aspectroid

exploring the aspect-oriented design space

Episode 1: Ribbons

©Carlo Pescio, 2014

Document Version 1.1

This booklet can be referenced as:

Carlo Pescio, Aspectroid Episode 1: Ribbons, 2014.

Full text, source code, binary files are available from 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 booklet, but you have to give appropriate credit (e.g. with a link to aspectroid.com); you cannot profit from the distribution, and you cannot alter the booklet or create derivative works for distribution.

As an exception to the aforementioned license, all the source code snippets in this booklet, and the full source code package you can download from aspectroid.com, are licensed under the GPLV3 license.

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 4
The Aspectroid Project 5
Prerequisites 6
Introductory material 7
Chapter 1: Why and What 9
Rationale behind the AOP goals 11
Choosing the right problem 13
The initial Design Vision 15
Chapter 2: How 19
It's an Android app 21
Handling touches 23
Keeping track of the Tracks 27
Where are we? 33
Sensing and Touching 34
Tracking 36
Where are we, now? 41
Let me see something 44
Updating the Screen 49
Two new Islands 53
Rendering 54
Show your colors 58
Lose your tail 63
Did you say simulator? 67
Blurring 71
It's a bug… it's a feature… it's an aspect 76
The whole thing 79

Miles to go ... 81
Chapter 3: Reflections ... 83
Objective evaluation (or lack thereof) ... 85
Dependencies ... 90
“Design style” ... 92
Classes and Interfaces ... 92
Reusable classes, specific aspects ... 92
“My” use of aspects ... 93
“Traditional” aspects (technological) ... 95
Monolith vs Cooperating independent pieces ... 96
Naming ... 97
Insights (or: things I’ve learnt by building it) ... 99
Reflections on the Process ... 99
The nature of the problem, or: why did aspects “work” here? ... 101
Identity and the role[s] of Surface ... 103
The different nature of Aging and Blurring ... 106
Coupling with the implementation ... 107
Idioms and Patterns ... 108
Testability and Testing ... 109
Testing in presence of aspects ... 112
Testing the aspects ... 114
Integration testing vs. Unit testing ... 115
Chapter 4: Wrap up ... 117
Appendix A: Tools and Setup ... 119
Bibliography ... 121
About the author ... 125

Acknowledgments

Daniele Pallastrelli and Michelangelo Riccobene went through both an early draft and the final version of this booklet. They helped me clarify parts of the text, proposed ideas that in turn led me to explore alternative solutions and (even more importantly) they provided me with what every author needs most: some comfort about the value of the book ☺.

Benedetto Fiorelli suggested several changes to improve the clarity and readability of the text. Being a bit stubborn about my writing style, I'm the one to blame for all the residual obscurity.

Paolo Bernardi found one of those hard-to-catch discrepancies between the code and the text.

The Aspectroid Project

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

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

For quite some time, I've also been pondering on how to best combine OOP and AOP, so that I could tackle design challenges unencumbered by old, crystallized yet sub-optimal strategies and patterns. AOP is a powerful, yet largely misunderstood and misrepresented paradigm. It is often depicted as a way to patch code, or with limited potential beyond a few technological aspects like logging. This is reminiscent of OO early years, before proper OOD techniques were introduced. Aspectroid will take AOP toward AOD, and use it as a full-fledged design vehicle.

The technological choices easily followed. AspectJ is the most mature aspect-oriented language. Developing yet another server-side application looked a bit boring. Android is more fun, and I've spent a lot of time in the past few years working on the Android platform. Therefore, Aspectroid will focus on aspect-oriented design within the context of Android apps, using AspectJ. As usual, most

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

of the notions and insights are completely platform-independent, and can be translated elsewhere.

I want to stress that my aim is to *explore the software design space*, not to build a best-seller, eye-candy app. Please come here looking for an aesthetics of code, not for the blending of colors. Also, *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.

Prerequisites

This is not a book for beginners, so to fully appreciate the 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, and a good exposure to object oriented design.
- A basic understanding of aspect oriented programming is beneficial. A few introductory works are listed below. While you can read this booklet 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 creating an Android app. Some knowledge about Android development will be useful. Again, you can follow the reasoning without a solid understanding of the Android fundamentals (activities, views, etc.), but to fully appreciate some details, and compare the final result with the average Android app, some experience is probably required.

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

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 play with code, you need to setup Eclipse with the Android Development Tools and the AspectJ Plugin. A few notes on how to set up the tools are in Appendix A.

Full source code is available on the *aspectroid website*.

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

Chapter 1: Why and What

I started this project with a design vision in my mind, and a set of hypothesis that I wanted to test. My vision was to use aspects to *improve* object-oriented design, or more exactly statically-typed object-oriented design, so that I could have the benefits of static typing, but also enjoy the flexibility usually associated with dynamic languages like Smalltalk. Actually, I wanted to move one step forward, and aim for a degree of compositionality beyond what is normally possible with OOP alone. Behind that vision, I had a few, relatively straightforward goals:

General design goals

- Align the architecture with the nature of the problem

- Allow for *extension and contraction* of application-level concerns

In theory, there is nothing new here. In practice, I often see technology-oriented, not problem-oriented architectures. People start with "let's use MVC" or "let's use a three-tier architecture", not with "let's understand the nature of the problem".

I also see fat applications that can be extended, but not easily contracted. I see classes getting larger as new features are brought in; and so on. As I've often said, people only got half the message in Parnas' seminal paper (*Designing software for ease of extension and contraction*, [Parnas1979]). So, even though these goals look rather naïve, they're probably not.

AOP goals

- Use aspects to separate application concerns, as opposed to what we commonly see in literature, where technological concerns like logging, exception handling, etc. are usually addressed.

- Keep aspects small. My hypothesis was that concerns could be kept in classes, with aspects enabling separation on one side, and weaving on the other.

Aspects are an enabling technology. With aspects, I have more freedom separating application-level concerns. In this sense, aspects are a disentangling construct at the conceptual level. On the other hand, because I have aspects, I can also compose those concerns together. In this sense, aspects work as gluing constructs at the pragmatic level. They allow me to separate concerns nicely in different classes while designing, because I know that I can glue them together later on with a little code. At least, that was the idea.

OOP goals

- No controllers, managers, etc.

- No stupid objects without methods.

- Avoid copying data between "layers" to "decouple" classes.

There isn't much to add here: a lot of contemporary "object-oriented" code is based on a large number of relatively small, stupid classes "managed" by large controller-like objects, often mapping data between similar objects. I wanted to show, once more, that it's not necessary to use that style to get highly decoupled, reusable classes.

A few, more pragmatic guidelines followed:

- Use classes to define behavior (and data).
- Use aspects to compose behavior (and supplement data).
- Never have a class depending on an aspect.
- Minimize dependencies between aspects (this is a prerequisite for extension and contraction).
- Keep classes and aspects small.

Most OOP literature suggests that classes should be small, but in practice, this is almost never the case in commercial code. If you peruse the Android source code, you'll find many classes with 5K to 10K LOC. I set a specific goal for my code: every class (and aspect) had to fit in one page of this booklet, and I'm using very small pages (A5, with some margin). I'm talking about full source code, including the imports; too many imports are a bad symptom anyway, so this rather arbitrary constraint will immediately expose problem areas[3].

Rationale behind the AOP goals

Most literature on AOP seems to be focusing on a small set of technological, cross-cutting, pervasive concerns like logging, exception handling, etc. That's reasonable, as those were the initial motivations for the paradigm itself: AOP was born to modularize code that OOP couldn't cleanly separate. Logging is the quintessential case of code that is normally sprinkled all over, a case of pervasive, cross-cutting concern[4]. Most pervasive cross-cutting concerns tend to be technological in nature, which explains the initial focus of most researchers.

[3] See [Pescio2006] for more on the idea of *backtalk* in a reflective conversation with source code.
[4] That is, a concern that cannot be isolated into a class or hierarchy, and instead is cutting across several unrelated classes.

However, using aspects for technological concerns has a relatively small impact on the larger application design. Sure, you may end up with a cleaner code, but overall, the design won't change much. That's perhaps one of the reasons why AOP hasn't been widely adopted: the case for AOP doesn't seem to be so strong.

There are, of course, papers talking about AOP as a full-fledged design paradigm; however, in most cases they stay at a rather abstract level, and it's hard to judge the final properties of a software product without having a chance to see the code, play around with it, etc. Nonetheless, in the past few years I've been exploring the idea of combining AOP and OOP in small toy projects, focusing on aspects as a way to separate application-level (or domain-level) concerns.

Borrowing from an old blog post of mine [Pescio2008], consider the usual subscription form for a service. The back end will register your data and activate your subscription. Suppose now that if you provide the email of a friend / referral, you get a discount and your friend gets a gift. You may treat that as part of the overall subscription logic, you may encode that in a flexible workflow if you have one, but you can also consider that as an application-level (or domain-level) aspect of the subscription process. That's very different from a technological concern, and may have a stronger impact on your design. I wanted to experiment more with this approach, to learn the actual consequences of walking that path.

There is more: a lot of aspect-oriented examples I've seen end up having significant code inside the aspects themselves. I wanted to explore a different style, where most code is in classes, and aspects contribute (mostly) with knowledge about the wiring. This is in line with the notion of aspects as a way to express when / where something must happen, but not the details of what is going to happen.

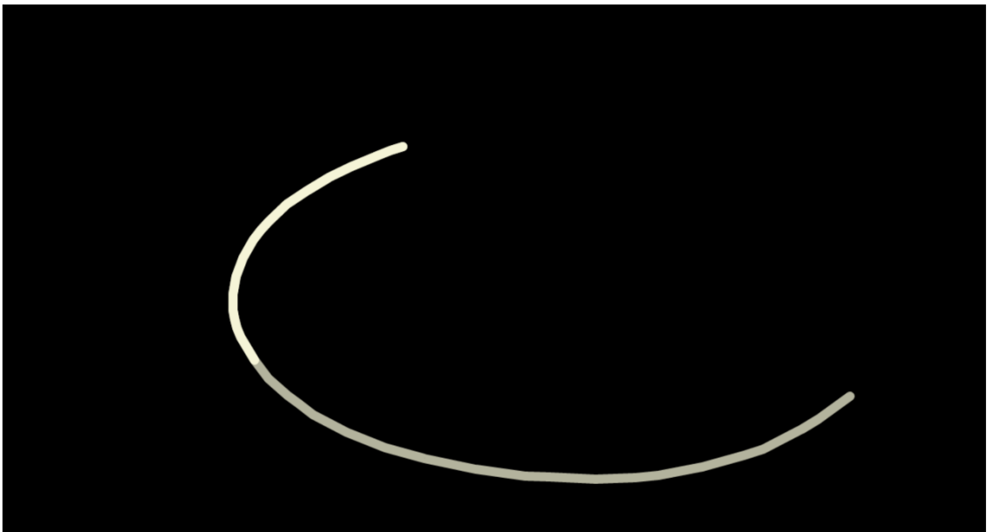
In the end, a good designer should occasionally explore new approaches, even radically new, and this was a nice opportunity to do so.

Choosing the right problem

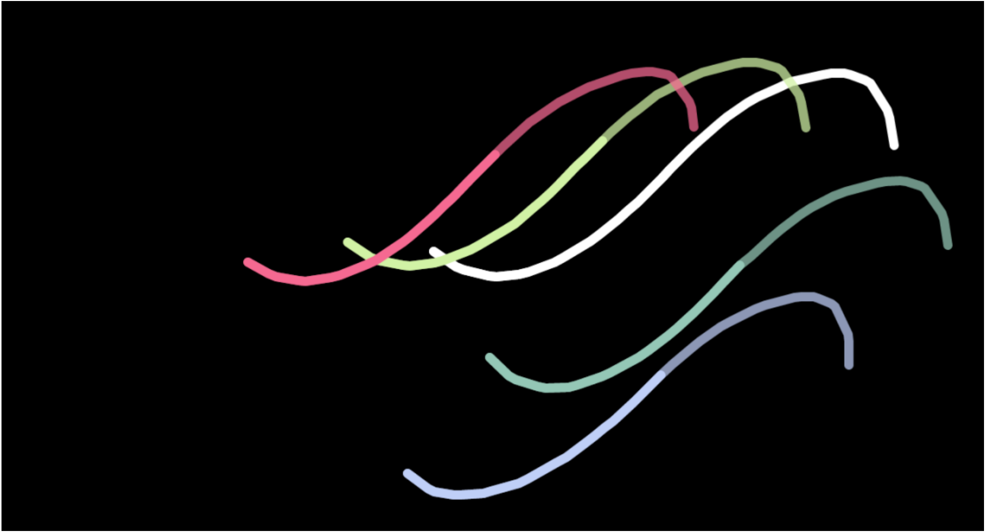
Looking for a problem where these techniques could be applied and explored, and probably influenced by early AOP literature on composition filters (see for instance [BA2004]), I had the intuition that the way I wanted to connect classes was similar to an aspect-defined pipeline, so it was natural to look for a problem where one could think of data moving through a number of stages, being transformed and enriched along the way.

My first idea was to create yet another activity tracking app. On one side of the pipeline, I would have had sensors, like GPS, accelerometer, etc. Stages could clean up data, merge different sensors to improve precision, compress the stream in various ways, detect the most likely type of activity, and calculate calories and whatever else. While that idea had merits, and I think it would be quite interesting to explore, I quickly realized that it would bog everyone down with a plethora of domain notions and some heavy math, while I wanted to focus this work on the choices we make while designing, and the consequences of those choices.

In the end, I came up with simple interactive application, where you draw simple sketches with your fingers, like this:



Or course, multi-touch should be supported too:



To make it slightly more interesting: the curves you draw should "follow" your movement, that is, the trail must not grow indefinitely. Also (this is visible in the pics above) the older part of the trail must be slightly dimmed.

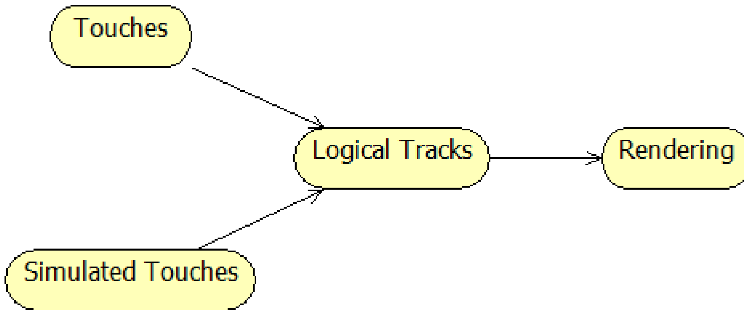
Describing dynamic behavior using text is always a bit difficult, so you can simply check out a couple of videos on the aspectroid website, or go ahead and try out the app; you can get it from google play or from the aspectroid website.

If you're in for a little coding, you may even want to go ahead and implement a similar app yourself *before* reading what follows, so that you can compare your design with what I'm discussing here.

The initial Design Vision

From the description above, one may reasonably think of this simple pipeline, which already shows multiple input sources:

Diagram 1

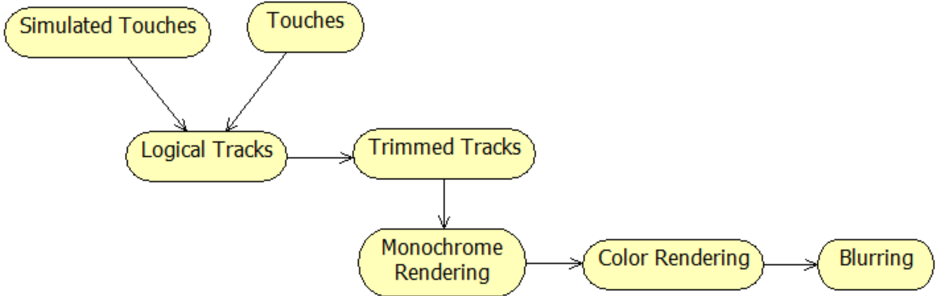


Of course, we could proceed to implement this, disregarding of any non-functional property. The leftmost stage would be rather trivial, as we only need to listen for touches. Some of the remaining logic would gravitate around the notion of a "logical track", some around the rendering block.

As we add more features (trimming, dimming) those blocks will get larger, and may require changes to the previous stages (it would be so handy for trimming if input provided a timestamp and not just the coordinates…). As we move further, the typical effect of those incremental changes is the emergence of a system that cannot be scaled down easily: features are there, more can be added, but removing or replacing parts is hard. Still, that was supposed to be one of the cornerstones of a pipelined architecture.

What if all the fine-grained concerns were pipeline stages? We could think of trimming as a stage, that's sort of simple. The rendering block may look a bit more impervious to partitioning. Yet, we have at least two concerns that could be isolated: associating a color to a track and blurring the tails. A rough pipeline may look like this:

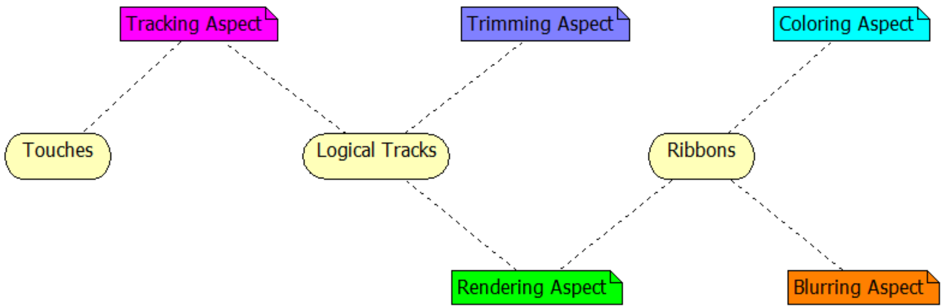
Diagram 2



Still, this representation is a bit old-school and does not really represent what we want to build. First, it is not obvious at all how we could add colors or blurring as later stages. Second, it seems like we need every stage in that sequence, while my design idea was to aim for maximum compositionality. I want the freedom to include or exclude any stage: having blurring but no colors, having colors but not trimming, etc.

In the end, a pipeline is *a good design metaphor*, but as we move from coarse-grained to fine-grained concerns it doesn't seem to scale so well. What I had in mind when I started was more like this[5]:

Diagram 3



I was thinking of three main domain classes, to represent touches, logical tracks, and their visual counterpart (ribbons), mostly disconnected, wired together by aspects and optionally supplemented by aspects to provide compositional behavior. In a sense, the initial vision was a mixture of a pipeline and a mixin architecture.

As it turned out, my initial idea evolved significantly when I actually stepped in and implemented specific parts. More classes and aspects emerged, a few things proved harder than I expected, etc. Still, the core of the idea, the "pipeline + mixin through aspects" vision, came through rather well. In what follows, I'll guide you through the code, and the design decision behind it, as every single piece comes to life.

---

[5] Yes, I have a rather colorful imagination ☺

Chapter 2: How

Writing this app was not a linear process. I had a vision for the properties my design should have, and I had a vision for the overall structure my software should take, but as I tried to turn vision into reality, I stumbled on a number of issues, ranging from unclear error messages from the AspectJ compiler, to design issues, to design opportunities that I hadn't considered before, to design insights that I gained by building something, and which suggested I should scrap some parts and make better ones. In fact, even the act of writing this booklet forced me to reflect upon my code at a deeper level and apply small and not-so-small changes.

