

# aspectroid

exploring the aspect-oriented design space

Episode 2: Inside The Core



©Carlo Pescio, 2015

## Document Version 0.2 (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](http://aspectroid.com).

### License:

This work is licensed under the **Creative Commons Attribution – NonCommercial-NoDerivatives 4.0 International License**. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

In short (and not as a substitute of the official license text) you can freely share this ebook, but you have to give appropriate credit (e.g. with a link to [aspectroid.com](http://aspectroid.com)); you cannot profit from the distribution, and you cannot alter the ebook or create derivative works for distribution.

As an exception to the aforementioned license:

- All the source code snippets taken from the Android source code naturally keep the original [Apache License, Version 2.0](http://www.apache.org/licenses/LICENSE-2.0).
- All **my** source code in this ebook, and the full source code package you can download from [aspectroid.com](http://aspectroid.com), are licensed under the [GPLV3 license](http://www.gnu.org/licenses/gpl-3.0).

In short (and not as a substitute of the official license text) the code is free to reuse with attribution, but derivative work must be distributed under the same license (GPLV3 with source code).

# Table of Contents

Acknowledgments .....	5
The Aspectroid Project .....	6
Prerequisites.....	8
Introductory material .....	9
Chapter 1: Why and What .....	10
Choosing the problem .....	12
Android version .....	13
Chapter 2: Charting the Unknown .....	14
Exploring the Settings app.....	14
Exploring the Power Manager .....	18
Am I debugging something?.....	23
A map of the problem .....	26
Chapter 3: Doing it with Aspects .....	27
Logical architecture .....	30
Physical architecture .....	33
The process I used .....	34
The code.....	35
Extending the Settings app.....	35
Extending the UsbDeviceManager .....	39
Extending the PowerManager .....	47
The ScreenState Contract and API.....	51
Adding the library to the build.....	54
Adding the ScreenState content provider .....	55
The Repository.....	58
The Makefile.....	63
Last few changes before we're done .....	64

Chapter 4: Reflections ..... 66

Chapter 5: Does it really pay off? ..... 68

Chapter 6: Wrap up ..... 69

Appendix A: Adding AspectJ to the Android build ..... 70

Appendix B: cross-cutting concerns inside Android ..... 71

Bibliography ..... 72

About the author ..... 73

# 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<sup>1</sup>. 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.

---

<sup>1</sup> 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<sup>2</sup>" 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.

---

<sup>2</sup> 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](http://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<sup>3</sup> 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.

---

<sup>3</sup> 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<sup>4</sup>. 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.

---

<sup>4</sup> 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 Chapter 3, version 5.1 is out, and M has been announced. 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](http://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](#)<sup>5</sup>, 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 as 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
<pre>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); }</pre>

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.

---

<sup>5</sup> Settings.Global is a system-wide value table, which can be read from regular apps but can only be written by system apps.

*com.android.server.power.PowerManagerService*

*com.android.server.wifi.WifiController*

*com.android.server.wifi.WifiServiceImpl*

So, that setting doesn't just keep the screen on; it keeps wifi active as well. This is a cross-cutting concern<sup>6</sup> that is dealt with simply by coupling all the modules to a system-wide table, 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. Looking at the code, the challenges and the techniques to be used there would not differ from those revolving around the power manager. In that sense, dealing with the wifi part would not add significantly to the book, except by making it longer. 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.

---

<sup>6</sup> 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).

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

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 value table as the other settings may seem fine.

However, doing so naively (by adding new explicit fields<sup>7</sup> to Settings.Global and having clients explicitly read / write it from / to there) would just perpetrate a non-modular design choice, and would also require more changes to the existing Android code. Although aspects make those changes non-invasive, maybe we can opt for a different alternative. I'll discuss that as part of the next chapter.

---

<sup>7</sup> Settings.Global is exposed as part of the android API. Adding new explicit field changes the. That in turn will cause the build process to recompile a large portion of android.

## 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<sup>8</sup> as part of the build process. As usual in most parts of Android, the power manager is built around a large (over 3KLOC) manager class, called [PowerManagerService](#).

Time to 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:

