Design Patterns in Mobile Architectures

Tomáš Chlouba

***Abstract*—**The paper inquires into classical design patterns, which are used to solve commonly appearing problems in software development. There is a new area of software development - the mobile platforms. In scope of these platforms, design patterns are given a new meaning. This paper describes the problematic of using these patterns within mobile architectures and provides a few examples on how this is done. The purpose is to present an overview of design pattern usage in mobile architectures and describe basic examples of this usage.

***Keywords*—** Mobile architectures, Design patterns, Android, Dependency injection, Factory method, Singleton

I. Introduction

DESIGN pattern in general is a reusable solution of a commonly appearing problem in software development. It is a term that comprises of a wide scale of software design problem solutions. There are also other types of patterns like architectonic patterns, so when we pursue other possible patterns, it is important, that the problematic actually concerns with design patterns.

Many new design patterns appeared with the coming of service oriented architecture, others emerge as additions to new programming techniques or areas of software development.

The emergence of design patterns is connected with the expansion of object-oriented programming and software design. It is object-oriented programming, that is the reason for existence of design patterns, because they provide effective solution of problems when using object-oriented techniques.

The basic definition of a design pattern is that it provides general reusable solution of commonly appearing problems in software development. These solutions are effective, because they have been used and tested in practice, but also because they simplify the overall application design. The idea of design patterns has been captured in [1] "Program to an 'interface', not an 'implementation'."

Design patterns usually start as a standardized solution for a commonly appearing problem, which has been solved many times before. These patterns were already used and applied in practice. They are a commonly used part of software applications. Their practical usage is extensive and with increasing complexity of applications, new patterns are emerging, sometimes derived from or composed of existing patterns. One of the examples is the Model-View-Controller pattern, which has grown to be a complex architecture thanks to many variations and extensions. There are also design patterns used specifically for mobile technologies. These patterns are based on techniques used when developing mobile applications.

We will now select a few design patterns and show the possibilities of using these patterns in mobile applications. We will also provide some examples of usage of these patterns. Some of the patterns need no modifications and can be used as are, with others it is necessary to change and revise them so that they can be used in mobile technologies. Some of the patterns cannot be used at all, for example because the problem the pattern attempts to solve no longer exists on mobile platforms.

There are also new patterns emerging with the coming of mobile platforms, such as the simple user interface patterns described in [2].

II. Design patterns in mobile architectures

It was already mentioned in the introduction, that the area of mobile technologies is one of the most quickly evolving areas of information technology. Mobile technologies are a perspective and well suited investment for many reasons. Most electronic devices are getting smaller, requiring less energy and a lower data transfer rate. This means that a lot more people start to use mobile devices as a helpful tool in many areas. At the same time the performance of these devices increases rapidly and extends the possibilities of using such devices. Based on innovations, new and more intelligent devices appear, providing the option of using a mobile device instead of a static one.

Hardware development is tied to software development, which evolves quickly as well. The quick development of the area is the cause for small amount of experts in the field and also for a lot of unexplored areas, which have only just been discovered. This is also the area of design patterns in mobile architectures in general.

Design patterns also fall into this category and research of these patterns can help the development of applications in mobile technologies.

Apart from application development specifically for mobile devices, we also come to contact with the integration of interfaces for mobile technologies in large information systems and expect extension of this area. It will be necessary more and more to incorporate mobile devices into corporate

Manuscript received October 31, 2010.

Tomas Chlouba is with the University of Hradec Kralove, Rokitanskeho 62, Hradec Kralove, 500 03 Czech Republic (phone: +420603175905; e-mail: tomas.chlouba@uhk.cz).

information systems, because not only managers but also other employers use mobile devices to log on to e-mail, electronic banking services and company information systems as well. For a manager for example, it is a great advantage to be able to access the company's information system from anywhere, because it allows him to have up to date information anytime.

The environment of mobile devices is very specific when compared to classical desktop application or client/server applications. These variations will be explained on the three following attributes.

A. Limitation

Mobile devices are because of their mobility limited especially by device size, processor speed, memory size, and also battery capacity. All these limitations are the reason for adaptation of mobile software to the boundaries set by these limitations. Programming for such devices must inevitably take these boundaries into account, and the concept of programming itself adapts to the properties of the mobile device.

Software developers are well aware of the constraints of mobile devices, the authors of [3] have even proposed a framework to address the issues, which are tied to the problem of limitations of mobile devices.

B. Ubiquity

Anytime, anywhere - that is the availability of modern mobile devices. This gives new meaning to device localization. Not only the current GPS location, but also the position of the device in three dimensional space. For example using the accelerometer.

C. Variability

Variability appears increasingly in mobile devices - there are various operation systems for these devices, different sizes and display resolutions, diverse performance and also miscellaneous additional technologies, that may not be available in all devices using the same operating system. That makes writing universal software far more difficult, because the programmer must take into account all the variations. This makes testing the applications even more difficult. Some of the examples of additional technologies may be the accelerometer, proximity sensor, or light sensor.