It's basically impossible to capture all that process in a book. I have indeed contemplated a narrative style where we could explore a solution, find problems, backtrack, etc. I'm sure it would be more effective; it would also require a longer text. In the end, at least for this episode, I settled for the traditional style of faking a rational design process [PC1986]. Just remember that it didn't go exactly that way; while the overall architecture didn't change much from my initial vision, many non-trivial choices and consequences were obvious only in hindsight.

Finally, I didn't try to make it "perfect". Real-world design is about making trade-off choices between several properties. I made some choices, and I've unraveled the consequences of those choices. Here and there, I saw different opportunities, but like in the real world, I didn't have the time to pursue them all. I will suggest a few interesting alternatives in case you want to explore that portion of the design space yourself. I might also come back to the same problem in a future episode, and explore a different approach in depth.

In the end: *what follows is not a blueprint architecture*. I'm not suggesting that you should implement any other application this way. I'm not even suggesting that you should use aspects at all. I'm definitely not suggesting that AOP is magic and that other paradigms are trash.

I consider the final design quite interesting and well aligned with the structure of the problem. That's it. While some people tend to be rather quick in extending the peculiarities of one case to universal truths, I am not. If you want to apply

similar techniques, my suggestion is that you spend some time understanding the nature of the problem I'm solving and why these techniques are working reasonably well here, and then consider the nature of the problem you want to solve, and maybe do some experiments to understand if they may work well there. With that said, let's the fun begin.

It’s an Android app

Like in any other Android app, we need a manifest, an Activity class, and a layout. I’ll skip the manifest, but let’s take a look at everything else, beginning with the main Activity class:

The RibbonsActivity class

```
package com.aspectroid;

import android.app.Activity;
import android.os.Bundle;

public class RibbonsActivity extends Activity
{
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
}
```

This is as uncluttered as it comes. I’m just setting the content layout, nothing else. Of course, this is a simple app and I had no need to override lifecycle methods, but it’s a good start anyway.

Before we move further: occasionally, in the following snippets, I've used rather short variable names (shorter than I would normally use), so that I could easily fit within the narrow page width. I still strived for good readability, and I’ve kept class, method and aspect names aligned with my usual conventions. A paragraph on naming in AOP can be found on chapter 3.

The activity is loading a layout, which as usual in Android is defined in an XML file:

The main layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <com.aspectroid.Surface
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="#000000" />

</LinearLayout>
```

At this stage, the layout is extremely simple as well: it's made of a custom view (Surface), filling the screen and with a black background.

We would usually expect to find quite some logic inside the Surface class, as it's the only place left:

The Surface class

```
package com.aspectroid;

import android.content.Context;
import android.util.AttributeSet;
import android.view.View;

public class Surface extends View
{
  public Surface(Context context, AttributeSet attributeSet)
  {
    super(context, attributeSet);
  }
}
```

Surface is uncluttered as well. It's actually more like a *marker*, a beacon which can be easily identified inside a pointcut definition. Most of the aspects will need to mention Surface, to add behavior or data. Surface will also provide *identity*, which is a non-trivial, resurfacing concept in Aspect Oriented Design; at this time, just remember that we want the app to work fine if we change the layout and declare multiple surfaces side by side.

Handling touches

An obvious concern for this app is sensing the screen, using the Android API to detect and decode multi-touch gestures. In practice, the touch screen is only one possibility among many: for instance, I'd like to have some simulation / test code feeding a predefined pattern without human intervention.

This is a variation point[6], and we already know how to handle that with Object Oriented techniques: interfaces and multiple implementations. I'm not going to give up on those techniques: I'm actually going to *blend* them with Aspect-Oriented composition. So let's start by introducing a new abstraction for things which can be touched, or which can provide a stream of touches: a **TouchPad** interface.

The TouchPad interface

```
package com.aspectroid;

public interface TouchPad
{
  public void onTouchDown(int pointerId, float x, float y);
  public void onTouchMoving(int pointerId, float x, float y);
  public void onTouchUp(int pointerId);
}
```

This rather minimalistic interface captures:

- What happened (encoded in the method name)
- Where (coordinates, when applicable)
- Using which finger / pointer (the pointer id) to allow for multitouch.

You can observe a small but significant departure from contemporary wisdom here. Here, a TouchPad is meant to notify events *to itself*. The idea is that you have a TouchPad object, you touch that object, and the object will say "hey, I've been touched here". There is no *listener* interface; the TouchPad is not telling

[6] It's actually a special kind of variation point: non-uniform behavior with a uniform interface; that's why OOP works well here.

someone else about the event. Although this is indeed a departure from the commonly adopted model[7], it is more in keeping with the original view of objects like living organisms [Kay1993]. If something bumps into you, *your body notices*. It doesn't go around telling listeners. In practice, this also means we don't have to come up with *non-functional abstractions*[8] like a listener, which is normally there simply to decouple classes. Decoupling is obtained in a different way here (we'll get to that shortly).

Of course, there are good reasons to adopt a listener model in Java (or similar languages; C# would use a delegate here, but the idea is the same). I'm not saying it's wrong. I'm saying, however, that AspectJ is a different material, which can be shaped in a different way, and that by exploring those shapes we may even get closer to the initial vision of objects. So if your spider sense is telling you that this design is wrong and you need to have a listener and so on, please suspend judgment a little longer ☺.

An interface is useless without one or more implementations, so let's move on to something more concrete. In practice, I'm going to touch a View object, so I need a special kind of TouchPad, where the Pad is in fact a View. I just need a TouchView class.

---

[7] At least in most widely used, statically-typed, single-inheritance OO languages.

[8] That is, abstractions which cannot be traced directly to the problem being solved, but are introduced exclusively to obtain some non-functional property (like reusability or decoupling), within the rules and limits of the implementation language.

The TouchView class

```
package com.aspectroid;

import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;

public class TouchView implements TouchPad, OnTouchListener
{
  public TouchView(View s)
  {
    s.setOnTouchListener(this);
  }

  @Override
  public boolean onTouch(View v, MotionEvent me)
  {
    int ptrIndex = me.getActionIndex();
    int ptrId = me.getPointerId(ptrIndex);
    int maskedAction = me.getActionMasked();

    switch( maskedAction )
    {
      case MotionEvent.ACTION_DOWN:
      case MotionEvent.ACTION_POINTER_DOWN:
      {
        onTouchDown(ptrId, me.getX(ptrIndex), me.getY(ptrIndex));
        break;
      }
      case MotionEvent.ACTION_MOVE:
      {
        int size = me.getPointerCount();
        for( int i = 0; i < size; ++i )
        {
          onTouchMoving(me.getPointerId(i), me.getX(i), me.getY(i));
        }
        break;
      }
      case MotionEvent.ACTION_UP:
      case MotionEvent.ACTION_POINTER_UP:
      case MotionEvent.ACTION_CANCEL:
      {
        onTouchUp(ptrId);
      }
    }

    return true;
  }
}
```

A TouchView basically grafts itself onto any given view (provided in the constructor), listening for [multi]touch events. Its only purpose is to transform MotionEvents into TouchPad calls. In this sense, TouchView is very much an adapter[9], even though it's got a decent name which reveals purpose, not implementation.

There is an ugly switch/case here, for which I'm sorry. MotionEvent has been designed in a way which basically requires you to switch/case, and I had no reasons to make the code overly complex by implementing something more flexible. After all, I don't foresee any relevant change in those cases: significant changes are probably better suited for a sibling class. For simplicity, I'm not handling the *historical events* embedded inside the MotionEvent. That's of course possible if you need a more fine grained resolution.

You can probably see that there is something going on here, because TouchView claims to implement TouchPad, yet it doesn't; still, it's not declared abstract, which would be an error in plain Java. The answer, of course, is in the aspects, but we need to postpone that part for a moment: aspects act as glue here, and you need at least two pieces to use some glue. The TouchView is one piece, and its job is done. Let's move further.

[9] As in the Adapter pattern, of course, although it doesn't look like it.

Keeping track of the Tracks

I obviously need some place to store all those points, so that later on I can draw them, and apply whatever logic is required to animate the trail. Some would suggest that I simply keep a list of Points, and then have a bunch of functions which can be applied to lists of Points. While that is a possibility, and while it might be interesting to investigate that side of the design space, I'm going for something more object oriented here. Objects are meant to be specialized little machines, and lists are basically like primitive types. So my first abstraction will be a Track. Intuitively, you begin a track by touching the screen, keep adding points while moving your finger, and then when you raise the track is completed. If you touch again, you'll start another track. That quickly leads to a class with the following responsibilities:

- Maintain a list of Points which can be incrementally populated.
- Implement a decimation policy: if you add a point that is too close to the last one, just ignore the request.
- Being in a state of open or closed. When a Track is closed, you can't add any more points.

A single track won't help much if we want to handle multi-touch: every finger will form its own trail (or Track), so I need a place to store a set of tracks. Again, while it might be tempting to just drop a List<Track> somewhere, I'll go for a TrackSet class. As we'll see, a significant portion of code can be placed there.

I keep talking about Points, so I have to make a decision: use a library class or roll my own. That's usually a no-brainer (use the library, Luke) but it so happens that the Point class in Android is in *android.graphics*, and by using that class I would tie my Track class with presentation[10], while I wanted this part to be totally unaware of sensing and rendering (which are meant to be specific *aspects* in my mind). This fact alone, plus the fact that we're talking about a trivial class, strongly suggests implementing our own Point. There is actually more: later on, we'll find it useful

[10] Actually, with a specific presentation technology: you don't use android.graphics.Point in android OpenGL ES.

to have our own aspects play with Points in a way that cannot be done with a library class living in a jar file we cannot touch[11] (*android.jar*). So, here is Point:

The Point class

```
package com.aspectroid;

public class Point
{
  public Point(float x, float y)
  {
    this.x = x;
    this.y = y;
  }

  public float manhattanDistance(Point p)
  {
    return Math.abs(p.x - x) + Math.abs(p.y - y);
  }

  public static final Point INFINITY =
    new Point(Float.POSITIVE_INFINITY, Float.POSITIVE_INFINITY);

  public final float x;
  public final float y;
}
```

Point is immutable, so it makes little sense to implement getters: it's easier to expose public data (no, I'm not really going to change the implementation to polar coordinates).

I like classes to do their parts within the overall computation. Therefore, even the humble point will take on some responsibilities besides carrying data:

- Calculate distance from another point. I'm choosing speed over precision here, and go with the so-called Manhattan Distance. Of course, a more traditional notion of distance could be added, but it's not needed here.

- Provide a constant for an infinitely distant point, which will come handy at some point, and it's generally useful.

[11] This is more a limitation of AspectJ under Android than a limitation of AspectJ itself or of Aspect Oriented technologies.

Track is straightforward as well:

The Track class

```
package com.aspectroid;

import java.util.Iterator;
import java.util.LinkedList;

public class Track implements Iterable<Point>
{
  public Track()
  {
    points = new LinkedList<Point>();
    lastPoint = Point.INFINITY;
  }

  public void addPoint(Point p)
  {
    if( !isClosed && p.manhattanDistance(lastPoint) >= MIN_DELTA )
    {
      points.add(p);
      lastPoint = p;
    }
  }

  public boolean isClosed()
  {
    return isClosed;
  }

  public void close()
  {
    isClosed = true;
  }

  @Override
  public Iterator<Point> iterator()
  {
    return points.iterator();
  }

  private static final float MIN_DELTA = 4;

  private Point lastPoint;
  private LinkedList<Point> points;
  private boolean isClosed;
}
```

The only remarkable thing about Track is that it implements *Iterable<Point>*, so you can use it in a *for(…)* loop.

TrackSet is more complex, so to speak (it still fits in one small page). Some of the code is here just for small-scale performance reasons. Overall, the responsibilities of a TrackSet are quite limited:

- Create a new Track given the initial point and pointer index
- Add points to an existing open track (by pointer index)
- Close a track
- Of course, keep memory of all the tracks, present and past (open / closed)

In theory, a single list would be enough, but it would also be inefficient. There are many micro-design alternatives here, but in the end, I choose to put all the tracks in a linked list, but also keep all the open tracks in an ArrayList, so that I can efficiently get a track given the pointer id. The immediate consequence of this is that it becomes useful to know the maximum number of simultaneous touches we want to handle (to size the array).

TrackSet is not synchronized; it sort of assumes that points will be added in the UI thread (which is always true when the touch screen is involved) and that will be pulled out in the UI thread as well (which is also true when the pulling is done for drawing purposes). Forcing synchronization on the class seemed unnecessary here; it could be interesting to investigate the value of using aspects here, but for this episode, I’m after application-domain concerns, not technological concerns.

Mimicking Track, TrackSet implements *Iterable<Track>*, so that we can use it in a *for(…)* loop.

The TrackSet class

```
package com.aspectroid;

import java.util.Iterator;
import java.util.LinkedList;

public class TrackSet implements Iterable<Track>
{
  public TrackSet(int maxOpenTracks)
  {
    tracks = new LinkedList<Track>();
    openTracks = new SizedList<Track>(maxOpenTracks) ;
  }

  public void startNewTrack(int trackId, float x, float y)
  {
    Track t = new Track();
    tracks.add(t);
    t.addPoint( new Point(x, y));
    Track prev = openTracks.get(trackId);
    if( prev != null )
      prev.close();
    openTracks.set(trackId, t);
  }

  public void addPoint(int trackId, float x, float y)
  {
    Track t = openTracks.get(trackId);
    if( t != null )
      t.addPoint(new Point(x, y));
  }

  public void endTrack(int trackId)
  {
    Track t = openTracks.get(trackId);
    if( t != null )
      t.close();
  }

  @Override
  public Iterator<Track> iterator()
  {
    return tracks.iterator();
  }

  private LinkedList<Track> tracks;
  private SizedList<Track> openTracks;
}
```

TrackSet uses SizedList, which is simply an ArrayList done right ☺ by adding a constructor which sets both the capacity and the size of the container:

The SizedList class

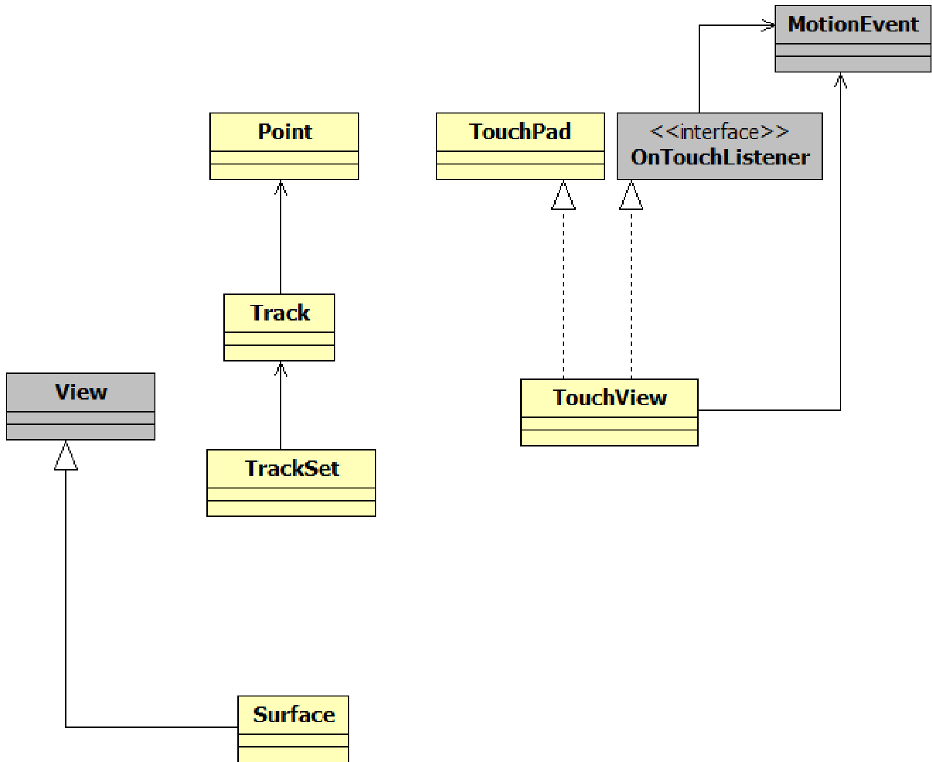
```
package com.aspectroid;

import java.util.ArrayList;

public class SizedList<T> extends ArrayList<T>
{
  public SizedList(int size)
  {
    // capacity
    super(size);
    // size
    for( int i = 0; i < size; ++i )
      add(null);
  }

  private static final long serialVersionUID = 1L;
}
```

Note that Point, Track and TrackSet have no dependencies on anything except the standard Java libraries (and themselves). As such, they can be tested independently and reused independently. They are domain objects, unaware of input (sensing) or output (drawing) concerns.

Where are we?

We already covered quite some code here, and it's easy to get lost when all you have at your disposal is a strictly sequential medium (storytelling)[12]. So, although UML is reported to be dead, I think that a simple class diagram[13] may help here:

Diagram 4



Gray classes are from Java or Android libraries, while all the yellow stuff is mine.

Unlike traditional OO design, where classes are expected to form a *connected* graph, here we can see small *islands* of classes: isolated clusters with no incoming

---

[12] A short analysis of the linear nature of textual descriptions is in [Armour2004], Chapter 5, on the Physical and Logical nature of models.

[13] I tend to follow a layout rule where all dependencies point upward. For an explanation of why, see [Pescio2006]

or outgoing relationships. This is in line with what I expected, and should be the norm when you mix OOP and AOP and you aim for application-level aspects, as opposed to the oft-discussed technological aspects. I'll get back to this in the next chapter.

Of course, unless something happens, this design will not work. The TouchView needs to be somehow connected with the Surface, so that it can sense touches on the main application view. Also, it needs to somehow populate a TrackSet with Tracks, coming from those touches.

All the wiring is provided by two aspects: the Touching aspect and the Tracking aspect. Touching is a specialization of Sensing, as the input may come from another source (for instance, it could be a recorded set of touches, or an algorithmic simulation; I'll provide a working example of that later on).

Sensing and Touching

The theory is simple: when a Surface is created, I want to create a TouchView as well, and wire the two together; TouchView takes a View as a parameter, and that would be the Surface, so that the TouchView will listen for touches on the Surface.

Again in theory, this could be accomplished by defining a pointcut on the Surface constructor and, in a corresponding advice, create the TouchView. This would work just fine, and that's indeed what I did in my first version. It also didn't extend nicely to simulated input, because in that case I needed to know the geometry of the Surface (width and height), and in android those are reported as zero inside the constructor. That's one of the many tiny differences between theory and practice. It turned out that that construction was too early, and I needed a better join point. A reasonable one is onSizeChanged, a method which is called on View when the layout has been [re]defined and the size is known. Since this pointcut will be used in all variants of the Touching aspect, I moved it into a base Sensing aspect; so, simulation, playback, etc. can simply extend Sensing like Touching does:

The Sensing aspect