---

<sup>8</sup> 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 value table, 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);
    //...
    updateSettingsLocked();
    //...
}
```

All the subscriptions get routed to the same observer (*mSettingsObserver*), which is an instance of a small inner class (*SettingsObserver*)<sup>9</sup>.

#### 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.

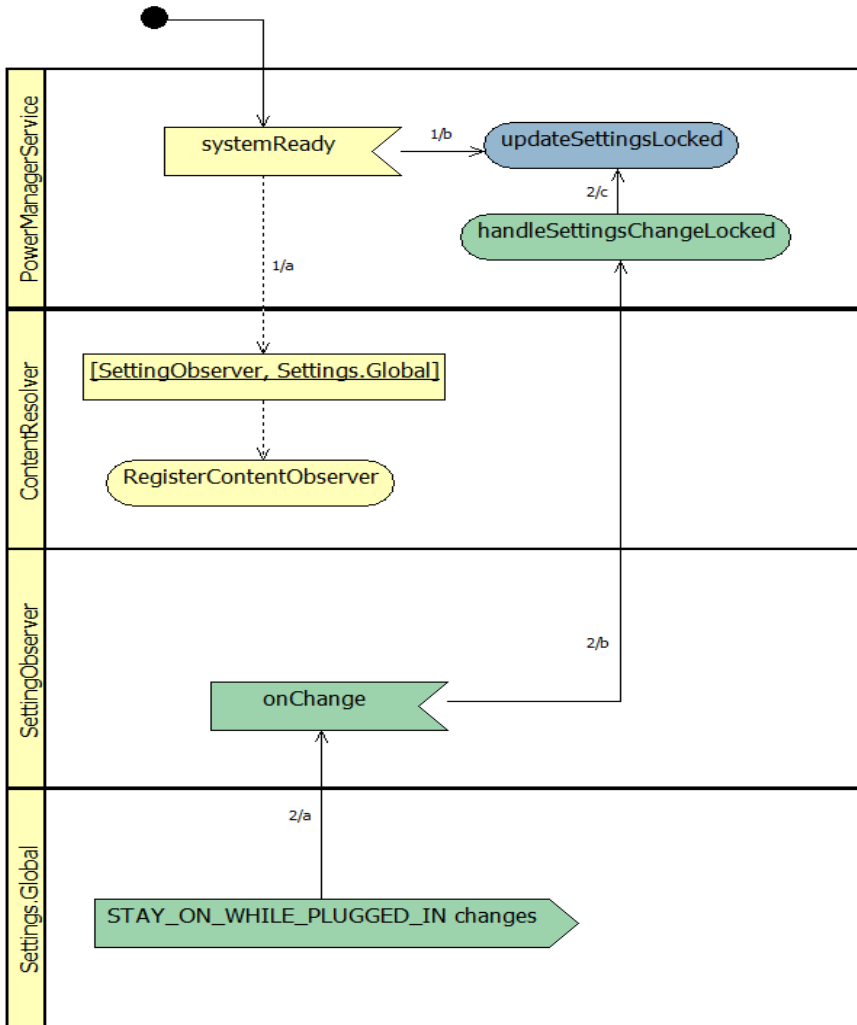
It's not that the logic inside *updateSettingsLocked* is too "complex"; there are a few conditionals, but nothing we can't understand in detail. The real issue is that this function is only an intermediate step, that is, there is further logic applied to the output of this function later on, before the decision to keep the screen on is taken. Therefore, this function is not yet 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.

---

<sup>9</sup> 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.

In a more visual way, using an activity diagram with swimlanes representing the performing objects<sup>10</sup>:

Diagram 1



<sup>10</sup> Actions in yellow execute in the PowerManagerService "systemReady" flow. Actions in green execute asynchronously, later on, when the global setting changes. The action in blue is executed in both execution flows, at different times.

Final 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<sup>11</sup>, 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` value table, 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`.

---

<sup>11</sup> The adb server, somewhat surprisingly, is sharing the entire source code with the adb client normally used on the PC side.

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 indirection as usual, with the help of the `UEventListener` class (see next diagram). Feel free to explore the Android source code to find out more, 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. The state of adb being connected can then be approximated by ANDing together `UsbDeviceManager.mAdbEnabled` and `UsbDeviceManager.UsbHandler.mConnected`<sup>12</sup>. 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:

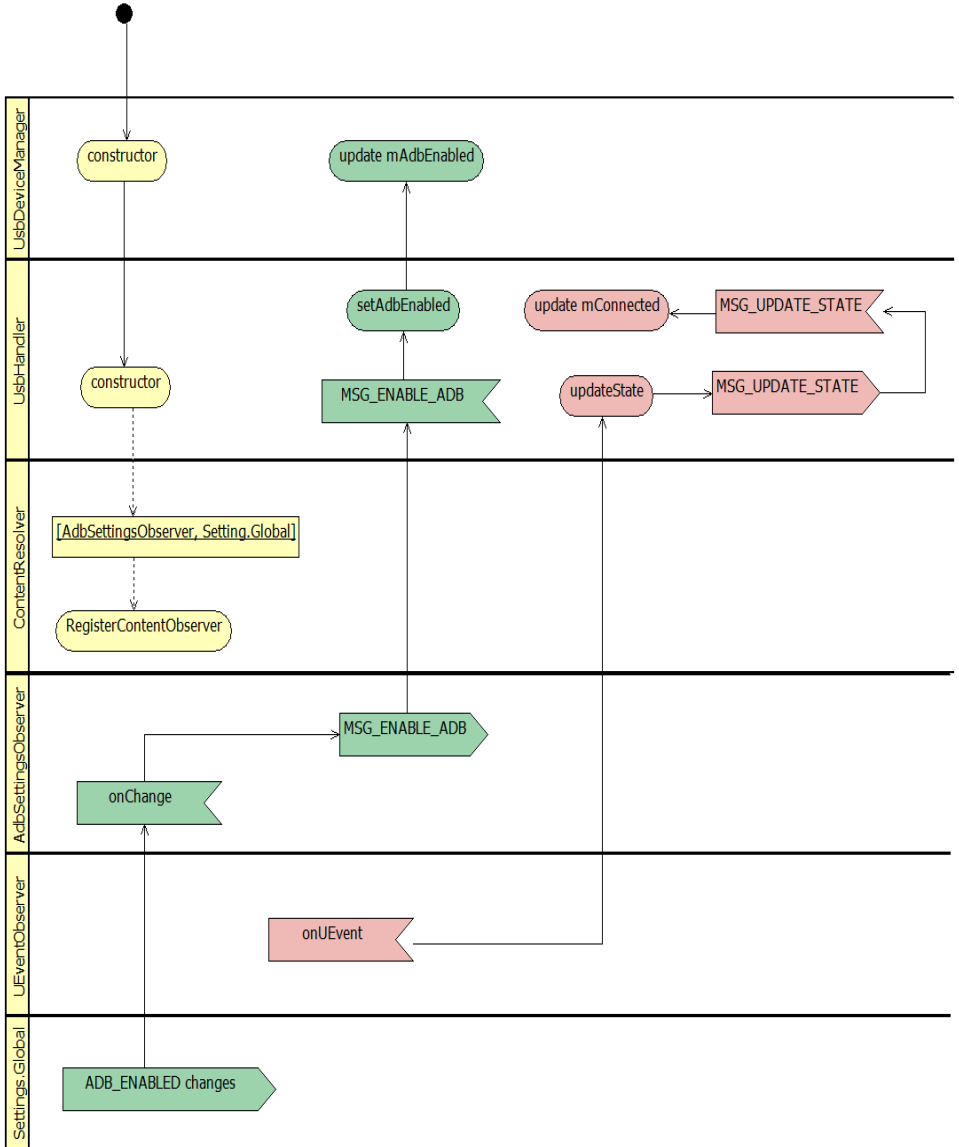
- Device charging: no effect.
- Device in host mode (USB OTG connected): no effect.
- Device with USB debugging turned off: no effect.
- Device with debugging turned on, connected to a computer: screen on.

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. Before that, I'll recap the interactions above in an activity diagram, once again using swimlanes to indicate performing objects, and colors to represent different flows of execution at different times.

---

<sup>12</sup> 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, my workarounds and then a different / better design in the next chapter.

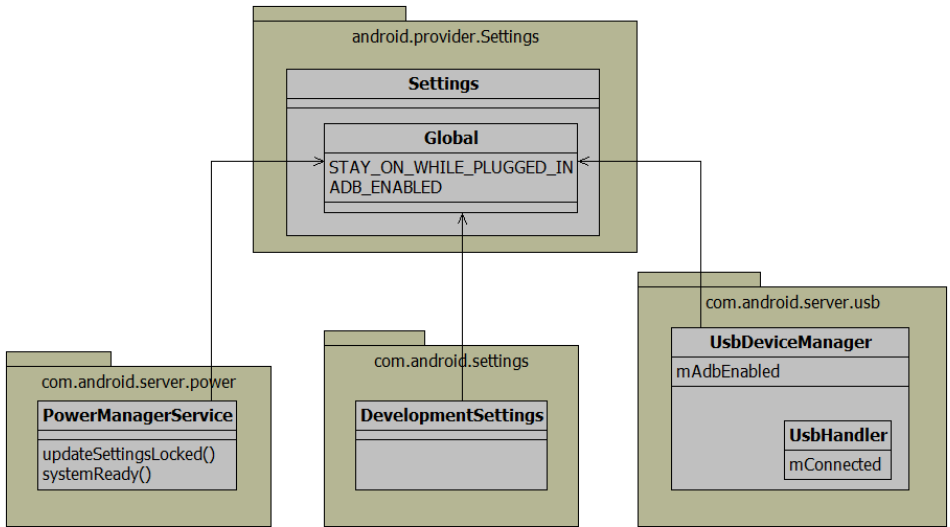
Diagram 2



## A map of the problem

Here is a simplified map of what we discovered so far<sup>13</sup>:

Diagram 3



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

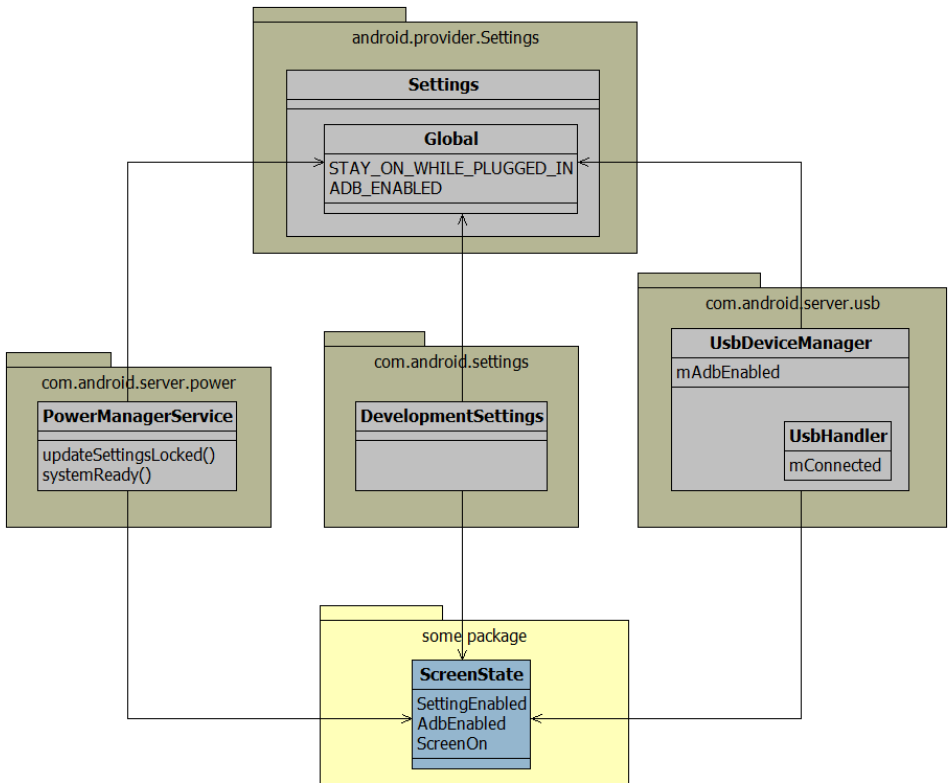
The data members, functions and inner classes that I've represented here are (so far) the most likely candidates to be advised. We'll learn more as we move from understanding the existing code to designing our own aspect-oriented solution, which is the subject of the next chapter.

<sup>13</sup> I'm using packages, not components, because technically the power manager service and the usb device manager are independently compiled into JAR files, but then merged into a single component (the system service) at compile time.

# Chapter 3: Doing it with Aspects

I often say that code and diagrams are talking to us [Pescio2006] in various ways, including their shape. Now, the previous picture (structure of the *problem*) immediately suggests a simple (not aspect-oriented) structure for the *solution* as well<sup>14</sup>:

Diagram 4



<sup>14</sup> I usually favor a “dependency go up” drawing style, however here and in the next diagram I want to emphasize the symmetrical nature of problem and solution.

Ignoring the possibility of aspects, but still trying to minimize changes to existing classes, the idea would be quite simple:

- We need to add a new option to the DevelopmentSettings class, say a “stay on while debugging” checkbox. The state of this checkbox must be persisted somewhere, along the lines of the Settings.Global value table. Since we want to minimize changes to the existing code, let’s say we’ll introduce a new ScreenState content provider in some package. The state of this new setting will be stored (and retrieved) in a SettingEnabled field.
- The UsbDeviceManager knows (by ANDing mAdbEnabled and mConnected) when adb can be truly considered “active”. It can store that information in the same content provider, say in the AdbEnabled field. Note that the AdbEnabled field does not need to be persisted, and actually it would be wrong to persist it. It’s being calculated on the fly by the UsbDeviceManager, and doesn’t make any sense to retrieve a value that was persisted some time ago. From an external perspective (see next point) this would be a write-only field.
- The ScreenState content provider could expose a read-only calculated field, again not persisted as it would not make any sense to do so. This field is again obtained by ANDing together our preference (SettingEnabled) and the adb state (AdbEnabled). From an external perspective, this would be a read-only field. Let’s call it ScreenOn, as it really says if the screen must stay on *now*.
- The PowerManagerService listens to ScreenOn and keeps the screen on accordingly.

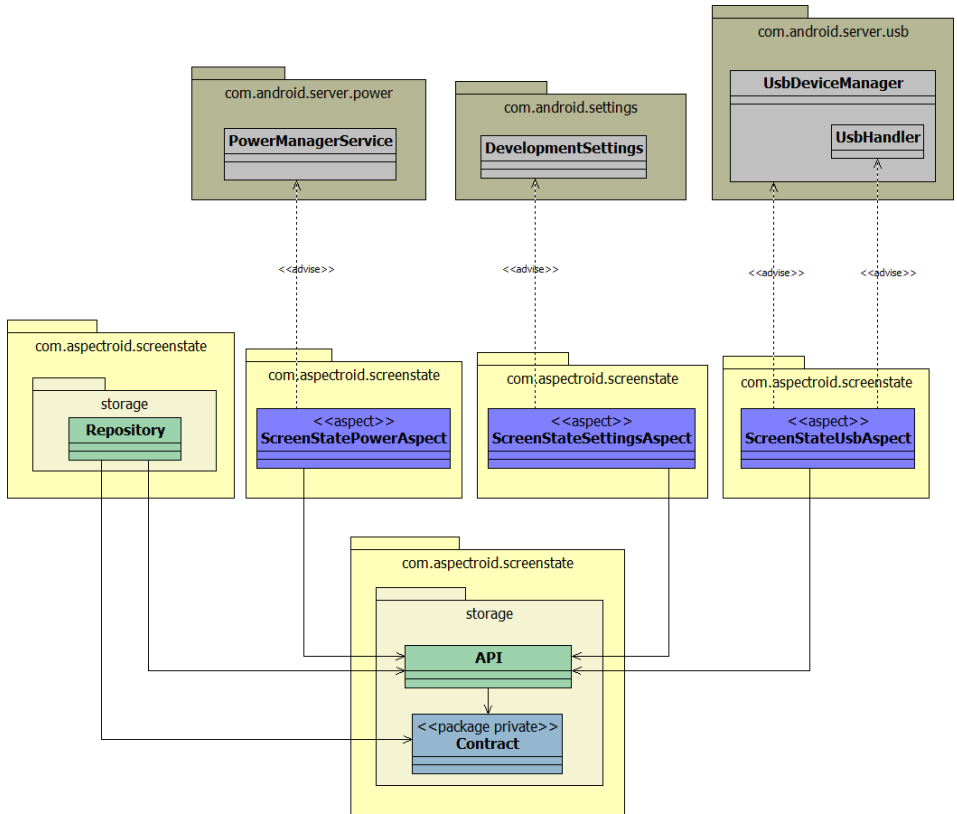
There is nothing intrinsically “bad” about this solution. Indeed, without aspect-oriented technologies, that’s probably what we would end up doing, by tweaking the code of the existing classes.

What would a more aspect-oriented solution look like? In a monolithic app, things would be rather straightforward. The amount of interception required is so small that a single physical aspect could take care of everything. Given that we’re dealing with different components running in different processes, we need to beef up that strategy. We’ll need multiple physical aspects, one for every compilation unit we want to advise.

# Logical architecture

A reasonable, high-level architecture is shown in the next diagram<sup>15</sup>:

Diagram 5



<sup>15</sup> I'm departing from the naming convention for aspects that I've proposed in Episode 1. There is a reason behind that, of course, that I'll explain better in Chapter 4. In short: Episode 1 was about *designing* with aspects; Episode 2 is about *extending existing code* using aspects.

On a conceptual plane, we're dealing with a single feature, which I called `ScreenState` for short and used as a prefix for aspect names. There are several *conceptual* aspects to this feature:

- A power aspect: keeping the screen on
- A settings aspect: providing UI to enable the feature, and enabling / disabling the feature.
- A usb aspect: monitoring the state of the adb process, that we do by observing flags in the usb device manager. This could also be called an "adb aspect", which would be a more problem-oriented way to look at it.
- Maintaining the current state, and storing the only persistent flag (the setting) could be considered a persistence aspect. In practice, one does not need a "physical" aspect to do that: a regular content provider would do<sup>16</sup>.

To make it more visible that these are just slices<sup>17</sup> of a single conceptual feature, I've placed them all inside the same logical package (`com.aspectroid.screenstate`), except the storage aspects and its API, which deserve a quick explanation.

