Convergence Mobile Application Architecture on Requirement View

*Haeng-Kon Kim and Yvette E. Gelogo

School of Information Technology, Catholic University of Daegu, Korea
**hangkon@cu.ac.kr, yvette@cu.ac.kr*

Abstract

Mobile applications development challenges the modeling activities that precede the technical design of a software system. The context of a mobile system includes a broad spectrum of technical, physical, social and organizational aspects. Some of these aspects need to be built into the mobile applications. Selecting the aspects that are needed is becoming increasingly more complex with mobile systems than we have previously seen with more traditional information systems.

In this paper, we discuss mobile application architectures. We start by describing some of the general concepts and terms behind client-server architectures and follow this by describing clients and servers and the connectivity between them. We then present several interesting architectural patterns and describe why they are useful as general mobile application architecture solutions. Finally, we discuss some of the tenets behind good architectural design and the considerations you need to be aware of when designing mobile applications. We also evaluate the mobile application architecture to apply an example case as best practices.

Keywords: *Mobile Applications Architecture, Architectures Requirements, Model View*

1. Introduction

The mobile applications support a much wider range of activities than desktop applications and leverage information about the user's environment to provide novel capabilities. From a technology perspective, mobility shifts the global computing infrastructure from static, homogenous, powerful desktop computers to highly dynamic, heterogeneous, resource-constrained handheld and wearable computers. This new computing context demands entirely new software architectural paradigms that address the challenges of mobile software development, are specialized for the nature of mobile devices and wireless networks, and take advantage of the opportunities afforded by mobile systems. This new computing context demands entirely new software architectural paradigms that address the challenges of mobile software development, are specialized for the nature of mobile devices and wireless networks, and take advantage of the opportunities afforded by mobile systems. Recent research has rapidly advanced the state-of-the-art in architectures for mobile software and systems. A mobile application will normally be structured as a multi-layered application consisting of user experience, business, and data layers. When developing a mobile application, you may choose to develop a thin mobile-based client or a rich client. If you are building a rich client, the business and data services layers are likely to be located on the device itself. If you are building a thin client, the business and data layers will be located on the server. Figure 1 illustrates common rich client mobile application architecture with components grouped by areas of concern [1].

* *Corresponding Author*

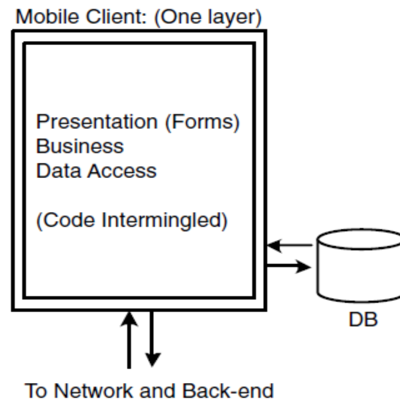


Figure 1. Common Rich Client Mobile Application Architecture

When developing mobile applications, there are a number of key challenges where architecture and design are fundamentally different from that of a typical enterprise application. Careful consideration should be given to these mobile architecture issues early in the development process in order to mitigate the downstream impact of poor architectural decisions. While some of these best practices also make sense for the development of non-mobile applications, many will become more readily apparent when developing on a mobile platform. The five most important areas for consideration, which are detailed throughout this document, include: performance, usability, data access, security, and connectivity. While more readily apparently in the previous years of mobile development, the computing power available on mobile devices still lags behind desktop and server counterparts and will continue to do so for the foreseeable future due to smaller device footprints and resource constraints. Even the most recent devices still boast only about one third to one half of the computing resources (CPU, RAM) of a low end desktop computer. Further, the quality of data connections available on a mobile device is often highly variable based on signal strength and is far inferior to broadband Internet access in most cases. Often during rapid application development, performance considerations are ignored until the end of the project and optimized only when necessary. In mobile development, more consideration to performance constraints of the mobile device may need to be given up front in the design process. Each platform has different code-level best practices for performance optimization depending upon the programming language and frameworks available on the platform. Some best practices, such as judicious usage of memory and limits on the number of unnecessary objects created, however, can be applied across all platforms [2]. One commonality between the most modern mobile platforms (iPhone, Android, Windows Phone 7) is that none of them offer any capability to connect directly to a database–for good reason. The current mobile architecture paradigm simply doesn’t support this scenario for modern database platforms in their current state.

In this paper, we discuss mobile application architectures. We start by describing some of the general concepts and terms behind client-server architectures and follow this by describing clients and servers and the connectivity between them. We then present several interesting architectural patterns and describe why they are useful as general mobile application architecture solutions. We present a set of requirements for future mobile middleware which have been derived by considering the shortcomings of existing approaches and the needs of application developers. Key among these requirements is the need to support coordinated action between application and system components and the resolution of conflicts

caused by the need to adapt to multiple contextual triggers. The paper concludes with the presentation of an architectural framework within which middleware researchers can deploy solutions to the problems identified. Finally, we discuss some of the tenets behind good architectural design and the considerations you need to be aware of when designing mobile applications. We also evaluate the our mobile application architecture to apply an example case as best practices.

