# aspectroid

exploring the aspect-oriented design space

Episode 2: Inside The Core

**Document Version 0.1 (DRAFT)**

This ebook can be referenced as:

Carlo Pescio, Aspectroid Episode 2: Inside the Core, 2015.

Full text, source code, binary files are available from aspectroid.com.

**License:**

# Table of Contents

# Acknowledgments

###############

# The Aspectroid Project

Aspectroid is an exploration of the design space, moving beyond conventional object oriented design and further into the aspect-oriented expanse. It takes place as a set of Episodes, where I present a small-scale but non-trivial problem, and I discuss a complete design, together with full source code and complemented by diagrams.

I launched this project because I was looking for a different kind of design narrative. Traditional design literature is not doing well[1]. Arguably, code is the contemporary form of design narrative; however, this position leaves many important notions and opportunities behind. I wanted to talk about realistic design issues, outgrowing the small, crafted examples one can easily show in a blog post, in a short paper, or in a few code snippets, moving closer to the complexity of full-blown applications. The straightforward way to do so is to actually develop an application as part of the narrative.

Episode 1 focused on the best way to combine OOP and AOP. The final result was quite different from traditional OO, and characterized by small islands of classes connected and complemented by aspects.

An exploration, however, makes sense if you keep moving toward the unfamiliar and the unknown. Therefore, I decided to move Episode 2 *from the app level to the platform level*, and use aspects to add a new feature inside the Android core.

This was not without challenges, and there is a lot to be learnt here: most literature on aspects is focusing on monolithic applications, but the Android core is spread out in a multitude of independently compiled, cooperating modules. Any non-trivial feature, including the one I'll be adding here, tends to cut across different modules.

[1] In early 2011, I wrote a blog post [Pescio2011a] asking a rather depressing question (*Is Software Design Literature Dead?*). I concluded that software design literature was basically dead, but although I proposed a few ways to bring it back to life, like creating Idea Books, I didn't step up to the challenge. This ebook is an example (in the small) of what an Idea Book could look like.

Even adding AspectJ support to the Android build was a small challenge in itself, which I solved by writing a compiler wrapper (among other things).

You can read more about the motivation for this work in the next chapter. However, *I think it's important to understand that aspect-oriented technologies can radically change the process of creating and maintaining a fork of a large-scale, open source application* (in this case, the Android core).

The common process is to fork using a version control system, and then periodically align with the trunk by using a merge tool, let's say a 3-way merge tool. This is fine for small or slow-moving applications, but the Android base code is huge and subject to radical changes between versions. My experience is that this process becomes significantly time-consuming after a while. Given the current Android modular structure, your features / changes tend to cut across many modules, and keeping track of *why* you changed some portion (so that you can port that change to the next version) requires quite some care.

AOP can help tremendously here, because you never change the base code. It's not a free lunch, as I'll discuss in the next chapter, but it's a game-changing approach. In fact, forking large-scale applications might well be the AOP technological sweet spot.

## Prerequisites

This is not a book for beginners, so to fully appreciate its contents there are a few technical prerequisites:

- You need actual coding experience. Without coding experience, it's almost impossible to relate to the fine-grained decisions discussed here.
- You need an appreciation of software design.
- A basic understanding of aspect oriented programming is beneficial. A few introductory works are listed below. While you can read this ebook without a significant experience with AOP, some exposure to the fundamental concepts is certainly useful.
- I'm using AspectJ, and therefore Java. Familiarity with the Java syntax is useful. The fine details of AspectJ are probably less important than an understanding of what can be done using AOP.
- I'm also working inside the Android core. While you don't need to have done so before, at least some knowledge about Android *apps* development will be useful. Again, you can follow the reasoning without a solid understanding of the Android fundamentals, but to fully appreciate some details some experience is probably required.

Overall, what you need more is an open mind. Many choices that I'll be making wouldn't be appropriate without aspects. Even if you have used aspects before, but only for technological cross-cutting concerns, you may find my usage surprising and somewhat going against common wisdom (like the "AOP is not for singleton[2]" advice in [FF2000]). You may have to suspend judgment until you see how pieces are woven together.

