

Developing for Mobile Platforms: General discussion and best practices

1.0 EXECUTIVE SUMMARY

Starting 2007 (launch of the venerable iPhone), Smart phones have changed the way we look at mobile communication. The erstwhile ‘mostly voice phone’ device suddenly transformed into a computing device and opened a new world of possibilities for mobile users.

Amazing User Experience (UX), small form factors, coupled with powerful development platforms & mobile broadband availability have opened up a Pandora’s box in terms of how these tiny devices can be used to improve our daily lives with almost as much productivity as a standard laptop PC. With more than 100,000+ (as of Nov 2009) applications on the iPhone and thousands on other competing platforms like Android, Blackberry, J2ME and Windows Mobile, it is apparent that we are living through a revolution; a paradigm shift, where the world is converging to mobiles and related technologies for erstwhile desktop centric applications. As this paradigm shift is occurring a lot of desktop programmers are being thrown into a new world mobile programming. While there are many tools like Rhomobile (www.rhobile.com), PhoneGap (www.phonegap.com), LiquidGear (www.liquidgear.com) and iWebkit (www.webkit.com) which try and bridge the gap between desktop and mobile programming, the seasoned mobile programmer will know these tools only scratch the tip of the iceberg and are only applicable for web based non real-time applications. Programming for limited power/resource devices is a much bigger challenge than the desktop environment. **This whitepaper is a general guideline for developers who are taking a leap of faith into the mobile development world, to help**

1.0	EXECUTIVE SUMMARY	1
2.0	CHALLENGING FACTORS IN A MOBILE ENVIRONMENT	2
2.1	BATTERY LIFE	2
2.2	SCREEN SIZE	2
2.3	CONNECTIVITY	2
2.4	BANDWIDTH	2
2.5	MEMORY	2
2.6	OPERATOR & OEM POLICIES	2
3.0	ARCHITECTURAL BEST PRACTICES	3
3.1	OPTIMIZING BATTERY LIFE	3
3.2	OPTIMIZING FOR SCREEN AND USER EXPERIENCE (UX)	5
3.3	CONNECTIVITY AND BANDWIDTH	8
3.4	MEMORY	9
3.5	ABSTRACT USING LAYERS: CODE FOR DIVERSITY / FRAGMENTATION	10
3.6	PERFORMANCE OPTIMIZATION	10
3.7	KNOW HOW TO TEST	13
4.0	A USEFUL DESIGN PATTERN: MODEL VIEW CONTROLLER	14
5.0	HSC EXPERTISE IN MOBILE APPLICATIONS	Error

them get a head start in terms of selecting the right approach to develop a mobile application. The paper is independent of any specific platform and simply concentrates on bringing out differentiation in applicable programming methodology.

2.0 CHALLENGING FACTORS IN A MOBILE ENVIRONMENT

Smartphones have fewer resources when compared to their desktop counterparts. Some of the notable ones are the battery, the screen size, connectivity, bandwidth and memory. In this section of the paper we will introduce several key challenges. The next chapter will address what we, as developers, can do to mitigate these challenges.

2.1 Battery Life

Unlike desktops, mobile phones are usually disconnected from a continuous power source. To add to that, batteries have not been able to keep up to speed in terms of technology advancements when compared to those of the chipsets which reside in the phone. In other words while the phone processors and applications got more powerful, the battery that powers both remained quite the same. Any compute operation performed by an application causes this precious resource to deplete. A compute intensive application would quickly eat up the available battery resource and eventually contribute to a degraded user experience and a potential rejection by users. The battery is not only a problem for application developers; often it is one the reason that leads to phone manufacturers having to make compromises on features for a device. For example, one of the primary reasons that Apple does not allow multitasking for 3rd party applications is that doing so will deplete battery resources more. In Apple's case, the phone features many software enhancements in terms of UI and functionality, which are already taxing to the battery. Adding multitasking capabilities for 3rd party applications would allow developers to create resource hungry applications that are likely drain the battery to unacceptable levels, quickly.

2.2 Screen Size

Irrespective of processor advancements, the form factor of these devices is small and the developers are left with a much smaller real estate to deliver their content on. This brings up several visual challenges. Just because a phone has a great web browser does not mean that it will be as convenient to *navigate* the same content on a phone as it is on a desktop. Therefore, screen size has a direct relationship to User Experience.