2. Related Work

2.1. Understanding Mobile Applications Architecture

Mobile applications architecture representing a common high-level abstraction of a system, is a description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns[3,4]. There are three reasons [5, 6] why software architecture is important to large, complex, software-intensive systems.

– *Architecture is the Vehicle for Stakeholder Communication*. Each stakeholder in a software system - customer, user, project manager, coder, tester, and so on - is concerned with different characteristics of the system that are affected by its architecture. Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems.

– *Architecture Manifests the Earliest Set of Design Decisions*. Software architecture represents a system's earliest set of design decisions. These early decisions are the most difficult to get correct and the hardest to change, and they have the most far-reaching effects.

– *Architecture is a Transferable, Reusable Model*. The earlier reuse is applied in the life cycle of software, the greater the benefit that can be achieved. While code reuse provides a benefit, reuse at the architectural level provides tremendous leverage for systems with similar requirements. Not only can the code be reused but so can the requirements that led to the architecture in the first place, as well as the experience in building the reused architecture. When architectural decisions can be reused across multiple systems, all of the early-decision consequences we just described are also transferred.

The Figure 2 shows the typical mobile multitier application architecture including mobile client and server.

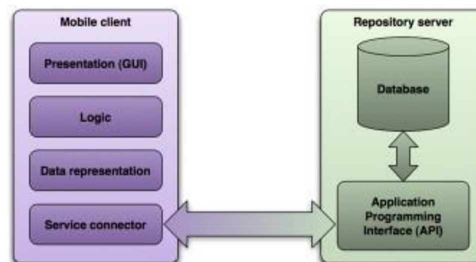


Figure 2. Typical Mobile Multitier Application Architecture

2.2. Software Architecture Model: The 4+1 View Model

Having good architectural documentation is crucial to the success of any architectural evaluation method [6]. The large variance of a quality assessment based on architectural analysis is associated with the granularity of the system description necessary to perform an evaluation. The goals of architecture documentation are to record the architects' decisions and to communicate the architecture. To meet these goals, the documentation must be complete and unambiguous. For these reasons, different views can be used to enhance the understandability of the architecture and to focus separately on particular concerns, as Kruchten pointed out [7]. In this respect, the 4+1 view model produces a mechanism to allow us to separate concerns while building or analyzing an architecture. Architects capture their design decisions in four views and use the fifth view to illustrate and validate them. The Figure 3 shows the 4+1 view model. Each view addresses a specific set of concerns as follows [6, 7]:

- *Logical View*. The logical view includes a set of abstractions necessary to depict the functional requirements of a system at an abstract level. This view is independent of implementation decisions and instead emphasizes interaction between entities in the problem space.
- *Process View*. The process view describes the design's concurrency and synchronization aspects. This view takes into account some nonfunctional requirements such as performance and system availability. It addresses concurrency and distribution, system integrity, and fault-tolerance.
- *Development View*. The development view describes the software's static organization in its development environment. This view supports the allocation of requirements and reasoning about software reuse, portability, and security.
- *Physical View*. The physical view describes the mapping of the software onto the hardware and reflects its distributed aspect. This view takes into account the system's nonfunctional requirements such as system availability, reliability, performance, and scalability.
- *Use case View*. The use case view describes an abstraction of important requirements as use case. This view is redundant with the other ones, but it plays two critical roles. One, it acts as a driver to help designers discover architectural elements during the architectural design. Two, it validates the architectural design.

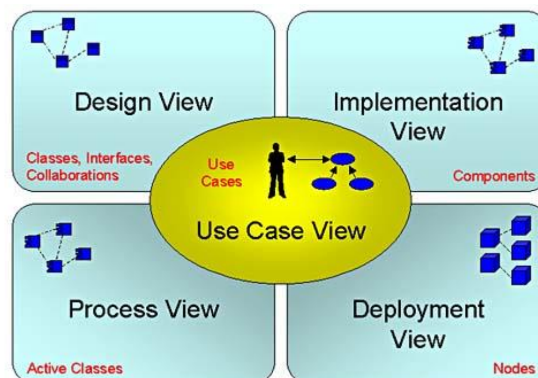


Figure 3. 4+1 View Model

2.3 Existing Approaches to Software Architecture Evaluation

Several research communities have developed techniques to perform architectural evaluation. In this section, the characteristics and limitations of some representative approaches are briefly illustrated.