Remember: *this is a book for thinkers*: if you're looking for a recipe book, you're probably better off with other sources. If, however, you like the idea of exploring new ways to structure your software, you'll probably enjoy this text.

[2] Which has nothing to do with the Singleton pattern ☺

## Introductory material

If you're new to AOP and/or AspectJ, the most readable material I've found is the three-part "*I Want my AOP*" series by Ramnivas Laddad. The examples are based on technological concerns (as usual) but it will give you a good introduction to both AOP and AspectJ without bogging you down with academic rigor.

If you're in for a more formal treatment, you can read one of the initial works from Kiczales et al [KLMMVLI97].

Finally, if you want to get a good overview of the field, and don't mind books, [FECA2004] is an excellent resource.

If you want to do your own experiments, you'll need to set up a regular build environment first. It's a rather long procedure, described online at source.android.com. I recommend that you try to build and install (or run in the emulator) before you move any further.

Once the regular build works, you can add AspectJ to the mix. The process is described in Appendix A. You'll have to install AspectJ on the build machine, add the AspectJ run-time library jar to the platform, compile my compiler wrapper and then modify a few build scripts to actually invoke the wrapper when needed.

Note that the features I'm adding in this episode require access to the USB; therefore, they won't work in the emulator. You'll have to flash the image to a real device (I did my testing on a Nexus 7 2013). However, you may simplify the code so that the USB is not used or (of course) develop a different extension with similar techniques.

Full source code for this episode is available on the *aspectroid website*.

# Chapter 1: Why and What

In the past few years, I've spent quite some time inside the android source code. I've been doing so as part of real-world projects, where android wasn't used in a tablet or a phone, but inside an industrial, fitness or entertainment device.

That experience opened my eyes to a number of facts:

- The android base code is deeply entangled with "mobile" concerns that do not apply (for instance) to industrial devices. The "battery" concern, to name one, is rather pervasive inside the entire code. However, some of the devices I've been working with require more than 1KW to operate; keeping the android device powered all the time adds nothing.

- There are in fact a large number of cross-cutting concerns inside the android base code. This leads to unexpected couplings, and can be detrimental to modularity. For instance, the audio manager ends up depending on the telephony manager; that's ok for a phone but a big non-modular decision for pretty much everything else.

- Along the same lines, many meaningful changes[3] to the android code ends up scattered among different compilation units. It's easy to lose sight of what is required for Feature A when it's implemented by changing a few lines in a couple of files in a module, other lines in a different module, and so on.

- Because of the above, and because of the high code volatility between versions, porting your customization to a later version requires a significant effort. I've found that keeping a detailed documentation about the changes was basically necessary (even when just a few lines were involved) to rebuild enough context and perspective to port those small changes to the new base code.

[3] That is, changes meaningful for the end users, or end-to-end functionalities.

All this was basically screaming “AOP” at different levels:

- It would have been extremely beneficial for the android team to manage cross-cutting concerns using AOP[4]. That would also make *removing unnecessary features* a breeze. I don’t have a phone, a gps, a power problem in any of my industrial devices. I do have Ethernet though, and that was hard because the “network” concern is not well modularized, but instead “embedded” in the wifi or 3g/data code.

- It would have been beneficial to customize the base code using aspects, instead of tweaking the existing code. That way, small changes to different files of the same module would not be scattered, and my changes would not be mixed with the base code. That would provide a much better context when reasoning about the changes.

There was also previous, encouraging literature [#########] on applying aspects inside operating systems. In short, it was an interesting learning opportunity. In practice, it proved a bit challenging to get started, because you can’t just install AspectJ and begin coding. The android build process is rather complex and does not easily accept a new language. With enough determination, however, I moved past this point. I just needed a small-scale, realistic problem that would make some sense on a regular tablet, because that’s what I expect most of you guys have and can easily relate to.