2.3 Connectivity

Unlike an always-on Ethernet connection via a DSL or Cable router, a mobile application must cater for sporadic connectivity. Even though mobile operators may advertise 'Always On' IP connectivity, in reality however, that is rarely true. High-speed train journeys, a visit to the

basement or an elevator are all prime examples or when a phone may drop its connection. Connectivity therefore, is a scarce resource that applications need to cater to.

2.4 Bandwidth

Different wireless access technologies coupled with subscriber service profile determines the bandwidth that would be eventually made available to the user. As developers we must understand that the available bandwidth, is typically a function of the network load in a particular geographical area. Further, data consumption associated tariffs are hefty and especially so, while roaming. Although with the introduction of one-tariff plans this is lesser of an issue, however still, there are many subscribers who use data services on pay as you use basis. Therefore, from an application perspective, not only should the application brace for changes in available bandwidth (example, coverage drops from 3G to GPRS), but by design, the application should be architected in a way that data consumption is as less as practically possible without affecting the quality of the service that is being provided by the application. Another key aspect to keep in mind while developing applications is that typically the downlink bandwidth is higher than the uplink bandwidth.

2.5 Memory

The reason why we have covered memory last is the fact that newer memory architectures being introduced in mobile phone platforms are making it lesser of an issue for the newer generation of devices. However still there are still millions of phones in use today which feature much less memory and as an application for masses we want to develop an application which can run seamlessly on different classes of devices available in the market today.

2.6 Operator & OEM Policies

OEMs and Operators usually build differentiation in their offerings by either customizing the phone platform or publishing guidelines for applications. Unless the application is being developed for a specific service provider and a mobile device, developers should be aware of different vendor policies and application development and deployment guidelines which must be adhered to, for the application to be deployed in a particular network or on a particular device. Please refer to section 3.5 for more details on this.

3.0 ARCHITECTURAL BEST PRACTICES

With these constraints in our mind, there are some key architectural methodologies, which can be adopted to write optimum code for mobile platforms.

3.1 Optimizing battery life

3.1.1 Be aware of the power management features of the OS

In order to prolong the battery life most of the mobile phone operating systems enter a dormant state incase there is no activity either by the user or from the underlying hardware platform. When the OS is about to enter a dormant state, applications are usually informed and asked for consent to action. The modern day mobile phone operating system features very complex power profiles and include multiple options for power management of audio, network and video components.

A good architecture should exploit these APIs, notifications and services and gel them well into the user experience in a way that it contributes to an elongated battery life. For example, for an audio player application which is playing background music stored locally on the phone, the video and the network components can be put to a dormant state, however, when the same application is playing back video, it prevents the OS from turning off the backlit display for the video component device.

Some other power management tips:

- Turning off or reducing display brightness for screens that are compute intensive and don't require continuous screen update
- Many power manager APIs provide the developer an API to check current battery charge level. Checking one in 10-15 minutes can help make intelligent processing decisions in the application.
- If the platform supports, check the power state of the phone before you execute a UI heavy operation (for example, if a phone is in Screen Off state, there is little point in doing a foreground animation unless your application wants to turn the screen on). In general, understand the power states of the phone well before you write an application and make use of the states in your code.
- Writing an application that utilizes memory management effectively is also important. Please refer to 2.5

3.1.2 Tradeoff between burning local CPU vs. using the network for compute heavy jobs

Carefully partition the business logic of the application to leverage network-based infrastructure for compute bound jobs. When adopting this strategy, make sure that you do not end up over architecting the solution, as network access 1) might not be ubiquitously available and 2) might consume relatively higher amount of battery if used excessively. However, if the application being designed is a client/server application and the user is expected to have network connectivity a correct architecture can stretch the battery a long way.

3.1.3 Select the right algorithm; go with as much approximation as allowable

Unless an application is processing something critical, as would be a case for a financial application, users are often ok with approximations in display and numbers especially in games and UI components. Higher precision often comes at a higher computation and hence battery costs. This concept is very useful when developing complex user interfaces and games. A good example is a video playback application, which can fallback to a lower resolution-rendering engine suitable for the form factor of the device's screen.