Techniques evaluating a specific quality attribute. Several research groups have developed techniques used for the specification and assessment of their particular quality requirements. Of those techniques, some techniques [8, 9] have adopted statistical models, *e.g.*, Markov Chain Model and Queuing models. On the other hand, the ADL(Architecture Description Language) research groups have developed various kinds of languages to represent architectural information relevant to their specific quality attributes, and they have analyzed architecture using them [16]. But these approaches tend to require considerable effort from the software engineer for creating specifications and making predictions. In addition, the applicability of a particular model or an ADL is restricted to narrow limits by the power of its representation.

Techniques using simulations or prototypes. These techniques require that the main components of the architecture are implemented, and other components are simulated resulting in an executable system [4]. But, these techniques require information about the system under development that is not available during the architectural design. Additionally, creating a detailed simulation or prototype for the purpose of evaluation is typically expensive [7].

Scenario-based evaluation techniques. A scenario-based technique is used to attempt to reduce the problematic nature of evaluating a high-level design with respect to software quality attributes [4]. To assess a particular quality attribute, a set of scenarios has to be developed to make concrete the actual meaning of the quality requirements. The technique focuses on architectural features that will reveal design biases and flaws early in the life cycle of the system. In these techniques, however, there are a number of uncertainties such as the granularity of representation and how representative the scenarios are in respects to their evaluation steps.

3. Convergence Mobile Application Architecture on Requirement View

3.1. Requirements for Mobile Applications Architecture

When getting started with mobile applications design, we suggest the key principles that will help to create architecture that meets “best practices,” minimizes costs and maintenance requirements, and promotes usability and extendibility. The key principles are:

• **Separation of concerns**. Break your application into distinct features that overlap in functionality as little as possible.

• **Single Responsibility Principle**. Each component or a module should be responsible for only a specific feature or functionality.

• **Principle of least knowledge**. A component or an object should not know about internal details of other components or objects. Also known as the Law of Demeter (LoD).

• **Don’t Repeat Yourself (DRY)**. There should be only one component providing a specific functionality; the functionality should not be duplicated in any other component.

• **Avoid doing a big design upfront**. If your application requirements are unclear, or if

there is a possibility of the design evolving over time, avoid making a large design effort prematurely.

• **Prefer composition over inheritance**. Wherever possible, use composition over inheritance when reusing functionality because inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes.

When designing an application or system, the goal of a software architect is to minimize the complexity by separating the design into different areas of concern. For example, the user interface (UI), business processing, and data access all represent different areas of concern. Within each area, the components you design should focus on that specific area and should not mix code from other areas of concern. For example, UI processing components should not include code that directly accesses a data source, but instead should use either business components or data access components to retrieve data. Follow these guidelines when designing an application:

• **Avoid doing all design upfront**. If you are not clear with requirements or if there is the possibility of design evolution, it might be a good idea not to do complete design upfront. Instead, evolve the design as you progress through the project.

• **Separate the areas of concern**. Break your application into distinct features that overlap in functionality as little as possible. The main benefit of this approach is that a feature or functionality can be optimized independently of other features or functionality. Also, if one feature fails, it will not cause other features to fail as well, and they can run independently of one another. This approach also helps to make the application easier to understand and design, and facilitates management of complex interdependent systems.

• **Each component or module should have a single responsibility**. Each component or module should be responsible for only one specific feature or functionality. This makes your components cohesive and makes it easier to optimize the components if a specific feature or functionality changes.

• **A component or an object should not rely on internal details of other components or objects**. Each component or object should call a method of another object or component, and that method should have information about how to process the request and, if needed, route it to appropriate subcomponents or other components. This helps in developing an application that is more maintainable and adaptable.

• **Do not duplicate functionality within an application**. There should be only one component providing a specific functionality—this functionality should not be duplicated in any other component. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.

• **Identify the kinds of components you will need in your application**. The best way to do this is to identify patterns that match your scenario and then examine the types of components that are used by the pattern or patterns that match your scenario. For example, a smaller application may not need business workflow or UI processing components.

• **Group different types of components into logical layers**. Start by identifying different areas of concern, and then group components associated with each area of concern into logical layers.

• **Keep design patterns consistent within each layer**. Within a logical layer, the design of components should be consistent for a particular operation. For example, if you choose to

use the Table Data Gateway pattern to create an object that acts as a gateway to tables or views in a database, you should not include another pattern such as Repository, which uses a different paradigm for accessing data and initializing business entities.

• **Do not mix different types of components in the same logical layer**. For example, the UI layer should not contain business-processing components, but instead should contain components used to handle user input and process user requests.

• **Determine the type of layering you want to enforce**. In a strict layering system, components in layer A cannot call components in layer C; they always call components in layer B. In a more relaxed layering system, components in a layer can call components in other layers that are not immediately below it. In all cases, you should avoid upstream calls and dependencies.

• **Use abstraction to implement loose coupling between layers**. This can be accomplished by defining interface components such as a façade with well-known inputs and outputs that translate requests into a format understood by components within the layer. In addition, you can also use **Interface** types or abstract base classes to define a common interface or shared abstraction (dependency inversion) that must be implemented by interface components.