The `Repository` is a content provider which exposes the same logical fields that `ScreenState` had in the non-aspect oriented solution discussed above. As we'll see later on, it's a peculiar content provider, as it's not backed by a `sqlite` DB as usual, and some of the fields are not persisted at all (as explained above, it would be wrong to do so). But still, it's a regular android content provider. Leveraging the content provider notion was quite useful, because we want to reach data from different processes (the `Settings` app runs as a different process; also, the fact that the power manager and the usb manager run in the same process is more of an android implementation detail than a law of nature. Moreover, content providers already have the machinery in place to notify observers when a value change, and

---

<sup>16</sup> This may sound peculiar to those who have been introduced to AOP as a way to deal with concerns like persistence. But as I said in the beginning, `aspectroid` is not about AOP "as usual".

<sup>17</sup> [### hyperplanes](#)

the power manager is definitely interested in knowing that the ScreenOn field (as it was called in the non aspect-oriented discussion above) has changed.

Now, what is usually suggested in android literature (see, for instance, [google's tutorial](#)) is to provide, along with the content provider, a Contract class. The contract is basically a mapping from logical names to physical names. It hides the physical location of the content provider (URI) behind a Java constant. It hides the physical names of the various fields behind Java constants. However, it doesn't hide the fact that you're using a content provider. Classes using the contract are still littered with occurrences of Cursor and ContentValues and so on.

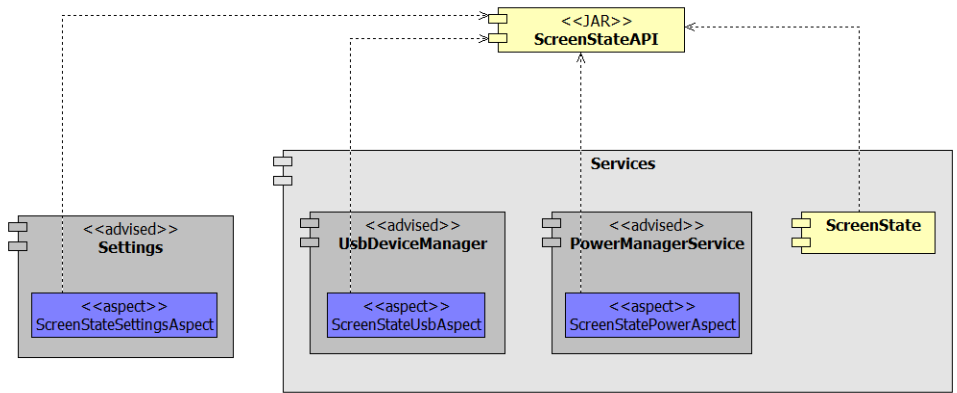
That's ok (sort of) when the content provider is a source of rich and structured data that you may want to query in any possible way. However, we have a much simpler problem here: 3 fields, one read-write-persisted, one read-only with notifications, one write-only. That's it. So, I did in fact provide a Contract class, but that's a package-private class shared between the Repository and a public API class. The API hides more than just the physical names of fields: it hides the content provider as well, exposing a simplified interface to the clients (the aspects). We'll go in details in a while when we'll look at the code, but just to complete the tour of the package-level view:

- Since all the aspects are in the `com.aspectroid.screenstate` package, I had to nest the Repository, Contract and API in a storage package, otherwise Contract being package-private would have been useless.
- I often look at the diagram for symmetries and asymmetries. They both reveal something, and should match our understanding of what we're building. Here we see a vertical symmetry, symmetry between aspects, and an asymmetry with the repository. They're all expected. I'll leave it as the dreaded exercise for the reader to spend a minute or so pondering on this, and on the usefulness of diagrams to reveal those symmetries, easily lost in code.

## Physical architecture

The contract class (in this case, hidden by a public API class) is usually independently compiled in a JAR file, so that it can be included in all the projects that need to interact with the content provider itself. This brings us to the slightly different component view, as opposed to the package view.

Diagram 6



All the components in gray are the existing Android modules that we need to change. I used nesting to emphasize once more that the `UsbDeviceManager` and the `PowerManagerService` gets merged into `Services` later on, but we're specifically advising those two modules. `ScreenState` is a new module, hosting the content provider, which I decided to host inside `Services` as well.

Small detour: I could have hosted that content provider elsewhere; however the `Services` module is being loaded pretty soon when android boots, before regular APKs are loaded (the `Package Manager` and `Activity Manager` are also hosted there), so to avoid any chicken-and-egg problem it's much simpler to host the content provider in the same module.

OK, enough diagrams, time to see some code. Well, almost.

## The process I used

As usual, I have incrementally refined most fine-grained design decisions by writing some code, according to the high-level architecture above. Some decisions were also forced by the tool (AspectJ), or rather by its limitations. Of course, some major decisions were already taken by the existing android source that I had to advise.

A highly iterative process, however, doesn't work well inside the android core. The build time is just too long. Although I have merged AspectJ well enough inside the build process that you can use *mm*<sup>18</sup> to compile individual modules, it still takes too long to inform an iterative *design* process. It also takes too long when you're writing pointcuts and advices: to see if they worked, you actually have to build the entire system, flash your device, etc.

What I've done, in practice, was to first mock the overall structure of the classes I wanted to advise. I created a simple project in eclipse, with placeholder classes for (e.g.) `UsbDeviceManager` and `UsbHandler`, respecting their relationships and the *most relevant* temporal dependencies, but otherwise gutted. I tested my aspects there, both syntactically and behaviorally. Then I moved my code into the platform.

That strategy *almost* worked. In practice, when I first moved the `ScreenState` content provider inside the System services, it failed to run (we'll see later why). It worked fine inside a regular app, but not inside System. Fixing that required some fiddling.

In the end, I didn't keep the mock project alive. Once I had it all working inside the core, I applied a few small tweaking there (changed a few names, etc.), and I didn't bring these changes back to the test project. I wish to say that I would certainly do that in the real world (as opposed to an experiment like aspectroid), but in practice I'm not sure about the long-term value of heavily mocked solutions like these. However, like scaffolding in manufacturing, they do help quite a bit in early stages.

---

<sup>18</sup> For those who never had to suffer the pain ☹ of building android: you can't simply move into a subdirectory, find a makefile and run `make` there. You need to use the *mm* command to build a submodule.

## The code

From a bottom-up perspective, we should start with the contract and the repository and move from there. However, this ebook is mostly about aspects, and those parts are not built with AOP, so I'll start with the UI and the service part instead. I'll mention the API in the code, but it's so simple that we don't need to see the implementation to understand its role. In fact, this specific API exists solely to support those specific clients, so writing the clients is also a way to specify what is required, in concrete terms, from the API.

## Extending the Settings app

We already know from Chapter 2 what we need to do; we can now be slightly more specific:

- Intercept `DevelopmentSettings.onCreate` and add a new checkbox option.
- Intercept `DevelopmentSettings.onResume` and refresh the option value using the value provided by the API.
- When the checkbox preference changes, store the new value using the API.

This is rather straightforward with aspects:

- We add a new data member to the `DevelopmentSettings` activity (the `stayOnPreference` checkbox). This is a regular inter-type declaration in `AspectJ`.
- When `DevelopmentSettings` is created, we create the checkbox preference and add it to the target category (I choose to put it into the `Debugging` category under the `Developer Settings` screen). This requires a pointcut intercepting `onCreate`; an "after" advice will do just fine.
- We subscribe changes to the checkbox state; when it changes, we store the new value using the API.
- When the activity is resumed, we retrieve from the API and set the checkbox state accordingly. This requires a pointcut intercepting `onResume`, and again, an "after" advice is just fine here.

It barely fits on one page, but mostly because of the long import list, `Preference`'s lack of a fluent interface for setters, and Java lack of lambdas.

## The whole ScreenStateSettingsAspect