3.1.4 Avoid floating point computations

Almost all of the mobile phone platforms do not feature a floating point co-processor and hence all floating point computations are executed using fixed point libraries which is quite a drag on the CPU and hence the battery. It is advised that either architects develop equivalent fixed-point algorithms or use the strategy laid out in 3.1.2 if such computations cannot be avoided.

3.1.5 Avoid loops

Resort to an event driven architecture as application loops prevent the OS from putting the application and hence the device in the low power state. This approach saves a lot of battery and also gels well with the modern mobile phone OS architectures as well, since most of the OSs are designed to enter a dormant state till an event to be serviced occurs on the phone.

3.1.6 Multi-tasking

3.1.6.1 Cooperative Multi-tasking

Cooperative multi-tasking is an application design pattern where applications willingly give up CPU whenever they don't need it, to give other tasks in the system a fair share of CPU. Cooperative multi-tasking mixed with event driven programming methodology not only leads to optimal battery performance but

also helps the OS to keep the platform responsive and interactive for other operations that the user might want to perform on the phone. While almost all OSs follow an application development methodology centered on cooperative multi-tasking as a defacto standard, implementation and realization depends on the platform.

3.1.6.2 Multi-threading

Different OSs take different approach towards Multi-threading in general. While in J2ME and Blackberry, it the way to implement multi-tasking, it is not a preferred approach on OSs like Symbian as it introduces un-necessary overhead in the runtime system.

3.1.6.3 General Rule

A good architecture should ideally abstract multi-tasking libraries of the OS from the core application code and adapt the abstraction to the platform depending upon the platform.

3.1.7 Use Compiler Optimizations

Applications can often be improved for performance and battery by optimizing the application using available compiler options. Study these options well.

3.2 Optimizing for Screen and User eXperience (UX)

While developing applications developers should ensure that nothing that they do affects the primary user experience in a negative way. Most of the operating systems today ensure such a behavior. As developers, we must also ensure that we follow the guidelines such that the application user experience integrates well with the native user experience of the phone platform. This will not only ensure that the users adapt well to the application but will also help users get comfortable with the application with little or no coaching. On the technical side, most of the UI components, which are natively supported by the platform, have already been optimized for the platform in addition to being tested for correctness/bugs. Reusing these components will not only reduce the amount of code but also add to stability of your own application. Some of the common guidelines for designing User Interfaces are as follows:

1. Avoid too small or fixed sized fonts and if possible allow the users to change the font sizes from the options menu.
2. Use internationalization libraries while displaying content. This ensures that future localization of the application is simpler (remember that different languages have different character representation schemes, let the resource libraries handle this for you).

3. Use relative positioning as much as possible and avoid absolute positioning. Use phone APIs to determine current font, window and screen size and ensure your application uses them instead of assuming values. This ensures the application will play well with different user settings.
4. Since many of the mobile phone operating systems feature a touch mode as well as physical keyboards, applications should be designed to support both interfaces (i.e. if the system exports touch events, it is a good idea to trap and process them too). This might require that the buttons and the UI components, which are placed on the views, are slightly larger than usual.
5. The color combination should be soothing and shall ensure readability. One good practice is to follow the color combination configured in the current theme installed on the device.
6. Make the user interface as consistent and predictable as possible. Users are more likely to adapt well to an application, which has consistent workflows than an application with dynamic workflows. What this means is don't re-invent well known functions (we've known applications that remap the back [not cursor left] key to move an on-screen object to the left!).

If the application is targeted for devices, which support multiple screen orientation, it is important that the application appears and behaves properly both in landscape and portrait mode and supports screen rotation whenever required. In case the developers choose to omit this guideline from their UI design, make sure the user of the application is appropriately informed of this decision. Also, if this is the case, if your platform supports it, restrict the application from switching to horizontal mode if the phone is turned (again, we recommend you test for horizontal mode instead). Although, not a widely adopted approach some architects follow the square based client area approach. This approach is most suitable for UIs, which have data entry screens. In this approach the entire UI is designed assuming the client area is a square with its dimension equal to the breadth or the shorter side of the mobile screen. This way, even when the screen is rotated, it is ensured that the user continues to see whatever was being displayed to him earlier in the other mode.