• **Do not overload the functionality of a component**. For example, a UI processing component should not contain data access code. A common anti-pattern named is often found with base classes that attempt to provide too much functionality. The object will often have hundreds of functions and properties providing business functionality mixed with cross-cutting functionality such as logging and exception handling. The large size is caused by trying to handle different variations of child functionality requirements, which requires complex initialization. The end result is a design that is very error-prone and difficult to maintain.

• **Understand how components will communicate with each other**. This requires an understanding of the deployment scenarios your application will need to support. You need to determine if communication across physical boundaries or process boundaries should be supported, or if all components will run within the same process.

• **Prefer composition over inheritance**. Wherever possible, use composition over inheritance when reusing functionality because inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes. This also reduces the inheritance hierarchies, which can become very difficult to deal with.

• **Keep the data format consistent within a layer or component**. Mixing data formats will make the application more difficult to implement, extend, and maintain. Every time you need to convert data from one format to another, you are required to implement translation code to perform the operation.

• **Keep cross-cutting code abstracted from the application business logic as much as possible**. Cross-cutting code refers to code related to security, communications, or operational management such as logging and instrumentation. Attempting to mix this code with business logic can lead to a design that is difficult to extend and maintain. Changes to the cross-cutting code would require touching all of the business logic code that is mixed with the cross-cutting code. Consider using frameworks that can help to implement the cross-cutting concerns.

• **Be consistent in the naming conventions used**. Check to see if naming standards have been established by the organization. If not, you should establish common standards that will

be used for naming. This provides a consistent model that makes it easier for team members to evaluate code they did not write, which leads to better maintainability.

• **Establish the standards that should be used for exception handling**. For example, you should always catch exceptions at layer boundaries, you should not catch exceptions within a layer unless you can handle them in that layer, and you should not use exceptions to implement business logic. The standards should also include policies for error notification, logging, and instrumentation when there is an exception.

3.2. Design Considerations for Mobile Applications Architecture

Mobile system development often involves using different technologies due to platform restrictions. In the three architectures below, two use both Microsoft and Java technologies. In both cases applications developed in these disparate technologies must communicate seamlessly with each other. Web services, HTTP, and TCP sockets were used to bridge these gaps. A mobile system development team must have the skill and experience to determine the best data transfer design. If the application is being deployed on new handheld devices, there is a good chance that some configuration will be required. After a device is cold booted, the deployed application must be reloaded and the 802.11 wireless configuration must be restored. Different manufacturers use proprietary methods for loading applications and configuration settings. If the users should not have access to the OS (*e.g.* to play solitaire) then a top-level menu application may be needed to run at system startup. Device scanners must be configured with the correct barcode symbologies and symbology options. Configuration options may need to be remotely managed as well. Beyond device configuration is software deployment. The application or suite of applications must initially be loaded or provisioned on the device. If there are many devices, this may be a formidable challenge. There are software packages that manage device software and configurations. These packages rely on a software client on the device. Proprietary packages must typically be written for the management applications that specify the software and configuration files to load. If no management package is used, the application should be self-updating. Having the users send in their devices to have software reloaded is usually unrealistic.

Designing the graphical user interface (GUI) on a mobile device can be challenging because of the small screen and difficult data entry. If the application or data is complex, the user will need to interact with many screen objects such as entry fields, lists, and radio buttons. Complex screens will need to be divided into separate screens or tabbed interfaces. A wizard-like interface may be appropriate for some applications. Some applications on pen-based devices may require that a stylus is not required and the device's physical keys must be used instead. If a lot of free-form data entry is required then a tablet or notebook PC should be considered. Servers and desktop computers have progressed significantly and performance is typically not an issue anymore. Handheld computing devices are another story however. Many are very slow by comparison. Complex user interfaces, CPU intensive algorithms, and data processing can easily make an application user-hostile. Care must be taken during design to avoid performance pitfalls. One pitfall in Compact Framework development is using ADO.Net DataSets. They are very slow on most handheld devices. Although memory is cheaper than ever, most mobile devices come with a set amount of memory and cannot be upgraded. If systems analysis shows that data requirements include having large amounts of data on the handheld, this may limit your hardware choices. Efficient data storage is necessary, and low-level interfaces may be required to make the most of the memory available. Because cold boots typically erase all non-volatile memory in the device, design must ensure that critical data is stored in non-volatile memory. Security is a concern in many systems and mobile systems are no different. Mobile systems introduce a few new issues

however. What if the mobile device is lost? A stranger cannot be allowed access to your sensitive business data. Some hardware includes thumbprint scanners to authenticate users. A user login may also be implemented so that users must present a set of credentials before application use. Data transfer is a significant part of mobile application architecture because of the number of 'hops' the data must make. The methods and protocols should be carefully considered during system design.