Programming for devices with similar performances has been done in the past, the difference is mainly that the modern devices are small, portable, and available to more people. Another difference is the fact, that one of the main factors, indicating the success of applications, is now the graphical form of the user interface. That leads to the development of techniques used to create user interface and also the emergence of design patterns for user interfaces.

Another area where new types of design patterns appear is the prediction of user behavior. The application attempts to predict what will be the user's next step and to provide relevant services and functions. It is an area, which is used mostly in advertisement and marketing, however one of other examples may be the automatic start of file download immediately when user clicks on a URL, so that the browser accelerates the file download by downloading even when user only chooses the file destination.

Such an extensive use of mobile devices raises the question of security of mobile applications on hardware and software level. Authors of mobile operating systems and experts in mobile software development are well aware of this problem and are attempting to solve it within the development platforms, so that additional applications do not have to be concerned with security that much.

III. Classical Design Patterns in Mobile Architectures

There is a group of patterns, which can be used the same or very similar way as in classical architectures. The level of abstraction allows the usage of these patterns almost the same way as in common desktop applications.

All of the examples in the following text will be presented on the Android platform, because it is open-source, which makes it easily accessible by everyone. The usage and possibility of using some design patterns may vary among the various mobile platforms, so we will also try to point to the right direction with other platforms as well. Specific examples however will be available for the Android platform only. Comparison of various mobile platforms is out of scope of this paper.

Open source tools provide the possibility of keeping up to date with the development of mobile architectures and that is a great advantage for developers, who can adapt their programming to current problems. This is also a step towards the inception of new patterns.

Programmers have been using design patterns for some time now and they got used to using them. This leads to the fact that they are seeking the same advantages provided by design pattern implementation in mobile technologies as well. That logically leads to the fact that programmers ask for adaptation of the same technologies they used while programming desktop applications to the mobile platform. This also means that widely used design patterns have been revisited and if possible reprogrammed for the mobile platforms. This is an approach a little bit different than that which was originally used to define the design patterns but also provides reflection of techniques used when programming for mobile platforms and leads to redefinition of the specific design pattern when used on the mobile platform.

A. Inversion of Control

Inversion of Control (IoC), is a design pattern, that allows loose coupling of tightly linked components. When programming the common way we create a class, which uses other classes and these classes use other classes etc. All of these classes are then tightly linked to each other and a change of a single class that is used in this model requires a change in the source code. IoC lets us loose this coupling, which is best explained by the Hollywood principle "Don't call us, we'll call

you." Basically it means that a class itself does not create instances of other classes, but the required instances are provided to the class from outside. There are several ways to achieve this, we will describe the three most used ones.

1) Constructor Injection

The classes which are injected with instances of other classes they require must have a constructor, which is capable of accepting the required types of objects. This technique is used by PicoContainer for example.

2) Setter Injection

Setters for class injections must be defined if setter injection is used. Setter injection is used by Spring Framework.

3) Interface Injection

At first, an interface is defined, which provides the methods, through which required class instances are set. Every class, which wants to use these instances must implement the relevant interface. Interface injection is used by Apache Excalibur, or Castle MicroKernel, which are based on the Avalon Framework.

B. Dependency Injection

Dependency injection (DI), is basically a newer name for IoC, however it also narrows the area of application of the pattern. It is a specific technique of using IoC. DI has several major advantages for programming as presented in [4].

a) *Simpler writing and maintenance of individual components which can be attributed to the lowering of mutual dependencies*

b) *It is much easier to test the code*

c) *Type cast is not necessary, because that is being taken care of by the IoC container*

d) *Component dependencies are defined explicitly, it is easier to find a way among them*

e) *IoC container usually requires no special intervention into the application code, individual components can then be used elsewhere without the need to modify the code*

There are also some drawbacks to DI, the main ones are these:

f) *It is difficult to find a way among the code for someone who does not know the IoC pattern*

g) *For simpler applications it is pointless to be written in difficult code, it is easier to write them in plain code*

h) *If the development environment does not know IoC, application development can be more complicated*

The IoC design pattern, respectively the DI pattern can be used in the area of mobile technologies without any change. Its advantage is more obvious in large projects, so in the area of mobile applications, it does not have to be such an advantage to use the pattern.

One of the usages of DI pattern is dealing with problems, which cannot be addressed any other way. Also, in case of mobile development environments, a DI framework may simplify the creation of software by allowing the developer to skip repeating steps when developing a mobile application.

DI on the mobile platform is provided for example by roboguice for the Android platform, the very lightweight Spring ME or Ninject for the .NET compact framework.

DI implementation on the Android platform using roboguice provides injections via annotations. The developer can use the *@Inject* (*@InjectView*, *@InjectResource*, etc.) annotations to inject class instances into methods. This is very useful to get rid of the initialization code and make the rest of the code easy to take in.