[4] I have taken the time to identify and quantify a large number of cross-cutting concerns inside the android platform code (Java code only). The method and the results are discussed in Appendix B. Many cross-cutting concerns were simply due to the lack of an *in-process plugin architecture* inside the base platform, but some would definitely require AOP-like techniques to be disentangled. Some of those were really unexpected (like the WifiStateMachine depending on the BackupManager).

## Choosing the problem

I decided early on that I would tackle an area of Android that I had not explored in depth before. I wanted this experience to be as realistic and representative as possible, and given the size of Android, it's quite likely that any real-world scenario will include some unfamiliar portions.

I also decided to choose a meaningful task, not just some abstract concoction. When a task is meaningful, it's easy to understand whether or not you reached the goal, whether or not a compromise is perhaps too much of a compromise, and so on. I also wanted something that would make sense on the average tablet / phone, not just on some exotic embedded hardware, because it's easier to relate to that.

In the end, I choose to implement something I always wanted as a developer. As you probably know, inside the Developer Options there is a "Stay awake" entry that you can select to prevent the screen from going off while testing / debugging. That option works by observing the presence of power, so if you check that option, your device will stay on when you provide power.

That's ok, sort of, when you use that device only for development. However, I also use my "personal" phone and tablet for development, although I have a few more devices lying around. I don't really like that option on my personal devices, because I normally don't want their screens to be on while *charging*.

In fact, that's not what I want at all, even on my development devices. I want the screen to stay on while *debugging*, not while charging. So, that will be my task: add a checkbox to the Developer Options so that, when checked, the screen will not go off while you're debugging, not merely because you're providing power. That's easier said than done, and there are a number of nuances we'll have to learn about Android before we can do that, but this is exactly what I wanted – a small, but realistic challenge involving multiple modules and services inside the Android core.

## Android version

The work described in this book is based on android 5.0. As I'm writing this, version 5.02 is out, and 5.1 is being discussed. I fully expect a later version to be available when I'm done, because I'm a slow writer these days.

Still, I'll try to turn this potential disadvantage (rapid obsolescence of details – the message is much more stable here) into an experimental advantage. After all, part of my thesis is that porting changes to a later version of the OS should be simpler, thanks to aspect technology. So, after releasing version 1.0 of this book, I'll go back, get the latest android version, and see what happens when I try to reapply my changes. However that goes, there will be something to be learnt, which is the entire purpose of this project.

# Chapter 2: Charting the Unknown

The first step, whenever you think about changing something inside Android, is to locate the best (sometimes, only) place where you can apply your change. This is not by itself trivial; the code base is huge, and documentation won't usually help you much. It's Jedi time: "use the source, Luke".

In practice, you have to start somewhere, and in this specific case, the best you can do is to get familiar with the most similar option (the "stay awake" option based on power). There is a rather clear place where you can start unraveling, and it's the UI itself. In the end, we'll want to add a "stay awake when debugging" option to the same screen, so it pays off to familiarize with the code anyway. To tell the truth, I already knew a bit about this portion of android, because we often add custom options when creating custom devices. The difference, however, is that we usually do that by changing the Android code.

Note: in what follows, I'll have to reference the android source code quite often. I'll copy the most relevant snippets here, but it's useful to have the entire source code handy. As I do not expect everyone to have the android source on his reading device, I added a hyperlink to a formatted, navigable, online version of each relevant file. There are a few websites offering this service, but I opted for grepcode.com, which I tend to use quite often.

## Exploring the Settings app

The Settings app (located under *packages\apps\Settings* in the source tree) is not plugin-based, that is, you can't somehow add entries without changing and recompiling the app itself. However, it's relatively simple to understand, and what we're looking for is in the *DevelopmentSettings* class, a Fragment containing all the logic for the (guess what) developer settings.

Inside that class, the "stay awake" logic gravitates around the *mKeepScreenOn* member, which reflects the checkbox in the settings UI. You can actually guess that from the name itself, but I did some cross-checking with the layout and resource files to confirm the intuition.