```
package com.aspectroid.screenstate;

import android.content.Context;
import android.database.Cursor;
import android.preference.CheckBoxPreference;
import android.preference.Preference;
import android.preference.PreferenceCategory;
import android.preference.Preference.OnPreferenceChangeListener;
import com.android.settings.DevelopmentSettings;
import com.aspectroid.screenstate.storage.API;

privileged aspect ScreenStateSettingsAspect
{
    private CheckBoxPreference DevelopmentSettings.stayOnPreference;
    pointcut onCreateExecuted() :
        execution(public void DevelopmentSettings.onCreate(..)) ;
    pointcut onResumeExecuted() :
        execution(public void DevelopmentSettings.onResume(..)) ;

    after(DevelopmentSettings self) : onCreateExecuted() && target(self)
    {
        final Context ctx = self.getActivity();
        PreferenceCategory targetCategory = (PreferenceCategory)self
            .findPreference(DevelopmentSettings.DEBUG_DEBUGGING_CATEGORY_KEY);
        self.stayOnPreference = new CheckBoxPreference(ctx);
        self.stayOnPreference.setKey("aspectroidScreenState");
        self.stayOnPreference.setTitle("Prevent sleeping while debugging");
        self.stayOnPreference.setSummary("only for usb debugging");
        targetCategory.addPreference(self.stayOnPreference);
        self.stayOnPreference.setOnPreferenceChangeListener(
            new OnPreferenceChangeListener()
            {
                public boolean onPreferenceChange(
                    Preference preference, Object newValue)
                {
                    API.persistPreference(ctx, (Boolean) newValue);
                    return true;
                }
            }
        );
    }

    after(DevelopmentSettings self) : onResumeExecuted() && target(self)
    {
        final Context ctx = self.getActivity();
        boolean on = API.retrievePreference(ctx);
        self.stayOnPreference.setChecked(on);
    }
}
```

Although this code is quite simple, and devoid of substantial logic, there are quite a few observations worth sharing:

- The pointcuts are quite robust here. I'm intercepting methods that are part of the lifecycle of the base class, and the logic I'm injecting is bound to make sense in that context. The only way to mess this up is for the `DevelopmentSetting` class to disappear, or to change its nature deeply (like not being a *PreferenceFragment* anymore).
- The API calls take a context as a parameter; most android APIs require a context, and inside my API class I'm calling my content provider, so I need a context.
- I choose to make API a static class. After all, it's a stateless façade over the content provider. A reasonable alternative would be to have a constructor taking the context, and then removing it as a parameter from the methods. Since my callers call 1 or 2 API methods, that would have made the callers longer, and the code above wouldn't have fit in one page anymore 😊.
- I had to use a privileged aspect because, of all things, `DevelopmentSettings.DEBUG_DEBUGGING_CATEGORY_KEY` is private. Honestly, although using a privileged aspect is often flagged as a "bad smell" in academic AOP literature, when you don't control the advised code it's almost impossible to do anything useful without it. We should leave this good / bad mentality behind and move toward a deeper understanding of consequences. Sure, it's possible that that constant will disappear in a later version. It's quite unlikely. We'll see when I get around writing Chapter 5, using the latest android version at that time.
- I didn't deal with localization. This is partially for simplicity / focus, because the real "core" of the code is about intercepting the lifecycle. But there is also a deeper problem here. Localization in android is handled through XML files, which I cannot reach from AspectJ. I'll discuss these issues in Chapter 4.

- Along the same lines, I had to modify the makefile, adding both the AspectJ runtime library and the screenstateapi JAR to the list of static java libraries to be used (snippet below). This is the only change I had to apply to an *existing* file, because once again .mk files are out of reach for AspectJ.

Changes to the Android.mk file

```
LOCAL_STATIC_JAVA_LIBRARIES := android-support-v4 android-  
support-v13 jsr305 aspectjrt screenstateapi
```

This change will have to be ported manually to newer android versions. Too bad.

## Extending the UsbDeviceManager

We're now ready to move into the core services and extend the usb device manager to publish the adb state. I honestly expected this to be rather straightforward, until I met a couple of limitations in AspectJ that required a more convoluted code than I expected. In fact, I evolved this portion of code quite a bit from the first iteration; it's a rather interesting story, so I'll show you my first approach, an intermediate step and then the final result.

Let's first recap our job from Chapter 1:

*The state of adb being connected can be approximated by `UsbDeviceManager.mAdbEnabled` && `UsbDeviceManager.UsbHandler.mConnected`.*

In theory, that would be trivial. We just need a privileged aspect to access `UsbDeviceManager.mAdbEnabled`. AspectJ can place a pointcut on a data member being set, and that would do it. We need to do the same for `UsbDeviceManager.UsbHandler.mConnected`. The advice logic can be the same for both: calling a function reading both values and updating the content provider through the API. The advice should call `proceed()` first so that the common function can just read the (updated) data members. That function needs a context to call the API, but the outer object (instance of `UsbDeviceManager`) conveniently has a context in a data member. Alas, it didn't work.

- The "common" function needed to access both the outer object and the inner object. No problems when you intercept `mAdbEnabled`: you have the outer and you can get the inner. But when you intercept `mConnected`, you have the inner and can't get the outer. This was [discussed on the AspectJ mailing list](#) back in 2013 and it's still true today.
- I also had some hard to justify problems reading the inner class field (compiled fine, ran erratically).

As we often do, I tried to "fix" this idea with a couple of workarounds. Let's see the code:

## The firstScreenStateUsbAspect