```
package com.aspectroid.aspects;

import com.aspectroid.Surface;

public abstract aspect Sensing
{
  public void Surface.onSizeChanged(int w, int h, int oldw, int oldh)
  {
  }

  protected pointcut
  onSizeChangedCall(Surface self, int w, int h, int oldw, int oldh) :
    execution(
      public void Surface.onSizeChanged(int, int, int, int)) &&
    args( w, h, oldw, oldh ) && target(self);
}
```

Don't let AspectJ syntax scare you; I'm just saying that I want to intercept Surface.onSizeChanged[14] here. AspectJ won't allow you to intercept a method that is not overridden, and since Surface had no need to override it, I'm doing it here.

The pointcut type matters. I choose *execution*, not *call*, because the call is inside the Android framework (which cannot be touched by AspectJ) while the execution is within my code, which can be easily intercepted.

I need access to the target object and the parameters, so the syntax is somewhat longish, but ok, AspectJ is like that, you need to get used to it.

Sensing is an abstract aspect, as it doesn't define any behavior; it's just factoring out a common portion of code for all the concrete implementations, like Touching.

[14] AspectJ won't allow me to hook into Surface.onSizeChanged, which is inherited from View, unless I redefined the function. It's quite harmless in this case.

At this point, Touching is trivial: after the Surface has been sized, I want to create a TouchView attached to it. While the TouchView object will get out of scope immediately after this call, it will be kept alive by the Surface, as it will become a touch listener.

The Touching aspect

```
package com.aspectroid.aspects;

import com.aspectroid.TouchView;
import com.aspectroid.Surface;

public aspect Touching extends Sensing
{
  after(Surface self, int w, int h, int oldw, int oldh) :
  onSizeChangedCall(self, w, h, oldw, oldh)
  {
    TouchView otp = new TouchView(self);
  }
}
```

We now have a working TouchView, but tracks are not being created. Time for the Tracking aspect to enter the picture.

Tracking

Once again, the theory is simple, but as we move to practice, we'll have to take a few non-trivial decisions. Let's look at the moving pieces here:

- On one side, we have a TouchPad (and its derived classes). Here we know that the screen has been touched.
- On the other side, we have a TrackSet; we definitely need to create a TrackSet and store it somewhere, so that we can add points to the TrackSet when touches are detected.
- Finally, we have the Surface; the Sensing and Touching aspects will connect a TouchPad to the Surface, but ideally, the Tracking aspect should not depend [too much] on that.

The first choice that we need to make is where to store the TrackSet. We don't have too many alternatives:

1. In the TouchPad
2. In the Surface
3. In the Sensing Aspect
4. (In some global associative container)

In my exploration of the Physics of Software, I've learnt to observe the forces at play and let them guide the correct allocation of data. Here the most relevant forces are channeled through the multiplicity of things:

- We want to allow multiple Surfaces on a single activity; since every surface must have a different TrackSet, alternative (3) isn't easy, that is, we can't use the default "singleton" allocation for aspects [CCHW2004], and we would need to find an appropriate instantiation model.

- We want to allow multiple TouchPads for a single Surface; for instance, a "live" TouchView and a simulated source of touches working at the same time. That discourages alternative (1), as we want to merge everything into a single TrackSet.

Alternative (4) is ugly, and I've mentioned it above just for sake of completeness. In the end, alternative (2) is the simplest, safest, most straightforward of the pack, so the Tracking aspect is going to add a TrackSet to every Surface.

Still, this opens up another minor problem (it may help to look back at Diagram 1). Touches are detected at the TouchPad level; the TrackSet is inside the Surface; I need to get hold of the Surface from the TouchPad, but I don't have any immediate way to do that. While the solution is simple, it pays to understand the reasoning process here.

In an interesting perspective on AOP [VideiraLopes2004], Cristina Videira Lopes talks about aspects (and beyond) as a way to talk about referential dependencies, like what should happen to whom and when. In this specific case, what I really want to say is "*when a TouchPad is created within a Surface context, I need to connect that TouchPad with that Surface*". Expressing that referential dependency

requires some AspectJ gimmick, but also a better understanding of what a "surface context" really is.

A first cut clarification could be: when a TouchPad is created within a call to Surface.onSizeChanged, I need to connect that TouchPad with that Surface. After all, that's what is going to happen: the Touching aspect is going to create a TouchView (which is-a TouchPad) after onSizeChanged is being called, and the simulation aspect (or any other aspect derived from Sensing) would do something similar.

That, however, would create a weak, invisible, yet present dependency between Tracking and Sensing. I know that I have to watch for onSizeChanged being called because I expect the TouchPad to be created from a Sensing-derived aspect. This is in contrast with my desire to keep the aspects as unaware (oblivious) of each other as possible. Theoretically, we could get rid of the Sensing / Touching aspects, and decide that we will wire the TouchPad directly in a subclass of Surface. This is within the allowed design space, and I don't want to place needless restrictions on design options.

The tension between an easy solution and its shortcomings is actually suggesting us a better definition of what "being within a Surface context" really means. *Being within a surface context is simply being in the control-flow*[15] *of any Surface method*. Even if we were manually creating the TouchPad from a subclass of Surface, that definition would work just fine. This lessens the (invisible) dependency between the Tracking the Sensing aspect, to a point where I'm honestly satisfied with the result.

The rest of the implementation is relatively straightforward: once we have a TrackSet in the Surface and a Surface in a TouchPad, we simply have to implement the TouchPad methods and forward calls from the TouchPad to the TrackSet (via the Surface).

---

[15] It's useful to understand control-flow based pointcuts here; see for instance [CCHW2004] or [Miles2004].

The Tracking aspect

```
package com.aspectroid.aspects;

import com.aspectroid.TouchPad;
import com.aspectroid.Surface;
import com.aspectroid.TrackSet;

public aspect Tracking
{
  private Surface TouchPad.surface;

  private pointcut derivedConstructorCall(TouchPad tp) :
   execution(TouchPad+.new(..)) && target(tp);
  private pointcut surfaceContext(Surface sfc) :
    execution( * Surface.*(..)) && target(sfc);

  // when a touchpad-derived object is created within a surface context,
  // put the surface in the touchpad
  after( TouchPad tp, Surface sfc ) :
    derivedConstructorCall(tp) && cflow( surfaceContext(sfc) )
  {
    tp.surface = sfc;
  }

  private TrackSet Surface.tracks;

  private pointcut constructorCall(Surface self) :
    execution(Surface.new(..)) && target(self) ;

  after(Surface self) returning : constructorCall(self)
  {
    self.tracks = new TrackSet(20);
  }

  public void TouchPad.onTouchDown(int pointerId, float x, float y)
  {
    surface.tracks.startNewTrack(pointerId, x, y);
  }

  public void TouchPad.onTouchMoving(int pointerId, float x, float y)
  {
    surface.tracks.addPoint(pointerId, x, y);
  }

  public void TouchPad.onTouchUp(int pointerId)
  {
    surface.tracks.endTrack(pointerId);
  }
}
```

A couple of notes on this code:

- For efficiency, the TrackSet wants to know the maximum number of open tracks; that's basically the number of simultaneous touches the Android device can handle. I slacked off and set it to 20 here.

- The Tracking aspect provides an implementation for the TouchPad methods. That's why the code compiles in the first place. This, however, is not the code that I'd like to write: it's what AspectJ allows me to write.

Expanding on the latter point: in my initial vision, classes were totally oblivious / unaware of aspects. Therefore, they could be somehow reused elsewhere, without those aspects, maybe in an OOP-only[16] code base. To have that property, TouchView should have been abstract (due to the TouchPad methods). So in an OOP-only approach, you would have to subclass TouchView and do something there (which is still a reasonable design). The Tracking aspect would have provided an implementation for those methods, and marked TouchView as no longer abstract.

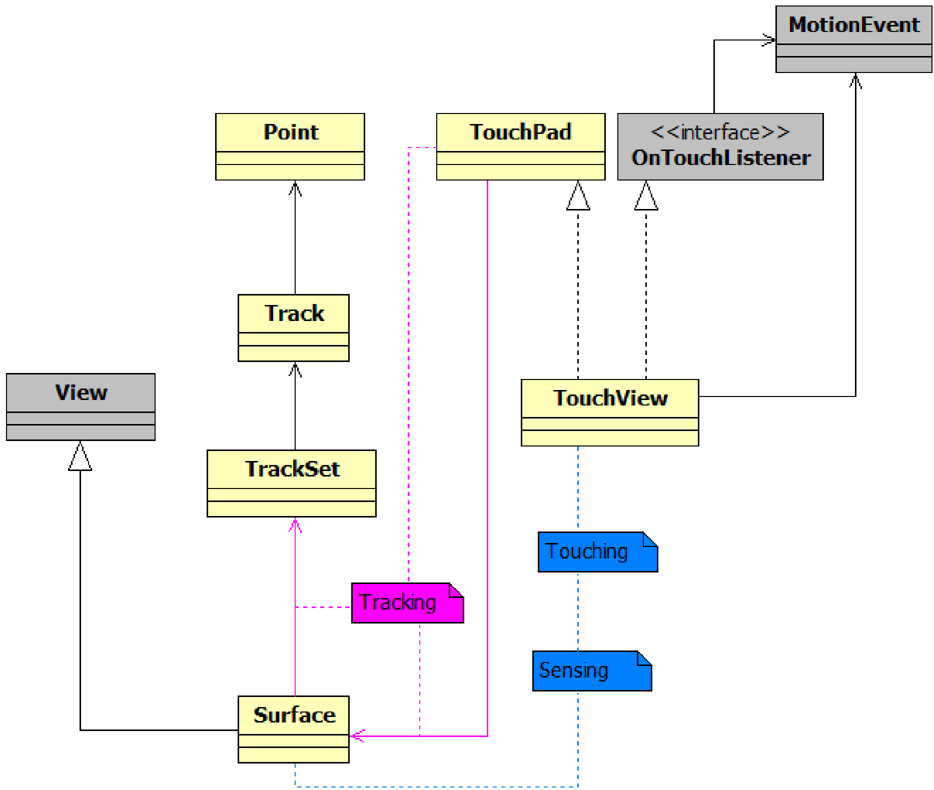
This is not possible using AspectJ. Looking at some debates over similar requests, I think the main issue is that the language is evolving toward a notion of aspects as "things you use to safely patch existing code", as opposed to "things you use to implement an aspect-oriented design", possibly because very few are talking about aspect-oriented design ☺.

[16] I use OOP-only, and not "pure OO" here, because purity is a pernicious notion when applied to engineering. Yes, I know, many of you have been educated with the idea that "purity" is bliss. Please try to forget that teaching.

Where are we, now?

Once we bring in the Sensing, Touching and Tracking aspects, the structure of our design changes quite a bit, becoming like this:

Diagram 5



Colored annotation boxes represent aspects; they connect to the classes (or other aspects) they depend on with dashed lines. Colored associations are added into classes from the correspondingly colored (and connected) aspect. So, in this case, the associations between TouchPad and Surface, and between Surface and TrackSet, are brought in by the Tracking aspect.

The dependency between TouchPad and Surface goes “down”, which is a violation of my “dependencies point up” rule. This is absolutely ok. TouchPad, as an artifact, is completely unaware of Surface. It can be [re]used in a completely different context, where Surface does not even exist. It’s the Tracking aspect that’s binding the two together. This is, in fact, the largest benefit of aspect-oriented technology from the perspective of software design: the ability to bind things at compile-time (with the strong type-checking guarantees coming from that), while still keeping the respective artifacts blissfully unaware of each other.

You can actually skip this part ☺

There are many proposals to extend UML with stereotypes and even entire UML profiles leaning toward AOP. The largest issue with the proposals I've seen (a dozen or so) is that they weren't designed to create *useful models*. They try to put too many details in the diagram, and while doing so, they still fail to include some. Trying (for instance) to represent the advice needed to add a Surface to the TrackPad in a UML diagram is pointless. That kind of detail is better left to the source code.

Unfortunately, those extensions also fail to support me while reasoning (which is while I'm modeling in the first place), or to make my AO design choices stand out clearly when I look at a diagram (for instance, they don't clearly show that a dependency is coming from an aspect). Therefore, I reluctantly did the worst possible thing: use my own notation.

My first attempt was based on abusing a few UML constructs like the association class. It failed, as I kept hitting UML metamodel walls. My tool of choice was not customizable enough at the metamodel level to allow me to define an association class-like classifier. In the end, I did the worst thing again, and simply used annotations and coloring. That, however, proved to be just what I needed while thinking. Ideally, my approach should be applied with a tool offering layers, but layers are not part of UML, so within most CASE tools we're limited to a single layer. Moving every aspect to a different layer, however, would be of great assistance while designing (as opposed to merely represent code in pictures).

My simple extension does not aim to be precise: for instance, it's not telling you which data a specific aspect adds to a class (unless you show that as an association), or which methods are intercepted, and how. While "seeing" this might be useful or even important, it's exactly the issue with most unsuccessful attempts at using UML for OOD: you end up with a clone of your code, with a significant maintenance cost, so you end up dreaming about keeping it in synch automatically, so you end up wasting a lot of time, so you end up not using UML anymore. I know the value of models, but I also know that to be valuable, they have to be *abstract models*, not the real thing in pictures.

Let me see something

A Track is a *domain object*, representing a filtered sequence of points. It's not a *presentation object*, concerned with colors and smoothness and similar things.

In fact, in my initial vision one of the pipeline stages was concerned with smoothing the input points using, for instance, a quadratic or cubic spline. Still, there is already a class in android.graphics which can do that (the Path), so it would be a waste to reimplement that logic. A Path seems a good visual counterpart for a Track, so it makes sense to somehow translate tracks into paths; however, drawing a Path on screen requires that we have a Paint object as well.

Once again, multiplicities act as forces. We need to transform a Track into a Path, but since in the end we want to assign a different color to each track, it would be simpler to keep the Path and the Paint together. This basically suggests that we create a new class, which I called Ribbon, with these basic responsibilities:

- Transform a Track into a Path: I settled for a quadratic spline, which is fast enough and good enough for this simple app.
- Keep track of a color, in form of a Paint
- Draw itself on a canvas (more on this in a minute)

I like to preserve symmetry between structures if possible, so it should come as no surprise that along with a Ribbon class (mapping a Track toward presentation) I introduced a RibbonSet class, again with a simple set of responsibilities:

- Transform each Track from a given TrackSet into a Ribbon, assigning a default Paint (white).
- Draw itself on a canvas

Within that over-arching symmetry, there is an inner asymmetry[17]:

- Ribbon will receive a Track, transform it into a Path, hold the Path and forget the Track.
- RibbonSet will receive a TrackSet and hold that forever.

[17] If you're in for some deep thinking, you may want to read [Alexander2009] and ponder on how this alternate symmetry / asymmetry is related to his Fifteen Properties of Wholeness.

To understand why, it pays to think about multiplicities and lifetime once again.

- Tracks come and go. Well, at this time, they just keep coming (every time you land a finger on the Surface you get a new track). But at some point, when we'll get to trimming their tail, tracks will eventually disappear.

- The Trackset stays. There is only one TrackSet per surface, and it never goes away.

- Accordingly, Ribbons will come and go. Symmetry again (at a finer level) suggests that we keep just one RibbonSet per Surface, holding into a TrackSet and producing a newly updated set of ribbons on demand, every time we need it.

Symmetry is a very good adviser in this case, because remember: we had to make a tough decision on where to store the TrackSet, but if we assume to have a RibbonSet permanently bound to a TrackSet, that would make the next choice (where do I store the RibbonSet) very simple: in the same place where I'm storing the TrackSet. This is again something I've explored in the Physics of Software: things with the same multiplicity and lifetime are attracted to the same place (center), unless there are other forces to keep them apart. As you may suspect, adding the RibbonSet to the Surface will be a job for the Rendering aspect.

Before we get to see the code, I'd like to spend a few words on the decision to add the rendering method inside Ribbon (and consequentially, in RibbonSet) instead of exposing the Path and Paint data, and have the drawing done elsewhere (say, in a RibbonView class).

In fact, that was my initial design, and in some sense it mimicked the MVVM (Model-View-ViewModel) approach, with Track being the Model, Ribbon being the ViewModel, and RibbonView being the View. It wasn't *extremely* bad. But in the end, it was more complex (one more class, plus RibbonSet implementing Iterator), it exposed data that I can now keep private, and didn't have any real benefit (Ribbon is a presentation object to begin with).

The reason I initially settled for it was that, making a parallel with a possible OpenGL implementation, I knew that I could not have leveraged Path, thought that Ribbon would become more complex in that case, and somehow wanted to

keep the smoothing logic and the drawing logic apart. Which is right, but I was doing it wrong. The "right way" to do that, even in the OpenGL case, would have been to create a Path-like class doing the smoothing (and perhaps less concerned with presentation than Path) and then use that inside Ribbon (the real presentation object). This is a much better separation of concerns, and going with the popular thing (MVVM) was just obfuscating my initial reasoning.

So here is Ribbon:

The Ribbon class

```
package com.aspectroid;

import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.Path;

public class Ribbon
{
  public Ribbon(Track t, Paint p)
  {
    paint = p;
    path = new Path();

    Point lastPoint = null;
    for( Point pt : t )
    {
      if( lastPoint == null )
      {
        path.moveTo(pt.x, pt.y);
      }
      else
      {
        path.quadTo(lastPoint.x, lastPoint.y, pt.x, pt.y);
      }
      lastPoint = pt;
    }
  }

  public void displayOn(Canvas c)
  {
      c.drawPath(path, paint);
  }

  private final Path path;
  private final Paint paint;
}
```

And here is RibbonSet:

The RibbonSet class

```
package com.aspectroid;

import java.util.LinkedList;
import android.graphics.Canvas;
import android.graphics.Paint;

public class RibbonSet
{
  public RibbonSet(TrackSet ts)
  {
    tracks = ts;
  }

  public void displayOn(Canvas c)
  {
    refreshRibbons();

    for( Ribbon r : ribbons )
    {
      r.displayOn(c);
    }
  }

  protected Paint colorForTrack(Track t)
  {
    return defaultPaint;
  }

  private void refreshRibbons()
  {
    ribbons = new LinkedList<Ribbon>();
    for( Track t : tracks )
      makeRibbon(t);
  }

  private void makeRibbon(Track t)
  {
    Ribbon r = new Ribbon(t, colorForTrack(t));
    ribbons.add(r);
  }

  private static final Paint defaultPaint = ThickPaint.make(-1);
  private LinkedList<Ribbon> ribbons;
  private TrackSet tracks;
}
```

Ribbon is almost trivial, while RibbonSet requires a longer explanation: the code itself is simple, but the reason behind the small private methods is deeper than it seems (yeah, it's not a case of "extract till you drop").

When OOP was still relatively new, people entertained the idea that one could easily redefine the behavior of a class by subclassing and overriding. There was no clear notion of "designing a base class". Every class was supposed to be a fair candidate for subclassing. In practice, it turned out people had to design classes to be "good" base classes. For instance, a method consisting of a long sequence of statements is not a good candidate for a base class, as you only have the option of using the entire method or rewriting the entire behemoth in a derived class. Simple design patterns, like Template Method [JVHG1994], could help here. Generally speaking, whatever the paradigm you choose, *you can only replace parts that have been enucleated from the rest*.

Now, although AOP is claiming to be "quantification and obliviousness", it's hard to move away from the notion above, which is almost a law in the physics of software. Factoring out replaceable behavior is as relevant in AOP as in OOP (or FP, where you usually adopt higher level functions for this).

So, here is the deal: at some point, I want to introduce a coloring concern; it helps to have a clearly factored out *colorForTrack* method that I can intercept and redefine. I may also want to play with the actual list of Ribbons to render (say, to implement the aging concern that makes ribbons shorten and disappear), so it helps having a *refreshRibbons* method. I may want to change the way a Ribbon is created (say, to implement blurring); it helps to have a *makeRibbon* method that I can intercept. Said otherwise, AOP without AOD may easily turn into a monkey patching nightmare, requiring extremely sophisticated trace-based quantification or significant duplication of code. By the way: I didn't get all these small-scale details right in my head. It was easier and more effective to build the thing, look back, refactor as needed.

The *defaultPaint* is initialized to a thick white paint using a factory, which unlike what I see in most code, is not called PaintFactory but ThickPaint, because it makes thick paint objects. Passing -1 to get an opaque white looks ugly, but -1 is the simplest way to express 0xFFFFFFFF in a language without unsigned integral types (good job, Java). TickPaint itself is trivial, and of course depends only on library classes:

The ThickPaint class