When needed, *mKeepScreenOn* is stored / retrieved from *Settings.Global*[5], using *Settings.Global.STAY_ON_WHILE_PLUGGED_IN* as a key. From there, other parts of the system will be able to read it and act accordingly. Interestingly enough, *mKeepScreenOn* is not stored a Boolean. It's stored as an integer, representing a mask. The stored value says *when* to keep the screen on, not just to keep it on or not. You can see it from this (reformatted) snippet, where the state of the checkbox is being stored:

DevelopmentSettings snippet

```
if (preference == mKeepScreenOn)
{
  Settings.Global.putInt(
    getActivity().getContentResolver(),
    Settings.Global.STAY_ON_WHILE_PLUGGED_IN,
    mKeepScreenOn.isChecked() ?
     (BatteryManager.BATTERY_PLUGGED_AC |
      BatteryManager.BATTERY_PLUGGED_USB) :
    0);
}
```

Ignoring that detail for a moment, this is the first piece of the puzzle, and in a sense is also a lead you have to follow to move further: find the users of *Settings.Global.STAY_ON_WHILE_PLUGGED_IN*, and you'll find the logic that is actually keeping the screen on.

While there are websites with the full android source code indexed and cross-referenced, they don't always get it right on static constants, so I just used a good old recursive grep here. I expected the Power Manager to be somehow involved in the "stay on" affair, but it turned out that it's not the only interested party: grep returned 3 hits.

*com.android.server.power.PowerManagerService*

*com.android.server.wifi.WifiController*

*com.android.server.wifi.WifiServiceImpl*

[5] Settings.Global is a system-wide content provider, which can be read from regular apps but can only be written by system apps. I'll talk a little more about this design detail later on.

So, that setting doesn't just keep the screen on; it keeps wifi active as well. This is a cross-cutting concern[6] that is dealt with simply by coupling all the modules to a global content provider, which acts as a global variable in a logically distributed system like Android.

Inside the wifi module, only WifiController contains actual logic based on the setting; WifiServiceImpl simply prints out the current value as part of the dump method. Not the best possible modular choice, but relatively harmless.

In the end, I decided to leave the wifi portion untouched. It would not add significantly to the book, as the challenges and the techniques do not differ from those revolving around the power manager. Therefore, I'll deal with the screen state only, as originally planned, ignoring the network state.

The settings app doesn't just provide a lead to the next module to explore. It's also the place where we want to add our own option. The key is *how*. The settings fragment is populated from an XML description of the options themselves. XML is outside AspectJ reach, so by going this way we would be back to the usual game. The alternative is to add the new option programmatically. This is entirely possible, with some consequences that I'll discuss in the next chapter. In practice, one can easily:

- Intercept DevelopmentSettings.onCreate and add a new checkbox option.

- Intercept DevelopmentSettings.onResume and refresh the option value from some storage, yet to be defined.

- When the checkbox preference changes, reflect the change in the same storage.

[6] That is, it is cutting across the modular structure of Android. The Wifi manager and the Power manager are two logically separated modules, yet the introduction of that option required changes in both. In terms of my Physics of Software, they both are C+D/U-entangled with that option (see [Pescio2010], [Pescio2011] for more).

That looks simple from an aspect perspective, and only leaves the option of *where* to store the value open. It must be accessible from other modules, so storing it in the same Global content provider as the other settings may seem fine.

However, that would just perpetrate a non-modular design choice, and would also require even more changes to the existing Android code. Although aspects make those changes non-invasive, maybe we can opt for a more modular alternative. I'll discuss that as part of the next chapter.

## Exploring the Power Manager

The Power Manager is deeply nested in the source tree, appearing under *frameworks\base\services\core\java\com\android\server\power*. It is one of the modules that get merged into the *services* module[7] as part of the build process. As usual in most part of Android, the power manager is built around a large (over 3KLOC) manager class, called *PowerManagerService*.

Time do dive in again, look for *Settings.Global.STAY_ON_WHILE_PLUGGED_IN* inside that class, and check how it's being used. I'll spare you some exploration (but on the other hand, you may find it interesting to read the code yourself).