7. Be aware of UI guidelines for certification: If your platform has a certification process, read it before you code! Sometimes, you may execute a UI operation, which is a direct contradiction of a certification guideline. In this case, your application will fail certification. If you implement an exception make sure you are confident of your chances of an exception approval.
8. Remember those nag screens: On many platforms (J2ME being an example here), if your application is not signed, the user will be presented with many 'nag' screens for many operations such as network access, SMS access, data storage etc. There is nothing that

destroys a User experience as much as this one fact. Make sure you know the potential nag screens, how to avoid them and how to alert the user to change phone settings appropriately as part of the overall UI workflow design.

9. Use alerts and system notifications and use them wisely. The information being displayed on an alert or a notification should be crisp and to the point, otherwise it will diminish the visibility of the relevant information.

On platforms which support system notifications, such notifications are usually a good medium to inform the user that the application has an updated content or information for user to review.

10. Some platforms like Windows mobile, Symbian and Android allow applications to place widgets in the phone's home screen. This is a good feature for applications which continuously run in background, as it provides users with:

- 1) Critical info updates without actually invoking the application
- 2) A shortcut access to the application for trivial tasks.

A good example is the Google Search Application for Symbian, where the users can directly enter the keywords, which need to be searched, and the application then shows the search results as the user selects the search button.

As an extension to the above, many platforms allow for dynamic widgets (i.e. you can add/change animation to the widget icon). This is also a great mechanism to update the user with brief data without popping up a new screen/dialog. The iPhone process of installing/upgrading a new application where the widget icon shows an embedded in progress bar and a calendar widget that shows the date and day as part of the icon are great examples.

11. Display only relevant information and keep it ordered and grouped. Some examples for the same could be:
 - a. Place commands for critical features in a prominent area such as a menu (on windows mobile), or a navigation bar or tab buttons (on iPhone) and Soft keys on Symbian, J2ME etc.
 - b. Likewise, place less-frequently used commands in a submenu or settings dialog.
 - c. Avoid placing commands redundantly
 - d. Errors should be notified to the user in a meaningful and concise way.

12. Instructions for using an application should be easily locatable and accessible.

13. The application should be responsive and shall give feedback to the user in reasonable time. Please refer to section 3.6 for more details.
14. If your application happens to be sending any user data to the network, it is a good practice to have a Terms Of Service that states this explicitly and have the user accept it before using the application.
15. Whenever the application needs to perform some time consuming work in response to user input, it is always recommended to show that progress is being made using some sort of progress indicator. It is a best practice in the mobile world to show such a progress-bar, slider, wheel etc. for any task that can take more than 1 second without a screen refresh.
16. If the application being developed has a time consuming initial setup phase, a splash screen should be displayed as soon as possible and with some sort of indicators that show progress is being made, or else the users will perceive that the application is frozen.

3.3 Connectivity and Bandwidth

Applications accessing the enterprise, social networking sites, Massively Multiplayer Online (MMO) gaming engines are already taxing the wireless networks. Although data connectivity may be good its availability and the size of the broadband pipe which is available to the subscriber, is largely variable. Hence, if an application needs online connectivity, it would be a good idea to plan for an online and offline mode during design.

In the offline mode the application typically caches data in the local database and makes it available to the user even there is little or no connectivity, and then the data is synchronized with the backend servers as better connectivity or bandwidth is available.

Are online applications better than offline applications? No one strategy is better than the other and shall be chosen by the architects based on its applicability to the problem they are attempting to solve. While the online mode is more secure and allows its users to access most up-to-date data, their main limitation is that such applications are typically slow and unusable when there is little or no coverage. The hybrid approach on the other hand takes care of the offline issue but designing such applications is typically complex and non-trivial and for online gaming engines this approach might not be applicable at all. This approach is slightly less secure as well, since, the data which is cached locally, can fall into wrong hands as the device is stolen or misplaced.

Architect applications so that it can work with the minimal level of data available (i.e. don't wait till you get the best datasets to work on, and till then process/render the best approximation that is viable for your app for now, since quick display is very important. A good example would

be the various mapping applications, which resort to approximations till they download either a better map or derive a more precise location.