The typical tradeoffs are-

- Security
- Ease of Implementation
- Reliability
- Cost of Ownership

Most mobile systems extend an existing business system or interface with an existing system. There are typically three major components to a mobile architecture figure 3.

- An existing system
- A middleware application
- A handheld application

Figure 4. Three Major Components to a Mobile Architecture

The reason a middleware application is usually needed is to provide data transformation, apply business logic, and be a central point of communication for the devices. If a new business system is being developed or rewritten then no middleware may be necessary; the appropriate logic can be built into the system to communicate with the devices from the start. However most business systems are not rewritten very often and it is economically unfeasible to rewrite them just to 'mobilize' them. Furthermore a middleware server may also serve a configuration management server. The architectures shown here are real-world architectures from actual projects. These mobile systems are in production in numerous locations.

Mobile application architectures are often modeled to highlight or illustrate the overall layout of the software (*e.g*., application code and platform) and hardware (*e.g*., client, server, and network devices). While there are many possible combinations of software and hardware, application architectures often fall into a series of recognizable patterns. Application architectures are commonly modeled in terms of a client-server architecture wherein one or more client devices requests information from a server device. In this paper, we proposed the our mobile applications development architecture as in Figure 4.

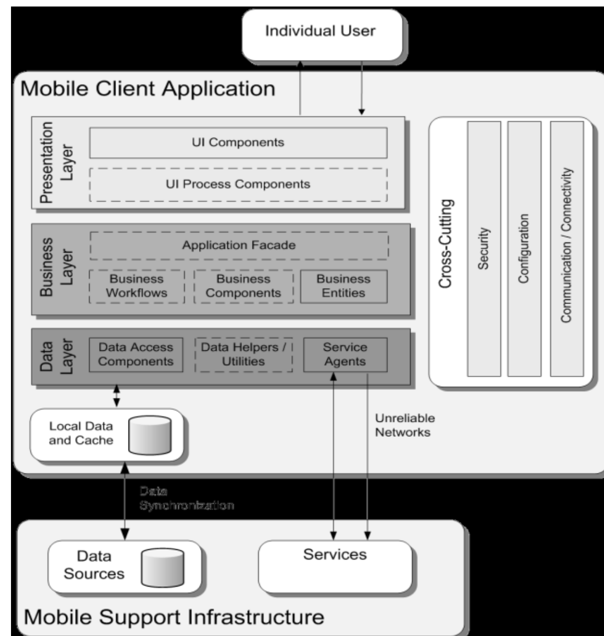


Figure 5. Mobile application architecture with components in this work

Application code functionality is not necessarily uniform throughout an application. Certain sections of application code are better suited for handling the user interface, while other sections are developed to manage the business logic or communicate with the database or back-end systems.

Layering describes the division of labor within the application code on a single machine. Layers are often no more than code modules placed in different folders or directories on the client or server. With client-side code, there are generally zero to three layers of application code. With server-side code, there are generally one to three layers of application code. This is partly a matter of good software design that helps code re-usability, partly a matter of security, and partly a matter of convenience.

A client with zero code layers essentially has no custom application code. This type of client is commonly referred to as a *thin client* and is possible in client-server architecture if the server holds all the custom application code. A client with one to three layers of application code is commonly referred to as a *fat client.* A server can also have one to three layers of custom application code. However, you cannot have zero code layers on a server by definition.

The code layer that interacts most closely with the user is often referred to as the Presentation Layer. The second layer is often referred to as the Business Layer, as it typically handles the business logic of the code. The third layer is often referred to as the Data Access Layer. It typically handles communication with the database or data sourceThe use of layers in a design allows you to separate functionality into different areas of concern. In other words, layers represent the logical grouping of components within the design. You should also define guidelines for communication between layers. For example, layer A can access layer B, but layer B cannot access layer A. Consider the following guidelines when designing layers:

• Layers should represent a logical grouping of components. For example, use separate layers for UI, business logic, and data access components.

• Components within a layer should be cohesive. In other words, the business layer

components should provide only operations related to application business logic.

• Consider using an **Interface** type to define the interface for each layer. This will allow you to create different implementations of that interface to improve testability.

• For mobile applications, implement a message-based interface between the presentation and business layers, even when the layers are not separated by a physical boundary. A message based interface is better suited to stateless Web operations, provides a façade to the business layer, and allows you to physically decouple the business tier from the presentation tier if this is required by security policies or in response to a security audit.

4. Mobile Applications Architecture Evaluation

4.1. Main component of the Architecture

We suggest the mobile platforms and architecture as shown in Figure 4. The key points to note from this figure are as follows. Firstly, mechanisms and policies for adaptation are tightly coupled and encapsulated in both applications and supporting middleware. This is a natural consequence of the trend towards applications being responsible for adapting to changes in context. There is no flow of control from the middleware to the applications, making coordinated responses to change impossible.