```
package com.aspectroid.screenstate;

import com.android.server.usb.UsbDeviceManager;
import com.aspectroid.screenstate.storage.API;

privileged aspect ScreenStateUsbAspect
{
    UsbDeviceManager outerThis;

    before(UsbDeviceManager self):
    execution(UsbDeviceManager.new(..)) && this(self)
    {
        outerThis = self;
    }

    pointcut adbSet() :
        set(private boolean UsbDeviceManager.mAdbEnabled);

    void around(UsbDeviceManager self) : adbSet() && target(self)
    {
        proceed(self);
        checkAdbOn(self);
    }

    void checkAdbOn(UsbDeviceManager self)
    {
        boolean isOn = self.mAdbEnabled && self.mHandler != null &&
            self.mHandler.isConnected();
        API.storeAdbState(self.mContext, isOn);
    }

    boolean UsbDeviceManager.UsbHandler.isConnected()
    {
        return this.mConnected;
    }

    pointcut connectedSet() :
        set(private boolean UsbDeviceManager.UsbHandler.mConnected);

    void around(UsbDeviceManager.UsbHandler self) : connectedSet() &&
    target(self)
    {
        proceed(self);
        checkAdbOn(outerThis);
    }
}
```

I am intercepting the construction of the outer object, and saving it inside the aspect. That was a suggestion from the AspectJ mailing list, and it applies nicely here because the outer object is a singleton (and the inner as well), so I can store the value directly in the aspect, without worrying about correlating the outer and the inner object. Problem solved.

I then set a pointcut when `UsbDeviceManager.mAdbEnabled` is being set. In the corresponding advice, I let the set call take place, and then call `checkAdbOn`. That's an aspect method, so I pass the outer object (self in this case) explicitly.

I do the same for `UsbDeviceManager.UsbHandler.mConnected`, but this time I don't pass self: I pass `outerThis` because that's the outer object, while self would be the inner object.

The `checkAdbOn` is trivial, just and-ing a couple of values together (I also need to handle the case where `mHandler` has not been created yet). Then I just call the right API for the content provider and store the adb state. Once again, I need a context, which I can get from the manager itself.

In theory, the introduction of `isConnected` through an inter-type declaration is not needed. I could just use `mConnected` instead. The code compiles just fine. However, I had random run-time failures, which I can't explain well. It seems that (in the version I used) AspectJ and inner classes don't get along too well.

This first version worked, and in the beginning I was basically OK with it, mostly because it took me a while to work around the issues with the inner class, and I suppose you know the feeling when you finally get something working.

Upon reflection, however, that code is rather crappy:

- I am fighting the tool. That's something I try to avoid as much as possible. Even though winning a fight with a tool can make a programmer feel good, it's always an indication that you're painting yourself in a corner.
- It's too fragile, compared for instance with the Settings aspect. Sure, it's just intercepting a couple of data members, but I won't count on `mConnected` staying where it is. It belongs to the outer class more than to the handler, and it seems like the entire `UsbManager` code has been *written*, not *designed*. It's reasonable that those two data members will still be there on a future release, but I won't count on one being in the outer object and the other in the inner object.

If we want to improve *significantly*, we need to understand the root cause of the problem. Here, I concluded, the root cause was my unconscious<sup>19</sup> preference to avoid redundancy. I wanted to reuse the existing data members: `mContext`, `mAdbEnabled`, `mConnected` are just there, waiting to be used. So I wrote `checkAdbOn()` expecting to find data where they already are. I didn't consider the option of cloning data in my aspect, which in this case is easy because we're dealing with a singleton.

However, if I accept to clone data, I only need to access the pointcut parameters (see the next snippet). No need to access either the inner or outer object. So instead of toying around with the code above and making small-scale improvements, I tried an intermediate step (because I could "see" a potential that will become clear in the final version).

---

<sup>19</sup> In [Alexander1964], Christopher Alexander (the father of patterns who also inspired the notion of coupling and cohesion exactly with that book) talks about the self-conscious and the unselfconscious design process, and how the unselfconscious process can generate beautiful works within the frame of a long tradition, but struggles to adapt to a changing context. Being more self-conscious about our design style, approach and biases always helps.

## The secondScreenStateUsbAspect

```
package com.aspectroid.screenstate;

import com.android.server.usb.UsbDeviceManager;
import android.content.Context;
import com.aspectroid.screenstate.storage.API;

privileged aspect ScreenStateUsbAspect
{
    boolean adbEnabled = false;
    boolean adbConnected = false;
    Context context;

    before(Context ctx): execution(UsbDeviceManager.new(Context))
    && args( ctx )
    {
        context = ctx;
    }

    pointcut adbSet() :
        set(private boolean UsbDeviceManager.mAdbEnabled);

    after(boolean newVal) : adbSet() && args(newVal)
    {
        adbEnabled = newVal;
        checkAdbOn();
    }

    pointcut connectedSet() :
        set(private boolean UsbDeviceManager.UsbHandler.mConnected);

    after(boolean newVal) : connectedSet() && args(newVal)
    {
        adbConnected = newVal;
        checkAdbOn();
    }

    void checkAdbOn()
    {
        boolean isOn = adbEnabled && adbConnected;
        API.storeAdbState(context, isOn);
    }
}
```

This code is way more straightforward and does not rely on any kind of trick. It's not fighting AspectJ. It's saving the context in the constructor and reading the new value as a parameter passed to the *set* advice.

However, it is not significantly more robust than the first. While `UsbDeviceManager` taking a `Context` as a parameter in its constructor is probably quite stable (you can't do much in android without a context), the aspect is still plagued by very specific *set* pointcuts on the inner and outer class members.

However, and this is what I was really after, now that I don't have to touch the inner and outer "this", I can write a more generic version of the pointcuts. What I want to say is something like this:

*"Intercept `mAdbEnabled` and `mConnected` within the lexical scope of `UsbDeviceManager`; I don't care where they are exactly (inner class, outer class)".*

The notion of lexical scope is worth an explanation and a contrast with the notion of control flow, which I used in Episode 1. I don't want to capture any data member called `mAdbEnabled` or `mConnected` *anywhere* in the entire module code. I want them to be inside the `UsbDeviceManager` or an inner class: that is, in the lexical scope of `UsbDeviceManager` as a compilation unit. The notion of control flow doesn't help here, because a call to the inner `UsbHandler` does not happen within the control flow of the outer `UsbDeviceManager`. We're after a static, compile-time notion here, not after a run-time notion<sup>20</sup>.

Fortunately, AspectJ helps here with the "within" static scoping pointcut designator, as it's called. It's easier to see it in action with the final code for the usb aspect:

---

<sup>20</sup> In some cases, the two notions give the same practical result. In that case, it's better to go with the static / lexical notion, because AspectJ can handle that more efficiently. A cflow pointcut usually has some measurable run-time overhead.

## The final ScreenStateUsbAspect

```
package com.aspectroid.screenstate;

import com.android.server.usb.UsbDeviceManager;
import android.content.Context;
import com.aspectroid.screenstate.storage.API;

aspect ScreenStateUsbAspect
{
    boolean adbEnabled = false;
    boolean adbConnected = false;
    Context context;

    before(Context ctx): execution(UsbDeviceManager.new(Context)) &&
args( ctx )
    {
        context = ctx;
    }

    pointcut adbSet() : within(UsbDeviceManager) &&
set(boolean *.mAdbEnabled);

    after(boolean newVal) : adbSet() && args(newVal)
    {
        adbEnabled = newVal;
        checkAdbOn();
    }

    pointcut connectedSet() : within(UsbDeviceManager) &&
set(boolean *.mConnected);

    after(boolean newVal) : connectedSet() && args(newVal)
    {
        adbConnected = newVal;
        checkAdbOn();
    }

    void checkAdbOn()
    {
        boolean isOn = adbEnabled && adbConnected;
        API.storeAdbState(context, isOn);
    }
}
```

The secret sauce here is:

```
within(UsbDeviceManager) && set(boolean *.mAdbEnabled);
```

Which means: I don't care about the class where mAdbEnabled appears, provided that it's in the lexical scope of UsbDeviceManager. Ditto for mConnected.

Cherry on top: I don't need this aspect to be privileged anymore. I'm just reading parameters. I'm not using any private or protected member (actually, no members at all).

The final code is simpler, shorter and more robust than my first attempt. In retrospective, the struggle with the inner class limitations in AspectJ was positive, as it forced me to re-evaluate my strategy and my bias against replicating data. I don't want to say it's always better to do so – I deeply reject the idea that we can form a “principle” based on a single lucky case and some rationalization. But it's certainly an option I need to keep myself open to when using aspects.

To conclude: I had to modify the makefile as well, along the same lines of the settings app. Here, the usb manager module didn't have any dependency on static java libraries, so I added a full line:

Changes to the Android.mk file

```
LOCAL_STATIC_JAVA_LIBRARIES := aspectjrt screenstateapi
```

This change will have to be ported manually to any new android version.

## Extending the PowerManager

In Chapter 2, we chased down the usage of `STAY_ON_WHILE_PLUGGED_IN` inside the `PowerManagerService`, only to find that it's being read inside `updateSettingsLocked`, where it's being stored as `mStayOnWhilePluggedInSetting`. That function also updates an `mDirty` flag mask, telling the service that "something has changed" and it should update the peripherals it's controlling.

We left with the idea that `updateSettingsLocked` didn't look too promising as a pointcut. The function is storing the choice to keep the screen on while plugged in in (guess what) `mStayOnWhilePluggedInSetting`.

We could just use an `around()` advice and "or" that value with what we get from our API when we ask if the screen must be kept on. However, that doesn't sound entirely right: I would be overwriting a *setting* saying when to keep the screen on, not a *decision* to keep the screen on *now*. Before we go ahead and do so, it's better to look around and see how that data member is being used. Lucky enough, only one function is reading its value:

### PowerManagerService snippet