3.4 Memory

Memory footprint of applications is another important aspect, which the developers need to consider while designing a mobile application. Most of the engineers often end-up over-engineering the solution by creating frameworks on top of which the user application is built. While this is a wise approach, often creating generalized frameworks introduces a lot of code, which could have been otherwise avoided, and hence contributes to memory footprint significantly. Care should be taken so as to strike the right balance between what should be developed generic and specific for the platform and the application. Here are some tips:

1. Memory allocation is never cheap. Not allocating memory is always cheaper than allocating memory, especially at runtime.
2. Allocate memory buffers at the start of the application and then reuse those buffers throughout the application lifespan. One should have a fairly accurate idea for memory allocations once the data structures and algorithms have been identified for the application being developed.
3. Do not allocate memory when executing screen update routines. This could introduce un-welcomed glitches/interruptions in the user experience.
4. Optimize the use of memory buffers by using the same buffer for manipulation wherever possible instead of allocating a new buffer every time. Generalizing the concept, avoid creating short-term temporary objects, if possible. An example for this approach could be to process the incoming data packet immediately instead of storing in the buffer queue for further processing which is an approach more suitable for server side multi-threaded high throughput environments.
5. For platforms which support exception handling, it is not a wise choice to throw an exception while the destructor is getting executed. First, it can cause the application to terminate, and second it can cause memory leaks since most of the library users usually do not trap the object deletion and hence it may lead to memory leaks.
6. Do not declare destructors as virtual, if the class would not be derived in the future. This will save some critical space, as the compiler will not create vtable entries.
7. Check the return value of all memory allocation calls so you know when you have hit a limit.
8. If the platform allows, use data sharing between related processes as much as possible (such as memory mapped files)

3.5 Abstract using layers: Code for Diversity / Fragmentation

Unlike in the desktop world, where there are a handful of well known Operating Systems the mobile world is fraught with fragmentation. Depending on country, operator, device and other factors the same mobile platform is often changed enough to cause nightmares for the developer. There are several causes for fragmentation, including:

1. Hardware: Examples include differences in screen parameters, memory size, processing power, input mode, presence of additional hardware, and connectivity options.
2. Software & Platform diversity: For example differences in platform/OS (Symbian, Series 60/40, RIM OS, iPhone OS, Palm OS, Mobile Linux, Android, BREW, Windows Mobile etc.).
3. User-preference diversity: Aspects such as in language, style, etc., or accessibility requirements
4. Environmental diversity: Such as diversity in the deployment infrastructure (e.g., branding by carrier, compatibility requirements of the carrier backend and application development APIs, gateway characteristics, opened ports, restrictions on access to outside the network etc.)

Application users, developers, content providers and distributors, network operators and device manufacturers are all affected by fragmentation. While designing applications developers must ensure that the solution architecture is generic enough so that it can be easily adapted to multiple targets. A good approach would be to identify layers, which should be specific to the platform. Once such demarcation has been identified, a large portion of code can be reused across platforms with development effort restricted only to platform specific modules.

3.6 Performance Optimization

1. As mentioned before, try and use native APIs provided by the SDK as much as possible since such libraries are usually optimized for both speed and space.
2. If you don't need to access an object's field specific to an instance, make them class fields. It can be called faster, because it doesn't require a virtual method table indirection.
3. For Java based platforms virtual references are better than interface references since the interface references could be about 2x time slower as compared to its counter parts.

4. On certain interpreted platforms especially the ones based on J2ME and .NET CLR, enums have large storage and performance overheads associated with them, and hence a wiser approach would be use constants instead of enums.
5. Embedded processors typically do not have hardware floating point support; so all operations on floats and doubles are performed in software. Some basic floating-point operations can take on the order of milliseconds to complete. Even for integers, some chips lack the support for divide operation and hence the integer divisions are performed in software, which degrades the overall performance of the software. A good approach would be to develop a revised algorithm, which replaces floats with fixed points representations and division with subtraction and shift approximations.
6. Most of the mobile platforms support some form of intent based IPC mechanism so as to allow maximum reuse of functionality provided by the OS or applications installed on the device. As an application architect, one should definitely look at reusing some of this functionality. This will not only help keep the memory footprint of the application low, but also allow developers to offload some of the capabilities to applications which have been designed ground up to offer such services. The exact terminology and implementation of this approach could be platform dependent.
7. Images are one of the most important elements in any user interface. A mobile application can use many types of images. Almost all mobile platforms publish a guideline for common image types. Developers should refer to guidelines before they create images. This not only provides consistency to the platform but also allows the platform to optimize image caches for optimum performance. (Also historically, many mobile platforms do a bad job of scaling images, so it is best to use dimensions that suit a particular view)
8. Keep error handling as simple as possible. A function that returns an error code should not throw error exceptions. This not only complicates application source code but also introduces unnecessary error checking making the memory footprint of the application bloat.
9. Avoid excessive try-catch or exception harnesses; they use excessive space when compiled since, most of the compilers use simulated exception harnesses as the support is natively missing from the platform.
10. Avoid global static data in your applications. Doing this has advantages:
 - a. It allows the code to be ported easily across platforms
 - b. The code can easily be ported for multi-threaded and re-entrant architectures
 - c. Better at adapting to an MVC architecture described earlier.