Somewhat unexpectedly, the *PowerManagerService* is not only reading that value, but also writing it. That happens in *setStayOnSettingInternal*, which in turn is called by *setStayOnSetting*, which is exposed in the service AIDL. From a comment, we learn it can be invoked through the (undocumented) "adb shell svc power stayon" command (which takes a boolean on the command line). Nice to know, and yet another cross-cutting concern.

Roundabout: if you explore the *svc* command, again well nested under *frameworks\base\cmds\svc\src\com\android\commands\svc*, you'll discover it's able to forward commands to the *PowerCommand* class, which in turn can talk to the *PowerManagerService* through its *IPowerManager* interface, and therefore can call *setStayOnSetting*.

In case you're wondering, *PowerCommand* is remapping the boolean taken on the command line to a bit mask, as required by the *PowerManagerService*. Not much separation of concerns and information hiding here, and in fact the mask ends up being slightly different than in the Settings app.

Here is a (reformatted) snippet:

[7] For an overview of this part of Android, see [Yaghmour2013].

PowerCommand snippet

```
if( "stayon".equals(args[1]) && args.length == 3 )
{
  int val;
  if( "true".equals(args[2]) )
  {
    val = BatteryManager.BATTERY_PLUGGED_AC |
          BatteryManager.BATTERY_PLUGGED_USB |
          BatteryManager.BATTERY_PLUGGED_WIRELESS;
  }
  // …
}
```

It is left as an exercise ☺ to learn why it's ok to send the command just to the power manager service, even though the wifi service is also affected by the setting (hint: it works). Once again, however, we can ignore this part. I never planned to expose the new feature through the svc command, and it would not add to the challenges, except by increasing the number of separate modules that are affected by a single end-user feature.

Moving on to more immediate matters, the Power Manager Service is also subscribing the *STAY_ON_WHILE_PLUGGED_IN* key from the global content provider, to get notified when the settings changes. This is reasonable and expected, and happens inside *systemReady*, where a bunch of other items are subscribed.

PowerManagerService snippet

```
public void systemReady(IAppOpsService appOps)
{
  // …
  final ContentResolver resolver = mContext.getContentResolver();
  // …
  resolver.registerContentObserver(
    Settings.Global.getUriFor(Settings.Global.STAY_ON_WHILE_PLUGGED_IN),
    false, mSettingsObserver, UserHandle.USER_ALL);
  //…
}
```

All the subscriptions get routed to the same observer (*mSettingsObserver*), which is an instance of a small inner class (*SettingsObserver*)[8].

SettingsObserver snippet

```
private final class SettingsObserver extends ContentObserver
{
  // …
  @Override
  public void onChange(boolean selfChange, Uri uri)
  {
    synchronized( mLock )
    {
      handleSettingsChangedLocked();
    }
  }
}
```

When any subscribed setting changes, *SettingsObserver* simply calls *handleSettingsChangedLocked* on the outer PowerManagerService instance. Once again, that's a short function, calling *updateSettingsLocked*, where *STAY_ON_WHILE_PLUGGED_IN* is finally being read.

It may seem like we have found the end of the bundle, that is, a place where we could somehow inject additional logic to keep the screen on, mimicking what is being done for *STAY_ON_WHILE_PLUGGED_IN*, but it's not quite like that.

The logic inside *updateSettingsLocked* is rather "complex", that is, it's calculating a few flags and masks. Nothing we can't understand in detail, but "complex" logic is usually a sign of potential instabilities. Therefore, this function does not look like the best candidate for a pointcut, but it's a good start, so I'll leave it for the next chapter to find a more suitable candidate.

[8] Lacking lambda functions, there is a lot of similar boilerplate code inside Android and inside Android apps. Upgrading the entire system to support Java 8, in my opinion, would have been a better move than simply switching to a different IDE for apps development.

To recap:

- The power manager service subscribes *STAY_ON_WHILE_PLUGGED_IN* inside its *systemReady* method. The subscriber (indirectly) calls *updateSettingsLocked*, which is also called explicitly inside *systemReady*, to give it an initial wake up call.