C. Factory method

The Factory method pattern is used to create a class instance on runtime. Its purpose is to allow the creation of various instances of classes, which implement the same interface and can be then delivered to a class which operates above them using the interface they implement. The classes using the interface don't care which implementation we deliver to them.

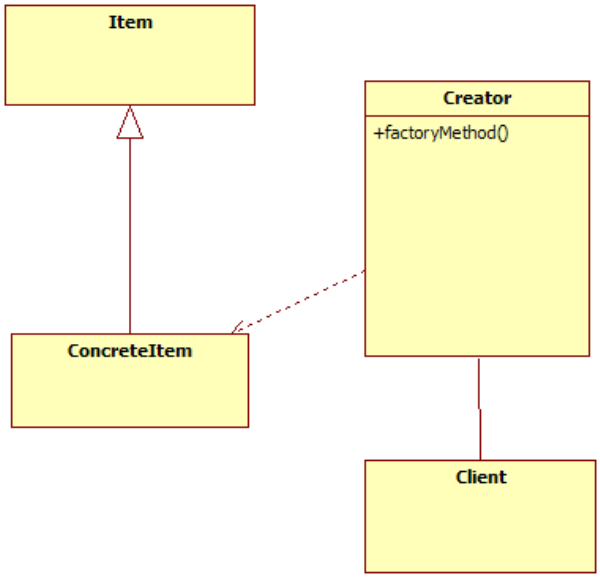

Figure 1: Factory method design pattern

The *ConcreteItem* implements the *Item* interface. The *Client* only has a reference for the *Creator* object. It requests a *ConcreteItem*, but does not specify which class exactly it should be. The *Creator* decides which instance to return and handles all the logic itself, so that the *Client* does not need to know any of this.

The Factory method pattern is implemented in Android itself, one of the examples of this implementation is the *SocketFactory* class and its subclasses. The *SSLSocketFactory* is used to create a specific *SSLSocket*, using the *createSocket* method, which opens a secure network connection and returns the *SSLSocket* instance. There can be more open connections and the socket factory provides the required instances.

This technique is widely used among programmers to solve issues when creating instances at runtime. There is no obstruction, preventing the usage of this pattern on various mobile platforms and it is being used already.

It is used for example to create proxy objects on runtime in the package of tools for parallel, distributed and mobile Java applications called Babylon, which is described in detail in [5].

D. Singleton

The goal of the singleton pattern is to have only a single instance of the specific class and to provide easy access to this instance.

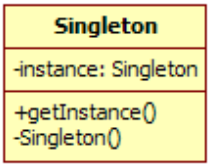

Figure 2: Singleton design pattern

To store the singleton instance we create a private attribute *instance*, the instance is created when the constructor is called. When an object requests the singleton instance, it calls the *getInstance* method, which checks if there is an existing instance of a singleton and returns that instance, or calls the constructor and returns the new instance.

The usage of the singleton pattern has some drawbacks even in desktop applications, bringing even more problems to the mobile platforms. There are several ways to create a singleton class, by using a static attribute, by finalizing the singleton class or by using a static method to create the instance.

On the Android platform, it is not recommended to use a static singleton, because the life cycle of a static object is not well under the control of the developer. Also, testing is far more difficult when using singletons. The modern approach is to make the application as lightweight as possible and when doing that, using a singleton is not a good idea either.

When using a DI framework, singleton may even be considered an anti-pattern, because of its use of static and private methods. Android may also terminate the singleton class if running out of memory, which brings a lot of other issues.

The singleton pattern should be used carefully and the developer should always thoroughly consider if using this pattern is really effective.

IV. Conclusion

The paper inquires into the usage of design patterns in mobile architectures and presents selected examples of classical design pattern application in mobile technologies. Included code examples and description of pattern application can be useful for developers to find a way in the application of design patterns in a mobile environment, especially when crossing over from desktop application development.

The basic examples may a base for a more complex and broader text, which cannot be in the scope of this paper. Another area which can be extended is mobile security and application of security issues to used design patterns.

From the introduced examples, it is obvious that classical design patterns will find their use in mobile technologies as well, however they always need to be revised to be properly used in mobile architectures.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns:Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995

[2] J. Narsoo, N. Mohamudally, *Identification of Design Patterns for Mobile Services with J2ME*, pp.623-643, Issues in Informing Science and Information Technology, vol. 5, 2008

[3] Girish Welling, B. R. Badrinath, *A Framework for Environment Aware Mobile Applications*, icdcs, pp.384, 17th IEEE International Conference on Distributed Computing Systems (ICDCS '97), 1997

[4] R. Harrop, J. Machacek, *Pro Spring*. New York: Springer-Verlag, 2005, pp. 49–92.

[5] W. van Heiningen, T. Brecht, S. MacDonald, *Exploiting dynamic proxies in middleware for distributed, parallel, and mobile Java applications,* ipdps, pp.247, Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, 2006