11. Do not ignore compiler warnings. Often there are clues in them as to what can be changed to improve the overall performance of the application. While this is also applicable for desktop environments, it is much more important for mobile platforms.
12. Most of the object-oriented platforms provide some sort of base class for objects. It is always advisable to use that class as the base of all classes we define. More often than not these classes implement some housekeeping functionality, which leads to better memory management, and hence less bugs in the application.
13. Make sure you disable all debugging and testing related code from the release build. Another important observation would be to make sure that all libraries to which the code is linking to, is also built in the release mode.
14. Functions should be defined local whenever possible. This reduces the amount of housekeeping code that is produced corresponding to each object by the compiler thereby reducing the memory footprint and results in neater applications.
15. Avoid multiple functions, which perform similar tasks. A good strategy would be isolate such functionality and abstract it within a single function with specific parameters. This approach helps reducing the memory footprint of the application.
16. Choose a good set of default values for you application and the algorithms it executes. This often helps optimize the application at runtime.
17. Mobile OSs can often suspend or terminate applications running in background so as to recover some runtime memory and launch them back when resources are available. As the application is terminated/suspended the user can loose some critical data or state information. To overcome this issue, almost all OSs provide application lifecycle events to the applications. Applications which are required to maintain a consistent state across invocations should register for these events with the underlying OS, and perform state management operations as the application transitions from one state to another.
18. If extracting maximum performance is key for you, don't use get/set APIs and access the class fields directly, especially for interpreted platforms like Android, J2ME and Blackberry (yes, we know this exposes us to potential data structure changes, but when it comes to performance optimization, you select the tradeoff)
19. Implement the UI and the Application Engine (or the business code) in two different tasks. Off loading computations to a different task often helps keeping the application responsive even when the application is performing a very compute intensive job and improves the overall user experience for users of that application. This strategy is also important because, in most of the mobile phone operating systems the OS monitors the main task, which normally hosts the UI, by sending it constant signals, which are automatically responded to by the UI. Hence, if the UI and the Engine components are

collocated and the application is involved in some intensive compute job the OS on not receiving the response for heartbeat may terminate the application assuming it is no longer responsive

3.7 Know how to Test

Developers often end up testing applications on device emulators and not target hardware or real devices. This could lead to problems when the solution is actually deployed across different platforms since, there could be notable differences in the emulator and the platform related to the application execution environment, the memory model and the performance parameters.

Several questions plague this community when it comes to testing:

1. How do I do automated testing on a mobile phone?
2. How does one test an application across hundreds of devices without the cost of buying them and/or paying for remote testing (via remote labs offered by some companies) for all those phones?

The answer to the above deserves a detailed whitepaper each. Suffice to say, at HSC we have tackled exactly these problems and have designed the right workflows and methodologies to meet such challenges for you.

4.0 A USEFUL DESIGN PATTERN: MODEL VIEW CONTROLLER

Model View Controller (MVC) is one of the most used and popular application design patterns today and yet so very few application developers use it in the mobile phone development paradigm for their own applications. The design methodology calls for an architecture, which decouples the rendering (view) from the logic (controller) and data (model) and isolating each of these layers from each other through a well-defined interface. In fact, the MVC architecture is used extensively by modern phone platforms like Android, iPhone etc. In this approach the view depends only on the model to render everything on the screen. The controller receives input, can gain additional information by calling the view, and at the end manipulates the model. The advantages of this architecture are:

- a. A modularized architecture, which is largely independent of how the view renders the information and how data is stored internally.
- b. Reusability of model and view across usage scenarios.
- c. Because the model is self-contained and separate from the controller and the view, it's much less painful to change your data layer or business rules.