```
package com.aspectroid;

import android.graphics.Paint;

public class ThickPaint
{
  public static Paint make(int color)
  {
    Paint p = new Paint();
    p.setAntiAlias(true);
    p.setDither(true);
    p.setColor(color);
    p.setStyle(Paint.Style.STROKE);
    p.setStrokeJoin(Paint.Join.ROUND);
    p.setStrokeCap(Paint.Cap.ROUND);
    p.setStrokeWidth(12);
    return p;
  }
}
```

Updating the Screen

A RibbonSet can display itself on a surface, by taking the current state of the connected TrackSet, bulding a set of Ribbons, and asking each one to display itself. Still, it won't do that unless someone tells it to.

We now have to make another choice, that is, when / how do we want to refresh the screen. We have basically two choices: when one of the tracks changes or periodically, for instance, 20 times per second. The first is basically a push mode: I move my finger on the screen, events get pushed, tracks changes, the display is updated. The latter is basically a pull mode: every nth of a second, we pull the current tracks, convert them to ribbons, and update the display.

While it may seem like the two choices are equivalent, or that the first might be preferable to avoid drawing the screen when nothing changes, in perspective it's actually the opposite. We *need* to update the screen even if nothing changes at the track level (that is, you don't move your finger, or you have already raised your finger). The idea of the app is that tracks will age, get shorter, and eventually disappear. That won't work with a push model, while it can be accomplished in a pull model. So pull model it is[18].

Once again, I will take this opportunity to create a couple of absolutely reusable, completely decoupled classes, which we'll mix in using aspects. After all, updating a view periodically is a common task, which shouldn't be tied to the current problem. Actually, we can generalize that a little more: executing a periodic task on the UI thread is a common problem; a specific instance of that is periodically refreshing a view.

So here is PeriodicUITask; it's not particularly nice, that is, there is quite a bit of Android stuffed in there. From the outside, however, it's relatively simple. PeriodicUITask is meant to be subclassed, hence the protected constructor; also, it won't start until you call play(), giving you the opportunity to set up any data member you need first.

[18] this is one of those cases where looking further than what we need "right now" makes for better choices and avoids invasive refactoring later on. Choosing between spending time thinking ahead or learning by doing is tricky, and despite what pseudo-gurus may have told you, there is a time for both.

The PeriodicUITask class

```
package com.aspectroid;

import java.util.Timer;
import java.util.TimerTask;
import android.os.Handler;

public abstract class PeriodicUITask
{
  public void play()
  {
    tick.schedule(new TimerTask()
      {
        @Override
        public void run()
        {
          handler.post(uiTask);
        }
      }, 0, rate);
  }

  protected PeriodicUITask( int rateMs )
  {
    rate = rateMs;
    tick = new Timer();
    handler = new Handler();
  }

  protected abstract void onTick();

  private final Runnable uiTask = new Runnable()
  {
    @Override
    public void run()
    {
      onTick();
    }
  };

  private int rate;
  private Timer tick;
  private Handler handler;
}
```

It's perhaps easier to understand PeriodicUITask by looking at a specific usage. Refreshing a view continuously is basically like playing a movie on that view, so I called that class Movie:

The Movie class

```
package com.aspectroid;

import android.view.View;

public class Movie extends PeriodicUITask
{
  public Movie(View v)
  {
    super(20);
    screen = v;
  }

  @Override
  protected void onTick()
  {
    screen.invalidate();
  }

  private View screen;
}
```

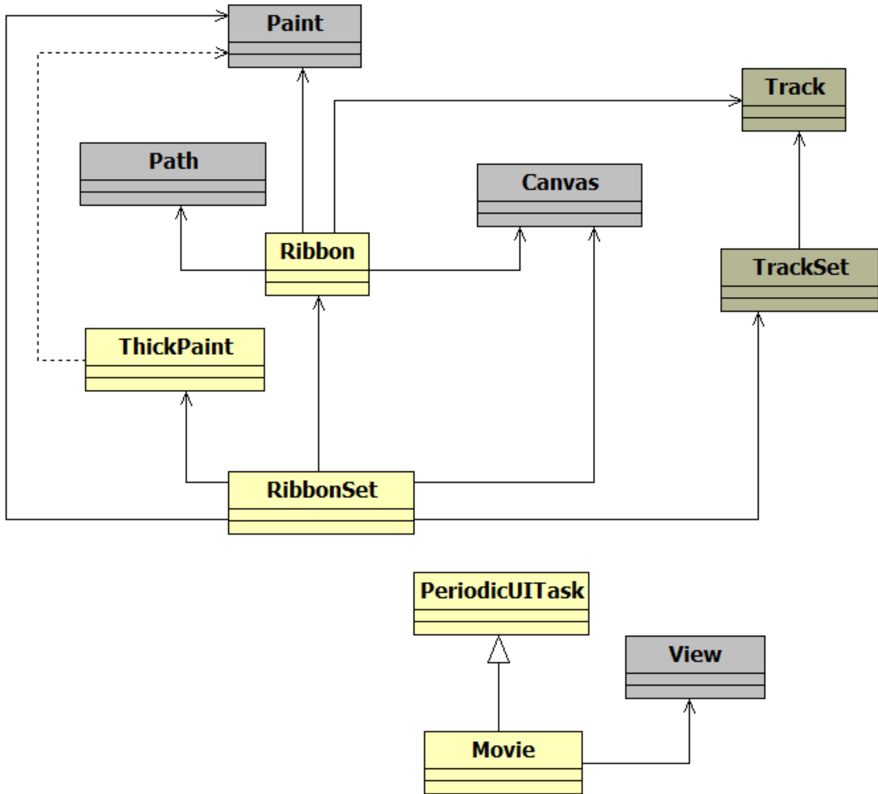
Movie is just setting up the base class so that when play() is finally called, the view is invalidated every 20ms, causing onDraw() to be called in the UI thread. That's when / where the real action will take place, as we'll see shortly in the Rendering aspect.

Movie can be grafted upon any view, as it's only using library methods: it does not require the view to implement any "special" interface.

Two new Islands

Here is how our code looks now: I've omitted the input-related cluster from this diagram (TouchPad / TouchView) because it's irrelevant for rendering:

Diagram 6



The upper cluster is growing: after all, the presentation object has to know the object it wants to present. Dependencies go in the right direction though.

Just like before, in traditional OOP this design wouldn't work. Nobody is creating a RibbonSet. Nobody is linking the RibbonSet with the right TrackSet. Nobody is attaching a Movie to a Surface, which is where we want to render the ribbons. Time for the Rending aspect to enter the scene.

Rendering

So, let's recap what we want to do:

- We want to store a RibbonSet inside each Surface.
- When the TrackSet is stored into the Surface, we want to pass it to the RibbonSet too.
- We want to attach a Movie to each surface; this will cause Surface.onDraw to be called periodically (that's what invalidate() does).
- When Surface.onDraw is called, we want the RibbonSet to render itself.

That translates quite nicely in AspectJ:

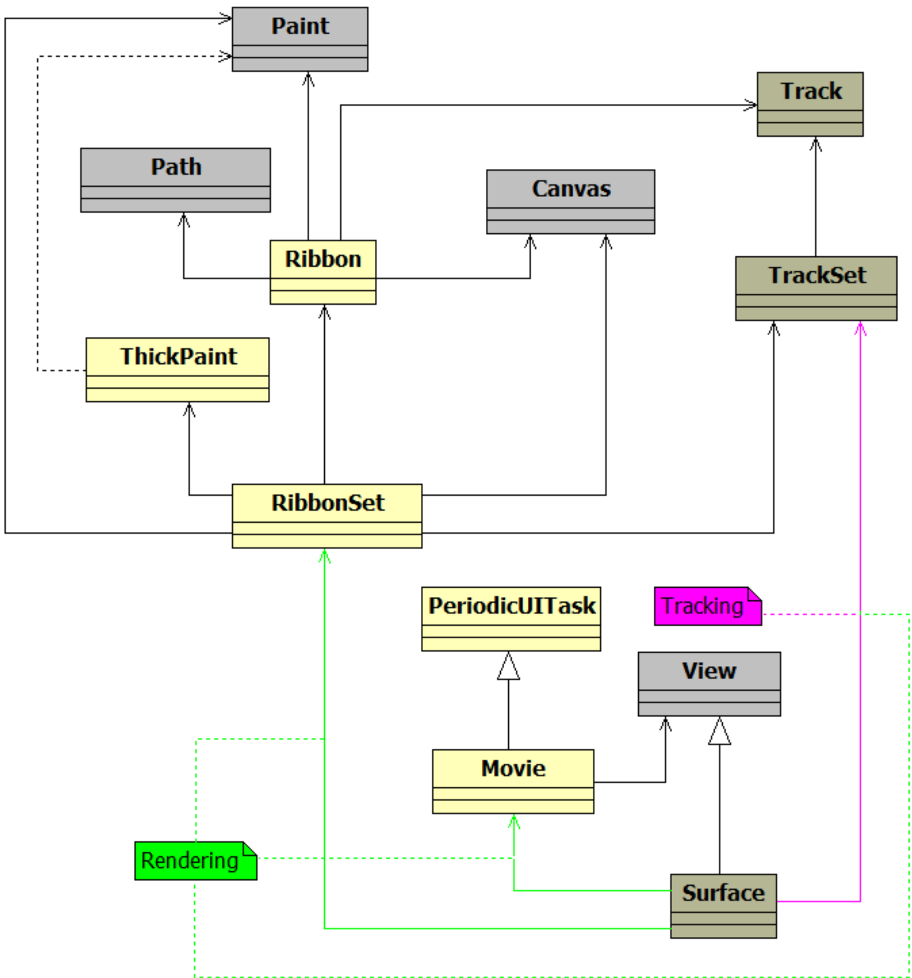
The Rendering aspect

```
package com.aspectroid.aspects;

import android.graphics.Canvas;

import com.aspectroid.TrackSet;
import com.aspectroid.Surface;
import com.aspectroid.RibbonSet;
import com.aspectroid.Movie;


public privileged aspect Rendering
{
  private pointcut setTrackSet() : set(TrackSet Surface.tracks);
  private pointcut surfaceConstructorCall() : execution(Surface.new(..)) ;

  private RibbonSet Surface.ribbons;
  private Movie Surface.movie;

  after(Surface self) : setTrackSet() && target(self)
  {
    self.ribbons = new RibbonSet(self.tracks);
  }

  after(Surface self) returning : target(self) && surfaceConstructorCall()
  {
    self.movie = new Movie(self);
    self.movie.play();
  }

  public void Surface.onDraw(Canvas canvas)
  {
    if( ribbons != null )
      ribbons.displayOn(canvas);
  }
}
```

Note the SetTrackSet pointcut: here I am intercepting the moment when a TrackSet is stored inside a Surface. In practice, this will happen inside another aspect (Tracking), but it doesn't have to. The call may originate from everywhere, and it would still work. However, Rendering is not really independent from Tracking, because Surface.tracks has been introduced by the Tracking aspect.

While I've been able to keep classes independent from aspects, there are still some dependencies between aspects which I haven't been able to remove completely. I have expressed this dependency with a green dashed line toward the pink Surface → TrackSet association in the following diagram, which shows the wireup between the two islands and Surface. Perhaps a better name for the Rendering aspect would be TrackRendering, as that's what it does, and it would make the dependency on Track[ing] more explicit.

Diagram 7



Guess what, it's finally working! Well, sort of. If you compile the code we have seen so far, you'll get something on screen which follows your finger[s] but it will be all white, the trail won't get trimmed as you move, and there will be no blurring.

The nice part is that we will be able to add all that, in 3 different steps, *without changing a single line of what we have already written*. What's even more interesting, we may choose to bring in the coloring, but not the trimming, or vice versa. It's really as compositional as we have been told many times OO software would be.

Show your colors

Time to make yet another choice, that is, how to pick a color for a given ribbon. We may use a random color, but the result might be aesthetically unpleasing.

I choose to adopt a fixed palette of 10 pastel colors. This logic is encapsulated in the Palette class:

The Palette class

```
package com.aspectroid;

import java.util.ArrayList;
import android.graphics.Paint;

public class Palette
{
  public Palette(byte transparency)
  {
    paints = new ArrayList<Paint>(ColorCount);
    int tr = transparency << 24;

    for( int c : Colors )
      addPaint(c | tr);
  }

  public Paint from(int index)
  {
    return paints.get(index % ColorCount);
  }

  private void addPaint(int color)
  {
    Paint p = ThickPaint.make(color);
    paints.add(p);
  }

  static final int[] Colors =
    { 0xFFFFFF, 0xD1F2A5, 0xFFC48C, 0xF56991, 0xBFCFF7,
      0x94C7B6, 0xFBCFCF, 0xA5AAD9, 0xF5F4D7, 0xFEFBAF };

  static final int ColorCount = Colors.length;

  private ArrayList<Paint> paints;
}
```

Palette takes a transparency value in the constructor; at this stage, I'm going to pass 0xFF, but transparency will prove useful when implementing the blurring aspect[19].

The idea of the palette is that I can ask for a color given an index, and it will return one of the fixed colors, using index modulo (number of colors). My original plan was to use the pointer index to color a track, so that every finger will get a different color, plus a given track would always get the same color, because it was generated by a specific finger. Remember that we create a new ribbon for a given track every time we draw the screen: therefore, whatever algorithm we choose to associate a color to a track, it must be stable, that is: it must always return the same color for a given track.

However, *there is no pointer index in the track*: it was lost in the input pipeline, because I had no use for it thus far. *This may seem like a minor point, but it's actually huge*. A common issue in pipeline architectures (whether you implement them using OOP, FP, or whatever) is that some data is added at stage X because you know that it will be used by stage Y, where Y > X + 1. That's really unpleasant[20], because it completely breaks the compositional nature of a pipeline: if you remove stage Y, you're then pushing useless data in the pipeline inside stage X. Worse than that, either you designed stage X with the idea that there will be a stage Y using that data, or you modified stage X to support stage Y. It's basically cheating.

The beauty of AOP in the context of a pipelined architecture is that I don't need to entangle stage X with stage Y. I can use an aspect to complement stage X with the necessary data for stage Y. That aspect will also wire Y inside the pipeline, so that consistency will be preserved: no aspect, no data for stage Y, but also no stage Y.

---

[19] This is yet another place where I'm faking a rational design process. I didn't have transparency in my first version.

[20] In fact, in [BMRSS96] when discussing the "pipes and filters" pattern, this is listed among forces: "*non-adjacent processing steps do not share information*". In practice, that's much less common than one might hope for.

Back to our problem, the coloring aspect will take care of:

- Adding an "index" member to the Track

- When a Track is built in the context of a *startNewTrack* call, set the index.

For simplicity, and for a more pleasant behavior, I choose to set the Track index to an incrementally growing counter, not to the pointer index. Note that I could have easily used the pointer index: it's a parameter of *startNewTrack*, and I would have had access to it. However, if you rapidly create multiple tracks using just one finger, under that implementation they would get the same color, while with an incremental counter they get different colors. I like this version better. What I like even more is that this decision is entirely encapsulated inside the Coloring aspect.

As you can see, I choose to add a new constructor to Track, taking the index as a parameter; that requires an around() advice, so that I can replace calls to the original constructors with calls to the new one. In a former implementation, I didn’t have a constructor. I used a before() advice, and set the index directly inside the advice. I refactored the code this way when I implemented the Blurring aspect, mostly to avoid some redundancy. In a way, the new code is slightly more coupled with the implementation of Track: it expects a constructor without parameters. The previous code was more general. It’s one of those small-scale trade-offs you have to make as you code, and my final choice is partially influenced by the “no longer than one page” rule, which pushed me slightly on the side of avoiding redundancy.

Associating an index to a track is only part of the job. We need to use that index at the right time, to change the color of the Ribbon that is rendering that Track. That's where one of those little methods in the RibbonSet class gets handy. All I have to do is to replace the execution of RibbonSet.colorForTrack(Track t), using an around advice, take the index from the track and use it to get a color from the palette. With that said, the Coloring aspect should be pretty clear:

The Coloring aspect

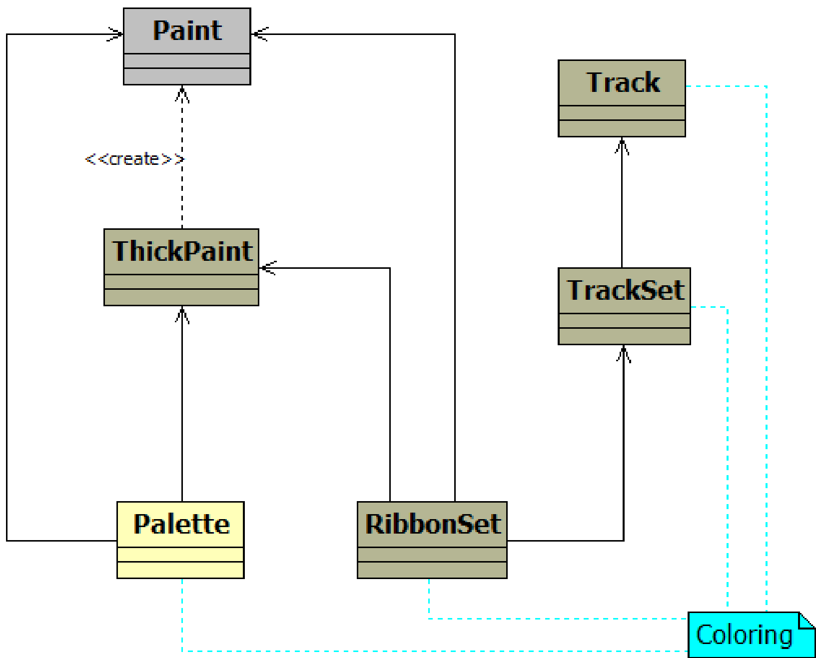
```
package com.aspectroid.aspects;

import android.graphics.Paint;

import com.aspectroid.Palette;
import com.aspectroid.Track;
import com.aspectroid.TrackSet;
import com.aspectroid.RibbonSet;

public aspect Coloring
{
  private pointcut trackConstructorCall() :
    call(Track.new()) &&
    cflow( execution( * TrackSet.startNewTrack(..)));

  private pointcut executingColorForTrack(Track t) :
    execution(protected Paint RibbonSet.colorForTrack(..)) &&
    args(t);

  private int Track.index;

  public Track.new( int trackIndex )
  {
    this();
    index = trackIndex;
  }

  Track around( ) : trackConstructorCall( )
  {
     return new Track(counter++);
  }