- *updateSettingsLocked* does a bit of logic to determine whether or not the screen must stay on, by reading *STAY_ON_WHILE_PLUGGED_IN* among other things.

- *updateSettingsLocked* doesn't look like a great candidate for a pointcut, while *systemReady* looks like a nice place where we could add a subscription for our own setting, or something along those lines (I'll explain in the next chapter why we don't want to observe the setting itself but something else).

With that in mind, we can move on and explore the last piece of the puzzle.

## Am I debugging something?

Finally, we need to understand how to capture the notion that the device is under debugging. This is not going to be trivial; the "real" debugging server is the adb server[9], which is written in C, therefore outside AspectJ reach. Having lost the most natural candidate, where do we begin to look?

Interestingly, the Settings module may help once again. As you know, there is a "USB debugging" setting in the Developer Options, which can be used to enable / disable usb debugging. Somehow, that option is influencing usb debugging, and might just show us the way. Time to dig in; we already know the drill.

Without much ado: inside the *DevelopmentSettings* fragment (the same class we explored earlier) the *ENABLE_ADB* checkbox is bound to the *mEnableAdb* member, which is finally reflected in the *Settings.Global* content provider, using the *Settings.Global.ADB_ENABLED* key. So once again we just need to find out who is reading that value, and we'll have our lead.

It doesn't take much to find out our candidate: *UsbDeviceManager*, another longish manager class at over 900 lines, nested under *frameworks\base\services\usb\java\com\android\server\usb*. The strategy used there will sound familiar:

An inner class *AdbSettingsObserver* is used to observe *Settings.Global.ADB_ENABLED*. When that value changes, the *MSG_ENABLE_ADB* message is sent to *mHandler*, which unsurprisingly is an instance of yet another inner class (*UsbHandler*). Inside *UsbHandler*, *MSG_ENABLE_ADB* is handled by calling *setAdbEnabled*. That function will cache the adb enabled state in an outer class member (*mAdbEnabled*) and then forward the call to the *UsbDebuggingManager*.

So, we know the class[es] responsible to enable or disable adb, but what about adb being *connected*, that is, our device being under [usb] debugging? Well, it turns out that the UsbHandler is keeping track of the connected state as well, in a data member properly called *mConnected*. That data is being set through some

[9] The adb server, somewhat surprisingly, is sharing the entire source code with the adb client normally used on the PC side.

indirection as usual (feel free to explore the Android source code to find out), but in the end the data member itself looks like a reasonable pointcut.

The *mConnected* is not being refreshed if you enable / disable the adb itself, so we need to check the *mAdbEnabled* value as well.

So, to recap:

The state of adb being connected can be approximated by ANDing together *UsbDeviceManager.mAdbEnabled* and *UsbDeviceManager.UsbHandler.mConnected*[10]. It's just an approximation, because:

- It will not handle the case where you're debugging through a wifi connection.

- The *mConnected* flag doesn't really tell us if we're connected to a debugger; it tells that we are connected in a way that allows the device to be debugged.

Is this a reasonable compromise? Honestly, I think so. It fits well with my scenarios:

- Device charging: no effect.
- Device in host mode (USB OTG connected): no effect.
- Device with debugging turned off: no effect.
- Device with debugging turned on, connected to a computer: screen on.