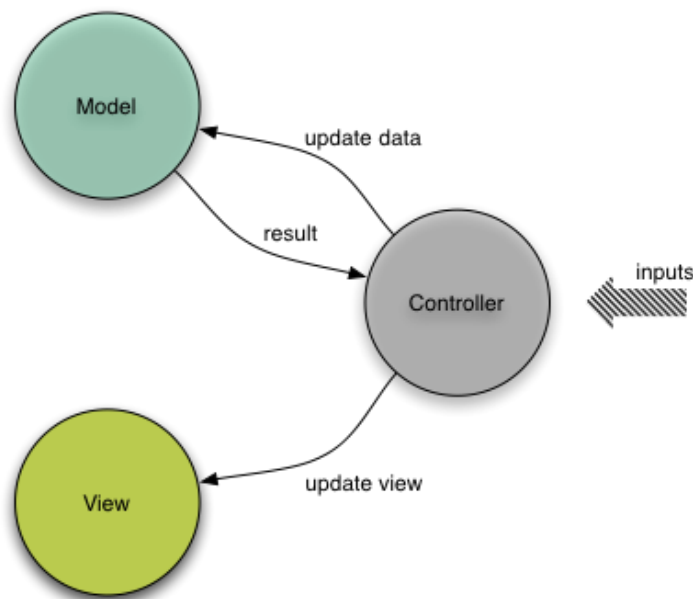


Figure 4-1: Model View Controller Solution Design Pattern

In this paper we propose a slightly modified MVC pattern be used in mobile applications. In this proposal, the architecture leverages the fact that most of the mobile applications are single threaded and co-operatively multi-tasked and hence there will be a consistent view of the model for both the controller as well as the view. The additional exchange between the model and the view where the view consumes the model in a read-only mode allows for an optimization where un-necessary parameter marshalling

can be avoided and the view can retrieve only that data which it needs from the model. The same architecture can also be extended for multi-threaded scenarios. However in that case, care needs to be taken that the consistency of data is maintained across the view and the controller, because the view might retrieve data from the model through a different thread.

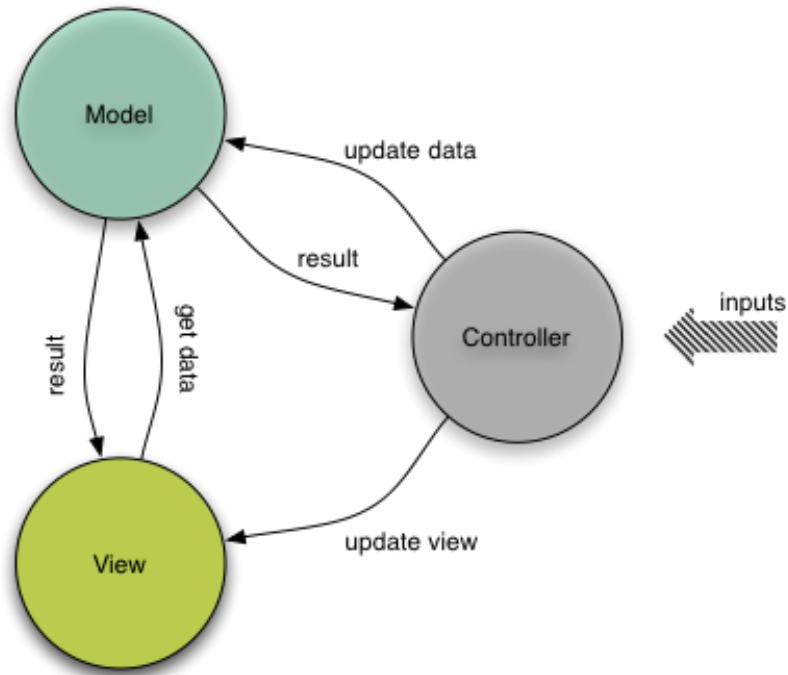


Figure 4-2: Customized Application View Controller Design Pattern

*

5.0 HSC EXPERTISE IN MOBILE APPLICATIONS

HSC is a world leader in mobile applications and offers turnkey consulting, development, testing services across mobile platforms. Our mobile solution ecosystem spans the entire lifecycle of application development.