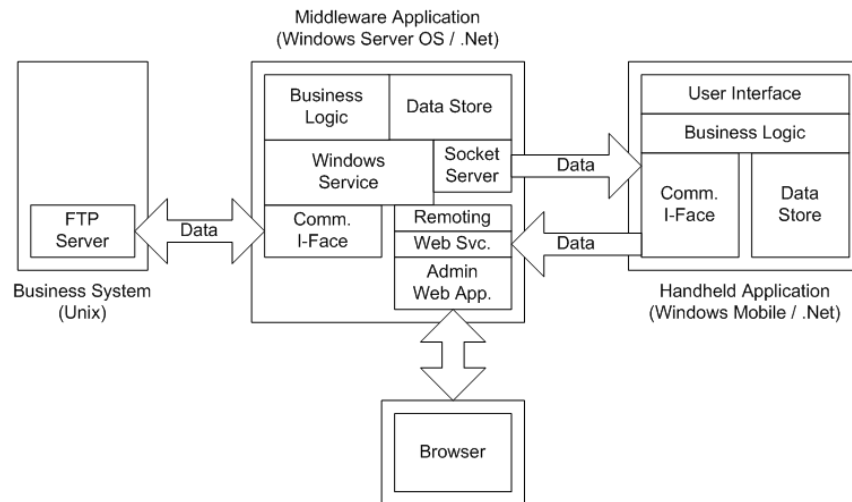


Figure 6. Suggested Mobile Applications Architecture

The middleware application uses a Windows service to configure the remoting infrastructure. The web service used by the mobile application accesses the application's business classes via remoting. The middleware is responsible for-

- Receiving and sending messages from the business system.
- Aggregating messages for a mobile device into a single message using a message envelope.
- Receiving messages from the mobile application.
- Processing messages from the mobile application in order.
- Creating a message envelope containing all messages for delivery to a mobile device.

- Storing messages from the business system for a device until that device connects.

Due to restrictions in the cellular provider's network, the middleware cannot "push" messages to a device. A web service was chosen to receive messages from the devices for this reason, and also because the mobile application is written in Java. Remoting was used to allow the web service to pass the message packet from the mobile application to the business layer. The business layer processes the message and always returns a message packet to the device via remoting and web service.

The main components of our architecture are as follows:

Context Space: Central to our architecture is the context space. This acts as a repository and distribution bus for information relating to QoS and context within the system. In particular, it is responsible for storing information from the device monitors, applications and middleware for use in determining the correct adaptation strategy in a given situation. The space must enable information from remote sources to be made available.

Device Monitors: Device monitors are typically simple daemon processes which monitor the state of devices and software components and report this information to the context space. Examples might include network device drivers and power management systems.

Applications and Mechanisms: Applications that include mechanisms for adaptation can register with the context space for information and control. It is the responsibility of the application developers to make the interfaces for adaptation mechanisms available.

Middleware and Mechanisms: In common with applications, middleware platforms can register with the context space for information and control. This enables the system to control and coordinate the actions of the middleware and the applications to avoid duplication of effort or conflicting actions.

Adaptation Control and Policies: The key aspect of our architecture is the adaptation control module. This is responsible for coordinating system responses to changes in the environment and resolving potential conflicts when multiple attributes change. The module is driven by a series of policies, which we envisage as being self-contained units that specify how a system should respond in a given situation.

The most novel aspect of our architecture is that we are hypothesising that policies can be constructed to support system wide adaptation to multiple triggers in an independent manner. Moreover, it will be necessary for such policies to be applicable in a wide range of system and application configurations and for the system to be able to understand and monitor the results of the policies' actions. While breaking up application code functionality into layers helps code re-usability, it does not automatically make the architecture scalable. In order to do so, it is important to distribute the code over multiple machines. *Tiers* describes the division of labor of application code on multiple machines. Tiering generally involves placing code modules on different machines in a distributed server environment. If the application code is already in layers, this makes tiering a much simpler process. The code that interacts most closely with the user is often placed in the Presentation Tier. A second tier, which holds the

application business logic and data access logic, is often referred to as the Application Tier. The servers that make up each tier may differ both in capability and number. For example, in a large-scale distributed web application environment, there may be a large number of inexpensive web servers in the Presentation Tier, a smaller number of application servers in the Application Tier, and two expensive clustered database servers in the Database Tier. The ability to add more servers is often referred to as *horizontal scaling* or *scaling out.* The ability to add more powerful servers is often referred to as *vertical scaling* or *scaling up.* Tiering the application code in such a fashion greatly facilitates the ability to scale applications.

4.2. Evaluation with example

In this section, we are planning evaluate the our proposed architecture with developing the example following the process. The existing architecture of the mobile POS(point On Sale) system should be presented based on the 4+1 view model with respect to all functional requirements defined by the evaluation contract. However, we here introduce just some significant parts useful for understanding the applicability of our approach which were identified as architectural spots. The use case view shown in Figure 7 shows us the primary purposes of sample POS system as use cases.