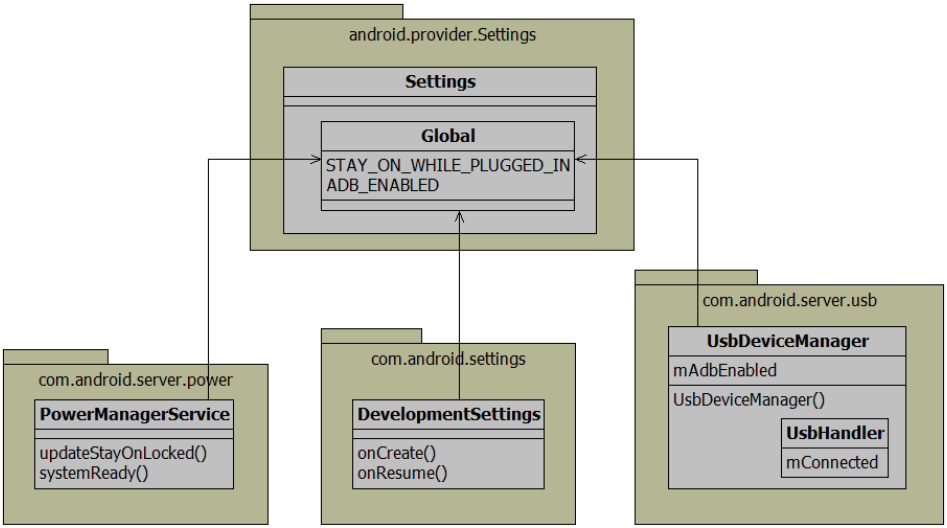
Although not exactly the same as being inside a debugging session, it's close enough to say that we can move on and try to make it work with aspects, without any change to the Android code.

[10] In practice, having to observe an inner class data member proved more challenging than I expected, due to limitations in AspectJ. I'll discuss that, and my workaround, in the next ### chapter.

## A map of the problem

Here is a simplified map of what we discovered so far:

**Diagram 1**




The power manager service, the USB device manager and the development settings module are cooperating through a global content provider, where the two options we've been investigating (stay on while powered, enable USB debugging) are being stored.

The member functions and inner classes that I've represented here are the most likely candidates to be advised as we move from understanding the existing code to designing our own aspect-oriented solution, which is the subject of the next chapter.

# Chapter 3: Doing it with Aspects

######

Macro-Structure of the solution

Main design decision; logical vs physical content provider

Interlude on “oo” content provider?

Components vs namespaces

Test outside the core / scaffolding

Finding “robust” code to advise

Concerns and source code;

Naming standard (different)

Other files: manifest xml; localization; makefiles; etc

# Chapter 4: Reflections

Previous literature on AOP in OS

Aspects vs distributed architectures

- Tools

Aspects and multi-language applications

- Xml, mk are languages too
- Native portions out of reach (adbd)

Aspects as a privileged solution

Why google should be using aspects in the android platform code

Why we should use aspects to customize the android platform

# Chapter 5: Does it really pay off?

Hypothesis / Validation: 5.1 or whatever

# Chapter 6: Wrap up

I welcome your feedback: in the spirit of this work, it would be more beneficial if we could share it with everyone else. I've set up a discussion board, which you can join from the aspectroid website.

Consider sharing this work with your colleagues and friends, sending a link to aspectroid.com through the usual social channels, mentioning it in your blog, on Hacker News, DZone or other sites, or by sharing the PDF itself. Sharing this ebook is the best way to say you liked it ☺.

# Appendix A: Adding AspectJ to the Android build

A small nightmare of its own

# Appendix B: cross-cutting concerns inside Android

Extraction Procedure and Results

# Bibliography

*Whenever possible, I've provided a hyperlink to a freely available version of the references. Hyperlinks were valid as of ########feb 2015.*


[FECA2004] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, Mehmet Aksit, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[FF2000] Robert E. Filman, Daniel P. Friedman, *Aspect-Oriented Programming is Quantification and Obliviousness*, Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000.

[KLMMVLI97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, *Aspect-Oriented Programming*, proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241, June 1997.

[Pescio2010] Carlo Pescio, *Notes on Software Design, Chapter 12: Entanglement*, published on carlopescio.com, November 2010.

[Pescio2011] Carlo Pescio, *Notes on Software Design, Chapter 13: on Change*, published on carlopescio.com, January 2011.

[Yaghmour2013] Karim Yaghmour, "Embedded Android", O'Reilly Media, 2013.

# About the author

I've been breathing software for over 35 years, learning, practicing, teaching and writing.

In my everyday life, I design software-intensive systems at different scales and in different domains, using a number of paradigms, languages and technologies.

I complement practice with a rather strong theoretical background, and I tend to go where no one has gone before.

Oh, I like Marvel comics too ☺

For a longer blurb, see [aspectroid.com](aspectroid.com).