Figure 5-1 HSC Mobile Porting Library

As part of the mobile solution ecosystem, HSC also offers a Mobile Porting Library (MPL) reusable component to its customers. The MPL is an advanced platform abstraction layer that hides platform specific details from the application developer for all important phone platforms such as Symbian, iPhone, Android, Windows Mobile etc. Written from ground up for mobile environments and tested in several live deployments by our customers, the MPL greatly enhances product development quality and reduces time to market for you.



PROPRIETARY NOTICE

All rights reserved. This publication and its contents are proprietary to Hughes Systique Corporation. No part of this publication may be reproduced in any form or by any means without the written permission of Hughes Systique Corporation, 15245 Shady Grove Road, Suite 330, Rockville, MD 20850.

Copyright © 2009 Hughes Systique Corporation

CONTACT INFORMATION:

Phone: +1.301.527.1629

fax: +1.301.527.1690

email: whitepaper@hsc.com

Web: www.hsc.com

APPENDIX A ABOUT HUGHES SYSTIQUE CORPORATION

HUGHES Systique Corporation (HSC), part of the HUGHES group of companies, is a leading Consulting and Software company focused on Communications and Automotive Telematics. HSC is headquartered in Rockville, Maryland USA with its development centre in Gurgaon, India.

SERVICES OFFERED:

Technology Consulting & Architecture: Leverage extensive knowledge and experience of our domain experts to define product requirements, validate technology plans, and provide network level consulting services and deployment of several successful products from conceptualization to market delivery.

Development & Maintenance Services: We can help you design, develop and maintain software for diverse areas in the communication industry. We have a well-defined software development process, comprising of complete SDLC from requirement analysis to the deployment and postproduction support.

Testing : We have extensive experience in testing methodologies and processes and offer Performance testing (with bench marking), Protocol testing, Conformance testing, Stress testing, White-box and black-box testing, Regression testing and Interoperability testing to our clients

System Integration : As system integrators of choice HSC works with global names to architect, integrate, deploy and manage their suite of OSS, BSS, VAS and IN wireless (VoIP & IMS), wire line and hybrid networks.: NMS, Service Management & Provisioning .

DOMAIN EXPERTISE:

Terminals

- Terminal Platforms : iPhone, Android, Symbian, Windows CE/Mobile, BREW, PalmOS
- Middleware Experience & Applications : J2ME , IMS Client & OMA PoC

Access

- Wired Access : PON & DSL, IP-DSLAM,
- Wireless Access : WLAN/WiMAX / LTE, UMTS, 2.5G, 2G , Satellite Communication

Core Network

- IMS/3GPP , IPTV , SBC, Interworking , Switching solutions, VoIP

Applications

- Technologies : C, Java/J2ME, C++, Flash/lite, SIP, Presence, Location, AJAX/Mash
- Middleware: GlassFish, BEA, JBOSS, WebSphere, Tomcat, Apache etc.

Management & Back Office:

- Billing & OSS , Knowledge of COTS products , Mediation, CRM
- Network Management : NM Protocols, Java technologies,, Knowledge of COTS NM products, FCAPS, Security & Authentication

Platforms

- Embedded: Design, Development and Porting - RTOS, Device Drivers, Communications / Switching devices, Infrastructure components. Usage and Understanding of Debugging tools.
- FPGA & DSP: Design, System Prototyping. Re-engineering, System Verification, Testing

Automotive TelematHSC

- In Car unit (ECU) software design with CAN B & CAN C
- TelematHSC Network Design (CDMA, GSM, GPRS/UMTS)

BENEFITS:

- **Reduced Time to market** : Complement your existing skills, Experience in development-to-deployment in complex communication systems, with key resources available at all times
- **Stretch your R&D dollars** :Best Shore” strategy to outsourcing, World class processes, Insulate from resource fluctuations

CONTACT INFORMATION:

Phone: +1.301.527.1629

fax: +1.301.527.1690

email: whitepaper@hsc.com

Web: www.hsc.com