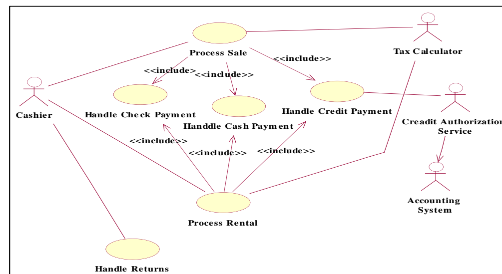
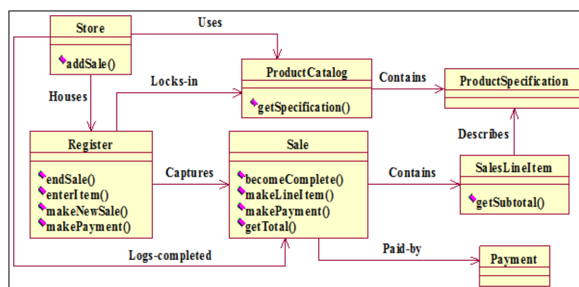
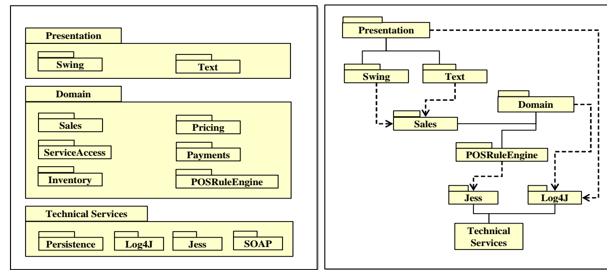


Figure 7. An example of use case view

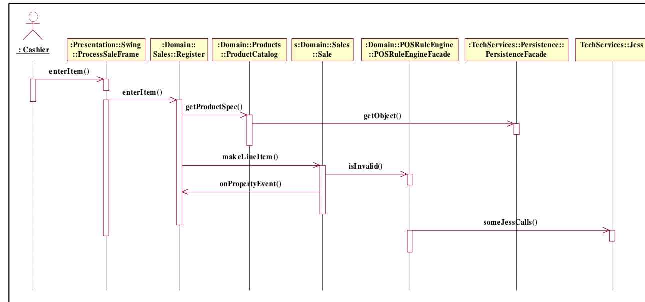
Subsequently, Figure 8 shows the logical aspects of the POS system. Figure 8(a) shows a set of key abstractions for the POS system and their logical relationships: association, usage, and composition. Figure 8(b) also represents a partial logical structure of layers in the POS system. In Figure 8(c), the interactions across the layers and packages are shown. Figure 8(d) shows how the system handles various pricing strategies, and Figure 8(e) shows that the system achieves protected variation with respect to location of services. The local cache of *ProductSpecification* is always searched for before attempting remote access for a "cache hit".



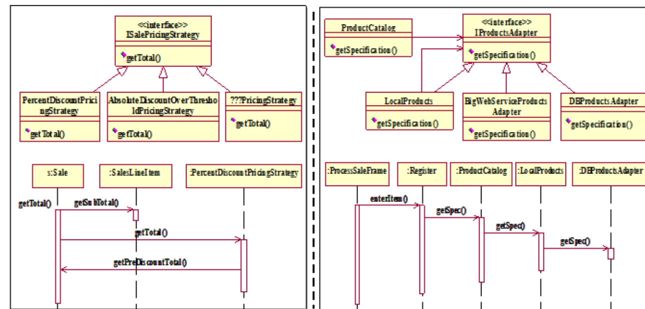
(a) a set of key abstractions of the POS system



(b) partial logical structure of the POS system



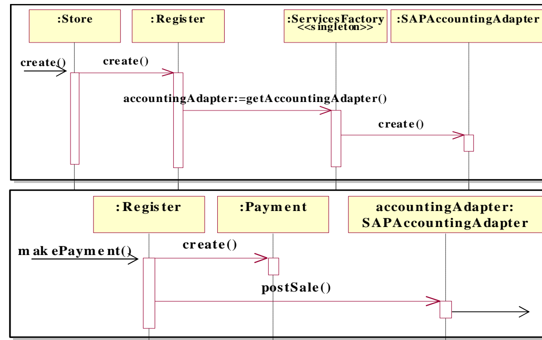
(c) interaction across the layers



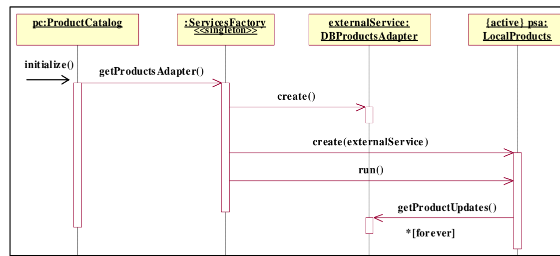
(d) behavior of pricing e) protected variation with respect to location of services