  Paint around(Track t) : executingColorForTrack(t)
  {
    return colors.from(t.index);
  }

  private static Palette colors = new Palette((byte)0xFF);
  private static int counter;
}
```

Once again, the most “complex” part is the definition of the pointcuts; the advice code is trivial, basically a couple of one-liners.

Note that the palette and the counter are static members of the Coloring aspect. Because of that, they will be shared among multiple Surfaces, if you create more than one on the same activity layout. Sharing the counter is intentional, so that colors cycle among all the surfaces.

Palette forms a little island, together with ThickPaint. I won't show you two diagrams this time, because that's the only new class that is being brought in together with the Coloring aspect. I'll show you directly the wiring, with only the involved classes:

Diagram 8



As expected, input-related classes are absent. Perhaps more interestingly, even the Surface and the Rendering aspect are absent, because all I need to do is set up the Ribbons properly, everything else is just fine. So, if you compile this version, tracks won't be trimmed, but every track will get a different color. Unlike Rendering, Coloring does not depend on any other aspect, which is more in line with my initial aim.

Lose your tail

Ok, coloring was simple. Trimming the tracks will prove just a little more complex, that is, we'll have to be careful with a couple of details.

So far, we're just adding more and more Tracks to the TrackSet. So we're eating memory, and whenever we draw, we draw everything, which loads the CPU/GPU quite a bit. Therefore, trimming is almost a necessity. The first choice, at this point, is which trimming logic we want to apply: based on length (we only keep the last N pixel or M inches) or based on time. In practice, it's not even a choice: we *must* trim the tracks based on time, because I eventually want the track to disappear after being in a state of "closed" for a while. In this sense, I have called this aspect Aging, because it makes it clear that it's about time, not space.

What is a decent algorithm for aging? Ideally, it would shorten Tracks, because we convert Tracks to Ribbons, and longer tracks take longer to convert. Also, at some point a track will become empty, and it would be better to simply remove it from the track list. Unless it's still open, that is: if you keep your finger down but you don't move it, the track will eventually become empty, but should not be removed as you'll be adding more points as soon as you move your finger again. In this particular implementation, I've decided to remove all points that are more than 600ms old.

Two main ingredients are needed here:

- We need to associate a timestamp with every point inside a track.

- We need to find a time and place for the aging logic to kick in.

I want to stress the first a little. In a more traditional architecture, at this stage we would:

- Modify Point to add a timestamp, or create a new class (derived from Point or containing Point) and replace all the occurrences of Point with that one.

- Go back to the input-related classes, and set the timestamp along with the coordinates.

Modifying Point is bad, in the same sense that adding an index to a track would have been bad. But then, so is replacing Point with another class everywhere, or modifying the input classes to set the timestamp. It's bad because is a pervasive change for the benefit of one specific aspect (Aging) that we may or may not want to have. That's how classes get fat, and often stay fat even if some data is no longer used. That's also why we occasionally switch to hashmaps or wish for a dynamic language, where adding data to an existing object is nice and easy (that was the Smalltalk way).

As I've observed before, AOP opens the statically-typed OO design space to new dimensions. This is, after all, the kind of cross-cutting concern that AOP is supposed to handle well, and indeed, it works fine here. Adding a timestamp from the Aging aspect is quite simple. We inject[21] a new field (timestamp) in Point, and we pointcut on Point's constructor to set the timestamp to *now*.

Of course, having the timestamp is useless if we don't make any use of it. Once again, one of those little private methods in RibbonSet will prove useful. Inside displayOn, ribbons are re-created from scratch every time, through a call to refreshRibbons. The default behavior is to iterate over tracks and convert each track into a ribbon. It would make sense to pointcut on refreshRibbons and, before invoking the default behavior, truncate the tracks based on the timestamp. After we have truncated the tracks, we can clean up any empty, closed track; then we can fall back to the default logic. Using the right advice type (before), invoking the default logic happens automatically at the right time.

With all that said, here is the Aging aspect:

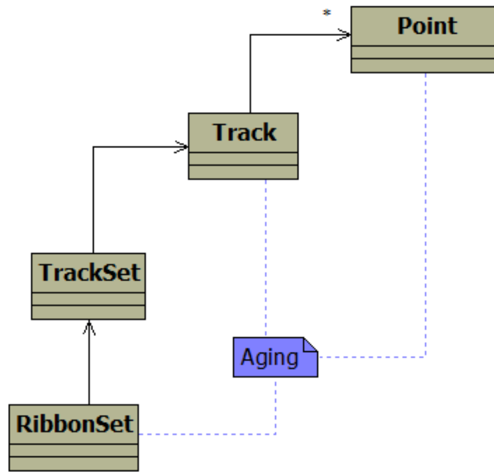
[21] The “correct” terminology would be “static introduction” or “inter-type declaration”, but I’m using “injection” informally in a few places in this text. I’m looking for expressivity and communication more than for rigor.

The Aging aspect

```
package com.aspectroid.aspects;

import java.util.Date;
import java.util.Iterator;
import com.aspectroid.Point;
import com.aspectroid.RibbonSet;
import com.aspectroid.Track;

public privileged aspect Aging
{
  public long Point.timestamp;

  private pointcut constructorCall() : execution(Point.new(..)) ;

  after(Point self) returning : target(self) && constructorCall()
  {
    self.timestamp = new Date().getTime();
  }

  private pointcut executingRefresh() :
    execution(private void RibbonSet.refreshRibbons(..)) ;

  before(RibbonSet self) : executingRefresh() && target(self)
  {
    long now = new Date().getTime();
    final Iterator<Track> i = self.tracks.iterator();
    while( i.hasNext() )
    {
      Track t = i.next();
      cleanup(t, now);
      if( t.isClosed() && !t.iterator().hasNext() )
        i.remove();
    }
  }

  private void cleanup(Track t, long now)
  {
    final Iterator<Point> i = t.iterator();
    while( i.hasNext() )
    {
      if( i.next().timestamp < now - 600 )
        i.remove();
    }
  }
}
```

Aging does not require any new class, so here is just the new wiring for the involved classes:

Diagram 9



There is an important issue here. Aging, in a sense, is taking on *two* responsibilities: Timing, in the sense that it provides Points with a timestamp, and Aging itself, which is more about trimming based on timestamps. I could have split these two aspects, and indeed, it would have been more appropriate. The final aspect we need to complete the app (blurring) is again using timestamps to "blur" the oldest part of the track. That makes blurring depending on aging, which is not strictly necessary. They could both depend on a more primitive Timing aspect. I didn't do so to make the storytelling shorter, but I'll get back to this in the next chapter.

Aging is also “richer” in logic, which is in a sense *domain logic*, than the other aspects we have seen so far. I’ll get back to this in Chapter 3.

Did you say simulator?

Before we move on to the last aspect (blurring), let's step back and introduce something I proposed from the very beginning: a simulation aspect, which can be applied instead of, or in parallel to, the touching aspect. After all, true compositionality is more than just swap and replace: we should be able to make them work at the same time.

Remember how the Touching aspect worked? There was very little logic in there: it simply wired a TouchView with the Surface, TouchView being a special kind of TouchPad. It would be nice to do something similar here, and simply wire a different kind of TouchPad, one that simulates touches instead of getting them from a View. We just need to get a few details right:

- Choose an algorithm. I choose to draw a circle, slow enough that the tail gets trimmed by the aging aspect.

- "Slow enough" means we want to simulate a touch event every N milliseconds. Also, remember that “real” touches happen in the UI thread, and we expect simulated touches to happen in that thread too.

Guess what, we already have a PeriodicUITask class which can do things every N millisecond in the UI thread. It's just a matter of dropping in a little math, and here comes the Simulated pad. I've set the tick to 100ms, slow enough for trimming to be visible.

Note that I don’t need to simulate touches at the android API level: having defined my own interface for that (TouchPad), I just need to make sure that the relevant methods get called. In this simple implementation, there is never a touch-up event. It’s just a never-ending circle.

The SimulatedPad class

```
package com.aspectroid;

public class SimulatedPad extends PeriodicUITask implements TouchPad
{
  public SimulatedPad(int w, int h)
  {
    super(100);

    this.w = w;
    this.h = h;
    r = Math.min(w, h) / 3;

    first = true;
    play();
  }

  @Override
  protected void onTick()
  {
    float x = (float) (w / 2 + r * Math.sin(a / 180 * Math.PI));
    float y = (float) (h / 2 + r * Math.cos(a / 180 * Math.PI));

    if( first )
    {
      first = false;
      onTouchDown(SIMULATED_POINTER_INDEX, x, y);
    }
    else
    {
      onTouchMoving(SIMULATED_POINTER_INDEX, x, y);
    }
    a = (a + 8) % 360;
  }

  protected static final int SIMULATED_POINTER_INDEX = 15;
  private boolean first;
  private float w;
  private float h;
  private float r;
  private float a;
}
```

A tricky part is setting a specific pointer index for the simulation. Remember that I slacked off in the Tracking aspect, setting the max number of pointers to 20. I'm slacking off here too, setting the simulated index to 15, which is ok for this purpose. A more refined implementation would discover the maximum number of touches supported by the device in the Tracking aspect, and use that. Then, in the simulating aspect, I would increase that by one (so to avoid any collision with real touches) and use that as my index. I kept it simple, but the more sophisticated version is definitely doable (with a more visible dependency between Simulating and Tracking).

With a SimulatedPad in place, the Simulating aspect is basically a one-liner (except, of course, all the declarative lines):

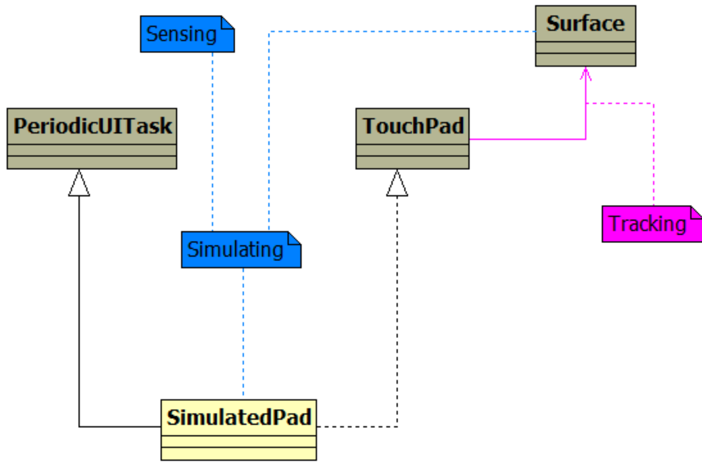
The Simulating aspect

```
package com.aspectroid.aspects;

import com.aspectroid.SimulatedPad;
import com.aspectroid.Surface;

public privileged aspect Simulating extends Sensing
{
  after(Surface self, int w, int h, int oldw, int oldh) :
  onSizeChangedCall(self, w, h, oldw, oldh)
  {
  SimulatedPad stp = new SimulatedPad(w, h);
  }
}
```

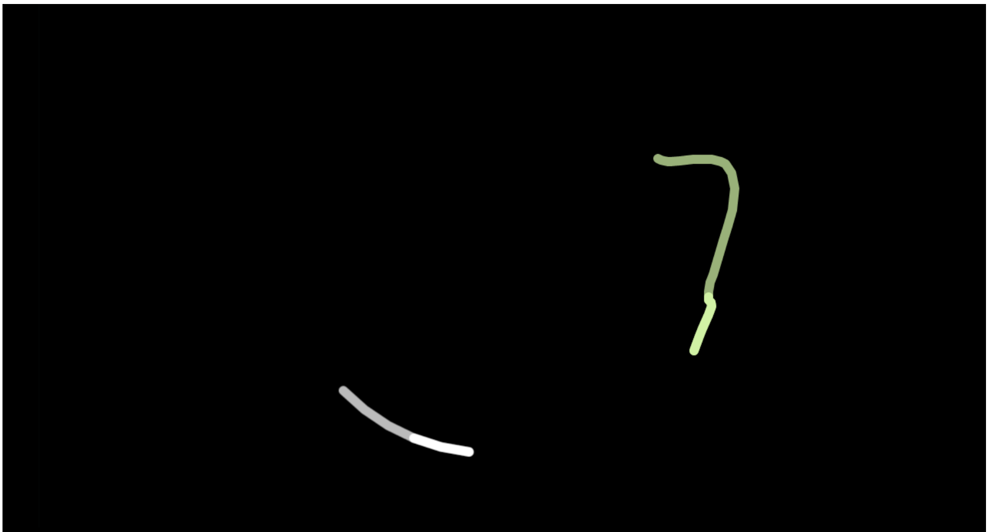
Like in the Touching aspect, I just need to create a TouchPad instance (stp is-a TouchPad) in a Surface context, and the Tracking aspect will kick in and wire it to the Surface.

Here is the wireup diagram; once again, SimulatedPad just adds to a small island (the input), while aspects do the wiring. For completeness, I added the Tracking aspect contribution to the picture.

Diagram 10



And here is some proof that it works. The gray circle is simulated; the green doodle comes from real touches:



The blurring aspect was on too, and as you see it will transparently work over the simulated touches as well.

Blurring

The previous picture shows blurring in action, as that aspect was compiled-in when I ran the simulation. The idea in itself is simple: the "old" portion of the ribbon should render in a blended nuance of the ribbon's main color.

How do we accomplish that without major changes to the existing code? A simple option would be to create two ribbons for each track, one representing the "old" portion, one the "new" portion. The blurring could be implemented using transparency, which is also why the Palette class takes a transparency parameter. With that in mind, it's mostly a matter of finding the right "hook".

Now I could just say that everything is fine, because the "perfect" hook happens to be already there, inside RibbonSet, waiting for us to use it. In practice, I have written RibbonSet that way because I knew it would give me the right hook. Let me show you the relevant portion:

RibbonSet snippet

```
private void refreshRibbons()
{
  ribbons = new LinkedList<Ribbon>();
  for( Track t : tracks )
    makeRibbon(t);
}

private void makeRibbon(Track t)
{
  Ribbon r = new Ribbon(t, colorForTrack(t));
  ribbons.add(r);
}
```

Note the way *makeRibbon* is declared and implemented. An obvious alternative would have been to return the newly created ribbon, and add it to the list inside the outer loop (in *refreshRibbons*). That would have been more in line, for instance, with the Factory Method pattern. However, that implementation would have been more limiting. If you declare Ribbon as a return type, you can only return one. If you declare a List<Ribbon> as a return type, you're definitely making RibbonSet aspect-aware.

I have chosen this form because it's the most aspect-friendly version. Of course, given the alternative implementation, we would still have a reasonable hook (*refreshRibbons* itself), at the cost of more duplicated code.

Is there a general lesson here? Probably yes. Paradoxically enough, in a time when functional programming is getting a lot of attention and mutable objects are generally considered "bad practices", it is exactly through mutability that I have obtained my "perfect" hook. As return types can somehow constrain the degree of freedom in a hooking aspects, controlled side-effects (mutating the object state) appear to be more flexible, and perhaps more in the spirit of an aspect-friendly structure. I certainly need more experience before I can say more about this specific point, but it has some logical integrity.

So, given the code as it is, we simply need to place an around() advice on *makeRibbon*, split the track into an "old" and a "new" one, build a first ribbon using the "old" track with a blurred color, and pass the "new" track to the original code, in place of the whole track, so that we will still get a regular ribbon for the new portion. Unfortunately, there are a lot of fine-grained consequences in doing so, and the blurring aspect ends up being coupled with the rest of the system in a stronger way than I wanted.

Even without looking at the code (yet) we can understand why we're going to have this coupling:

- Blurring requires the timestamp in Point to split the track, so it's coupled to the Aging aspect. As I've already mentioned, this could be somehow soothed by splitting Aging in Timing + Aging, but Blurring will still be coupled to Timing.

- Blurring is creating two Ribbons from one Track. It does so by creating two tracks, and then a ribbon for each track. Since the color of the track depends on the track index, it's useful if both tracks should get the same index. Still, the index is being added by the Coloring aspect, and this effectively ties Blurring with Coloring. As we'll see shortly, in my implementation the coupling with Coloring is even stronger (for simplicity + efficiency).

The Blurring aspect