```
private void updateStayOnLocked(int dirty)
{
    if( (dirty & (DIRTY_BATTERY_STATE | DIRTY_SETTINGS)) != 0 )
    {
        final boolean wasStayOn = mStayOn;
        if( mStayOnWhilePluggedInSetting != 0 &&
            !isMaximumScreenOffTimeoutFromDeviceAdminEnforcedLocked() )
        {
            mStayOn =
                mBatteryManagerInternal.isPowered(mStayOnWhilePluggedInSetting);
        }
        else
        {
            mStayOn = false;
        }

        if( mStayOn != wasStayOn )
        {
            mDirty |= DIRTY_STAY_ON;
        }
    }
}
```

So, overwriting `mStayOnWhilePluggedInSetting` wouldn't have been the best choice; it is getting AND-ed with another value, and I don't really want my setting to be influenced the same way. On the bright side, `mStayOn` looks really good. It's not telling the `PowerManagerService` about a *setting*, it's the final decision to keep the screen on. Also the `mDirty` flag is being manipulated in a meaningful way here: if *now* we need to keep the screen on, we say so. The "now" part is mostly to avoid some logic later on (if the screen was already on, there is nothing to change). I'll keep this optimization when adding my logic too.

So, `updateStayOnLocked` looks like a pretty good candidate for an `around()` advice. We need to call our API from the advice, so as usual we need a context, but conveniently the `PowerManagerService` has a context in a data member. We need a privileged aspect this time, because we'll be accessing private fields anyway, so I can just that context member as well.

Before we forget: we also need to subscribe changes coming from our API, and make sure they trigger the same update chain that is triggered (for instance) when the "keep on while debugging" field is updated. That's simple; we just need to register the common observer (`mSettingsObserver`, see Chapter 2) as an observer for our own "should keep screen on" value. We can do so after `systemReady` has been called and all the other subscriptions are in place.

Overall, the entire code is quite short and simple:

## TheScreenStatePowerAspect

```
package com.aspectroid.screenstate;

import com.android.server.power.PowerManagerService;
import com.aspectroid.screenstate.storage.API;

privileged aspect ScreenStatePowerAspect
{
    pointcut settingsUpdated() :
    execution(private void PowerManagerService.updateStayOnLocked(int));

    void around(PowerManagerService self, int dirty) :
    settingsUpdated() && args(dirty) && target(self)
    {
        proceed(self, dirty);
        boolean keepOn = API.shouldKeepScreenOn(self.mContext);
        if( keepOn && !self.mStayOn )
        {
            self.mStayOn = true;
            self.mDirty |= PowerManagerService.DIRTY_STAY_ON;
        }
    }

    pointcut systemReadyExecuted() :
    execution( public void PowerManagerService.systemReady(..));

    after(PowerManagerService self) : systemReadyExecuted() && target(self)
    {
        API.registerScreenOnObserver(self.mContext, self.mSettingsObserver);
    }
}
```

Is this aspect robust<sup>21</sup>? Honestly, not as much as the former two. It's intercepting a private function, reading a private field (the context), leveraging a private observer, updating a couple of private fields. However, if we try to go further down the rabbit hole and see how `mStayOn` and `mDirty` are being read, looking for a better place to advise, we find out that their usage is spread out in several different places. Overall, `updateStayOnLocked` still looks like the best candidate.

---

<sup>21</sup> I don't want my concerns with the robustness of the aspects give you the impression that this approach is somewhat "worse" than patching the existing code as people would normally do. If the `updateStayOnLocked` member were to disappear on a later version, my aspect would break but so would any local patch applied there.

We're not done yet: as in the previous cases, we need to update the makefile to add the AspectJ runtime library and the API jar file. However, while the UsbManager had its own makefile, the PowerManager doesn't. There is only one makefile for all the modules in the "core" folder, so that's where we'll have to add our magic line:

Changes to the Android.mk file

```
LOCAL_STATIC_JAVA_LIBRARIES := aspectjrt screenstateapi
```

Just like before, this change will have to be ported manually to any new android version.

## The ScreenState Contract and API

The aspect-oriented part is over. We're now simply inside traditional android / java programming. I've already explained my choice to have a private contract and a public API for the ScreenState content provider, so we can simply have a quick look at those. There isn't much to learn here, except how to put this library into the android core itself.

### The Contract class

```
package com.aspectroid.screenstate.storage;

import android.net.Uri;

class Contract
{
    static final String AUTHORITY = "com.aspectroid.screenstate";
    static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY);

    static final class SettingEnabled
    {
        static final String NAME = "settingEnabled";
        static final Uri CONTENT_URI =
            Uri.withAppendedPath(Contract.CONTENT_URI, NAME);
        static final String COLUMN_NAME = "value";
    }

    static final class AdbEnabled
    {
        static final String NAME = "adbEnabled";
        static final Uri CONTENT_URI =
            Uri.withAppendedPath(Contract.CONTENT_URI, NAME);
        static final String COLUMN_NAME = "value";
    }

    static final class ScreenOn
    {
        static final String NAME = "screenOn";
        static final Uri CONTENT_URI =
            Uri.withAppendedPath(Contract.CONTENT_URI, NAME);
        static final String COLUMN_NAME = "value";
    }
}
```

I choose to give each field a separate URI, instead of making them columns of a single URI / table. As we have discussed, one of those field will be R/W and persistent (SettingEnabled), one will be W/only and not persisted (AdbEnabled) and one will be R/only and not persisted (ScreenOn). Making those largely different fields “columns” of a single virtual table simply hurt my sense of aesthetics (yes, sometimes I take design decisions based purely on aesthetics).

The API is using the familiar android classes to read and write from a content provider, while enforcing the restricted access as above. The only “logic” here is some conversion from int to boolean and vice-versa. This class is trivial but won’t fit on a book page. I won’t even try.

#### The API class

```
package com.aspectroid.screenstate.storage;

import android.content.Context;
import android.database.Cursor;
import android.content.ContentValues;
import android.database.ContentObserver;
import android.os.UserHandle;

public class API
{
    public static boolean shouldKeepScreenOn(Context ctx)
    {
        Cursor c = ctx.getContentResolver().query(
            Contract.ScreenOn.CONTENT_URI, null, null, null, null);
        c.moveToFirst();
        int n = c.getInt(0);
        c.close();
        return n != 0;
    }

    public static void registerScreenOnObserver(Context ctx,
        ContentObserver obs)
    {
        ctx.getContentResolver().registerContentObserver(
            Contract.ScreenOn.CONTENT_URI, false, obs, UserHandle.USER_ALL);
    }

    public static void storeAdbState(Context ctx, boolean isOn)
    {
        ContentValues cv = new ContentValues();
        int n = isOn ? 1 : 0;
        cv.put(Contract.AdbEnabled.COLUMN_NAME, n);
    }
}
```

```

    ctx.getContentResolver().update(
        Contract.AdbEnabled.CONTENT_URI, cv, null, null);
}

public static void persistPreference(Context ctx, boolean pref)
{
    int s = pref ? 1 : 0;
    ContentValues cv = new ContentValues();
    cv.put(Contract.SettingEnabled.COLUMN_NAME, s);
    ctx.getContentResolver().update(
        Contract.SettingEnabled.CONTENT_URI, cv, null, null);
}

public static boolean retrievePreference(Context ctx)
{
    Cursor c = ctx.getContentResolver().query(
        Contract.SettingEnabled.CONTENT_URI, null, null, null, null);
    c.moveToFirst();
    int n = c.getInt(0);
    c.close();
    return n != 0;
}
}

```

## Adding the library to the build

Most of the usage of the API library (except for the Settings app) is from the system server modules, so I decided to add it there. The way the android build works, you don't have to copy the JAR around anyway.

I've just created a *screenstateapi* folder under *frameworks/base/services/*, under which we'll place the API and Contract source code with the usual nesting of Java folders, and added an *Android.mk* file there. This entire folder (*screenstateapi*) will need to be ported to any new android release, but that's ok: it's new stuff we're *adding*, not something we're *changing*.

The API *Android.mk* file

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := screenstateapi

LOCAL_SRC_FILES += \
    $(call all-java-files-under,java) \

include $(BUILD_STATIC_JAVA_LIBRARY)
```

If you've never seen an android makefile, that might not make much sense, but it's really just a boilerplate makefile that you use whenever you want to build a static java library (as the last line is implying).

With this done, we're only left with the content provider itself.

## Adding the ScreenState content provider

Paradoxically enough, this might be the most controversial part of my code. While it's probably safe to assume that most people have no experience using AspectJ inside the android core, quite a few of you have certainly implemented a content provider before, and they know "how it's done". There are infinite blog posts and samples and whatnot, and they all do the same thing: create a sqlite database (with its baggage of SQLiteOpenHelper and stuff) and then implement a lengthy, custom class derived from ContentProvider with a switch-case inside, along the lines of the [official android tutorial](#). Been there, done that, and I didn't like the code too much.