Figure 8. Examples of logical view

Finally, Figure 9 shows the POS system description in the aspects of a process view. Figure 9(a) shows that the system provides protected variations from the varying interfaces of external services such as external tax calculators, accounting systems, and so forth. Figure 9(b) also shows how the system solves the stale cache problem. Since product prices change quickly, the cache contains stale data, which is always a concern when data is replicated. One solution is to add a remote service operation that answers today's current changes. In Figure 9(b), the *LocalProducts* queries it every *n* minutes and updates its cache accordingly.



(a) Protected variation from the varying interfaces of external services



(b) stale data caching

Figure 9. Examples of process view

In our example, some architectural design decisions were identified. The architectural design decisions are presented with their rationale. Here, we illustrate only an example of the architectural design decisions shown in Figure 7 through 9. We evaluate a question, "How the layers can be connected?", could be raised in relation to the logical inter-connection mechanism represented in logical view. According to the question, a design issue, 'inter-layer connection mechanism', was determined as a decision variable. The decision value as a solution, 'using façade', was identified from architectural spots.

6. Conclusion and Further Works

These architectural requirements have then been used to develop a high-level architectural framework for supporting adaptive mobile systems. We hope that these requirements and the associated architectural framework will provide input into existing and future research efforts in the field of adaptive mobile systems. In particular, we hope that future middleware will provide better support for developers of applications which need to adapt to multiple contextual triggers in a cooperative environment. This architecture addresses these high-level requirements-

- High reliability.
- Easy installation and administration.
- Complex pricing rules must be implemented.
- The handheld application must be a web application; there should be no code on the handheld device.

- The system must be asynchronous, i.e. users can scan a barcode to activate a price and not have to wait to scan another item.
- The system must operate in near real-time, i.e. price changes must be seen at the registers immediately.

In this paper, we discuss mobile application architectures. We start by describing some of the general concepts and terms behind client-server architectures and follow this by describing clients and servers and the connectivity between them. We then present several interesting architectural patterns and describe why they are useful as general mobile application architecture solutions. We present a set of requirements for future mobile middleware which have been derived by considering the shortcomings of existing approaches and the needs of application developers. Key among these requirements is the need to support coordinated action between application and system components and the resolution of conflicts caused by the need to adapt to multiple contextual triggers. The paper concludes with the presentation of an architectural framework within which middleware researchers can deploy solutions to the problems identified. Finally, we discuss some of the tenets behind good architectural design and the considerations you need to be aware of when designing mobile applications. We also evaluate the our mobile application architecture to apply an example case as best practices.

References

[1] http://robtiffany.com/wp-ontent/uploads/2012/08/mobile_architecture_guide_v1.1.pdf.
[2] John Sprunger, "Mobile Architecture Best Practices for Mobile Application Design and Development", West Monroe Partners white paper.
[3] L. Bass, P. Clements and R. Kazman, "Software Architecture in Practice", Addison-Wesley, **(1998)**.
[4] J. Bosch, "Design and Use of Software Architectures", Addison-Wesley, **(2000)**.
[5] P. Clements, R. Kazman and M. Klein, "Evaluating Software Architectures", Addison-Wesley, **(2002)**.
[6] R. Kazman, "Experience with Performing Architecture Tradeoff Analysis", Proceedings of the 21st International Conference on Software Engineering, **(1999)** May, pp. 54-63.
[7] P. Kruchten, "The 4+1 View Model of Software Architecture", IEEE Software, vol. 12, no. 6, **(1995)** November, pp. 42-50.
[8] H. Eriksson and M. Penker, "UML Toolkit", Addison-Wesley, **(1998)**.
[9] M. Klein and R. Kazman, "Attribute-Based Architectural Styles", CMU/SEI-99-TR-022, Carnegie Mellon University, **(1999)** October.
[10] P. Inverardi, C. Mangano, F. Russo and S. Balsamo, "Performance Evaluation of a Software Architecture: A Case Study", Proceedings of the 9th International Workshop on Software Specification and Design, **(1998)** April, pp. 116-125.
[11] K. M. Cho, "Framework of Content Distribution in Mobile Network Environment", in Proc. the 2003 International Conference on Internet Computing (IC '03), **(2003)**, pp. 429-434.
[12] C. Mascolo, L. Capra, S. Zachariadis and W. Emmerich, "2002. XMIDDLE: a data-sharing middleware for mobile computing", International Journal on Personal and Wireless Communications,
[13] N. Medvidovic and G. Edwards, "Software architecture and mobility: A roadmap" The Journal of Systems and Software, vol. 83, **(2010)**, pp. 885–898.