```
package com.aspectroid.aspects;

import java.util.Date;
import android.graphics.Paint;
import com.aspectroid.Palette;
import com.aspectroid.Point;
import com.aspectroid.Ribbon;
import com.aspectroid.Track;
import com.aspectroid.RibbonSet;

public privileged aspect Blurring
{
  private pointcut executingMakeRibbon(Track t) :
  execution(private void RibbonSet.makeRibbon(..)) && args(t);

  void around(RibbonSet self, Track t) :
  executingMakeRibbon(t) && target(self)
  {
    Date d = new Date();
    long now = d.getTime();
    Track old = new Track(t.index);
    Track recent = new Track(t.index);
    for( Point p : t )
    {
      if( p.timestamp < now - 300 )
        old.addPoint(p);
      else
        recent.addPoint(p);
    }

    if( old.iterator().hasNext() )
    {
      Ribbon blurred = new Ribbon(t, colorForTrack(t));
      self.ribbons.add(blurred);
    }

    proceed(self, recent);
  }

  private static Paint colorForTrack(Track t)
  {
    return blurredColors.from(t.index);
  }

  private static Palette blurredColors = new Palette((byte)0xBB);
}
```

Is this reasonable? Have I just given up on compositionality of behavior? After spending some time pondering on this, I can safely say that yes, blurring is non-compositional in the sense that it requires Aging (Timing). Although my initial vision for the app was that of a compositional pipeline made of optional stages, talking about Blurring without Aging or at least Timing makes no sense, as the very notion of Blurring the “old” part requires timing. The coupling with Coloring is less stringent on a conceptual level, and it is more of a byproduct of the implementation. Let’s see why.

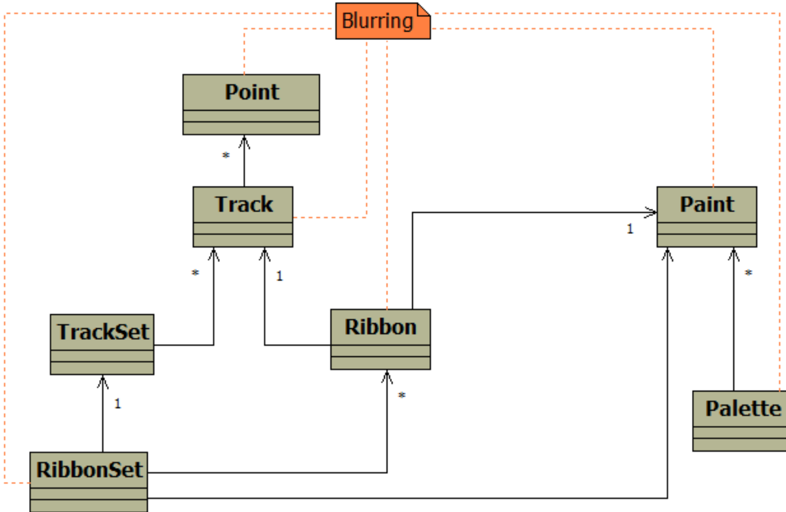
In my implementation, I’ve chosen to create two new tracks and add old and new points to the respective track. An alternative would have been to simply clone the track twice, and remove points from each one. That would have spared me the need to know about the index (at the cost of defining the right clone method for deep cloning in Track). Having the right index in the “new” track is required (when there is an active Coloring aspect) so that the call to *proceed* yields the expected results.

The need to have the same index in the “old” track is simply because it makes my implementation very simple: I use a more transparent version of the Palette used by Coloring, and use the same strategy to convert from a track index to a color. That’s simple, but in a sense ugly: I’m not just coupled with Coloring because I use Track.index: I know its coloring strategy as well (this is a case of entanglement as discussed in [Pescio2010]). Can this be avoided? Yes, easily, at the cost of a performance hit. I could simply call the old behavior, then take the color and make a more transparent version of it. This would work with or without the Coloring aspect, as the default ribbon still had a default color (white). Creating a new Paint object is inefficient, though. By giving up on simplicity, I could still implement a caching aspect to take care of that: it would be one of those non-functional aspects often seen in other works.

In the end: my implementation of blurring is simple, but unnecessarily coupled with Coloring. Better implementations are possible, and once again, this is an interesting opportunity to stop reading and get to write some code ☺.

This is the resulting wiring: as you see (and as was implied by the long import list) the Blurring aspect knows about quite a bit of classes:

Diagram 11

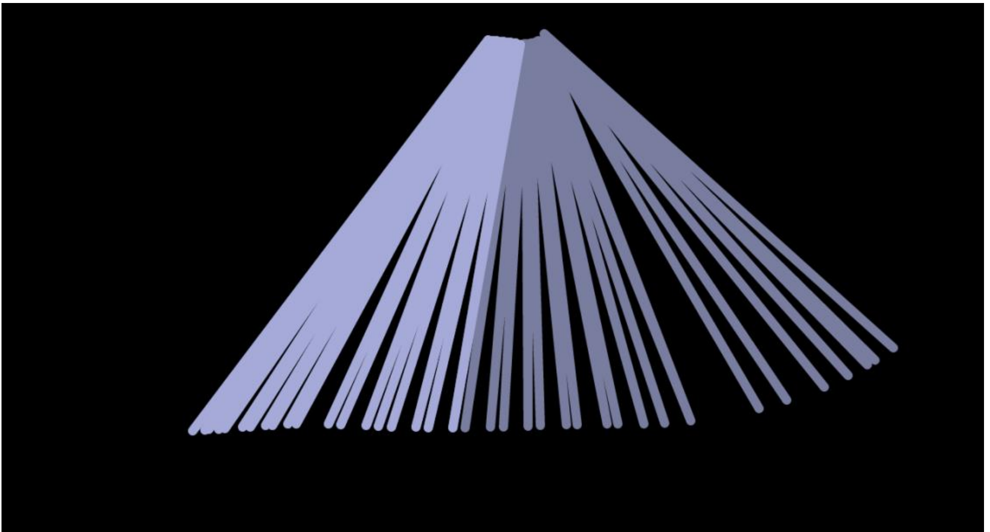


Like Aging, Blurring is not simply setting up some wiring: it contains what could be defined as “domain logic”. Again, I’ll discuss this more in depth in Chapter 3. I still deem necessary, in front of my previous critique of this solution, to point out a relevant aspect (pun intended). Even if we move away from aspects, and we fall back to another programming paradigm, the blurring logic will still involve splitting a track, creating two ribbons, choosing the right color, etc. This is part of what Brooks defined as “essential complexity” [Brooks1986]. We can partially fool ourselves and remove domain notion like tracks, ribbons, etc., and treat everything as a list or a map, but this won’t make the task any simpler or any more disentangled. The beauty of modeling Blurring as an aspect is exactly that: although it requires some predecessors in the pipeline, it can be removed entirely, and its implementation discarded in favor of a better one, without touching other code. The implementation suffers a bit from Java 6 lack of lambdas, and overall from a rather primitive container library, but this has little to do with aspects.

It's a bug… it's a feature… it's an aspect

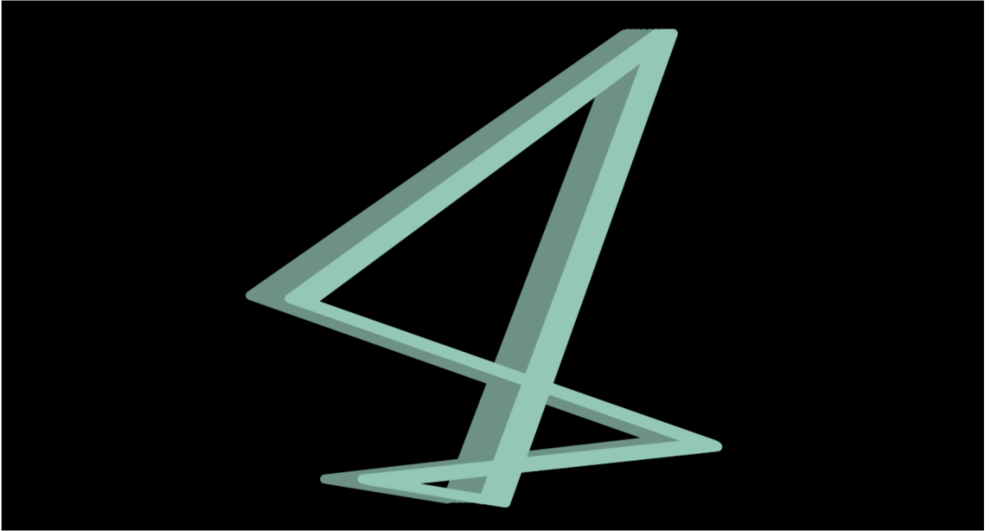
I would be happy to tell you guys that I never make a mistake while coding, but that of course would not be true. Actually, I occasionally make trivial mistakes. For instance, when I first implemented TouchView, I used *ptrId* in the ACTION_MOVE case (as in ACTION_DOWN), instead of *me.getPointerId(i)*. Still, the result was… interesting. In a sense, even better than what I wanted as the real behavior (doesn't happen that often, does it?)

In practice, with all fingers getting the same pointer id, you could keep a finger on a spot, and swipe another with a semicircular movement, getting a shape like this:



(see also the video on aspectroid.com to see this in action)

You could also use four fingers, and depending on the timing of the touch down, get a dynamic rectangle or a more connected shape like this:



It was a bit too good to let it go, but I wanted to fix the bug and proceed with the expected behavior, so that I could talk about different colors etc. The original behavior is also easier to explain. So, I had to let this bug go.

The nice thing is that I can bring the serendipitous behavior back with an aspect:

The Zeroing aspect

```
package com.aspectroid.aspects;

import android.view.MotionEvent;

public aspect Zeroing
{
  private pointcut getPointerIdCall() :
    call(public int MotionEvent.getPointerId(..)) ;