Now, if you think about it, a sqlite database is really a poor fit for the ScreenState content provider. Three integer fields, two of which we don't really want to persist. Down to one. Do I need a database for one integer field? Do I have to write all that custom code to persist one field? Of course, a database is not strictly required. Indeed, the android tutorial says "*a common way to store this type of data is in an SQLite database, but you can use any type of persistent storage*". But in practice, the ContentProvider interface is so slanted toward an SQL implementation that all you find around is sqlite-based.

Don't get me wrong; in a number of cases, a sqlite database is a good fit. If you have structured data, connected by relationships, and you want to expose a rich data model with a flexible query mechanism, implementing it on top of sqlite is just fine. However, some apps (and systems of apps) share simple contents, often marginally structured, and more often than not transient in nature. This is really common when you interface with hardware, for instance. You may want to expose a soft real-time snapshot through a content provider, but you don't want the overhead of sqlite and you definitely are not exposing structured data.

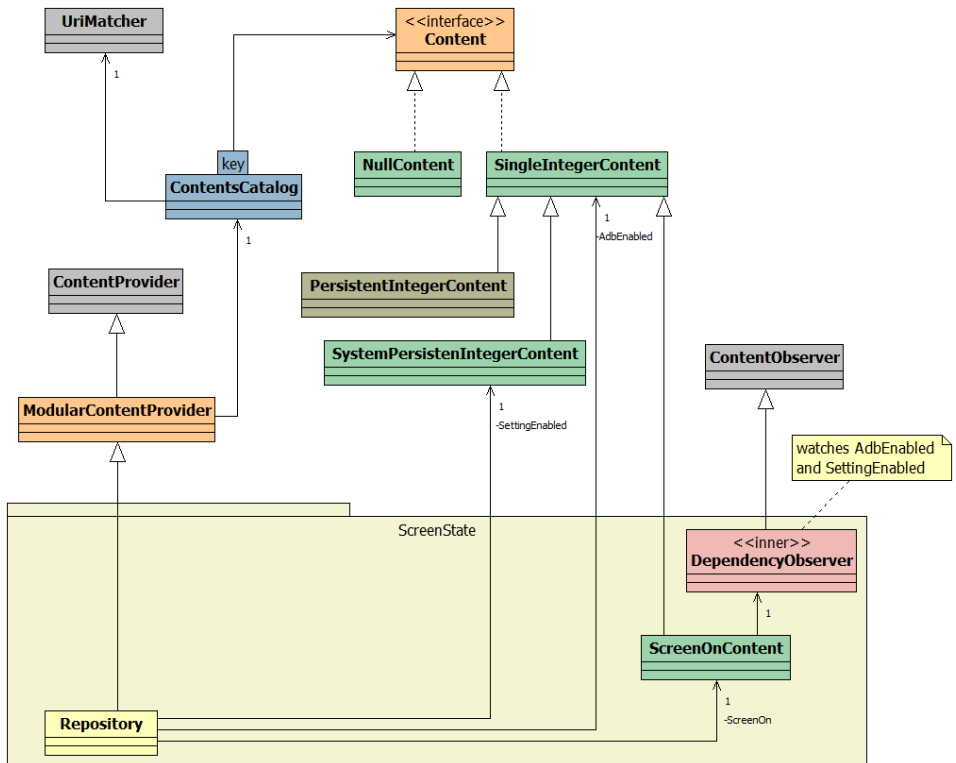
There is more. The ContentProvider's interface, by nature, will lead to long classes and switch cases etc. The problem is, as usual, that it's a fake-OO veneer over procedural thinking. In fact, in the android library we have a notion of ContentProvider, but we don't even have a notion of *Content* 😊.

Faced with the ugliness of the code I usually see, and with the fact that I often needed to expose simple, transient data, over time I built a small set of classes to add some OO-goodness to this side of android.

- I introduced a notion of Content.
- Content does not have to be persistent.
- The Content decides where to store itself.
- The ModularContentProvide is populated by adding Contents.
- The purpose of a Repository class is mostly to designate the contents, that is, to act as a Builder.
- Occasionally, you may need project-specific contents.

Here is a diagram of the involved classes:

Diagram 7



The gray classes are from the android library. The classes outside the ScreenState package are general-purpose; PersistentIntegerContent is not actually used in this project, and I kept it around just because it's mentioned later on in this ebook. The only project-dependent classes are the Repository and the ScreenOnContent, which actually contains DependencyObserver<sup>22</sup> as an inner class.

A discussion of the entire code would be probably out of place here, and I'm also using a small subset of the library in this project. That subset is however included in the source code distributed on [aspectroid.com](http://aspectroid.com), so you can take a look for yourself. Here, I'll limit myself to a few general notes, and then I'll focus on the project-specific code and about a few project-specific issues that I had to deal with.

---

<sup>22</sup> I'm totally against classes ending in -er (see [Pescio2006]), which usually end up supporting procedural thinking just like the ContentProvider does. However, here I kept 2, ModularContentProvider and DependencyObserver, mostly for consistency with the base class.

## The Repository

You can get a gist of what it's like to use the modular content provider by looking at the entire code for the Repository, which is actually quite small:

### The Repository class

```
package com.aspectroid.screenstate.storage;

import com.aspectroid.cp.ContentsCatalog;
import com.aspectroid.cp.ModularContentProvider;
import com.aspectroid.cp.SystemPersistentIntegerContent;
import com.aspectroid.cp.SingleIntegerContent;

import android.content.ContentResolver;
import android.content.Context;

public class Repository extends ModularContentProvider
{
    @Override
    protected boolean addContents(ContentsCatalog catalog, Context ctx)
    {
        ContentResolver contentResolver = ctx.getContentResolver();

        SystemPersistentIntegerContent ec =
            new SystemPersistentIntegerContent(
                Contract.SettingEnabled.CONTENT_URI,
                Contract.SettingEnabled.COLUMN_NAME, ctx);
        catalog.addContents(Contract.AUTHORITY,
            Contract.SettingEnabled.NAME, ec);

        SingleIntegerContent adbState = new SingleIntegerContent(
            Contract.AdbEnabled.CONTENT_URI,
            Contract.AdbEnabled.COLUMN_NAME, contentResolver);
        catalog.addContents(Contract.AUTHORITY,
            Contract.AdbEnabled.NAME, adbState);

        ScreenOnContents soc = new ScreenOnContents(contentResolver);
        catalog.addContents(Contract.AUTHORITY, Contract.ScreenOn.NAME, soc);

        return true;
    }
}
```

If you have ever created a ContentProvider, you can see the difference – I'm merely building up the Contents here; there is no other logic in place.

## The ScreenOnContents class

```
package com.aspectroid.screenstate.storage;

import com.aspectroid.cp.SingleIntegerContent;
import android.content.ContentResolver;
import android.database.ContentObserver;
import android.database.Cursor;
import android.os.Handler;

class ScreenOnContents extends SingleIntegerContent
{
    public ScreenOnContents(ContentResolver contentResolver)
    {
        super(Contract.ScreenOn.CONTENT_URI,
            Contract.ScreenOn.COLUMN_NAME, contentResolver);
        DependencyObserver depObs =
            new DependencyObserver(new Handler(), contentResolver);
        contentResolver.registerContentObserver(
            Contract.SettingEnabled.CONTENT_URI, false, depObs);
        contentResolver.registerContentObserver(
            Contract.AdbEnabled.CONTENT_URI, false, depObs);
    }

    private class DependencyObserver extends ContentObserver
    {
        public DependencyObserver(Handler handler,
            ContentResolver contentResolver)
        {
            super(handler);
            resolver = contentResolver;
        }

        @Override
        public void onChange(boolean selfChange)
        {
            Cursor c = resolver.query(
                Contract.AdbEnabled.CONTENT_URI, null, null, null, null);
            c.moveToFirst();
            int adbEnabled = c.getInt(0);
            c.close();
            c = resolver.query(
                Contract.SettingEnabled.CONTENT_URI, null, null, null, null);
            c.moveToFirst();
            int settingEnabled = c.getInt(0);
            c.close();
            setValue(adbEnabled * settingEnabled);
        }

        private ContentResolver resolver;
    }
}
```

ScreenOnContent is a special content, as it's AND-ing together two other fields from the same Repository. Instead of using my own observer chain I have just leveraged the built-in notion of ContentObserver. So ScreenOnContent is basically a transient SingleIntegerContent which value is updated whenever one of the two observed fields changes.

That's it, I could say. However, as I have hinted at, the real story was a little more troubled than it seems by looking at the code, and it's something worth sharing.

As I mentioned, I prototyped most of this code *outside the core*. While doing so, I leveraged a class I already had (PersistentIntegerContent), which could save an integer without the entire machinery of a sqlite database. It's based on a preference file instead, and basically answers the frequent question "can I store the contents of my content provider into a preference / xml file instead of using sqlite" in 25 lines of code + the usual import list. All was good.

