state of the mobile phone. If there is connectivity, it will perform the data synchronization task.

If there is no connectivity, the sender manager will add the message to a queue and listen for event from connectivity monitor. The moment there is a connection, the connectivity monitor will send an event to the sender manager where the sender manager can loop through all the messages in the queue and send it.

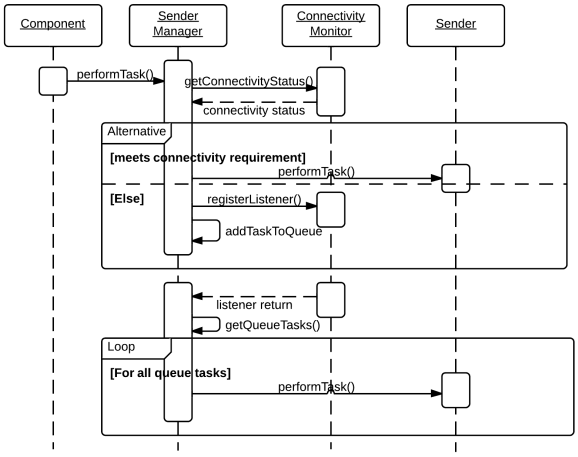

Figure 18: Offline Usability Interaction

Consequences

Benefits

- Usability improves – user will not get error message trying to perform data synchronization when they are lost connectivity. Also if there is low battery, huge data synchronization can be delayed till the user has charged the phone till a certain level.

Liabilities

- Developer have to have a way to inform the user that the data synchronization task is added to the queue or the user might feel that it's a bug from the application when the user checked and noticed that the task was not performed.

Known uses

This pattern is heavily used in the industry. If a user were to use Google's Gmail application to send an email when they are not connected, Google adds the email to the queue and only send when the user is connected back to the Internet.

Whatsapp [28] utilize a form of this pattern too, if a user were to send a message when they are not connected, Whatsapp will add the message to the queue and send only when the user is connected back to the Internet.

Related patterns

Sensor Manager

Related works

This pattern was discussed during the Google IO 2012 by Reto Meier, a tech lead at Google [23].

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a pattern language initiative by compiling 20 design patterns into 4 categories: architecture, controller, view and model concerns. We believe this is useful for new mobile application developer to understand the different type of concerns for mobile application. However, this compilation by no means is comprehensive to cover all the possible concerns of mobile application domain. In the POSA series, a total of 13 possible concerns areas have been identified for the enterprise application domain and we've only covered 4 in this paper for the mobile application domain. Future work will be to continue exploration of possible concerns and design patterns and document the pattern down into the same format as this paper.

8. ACKNOWLEDGEMENTS

We would like to thank our shepherd Prof. Youngsu Son for his useful feedback for polishing the paper. We would also like to thank the following people for their help reviewing in this paper at AsianPLoP 2014: Ademar Aguiar, Ed Fernandez, Oscar Encina, Yu Chin Cheng and Emilian Tramontana.

9. REFERENCES

[1] Chapter 24: Designing Mobile Applications. (n.d.). Retrieved from http://msdn.microsoft.com/en-us/library/ee658108.aspx

[2]Buschmann, F. Pattern-Oriented Software Architecture Volume 1: A System of Patterns. Wiley, 1996.

[3] Alexander, C. (1977). A Pattern Language: Towns, Buildings, Construction. USA: Oxford University Press. ISBN 978-0-19-501919-3.

[4] Mobile Design Pattern Gallery. (n.d) Retrieved from http://www.mobiledesignpatterngallet.com/

[5] Welie, M. V., & Trætteberg, H. (2000). Interaction patterns in user interfaces.

[6] Patterns | Android Developers. (n.d.). Retrieved from http://developer.android.com/design/patterns/index.html

[7] McCormick, Z., & Schmidt, D. C. (2012). Data Synchronization Patterns in Mobile Application Design.

[8] Gu, Y., March, V., & Lee, B. S. (2012). GMoCA: GreenMobile Cloud Applications.

[9] Richard, E. H. Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series). Addison-Wesley Professional, 1995.

[10] Michaux, P. (n.d.). Maria - The MVC Framework for JavaScript Application. Retrieved September 9, 2013, from http://peter.michaux.ca/maria/

[11] Kendo UI - jQuery HTML5 framework for desktop, mobile app development, HTML5 data visualization. (n.d.). Retrieved September 9, 2013, from http://www.kendoui.com/

[12] Knockout : Home. (n.d.). Retrieved September 9, 2013, from http://knockoutjs.com/

[13] AngularJS — Superheroic JavaScript MVW Framework. (n.d.). Retrieved September 9, 2013, from http://angularjs.org/

[14] android-binding - Providing a framework that enabes the binding of android view widgets to data model. It helps to implement MVC or MVVM patterns in android applications. - Google Project Hosting. (n.d.). Retrieved February 4, 2014, from https://code.google.com/p/android-binding/

[15] Arcbees (n.d.). ArcBees/GWTP · GitHub. Retrieved October 12, 2013, from https://github.com/ArcBees/GWTP

[16] Fowler, M. (n.d.). Passive View. Retrieved September 1, 2013, from http://martinfowler.com/eaaDev/PassiveScreen.html

[17] Repository Pattern. (n.d.). Retrieved from http://msdn.microsoft.com/en-us/library/ff649690.aspx

[18] Core J2EE Patterns - Data Access Object. (n.d.). Retrieved from http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

[19] Parmer, K. (2013, May 24). Volley: Easy, Fast Networking for Android - Example | KP Bird. Retrieved February 4, 2014, from http://www.kpbird.com/2013/05/volley-easy-fast-networking-for-android.html

[20] B. Chun and P. Maniatis, "Augmented smart phone applications through clone cloud execution," in Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS), 2009.

[21] March, V., Gu, Y., Leonardia, E., Goh, G., Kirchberga,, M., & Lee, B. S. (2011). µCloud: Towards a New Paradigm of Rich Mobile Applications.

[22] FunfArchitecture - funf-open-sensing-framework - An overview of the Funf package structure - Android-based framework for phone-based sensing and data-collection. - Google Project Hosting. (n.d.). Retrieved February 4, 2014, from https://code.google.com/p/funf-open-sensing-framework/wiki/FunfArchitecture

[23] Google (2013, June 30). Google I/O 2012 - Making Good Apps Great: More Advanced Topics for Expert Android Developers [Video file].

[24], F., Henney, K., & Schmidt, D. C. (2007). Pattern-oriented software architecture: V. 4. Chichester, England: John Wiley.

[25] Navigation Drawer | Android Developers. (n.d.). Retrieved from http://developer.android.com/design/patterns/navigation-drawer.html

[26] Spinners|Android Developers. (n.d.). Retrieved from http://developer.android.com/guide/topics/ui/controls/spinner.html

[27] Tabs | Android Developers. (n.d.). Retrieved from http://developer.android.com/design/building-blocks/tabs.html

[28] *WhatsApp :: Home*. (n.d.). Retrieved September 9, 2013, from http://www.whatsapp.com/