  int around() : getPointerIdCall()
  {
    return 0;
  }
}
```

I’m just intercepting the call to MotionEvent.getPointerId and return 0 for every pointer / finger. As it is not a “real” feature, I didn’t bother defining a more specific pointcut (I want to replace the pointer id only inside TouchView). We have already done so before, so if you want you can try that as the dreaded exercise for the reader.

The whole thing

The final result is composed of many small classes and a few aspects. The page format I've chosen is probably too narrow to make the following diagram justice, so I've provided a high-resolution version on aspectroid.com. Just click on the picture to follow the hyperlink.

Diagram 12

The two circles you see are not part of UML: they merely indicate a couple of relationships that would be problematic if expressed directly in the source classes. Having the touchpad containing Surface, or the SimulatedPad talk to Surface for that matter, would create a dependency in the wrong direction, from something abstract / reusable to something specific. It would also hinder our ability to swap Surface for something else (see next chapter).

The diagram *is* complex. I won't try to hide that. That's why I'm using diagrams in the first place: to deal with structural complexity the way I do it best (visually). I'll discuss the trade-off between algorithmic and structural complexity, as well as OOP / AOP and delocalized plans, etc., in the next chapter.

There is an interesting point to observe here: gray classes are my contact points with the Android framework. The top/right portion of the diagram is event-driven by nature. The role of those classes is to provide input. The "core" islands of the application are framework-agnostics, and are connected to the input via aspects. Classes on the left side are in touch with the Android framework again, this time to provide output (graphics). Here I haven't made any serious effort to isolate my code from Android. It could be another interesting challenge to tackle (see also the next paragraph on using OpenGL). As usual, there is a strong relationship between the shape of the software and many of its properties, like testability. I'll talk about this in Chapter 3.

Miles to go

Even a simple app like this is never finished, and it's easy to conceive a few additional features or variants:

- More precision: a MotionEvent doesn't simply contain the latest coordinates. It contains an array of coordinates which have been coalesced into a single event, the so-called *historical* coordinates. It would be interesting to introduce an aspect concerned with increasing resolution.

- Some instrumentation: especially if you add the above, it would be interesting to know how many points are being dropped because of the decimation strategy. Again, an aspect can take care of that.

- Some settings: the thickness of the paint, the palette, the decimation and blurring thresholds, etc. are all hard-coded at this time. It would be interesting to make them editable from a settings page. Doing this will expose a mismatch between the compositional nature of the architecture and the simplistic (but also simple!) notion of a single-page settings activity, driven by a single XML file, which is out of reach from AspectJ.

- A capture / replay feature. While in itself this is quite simple to implement using aspects, there are again some non-trivial nuances. It would make sense for a capture / replay feature to use some space on an *action bar*. I would expect the settings aspect to do the same. Once again, the simplistic (but simple) notion of a single menu resource, described in an XML, would prove at odd with the compositional nature of aspects.

- Having a knob to turn Zeroing on/off would be interesting as well, again from the perspective of sharing either configuration or the action bar with other aspects.

In a sense, the mismatch between the Android approach (leaning toward a single-XML description of things) and aspects has relatively little to do with AOP *per se*, and more to do with the fact that we're building a compositional application,

where features can be added or removed. Those “single files” artifacts tend to create an hourglass shape, with that artifact at the center, which is very non-compositional by nature. If you want to think deeper on this subject, you may want to read [Pescio2012a].

Chapter 3: Reflections

I started this booklet with a set of goals / hypotheses, so it makes sense now to look back and see if those goals have been attained, and those hypotheses confirmed. For various reasons, it's easier to start with the "OOP goals":

1. No controllers, managers, etc.
2. No stupid objects without methods.
3. Avoid copying data between "layers" to "decouple" classes.

I think I can honestly claim these goals as attained. Every class is small (I'll do a quantitative analysis further on) and contributes with some intelligence to the whole. There are no "managers". In no instance I'm copying data around between similar structures.

Let's move to the AOP goals:

1. Use aspects to separate application (not technological) concerns.
2. Keep aspects small (concerns in classes).

I think goal #1 has been attained, while I can only claim partial victory about #2: aspects are small on an absolute scale, but not on a relative scale, that is, compared to classes. On one side, the AspectJ syntax is rather verbose, but on the other side, a couple of aspects ended up containing some kind of logic that I couldn't easily allocate to classes. I'll get back to this later on.

It's also interesting to review my guidelines:

1. Use classes to define behavior (and data).
2. Use aspects to compose behavior (and supplement data).
3. Never have a class depending on an aspect.
4. Minimize dependencies between aspects.
5. Keep classes and aspects small.

#1 and #2 are strictly related to the AOP goals above. #5 was intended in absolute terms, so I would say it has been respected. #3 has been followed strictly. #4 is a bit more controversial, as I do have a few dependencies between aspects, and

“minimizing” is a bit too informal for an objective discussion (unfortunately, I don’t have a better frame of reference that would make it more formal).

It would pay to spend a few more words on the idea that behavior should be in classes, not in aspects. It’s not a made-up principle based on personal taste. It has a few strong motivations, all rooted in the fact that aspects (in AspectJ) can only attach behavior to other entities, and can’t stand on their own. Therefore:

- Behavior in classes can be unit-tested with the class as the unit. Behavior in aspects requires a larger unit.

- Behavior in classes can be reused more easily, in some cases even in an OOP-only project. Reusability has never been a goal for aspects, and although there is progress on that side (with abstract aspects, generic arguments, etc.) they tend to be more concrete.

This also explains why I’m aiming for classes with no dependencies on aspects. They are reusable in OOP-only contexts, and tend to create smaller units for testing and reuse. They also make their *essential* dependencies more manifest, which is not a bad thing.

Let’s move back to the initial goals. I had a couple of “general” design goals too:

1. Align the architecture with the nature of the problem
2. Allow for *extension and contraction* of application-level concerns

#1 is informal / complicated. I’ll try to discuss it in a separate paragraph below. #2 has been partially (or largely, depending on perspective) attained. The architecture itself is largely compositional, and can be easily moved up and down in the concern scale. Still, it’s not perfectly compositional: as we have seen, there are a few hard-to-remove dependencies among aspects, which prevents *arbitrary* composition (no blurring without aging / timing). I have already discussed a few possibilities for improvement, but I’ll get back to this and discuss yet another alternative in a following paragraph.

Objective evaluation (or lack thereof)

Warning: this section is a bit weak; thankfully, it's also short.

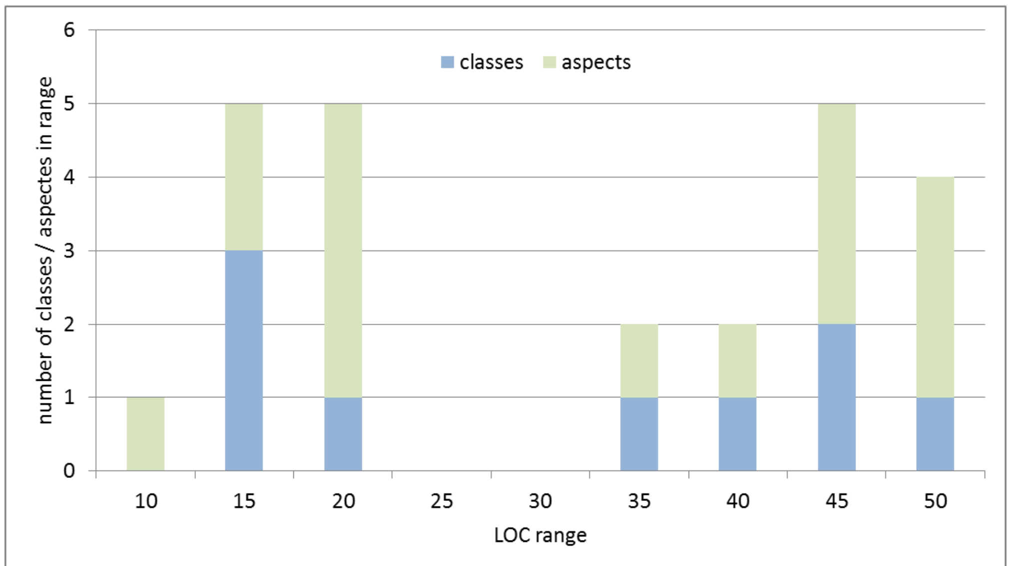
Talking about design quality is always complicated, because there are relatively few objective measures of quality. Some tend to equate adherence to principles with quality. Others confuse familiarity with quality. In the end, the biggest quality indicator is ease of change, and in this sense, the evolution of the app from a monochrome, continuous trail to the final result gives me an informal measure of quality that I consider rather good, although not perfect.

Still, we could also try a few well-known objective metrics and criteria and see how we're faring. For instance, using a well-known metrics suite (Chidamber and Kemerer, see [CK1994]), we could take a look at some simple measurements, like LOC (lines of code) CBO (a measure of coupling) and LCOM (lack of cohesion). My hypothesis: CBO and LCOM should be small for classes and still moderate, but higher, for [some] aspects. This makes sense as aspects here are gluing together different parts, and contributing members to different classes. In practice, one requires a tool to do the calculations, and the tool has to understand aspects as well. That's where things broke. I found only a couple of (mostly abandoned) tools which could deal with aspects. One didn't even build, the other didn't run. I fixed the latter but soon discovered it choked on my project, because it used an old version of AspectJ inside which couldn't parse my source files. I brought in the new AspectJ jars, but guess what, the tool was strongly coupled with the previous jars, and way too many classes weren't there anymore, with no obvious replacement in sight. After a while, I decided to let it go, at least for this time. So, sorry, no cool metrics, not even sanitized LOC.

I can still offer raw LOC, that is, lines of code including empty lines etc., as formatted inside my IDE. It's not much, but it's still interesting to look at absolute values and at the distribution.

Class	LOC
TouchPad	8
Surface	13
RibbonsActivity	14
SizedList	17
ThickPaint	19
Movie	20
Point	20
Palette	35
Ribbon	36
SimulatedPad	41
PeriodicUITask	42
Track	44
RibbonSet	46
TrackSet	47
TouchView	48

Aspect	LOC
Touching	12
Sensing	14
Simulating	14
Zeroing	16
Rendering	35
Coloring	37
Aging	45
Tracking	45
Blurring	46



Distribution of LOC for classes and aspects

So, classes range from 8 to 48 LOC and aspects from 12 to 46 LOC; overall, there are 714 LOC in the entire project (aspects + classes). The distribution is slightly more interesting: there is one class in the [0,10] LOC range, 3 aspects and 2 classes in the [10-15] LOC range, etc. It has long been known that the distribution of LOC tend to follow a power function even in relatively small code bases (see for instance [Hatton2009]; there is a lot of literature on the subject), but that does not seem to happen here. Of course, I had an artificial constraint (fitting on one page) and I tend to favor small classes anyway, so it's hard to say if aspects had anything to do with this.

These numbers alone don't tell much of a story: it would be more interesting (even in presence of more meaningful metrics) to compare them with a similar project written in a more "traditional" style. With no claim whatsoever to be "scientific", I'll try to do just that. Among the android API demos there is indeed a simple project (graphics) which contains a remotely similar app (FingerPaint). However, FingerPaint:

- Is not multitouch
- Does not trim or blur tracks, therefore there is no animation
- Has a few extra features (select color, draw embossed) which all resolve in using a different pen when drawing.

The whole graphics demo is a bit tangled, but after removing all the unnecessary classes (for the sole purpose of the FingerPaint demo) I have shrunk it down to a mere 3 classes + 1 interface, contained in just two source files. As you might guess from that, there is no notion of "domain" objects like Tracks inside FingerPaint. Everything is basically handled inside the user interface. That should allow for shorter code, and in fact, here are the numbers (excluding the long copyright comment):

Class	LOC
FingerPaint + MyView	195
ColorPickerDialog + OnColorChangedListener	217

For a whooping total of 412 LOC.

In practice, FingerPaint is more equivalent (as a feature set) to this subset of classes / aspects:

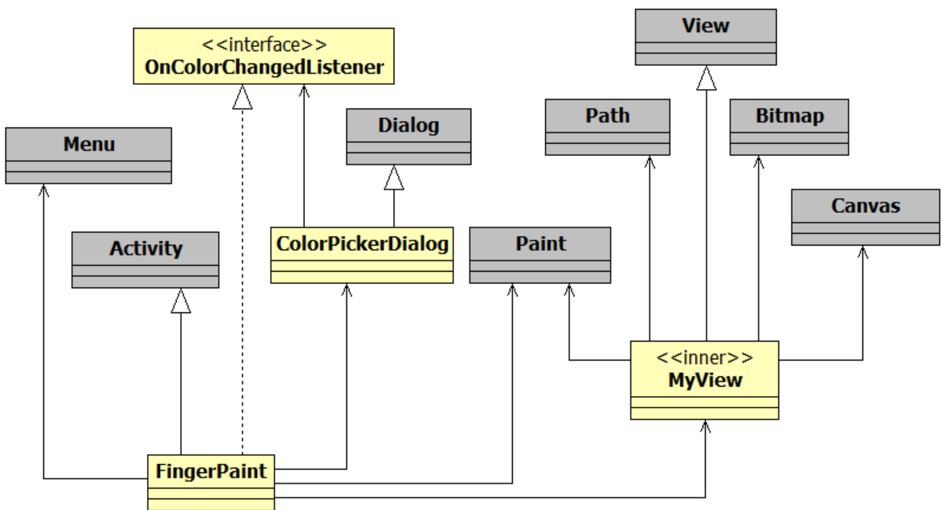
Class	LOC
TouchPad	8
Surface	13
RibbonsActivity	14
SizedList	17
ThickPaint	19
Movie	20
Point	20
Palette	35
Ribbon	36
PeriodicUITask	42
Track	44
RibbonSet	46
TrackSet	47
TouchView	48

Aspect	LOC
Touching	12
Sensing	14
Rendering	35
Coloring	37

For a total of 507 lines, which I would claim is not much of a difference.

From a structural perspective, here is FingerPaint:

Diagram 13



Everything in gray is in the Android library, so as anticipated we only have 4 classes here. Of course, the issue with this design is that it doesn't scale well. We have set up a monolithic gravitational field, and code will be attracted either to FingerPaint or MyView. Making MyView an inner class of FingerPaint further confuses the picture, by making less clear why a few things are dealt with in one or in the other.

The common remedy would be to move some of their logic into a "controller". Note, however, that a monolithic controller would only solve part of the problem. A monolith does not accept extension and contraction: you can only change it (usually making it larger). This, however, is more of a subjective than an objective observation, so, I'll leave it to further down in this text.

A "real" MVC structure would also require a Model, therefore something along the lines of my Track / TrackSet. This may again take away some part of the logic, and in a sense bring the design closer to mine, but at the same time will also increase the LOC, bringing them closer too.

Bottom line: the android demo wasn't created to show how to properly structure an app (I hope), just to show how to use a few APIs with a "quick and dirty" approach. It is still interesting to observe that there is very little advantage, from a LOC perspective, to adopt the dirty style. Bringing in some other metrics (like cyclomatic complexity or WMC) would further prove that the dirty style is indeed dirty (for instance, much harder to test). A "traditional" monolithic controller, with or without a model, will only make the LOC counter worse.

Dependencies

A little disappointed by the lack of tools on the metrics side, I've also tried to gain some objective insight by creating a Dependency Structure Matrix (also known as a Design Structure Matrix), a popular tool to evaluate modularity [SJSJ2005].

A DSM shows dependencies between modules (here: aspects and classes). A pink square indicates that a row depends on a column: for instance, OnTouchListener depends on MotionEvent.

	Paint	View	Point	Canvas	Path	MotionEvent	OnTouchListener	Surface	ThickPaint	PeriodicUITask	Palette	Movie	Track	TrackSet	Ribbon	RibbonSet	TouchPad	TouchView	SimulatedPad	Sensing	Touching	Simulating	Zeroing	Tracking	Rendering	Aging	Coloring	Blurring
Paint																												
View																												
Point																												
Canvas																												
Path																												
MotionEvent																												
OnTouchListener						■																						
Surface		■																										
ThickPaint	■																											
PeriodicUITask																												
Palette	■								■																			
Movie		■								■																		
Track			■																									
TrackSet													■															
Ribbon	■			■	■								■															
RibbonSet	■			■					■					■	■													
TouchPad																												
TouchView						■	■																					
SimulatedPad										■							■											
Sensing								■																				
Touching								■										■		■								
Simulating								■											■	■								
Zeroing						■												■										
Tracking								■						■			■											
Rendering								■				■				■								■				
Aging			■										■			■												
Coloring											■		■	■		■												
Blurring	■		■								■		■		■	■											■	

You can basically read the matrix by row (so you'll see all the components that the component on that row depends on) or by column (so you'll see all the components that depend on the component on that column).

There are many tools available to analyze a DSM, but once again, they didn't prove useful. The matrix is already in a sort of "ideal" state: there are no cycles, it's lower-triangular and there are basically no clusters to be found. A matrix like this is called "layered".

I have ordered classes so that library classes come first (gray), then my classes (green) then aspects (blue). It is important to single out (or remove) library classes, because we can expect them to be stable. Therefore, having many classes depending on a library class is not overly bad (for instance, several of my artifacts depend on Paint). I still prefer to show them in the DSM because it helps to answer a few questions (like "what if I want to port this app to …").

What can we see here?

- There are no "change propagators", that is classes that depend on a large number of classes and have a large number of classes depending on them. Ribbon and RibbonSet almost qualify (within the limits of a small number of dependencies) but they depend mostly on library classes.
- The middle – right area is "forbidden" as it represents a class depending on an aspect. In fact, it is empty.
- The bottom – right area is "discouraged" as it represents an aspect depending on another aspect. Besides the two occurrences of inheritance (which is ok) we find two known dependencies here.
- Several aspects depend on Surface. I'll talk about that later on.
- Overall, the system is in a very good shape. Not much of an insight.

In the end, the "objective" evaluation didn't provide much learning. Therefore, I'll move back to a more "subjective" style and ponder a little more on the design style of this app.

“Design style”

Classes and Interfaces

Compared to the average OO application, I’ve created just a few interfaces (one: the TouchPad) and abstract classes (one: PeriodicUITask; two if you count the abstract Sensing aspect). Still, the design is quite flexible / extensible and with no undesirable coupling: in fact, classes tend to form small islands.

I consider the removal of those abstractions *a good thing*. I know, years of OOP have taught us that the key to extendibility and to decoupling lies in adding the right dose of interfaces and abstract classes. The practice of OOP also shows that in many cases those abstractions do not match the language of the problem, so they’re hard to name properly. They’re here for *non-functional* reasons, and therefore are part of the solution, not part of the problem.

Using aspects to decouple classes allowed me to remove those unnecessary abstractions from the primary decomposition, which is now very close to the structure of the problem. This property has been observed before, and the interfaces which can be removed in this way have been called “aspectizable interfaces” [TC2005]. As I said in Chapter 1, aspects are a disentangling construct. They allow decoupling via separation, instead of via abstraction. Therefore, we don’t have observers, decorators, or any of the more classical OOD traces here.

Reusable classes, specific aspects

Basically all the classes I’ve presented are reusable elsewhere, some more than others (with the Ribbon + RibbonSet pair as the least reusable, because it’s more closely tied to the problem I’m solving). Aspects, as I’ve used them, tend to be very specific instead, and hardly reusable elsewhere[22]. This may seem odd if you’ve only seen aspects like logging or exception handling, where extremely general pointcuts and largely problem-independent semantics allow them to be widely reusable. Still, when used primarily for wiring concerns implemented in

[22] With the exception of the Sensing hierarchy, which could be made reusable – see below on the role of Surface.

classes, aspects tend to become specific (hardly reusable, as they name specific classes and methods) while classes become more independent.

It is probably possible to make some of the aspects slightly less specific by adopting a style based on *crosscut programming interfaces* (XPI for short) or adopting some ideas from generic programming. See below for a few more ideas on this side.

"My" use of aspects

Most of the code we have seen follows a simple intuition: *that I could use aspects to remove dependencies from classes, instead of abstracting them*. This is a rather powerful notion, which deserves a few more words.

Consider a similar design, carried out with OOP only. The central class would become Surface: in my design, Surface ends up containing TrackSet and RibbonSet, receiving input and dispatching Points to the TrackSet, receiving refresh requests and dispatching them to the RibbonSet. In practice, the Surface becomes a Presenter in a Model-View-Presenter architecture. Still, Surface as a class is empty. There is a large gap from the structure of Surface *before weaving* and the structure of Surface *after weaving*.

Along the same lines, Ribbon and RibbonSet are only concerned with transforming a Track/TrackSet into a monochrome Path and plotting that to a Canvas. There is no code in place for sake of extension. There is no "anticipated complexity" there.

In an OOP setting, one may instead create a small number of interfaces to provide hooks for extension. For instance, we may want to set up a transformation chain between the ribbon and the canvas, so that we may apply coloring, trimming, blurring, etc. We may then use a dependency injection technique to wire up concrete instances. Dependencies would still be there, but "abstracted" into an interface. There are a couple of problems with this approach:

- It's "anticipated complexity", not just in the sense that we must foresee the need for the filter and put an interface there, but that the interface

will still be there even if we shrink down the application and don't use any filter.

- The actual interface for the filters is hard to specify in a flexible, yet type-safe way. What is needed for Coloring is very different from what is needed for Blurring. Remember: Blurring created more Ribbons where we just had one. It's hard to get the interface "right" from the very beginning, but that's only part of the problem. The real problem is that the interface you get in the end won't shrink down easily if you remove the blurring requirement. It will keep that form because, at some point, blurring existed. Note that this would be true for a (type-safe) functional design as well.

Using aspects, I haven't merely abstracted those dependencies from a concretion to an interface. I have removed them from the main artifacts (classes). If I remove the aspect which needs some dependency, the extra complexity is removed with it. This is a property which is rare to find in OO-only code, but also in generic programming or functional programming, because the types involved tend to get more complex to accommodate the buffed-up cases, and won't shrink down when you don't need those cases.

Of course, life is different in highly dynamic languages, like Smalltalk or javascript. You can just pop a class (or an object) open and add members as needed. Those languages afford the designer with a larger solution space, at the cost of losing type safety (for those who believe in the value of type safety – it's an age-old debate).

From this perspective, I am using aspects to grant myself the flexibility (in terms of modular design) of dynamic languages, while still keeping the code type-safe. Indeed, you'll see no casts in the entire code (with the exception of a couple of casts to float in simulated pad, which are more of a byproduct of Java than anything else).

To conclude, and to say something big: in [Pescio2012c] I tried to define an object-oriented structure in a way that wasn't dependent on implementation

technologies like classes and polymorphism (you probably need to read the post to understand the perspective). I came up with this (slightly edited here):

Given a set of meaningful changes occurring over time, an OOS is one that minimizes the total size of the artifacts you have to update, encouraging the creation or deletion of entire artifacts instead.

Within that perspective, I would say that aspect-like constructs are a necessary extension to the traditional object-oriented constructs if we want to accomplish that goal within the boundaries of type-safe languages.

"Traditional" aspects (technological)

Perhaps because of my initial outlook, I haven't found many opportunities to apply the more traditional, technological aspects. I've mentioned the caching of colors as a possibility. Threading has also been considered an aspectizable concern, so one may think that I could use an aspect, instead of inheritance from PeriodicUITask, to make the SimulatedPad execute in the UI thread. I've mixed feelings about that choice, because it's not something "optional" that we may want to remove. SimulatedPad *must* send touches in the UI thread. It would be more interesting to separate the computational task from the touch simulation logic, but of course that was just a simplified example. Wiring a calculation strategy using an aspect would be more in line with the ideas in this booklet.

Yet another technological aspect could be the adoption of android.graphics as opposed to (e.g.) opengl. I'll get back to this while discussing testing and testability, but overall, an OO solution may work well here.

A technological, cross-cutting, pervasive concern in this app is "android", that is, in many places my code is strongly influenced by the platform. I didn't even try to remove this dependency, as it would not be representative of what I'm usually doing while writing Android apps. It doesn't seem feasible, even with AOP, without a lot of complexity creeping in, and with dubious advantages. Given that requirement, it would be better to use a cross-platform development environment from the very beginning.

Monolith vs Cooperating independent pieces

The final architecture is composed by many independent pieces. This type of architecture has been criticized before[23], usually accused of "overengineering" or simply of being hard to understand. I could just say that this is the OO way, but let's dig a bit deeper. We don't do many real-world experiments in software engineering, but occasionally, we do. Back in 2004, Erik Arisholm and Dag Sjøberg conducted an interesting experiment [AS2004] comparing the ability of novices and expert programmers in understanding and changing software written in the "centralized controller" style vs. the "delegated intelligence" style. Unsurprisingly (for me) they discovered that novices found the centralized style easier to understand, while experts found the delegated style easier to follow and adapt.

As I said, this didn't surprise me. Novices are usually coming with the mindset that *what happens next has to be found next*; that is, they are ill-equipped to reason and act in presence of so-called *delocalized plans* [LS1986]. Looking at my own experience, I contacted the authors and asked if they had tried a simple change to their experiment: add some diagrams (class diagrams, sequence diagrams) to the code, and see what happens. My theory was that diagrams would partially close the gap between novices and experts (remember: this was a comprehension / maintenance task, not a design task). This is an excerpt from Erik's answer: "*I agree! […] we have now replicated the experiment on three separate occasions […] where the subjects also received use-case, class and sequence diagrams. As you expected, the preliminary results suggest that the novices have much less difficulties with a delegated control style when they have access to such UML documentation*".

Now, I understand diagrams and UML have got a bad reputation in the past few years. I also hope you have seen how lightweight my usage of UML has been in this booklet. In presence of highly structured software, with several small parts and decentralized intelligence, diagrams help. A significant part of our brain is engaged in visualization. It's as simple as that.

---

[23] See, for instance, the 3-part discussion on "living without stupid objects" on carlopescio.com.

Naming

I've always considered naming important. The naming of classes and methods is more than a way to convey intent to others. It's also a sounding board for our own reflections, as we progress in our understanding of the problem (see, again, [Pescio2006]). Naming "conventions" are commonly adopted within a community (paradigm / language adopters) as a way to encode common notions, knowledge or understanding. Unfortunately, several naming conventions are rather at odd with the paradigm they're supposed to support: for a few examples applied to OOP, you can see my relatively popular post "Your coding conventions are hurting you" [Pescio2011b]. While writing aspect-oriented code, it was only natural for me to ask myself "what makes a proper name for an aspect?"

There are, of course, some examples in literature. In [Wampler2004], the author considers aspects to be adjectives, and therefore suggests an –able prefix, like Draw*able*Shape. This approach stems from his usage of aspects primarily as a device for subclassing, not dissimilar to an Inversion of Control Container. In that case, Shape is the noun, and DrawableShape is prefixed with an adjective modifying the noun (and "adding" the ability to draw itself).

Although reasonable, I've found that approach limiting:

- It works best when the aspect is dealing with a single class, as the name of the aspect contains the name of the class. This is at odd with the cross-cutting nature of aspects.
- It is skewed toward a specific usage of aspects (for subclassing) that won't fit well, for instance, with my usage in this booklet.

Of course, a reasonable convention cannot be formed unless we first agree on the idea we want to convey. For instance, there is a widely accepted notion that classes should be named as *nouns*, as they represent a category of objects. Aspects might be more elusive, because:

- In a sense, they are things (which suggests nouns). I'd like to say "the <noun> aspect".

- In a sense, they often express an action or intention. Remember the quote from Cristina Videira: *"what should happen to whom and when"*. An aspect should "happen", which suggests a verb.

Luckily enough, grammar comes to help, in the form of the –ing suffix, which can be used to form both a noun and a past principle or gerund (see also the notion of adverbial phrase). So I can say things like "*coloring* adds a track index" (usage as noun), or "*coloring* the ribbon" (usage as verb).

Following this idea, I've tried to adopt this convention throughout the entire booklet. I can say it served me well, forcing me to think about aspects in the "right" perspective, with one exception: the Simulating aspect. Honestly, I would have usually called it Simulation, not Simulating; I did so for sake of consistency.

On the other hand, the –ion suffix plays a similar role in natural language, providing a way to form a noun or an action. In the end, my current view on this matter is that it's probably better to adopt *both* –ing and –ion as plausible suffixes when forming an aspect name. This will allow some very reasonable name like "configuration aspect", while staying in line with the notion of an aspect as both a noun and a verb.

Insights (or: things I've learnt by building it)

I started this booklet as an experiment. I had some hypothesis, but I also knew from experience that I would learn more by actually *building* the thing[24], and even more by writing about it. As they say, hindsight is always 20/20, so quite a few things make sense only when you look backward and ponder on what you have done.

Reflections on the Process

As I mentioned in the beginning, the design / development process was slightly less "rational" than it may seem. I had a vision, a set of goals, and as I started to put that in practice I readjusted both fine-grained details and the coarse-grained organization. Fine-grained refactoring was mostly needed to provide the "right" hooks for aspects, while maintaining a clean decomposition in meaningful methods. Coarse-grained reorganization took place mostly when I discarded some alternatives (e.g. the viewmodel-like class) or moved intelligence around, mostly by pushing it down to the class where it belonged.

As I believe it's often the case for experienced developers, I didn't really go through any "formal" refactoring stage: especially for the fine-grained changes, it was a continuum, whereby I would add an aspect, see the opportunity to create the "right" hook, provide the hook, and so on. I like the idea that while doing so, we decompose methods into semantically meaningful micro-behaviors that can be replaced or enriched, much like we would do in an inheritance-based design.

In retrospective, I don't feel like most of the theoretical aspect-oriented design approaches would have worked well for me. The notion that I could keep some sort of "main" code untouched while adding aspects (the "obliviousness" approach as defined in [FF2000]) would have been very hard to practice. As others have discussed, it would have led to extremely complex pointcut definitions, because without hooks at the right granularity you need sophisticated cflow statements to capture the exact points. Those statements are also fragile.

[24] As Fred Brooks said, *"Scientists build to learn; engineers learn to build"*. I must say I approach projects like this aiming for both. I build to learn, and I'll use what I'm learning to build more (and better) things.

The XPI approach would work better for me, but I would find it difficult to apply it as a top-down, up-front design of the interfaces followed by independent development. In part, it's certainly because I lack the necessary experience; in part, it's an elusive goal even within the more familiar OO paradigm, where interfaces are "obvious" in some cases, but need to emerge through coding in others.

So in the end some design was planned, some emerged, some was badly planned and was fixed. For instance, in my first implementation the Track contained an index field; only later on I realized it was "part" of the Coloring concern, and moved it there.

Finding the "right" aspects was not trivial. In some cases (see Timing vs Aging) the more elegant decomposition was clear only in hindsight. Visualization of the dependencies helped a lot: I tend to visualize the collaboration structure when doing OOD as well, but here it felt even more necessary, and I would have appreciated a more aspect-friendly modeling tool. Conversely, the facilities provided by Eclipse in the AspectJ perspective were of little or no use to me (possibly because I had more useful diagrams on the side). It was very useful to see which lines were targets of an advice. However, I used this more like a "static debugging" tool, answering questions like "am I writing the right pointcut expression". It didn't provide any value as a design tool, and I've found basically no use for the other facilities like the Visualizer – they just don't give me the kind of information I need. Perhaps they're more useful when you adopt AOP with different goals / approaches, like patching existing code.

The effect of the rather artificial constraint on class / aspect size (one page) definitely pushed me to write better classes. For instance, Point didn't initially have a manhattanDistance method: that logic was in the caller. As I looked for opportunities to make the caller shorter, it was obvious that it should be moved to Point[25].

[25] Some would say it exhibited "feature envy", a classic bad smell documented in [FBBO1999]. Still, in my mind this is just an example of the long-standing "I'm alive" and "Er-Er" principles formulated by Peter Coad [CN1993], who recommended early on to avoid controller-like behavior and push intelligence down the chain.

Back to the process: as you have certainly noticed, I didn't use TDD, or BDD, or any other kind of "driven development"[26]. My design was also not based on "principles", that is, I didn't look to principles as either an inspiration for design ideas or as a steering force driving my design. Although I've been teaching design principles for a long while (and yeah, there is more to principles than SOLID), I honestly consider excessive reliance on principles as a clear sign of an underdeveloped discipline, and of a relatively low rank in the Dreyfus model. Experienced designers have internalized principles (and many patterns) and apply them situationally, mostly at the intuitive level; it's only when asked to rationalize that they may need to recast their work in terms of principles.

I've also *chosen* not to approach this project by borrowing from the literature on AOP patterns. I'll elaborate on this at the end of this chapter.

The nature of the problem, or: why did aspects "work" here?

I mentioned that part of the reasons why aspects worked "well" here is that the solution space they open is well aligned with the *nature* of the problem. It's probably worthwhile to elaborate a little on that "nature". Unfortunately, we don't have a reasonable formal language to describe the nature of problems; common formalisms tend to focus on the solution space, and the few exceptions, like Jackson's Problem Frames, are not refined enough to express the nuances I need. I'll simply have to describe the way I *perceive* the problem.

I see this problem as a push-pull pipeline, where "stuff" (touches) enters at one side, get enriched along the way, and is then being pulled at the other side (screen updates). The "enrichment" part is crucial, as it allows for decomposition and recombination. It is natural to think of the various features as optional, and every feature is also bringing in a small dose of data and behavior. This long, articulated sequence of reconfigurable stages maps very naturally to classes woven together by aspects.

---

[26] Well, it was *brain-driven design* as I call it, but that doesn't really count.

There are actually many problems sharing this “nature”, from very different domains. Consider, for instance, the nature of a web server: it fits very well with the notion of a pipeline with optional stages. In fact, several server-side frameworks are internally structured as pipelines (for instance, ASP.NET MVC), but stages are hard-coded (possibly empty, but present), and they stop at the boundary with your code, while an aspect-oriented design could re-shape the way we approach at least some parts of server-side web development.

Still, many problems don’t fit well with that family. They may have sub-problems which fit, but the overall shape can be very different. For instance, if your problem is to set up some complex equipment, moving through some kind of hierarchical configuration, you’re looking at a substantially different thing, and aspects may or may not work well there.

Of course, aspects are not limited to pipelined software: in fact, I intend to explore radically different problems in future episodes. It would be inappropriate, however, to suggest that aspects are the “best” solution to all problems, just like it would be for any other paradigm.

Back to the problem being solved here, it’s again worth noticing that true compositionality is extremely hard. Although it’s simple to draw a pipelined architecture with a few boxes and lull ourselves in the idea that we can freely add and remove stages, in practice it’s a very elusive goal. Everything is simple when stages don’t alter the “shape” of whatever we move through the pipeline. For instance, unix pipes are often claimed to be the quintessential example of a true compositional, component-oriented architecture. However, unix commands can be easily pipelined inasmuch as they only pass along streams of text, usually streams of text *lines*. Add something along the way, and everything breaks down rather quickly.

Conversely, when we enrich information along the way in hard-coded structures, it’s common to bloat whatever structure we use. We need time, so we add that in. We need color, so we add that in. Dynamic languages don’t care; statically typed languages tend to suffer. Aspects work well here because they can enrich both the structure being passed and the processing we perform, inside a modular unit (aspect) which can be added or removed.

When using aspects to create a compositional, pipelined architecture, it's necessary to pay extreme attention to *hidden dependencies*. As soon as one stage is adding information that is used later on in another stage, we have created an invisible dependency, and it's no longer possible to remove stage 1 and keep stage 2. We have seen this happening in this episode as well, and even if we moved to a more sophisticated architecture, some aspects would depend on Timing. It's probably one of the small differences between theory and practice that we need to accept in order to build something that actually works.

Identity and the role[s] of Surface

Let's step back from AOP for a moment and look back at traditional OOP patterns. Several patterns are concerned with extending behavior, like Template Method, Decorator, Strategy, and Visitor. There is a subtle but fundamental distinction between the first (Template Method) and the other three: in Template Method, you end up with a single object. Using the others, you end up with two or (for Decorator) more objects. As usual, there are specific advantages in those patterns, but splitting an object (as a unit of behavior) in multiple objects comes with a cost.

Consider the simple case of a Template Method. Suppose that in the subclass you somehow need to call a method of the base class. That's quite simple: you just call that method. Not so in Strategy, for instance: a Strategy is a different object, and if you want to transfer control back to the Context (as it's named in the GoF book) you need to pass a reference to the Strategy, possibly by using a new interface. A similar problem will surface every time we split an object into two or more objects. This issue has long been known in delegation-based languages (which will usually provide the context as part of the call), and it has been defined in literature as the "self problem" or as "object schizophrenia" [SR2002].

In contrast, aspects allow us to extend a class without any need to split it. What is usually obtained by a chain of Decorators, for instance, can be easily implemented by having a number of aspects adding behavior and data to a class. It has been suggested, therefore, that aspects may help solving object schizophrenia [CE2000], [Kendall1999]. While that is certainly true in some cases, there is a

wider notion of Identity that we still need to solve when designing aspect-oriented software.

To understand why, let's look back at the original problem. I want to move my fingers on the screen and see colored ribbons. In practice, the screen might be split in different areas, and I want that to happen on a specific area. As I mentioned in Chapter 1, I may even want *multiple, independent* areas with that behavior on a single activity screen. Independent, of course, means that the touches belonging to an area end up forming Tracks for that area, which end up forming Ribbons for that area, which have nothing do to with Ribbons in other areas. That strongly suggests that we need to have a TrackSet and a RibbonSet per area, and that instances of TouchPad must be also per-area.

In the end, the area that I called Surface happens to determine the multiplicity of every other entity; in terms of my Physics of Software, TouchPad, TracksSet and RibbonSet have a C/C (creation/creation) entanglement with Surface in the Run/Time space (for various reason, I tend to say they have the same *spin*).

Information particles with the same spin would very much want to share Identity as well. In fact, I could have used a different design approach, moved the code for the Surface-entangled classes into aspects, and injected that code into Surface. That would have solved all problems of identity, at the cost of having behavior in aspects, not in classes. Since I wanted to keep behavior in classes, I needed to find a way to "reference" the original information particle, which in this case happens to be Surface. That's why aspects need to mention Surface, and that's why I said it's acting as a beacon, or a marker. I think this will be a recurring problem whenever we try to use a similar design: aspects will need to glue together classes with the same run-time multiplicity, and using a common "center"[27] like surface will be a practical, simple solution.

Curiously enough, both my early reviewers argued that I could solve a few issues (for instance, the dependency between Tracing and Rendering) by accepting the

---

[27] I actually tend to see this as a gravitational center with all the other entangled particles orbiting around it, sharing spin. For a zen-like analogy, you may want to see [Pescio2012b] on the Value of Emptiness – Surface is in fact an empty class here.

idea that Surface is not just a marker, but a concrete class hosting TrackSet and RibbonSet as data members.

I could argue that Surface is a presentation object and that I don't want my presentation objects to contain domain objects, but that looks like the usual rethoric-based design. I can offer a better reason, which is not immediately apparent from the code: Surface does not really need to be mentioned in the aspects. The code could be changed so that any View could act as a Surface, but if I move TrackSet and RibbonSet into Surface, I'll lose that generality.

In fact, right now the only practical requirement on Surface is that is extends android.view.View. I wrote all code using Surface because it was much simpler to talk about it that way, without entering the realm of generic aspects and the half-baked support we get in AspectJ. Theoretically, we should indeed be able to write (e.g.) the Sensing aspect as an abstract, generic aspect:

Generic Sensing aspect (not working)