Then, I moved the project inside the core and bang, it failed. It failed because, of course, the preference file machinery is built to run within an app<sup>23</sup>, and the system server is not an app. Enter SystemPersistentIntegerContent, which is basically its alter-ego with the superpowers required to run inside the code. How does it do that? I could have written some code to deal with files and formats and whatnot, but that would have been a lot of work.

Whenever I face a "that's a lot of work for a rather small accomplishment" situation, I tend to look back and see which chain of decisions put me there. I do so because even though we all like to think that we take our decisions based on ineffable logic, we often mix in some bias, some quick judgment, some gut feeling, or we simply take the decision at the "wrong" time, when our understanding of the forces is too limited to choose best.

Now, in hindsight it's rather obvious that I'm facing this issue now because I choose not to (publicly) store all my data inside Settings.Global, ignoring also the difference between persistent and non-persistent contents. That would have made my life so easier. So why exactly didn't I like that option?

---

<sup>23</sup> Yeah, I consider this an oversight.

The way it's written, with explicit public constants for each setting, Settings.Global is an inherently unstable API class. It mimics a content provider (you also have the ability to get a URI for a specific content, so that you can use it with a ContentObserver) but it breaks the familiar idea that inter-app data is shared through *specialized* content providers. That's part of the reasons I wasn't fond of the idea to extend it. But it was not the real issue. Looking deeper into my dislike for Settings.Global, I realized that what I really didn't like was the way it was used, that is, as the *public container* of those data.

In theory, there is nothing wrong with the idea of a system-wide table of data. That, however, doesn't mean that every client application should know that your data (or part of it) is coming from that table, any more than it should know that it's coming from a specific sqlite database or xml file. Settings.Global should be a *private* storage mechanism for system-wide data, which should then be exposed through regular content providers. That would remove the instability, and would also remove it from the SDK API, moving it where it belongs (core development).

To explain my point better: one of the predefined keys (constants) in [Settings.Global](#) looks like this:

String	USE_GOOGLE_MAIL	If this setting is set (to anything), then all references to Gmail on the device must change to Google Mail
--------	-----------------	---

Does that belong in the public API of the AOSP? Of course not. It's a google-specific thing. Gmail is part of the google apps, and those are not part of the android core (and are actually licensed separately to OEMs). That constant is there only because ..., well, *because google*. I don't really question the need for that setting be we available system-wide. I question the fact that it's exposed through Settings.Global instead of a gapps-specific content provider. That's what I didn't really want to do.

However, from that perspective, it's ok to use Settings.Global to store a system-wide value, say a SystemPersistentIntegerContent, inasmuch as:

- You don't break the Settings.Global interface. Fortunately, I don't need to. I can just use *getInt* and *putInt*.
- It stays hidden as an implementation detail. Here will be twice hidden:
  - o Public access is through a content provider, not through System.Global
  - o Even the modular content provider won't know the exact persistence mechanism of a system-persistent [integer] content.

So here is SystemPersistentIntegerContent in its full glory – quite short in the end:

### The SystemPersistentIntegerContent

```

package com.aspectroid.cp;

import android.content.Context;
import android.net.Uri;
import android.provider.Settings;

public class SystemPersistentIntegerContent extends SingleIntegerContent
{
    public SystemPersistentIntegerContent(
        Uri contentUri, String fieldName, Context context)
    {
        super(contentUri, fieldName, context.getContentResolver());
        ctx = context;
        key = contentUri.getEncodedPath();
        setValue(getStoredValue());
    }

    private int getStoredValue()
    {
        return Settings.Global.getInt(ctx.getContentResolver(), key, 0);
    }

    @Override
    protected void setValue(int v)
    {
        Settings.Global.putInt(ctx.getContentResolver(), key, v);
        super.setValue(v);
    }

    private Context ctx;
    private String key;
}

```

I am using the contents URI as a system-wide key, which is ok – *a content URI is supposed to be unique system-wide anyway for any content exposed through a content provider*; there is no need to put an explicit static constant inside Settings.Global and break the interface every time.

### The Makefile

The ScreenState is a new module, so we need to add a makefile as well. This time we need a special makefile, because all modules to be merged inside the system server follow a specific convention about the module name (see also the next paragraph where part of the system server makefile itself is being shown).

Despite that (one really has to know a gazillion little details to play inside the android core) it's still pretty much a boilerplate makefile for this kind of module:

#### The ScreenState Android.mk file

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := services.screenstate

LOCAL_SRC_FILES += \
    $(call all-java-files-under,java) \

LOCAL_JAVA_LIBRARIES := screenstateapi

include $(BUILD_STATIC_JAVA_LIBRARY)
```

This module isn't built with aspects, so there is no need to add the aspectjrt library here.

### *Last few changes before we're done*

We're very close to the end, but it's not finished yet. We can't simply add a new module to the system server by creating a folder in the right place. We need to change the system server makefile as well, because it contains a list of all the modules that have to be merged. As before, changes are in red:

#### Changes to the Android.mk file

```
# ...
# merge all required services into one jar
# ...
# Services that will be built as part of services.jar
# These should map to directory names relative to this
# Android.mk.
services := \
    core \
    accessibility \
    appwidget \
    backup \
    devicepolicy \
    print \
    restrictions \
    usage \
    usb \
    voiceinteraction \
    screenstate
# ...
LOCAL_STATIC_JAVA_LIBRARIES := $(addprefix services.,$(services))
screenstateapi
```

I also had to add the screenstateapi jar to the local static java libraries. Honestly, I'm not sure why. In fact, I didn't need to add the aspectjrt library. I suppose it's more a byproduct of the way the individual JAR gets merged into one than a design-level issue.

Almost over now. There is one problem though. I added a content provider to the system server. Unlike, for instance, a broadcast receiver, which can be declared in a manifest file or programmatically created, android requires content providers to be declared through a manifest.xml file<sup>24</sup>. However, sub-modules of the system server don't have a manifest.xml file, because they're being compiled into simple JAR files. That got me stuck for a moment ☺; then I realized I had to add the relevant lines to the manifest.xml file of the system server.

Note: this is yet another portion of android where a half-hearted approach to modularity raises its ugly head. In theory, the system server is modular; in practice, its manifest knows about all the contained modules.

The manifest itself is as usual well hidden under frameworks\base\core\res, and it's rather big – over 3000 lines. That's to be expected because it is collecting concerns from all the included modules. Here are my changes (additions):

#### Changes to the manifest.xml file

```
<uses-permission android:name="android.permission.WRITE_SETTINGS"/>

<provider
android:name="com.aspectroid.screenstate.storage.Repository"
android:authorities="com.aspectroid.screenstate"
android:exported="true" />
```

Honestly, when I first switched from PersistentIntegerContent (which didn't work in the core) to SystemPersistentIntegerContent I didn't add the first line. That made the image to fail again (quite depressing ☺), but adb logcat was enough to understand the problem, after which the fix was rather obvious. I'm writing the system settings, and I need a permission to do so.

**We're done! We added an interesting new feature to the android core, without any changes to the actual (java) source code.**

---

<sup>24</sup> This part of android seems to have been designed from someone coming straight from the '90s way of writing J2EE applications, with XML files referencing class names etc. instead of using annotations, marker interfaces, etc. Oh well.

# Chapter 4: Reflections

## Random notes so far:

Previous literature on AOP in OS

Aspects vs distributed architectures

### Tools

Aspect in a jar, link with apps; would need a way to declare dependencies. Pretty much like a cross-cutting interface based on structural conformance.

Aspects and multi-language applications

Xml, mk are languages too

Localization – extendibility – modular level (e.g. a new layout is ok )

Native portions out of reach (adbd)

Naming standard (different)

Other files: manifest xml; localization; makefiles; etc

Concerns and source code; [?]

Hyperplanes / hyperJ

Aspects as a privileged solution

Why google should be using aspects in the android platform code

“std” android arch layered diagram

once we move past the markitecture [quote] or powerpoint architecture of clean modules, the actual code reveals a different story.

Microservices

## Notes from ccc appendix

Why we should use aspects to customize the android platform

Alternative approaches: xda tech. pointcuts expressive power; granularity.

more technological, more "hacking"

why pursue this then?

- no arch. astronaut, confront reality

- part of the exploration of a design space is to understand the limits, the complexity of moving from theory to practice

- to learn. aspectroid is a learning exercise.

why needed

oem layer vs "open source"

next versions

ref. to biblio aop/os

simpler than mega-refactoring

## Chapter 5: Does it really pay off?

Hypothesis / Validation: M 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.

[Alexander1964] Christopher Alexander, *Notes on the Synthesis of Form*, Harvard University Press, 1964.

[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.

[Pescio2006] Carlo Pescio, [Listen to Your Tools and Materials](#), IEEE Software, Vol. 23 No. 5, Sept.-Oct. 2006.

[Pescio2010] Carlo Pescio, [Notes on Software Design, Chapter 12: Entanglement](#), published on carlopesccio.com, November 2010.

[Pescio2011] Carlo Pescio, [Notes on Software Design, Chapter 13: on Change](#), published on carlopesccio.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](https://aspectroid.com).