```
public abstract aspect Sensing<T extends View>
{
  public void T.onSizeChanged(int w, int h, int oldw, int oldh)
  {
  }

  protected pointcut
  onSizeChangedCall(T self, int w, int h, int oldw, int oldh) :
    execution(public void T.onSizeChanged(int, int, int, int)) &&
    args( w, h, oldw, oldh ) && target(self);
}
```

That, however, *does not work*, more for limitations of AspectJ than anything else. There are alternatives, as usual, to move the idea closer to the (current) expressive power of the language. It is possible to redefine Surface as an interface, and implement all the aspects in term of that interface alone. While I won't discuss the entire implementation here, the full source code is available on the aspectroid website inside the *IfcRibbons* project. There, I have two concrete classes (RibbonSurface1, RibbonSurface2), both implementing Surface, both present on the activity layout, and both being advised as expected.

The different nature of Aging and Blurring

I couldn't make Aging and Blurring follow the idea of having logic in classes and wiring in aspects. They both contain some "business" logic.

Let's focus on Aging first. It works by "augmenting" the RibbonSet logic, and that kind of change is not easily allocated to a meaningful collaborator. In hindsight, that's because the most natural way to "augment" RibbonSet that way would be to use inheritance, not collaboration. We could easily define an AgingRibbonSet class, derived from RibbonSet, and redefine the executingRefresh() there instead of doing so in the Aging aspect. Then, the Aging aspect could simply intercept creation of RibbonSet instances, and replace them with AgingRibbonSet instances. This would make Aging simpler and the overall design more in line with my goal of having most of the logic in classes, not in aspects.

That benefit, however, would come with a cost. The AgingRibbonSet class would depend on the Aging aspect (a circular dependency) because it would have to use the timestamp injected by Aging. As suggested before, I could then move that part on a Timing aspect, breaking the circular dependency. I would still have a type of dependency I strived to avoid (from a class to an aspect, AgingRibbonSet to Timing).

I never approach design with religious beliefs and hard rules. I like the idea of listening to the backtalk from my material as I'm shaping it. So perhaps there is something to be learned here – for instance, that my idea of having behavior in classes and also avoid dependencies from classes to aspects cannot realistically work, and that I have to choose one of those goals, either everywhere or on an instance basis. If I accept dependencies between classes and aspects, I open up an entire new dimension in the design space. There is a strong similarity between the design option I've mentioned above and what you get by following the mixin layers approach to AOP (see [CBML2002] or the similar paper from the same authors in [FECA2004]). It's something that I haven't explored in practice, but it would be interesting to.

A similar solution could be applied to Blurring as well, with the introduction of a BlurredRibbonSet. Note, however, that here we'll quickly come to face the main

issue of mixin technologies – the need to linearize the inheritance hierarchy. When I apply both Aging and Blurring, I need to place Blurring below aging.

A different approach could be to move the balance more on the explicit / OO side here, and introduce the notion of applying some Effect to the RibbonSet, with Aging and Blurring being effects. As I mentioned, the main issue here is dealing with the type system. Once again, it's only by trying that we can really appreciate the nuances of every single approach.

Coupling with the implementation

Aspects may end up being too coupled with implementation details of the classes they advise. My aspects are not an exception, and as I've discussed, there was some interplay between aspect design and fine-grained class design. Coloring "works" because I have enucleated a colorForTrack method.

My aspects are also quite often declared as privileged, that is, they can access private data and methods. This is often discouraged, but in practice, it reinforces encapsulation, while the obvious alternative (make the necessary members public) makes those members available to every other class /aspect. In this sense, it's a typical case where a localized violation of encapsulation is strengthening system-wide encapsulation.

Still, there are options I haven't pursued. I like the idea of crosscut programming interfaces / XPI, and I will probably explore this notion in a future experiment.

From an OO perspective, the degree of coupling I got is not substantially different from what you get using implementation inheritance. In fact, the similarity is more pronounced than it may seem at first. When we inherit from an interface, we have a clearly defined contract (the interface itself). Implementation inheritance is less explicit: some methods might be implemented, but isolated so that derived classes can still override their behavior. This is strikingly similar to enucleating a method so that an aspect can redefine its behavior. In this sense, the XPI approach is trying to bring back some explicitness, and as I said, I definitely want to try it out at some point.

Idioms and Patterns

As I mentioned, I didn't approach this project from a pattern perspective. Still, if we look at the problem under the OO-only lens, we can easily find a number of patterns that might apply. Pipes&Filters [BMRSS96] provides some sort of architectural guidance for pipelined architectures. An alternative to aspects could be to set up a chain of Decorators [JVHG1994]. Some would start with Model-View-Controller [BMRSS96] anyway; etc. If you're an experienced OO designer, you're probably familiar with the kind of trade-offs that usually come with those patterns, and it might be interesting to compare that kind of design with the aspect-oriented version I've presented. I would suggest spending some time pondering on the need, when we approach this problem from the OO perspective, to *anticipate* some complexity and introduce a few nonfunctional / aspectizable interfaces.

I've also consciously avoided the temptation to borrow from the aspect-oriented pattern literature, because I'm not sure we have enough knowledge about AOD to talk about "patterns" yet. Alan Kay recently criticized the pattern movement in software [Binstock 2012], saying that Alexander found a few hundred patterns from 2000 years of architecture, while we're rushing to classify as a "pattern" everything we find. There is some truth in that. I would say that some *idioms* (like the "classic" AspectJ approach to implement a role-based design documented in [NSPB2007]) are actually worth classifying (as such) but it's way too early to talk about patterns.

There is of course value in classifying things. While some of the "patterns" I've seen (for instance, *planned extension points* in [NSPB2007]) or the whole classification of cross-cutting concerns provided in [MMD2005]) are too general to provide any real guidance while designing, they form an interesting *vocabulary*, when talking about AOP but also outside AOP. A common design vocabulary is one of the benefits of patterns, but we don't strictly need patterns to form a vocabulary.

In the end, I also had a strong desire to explore the solution space without any external influence. That doesn't mean my design ended up being better because of that (maybe the opposite), but I see Aspectroid as an exploration of the design space: I didn't want to begin with a sort of guided tour.

Testability and Testing

Testability is one of the many non-functional properties of software; for some, it has turned into the primary non-functional property, "driving" the shape of the software itself. On my side, I see testability as a property that has to be balanced with others, so it was not the primary criteria driving my design. It is interesting, however, to review the results from a testability perspective. As we'll see, it's useful to discuss testability of classes first, and then consider the more challenging goal of testing classes within an aspect-oriented program, saving testing the aspect themselves last.

Looking back at Diagram 12, the simplest candidates for testing are the "core model" classes, that is, Point, Track, TrackSet. Personally, I would not change the design to allow any kind of mocking here. Point is basically trivial, Track can be easily tested by adding / retrieving points, and TrackSet too. The isolation of the core from either input or output came "for free" through the usage of aspects (and from the decision to roll my own Point class), so all is well in this area.

The right part of the diagram (TouchView, SimulatedPad, Movie) it's hard to unit-test. TouchView requires real input: simulated input is already handled by SimulatedPad; removing real input from TouchView makes the test useless. Movie works by periodically invalidating a View, and although this could be automatically tested somehow, the complexity of the testing code would exceed by far the complexity of Movie (try if you feel like it). SimulatedPad could be tested by subclassing; again, it's very likely that the complexity of the testing code would exceed the complexity of the class under test. This is an issue I've brought up years ago in my blog, raising some anger in the agile community, so I won't repeat myself here. It would be interesting to see how true believers would approach a meaningful automated test there, while keeping the complexity of the testing code below that of the tested code. Note that I'm speaking about complexity, not length. So, in the end, and at the cost of being labelled as unprofessional in some circles, I would probably test those classes interactively. I don't see them as changing; they're so small they're better swapped out if you need different behavior, so the chance of regression is quite small.

The left part of the diagram (ThickPaint, Palette, Ribbon, RibbonSet) is more interesting. ThickPaint and Palette are not challenging. Their purpose is basically that of building things. We can access those things and check they match the specification, no big deal. Ribbon and RibbonSet are more challenging, because their behavior, as implemented, is to transform a Track / TrackSet into a series of Paths and then plot those Paths on a Canvas. They do not expose the Paths, but even if they did, one cannot simply go through a Path and "check" that is was built according to the specification. At least, not with the "standard" Android Path.

Here things become more interesting. If we decide that we want to test Ribbon / RibbonSet without looking at the screen, we're imposing a new force upon our design, and the shape must be changed accordingly. A common way would be to add the ability to swap Path away, and replace it with our own, more test-friendly class. That could be easily done by placing an interface between Ribbon and Path, which does not have to mirror the whole Path public interface, but just the few methods we are using. Path is in the Android library, so we cannot have it implement our own interface, and an adapter would be needed. In turn, that would open the problem of creating concrete objects, leading to the common solution of dependency injection (or, if we are inclined to use aspects for testing, we could simply use an aspect as a factory).

What is a more "testing-friendly" version of Path? It's something that we can easily check for compliancy with a specification. For instance, keeping the instructions (MoveTo, QuadTo) into a syntax tree (or simply as a string) makes it easy to check that everything is going as expected.

That's just half the picture though. Path is not accessible from Ribbon; it's an implementation detail. Now we face another choice:

- We can go the easy way, and make it public.
- If we use aspects for testing, we can use a privileged aspect and get the Path-like object.
- We can maintain encapsulation, but swap out Canvas as well.

The latter option is probably the most interesting to discuss. We may easily maintain encapsulation, as long as we're not using a real Canvas inside the Ribbon

/ RibbonSet pair. Once again, we can interpose our own interface, and use an adapter to forward the calls to a real Canvas, or swap that for our own Canvas, which will receive the Path-like object and check for compliancy. Once again, we'll face the issue of creation, and if you build it, you'll find you'll have to perform a downcast as well (because at some point, you need the real Path to plot on the real Canvas).

Now, we can look at all that under two different perspectives:

- It's "design damage". As I'm writing this booklet, just a few weeks have passed since a blog post from David Heinemeier Hansson (Test-induced design damage) spurred a debate on the value and cost of TDD. If you subscribe to this point of view, all the additional complexity of what I've just discussed (with the sole purpose of testing 36 + 46 LOC) is just design damage and should be avoided.

- It's actually an improvement over the original design. There is a school of thought according to which testability is a force "aligned" with other forces, like low coupling, and by making the design more testable we're therefore choosing to improve other properties as well, perhaps at the expense of some additional structural complexity.

As usual, there is merit in both, and I consider it useless to discuss this matter outside a context. So, let's stay within *this* context. I've mentioned the possibility of moving the rendering from android.graphics to android.opengl a couple of times already. That's interesting, because there is no Canvas and there is no Path in android.opengl. There is a GLSurfaceView which may act as a canvas, and we'll have to roll our own Path-like object to implement a quadratic spline. In this perspective, improving the testability of Ribbon and RibbonSet proves to be in fact aligned with another goal: extendibility. I'm not claiming that this is generally true, or that it's even worth doing it this way (as opposed to create an entire different subsystem for opengl, with its own Ribbon and RibbonSet, and have a different OpenGlRendering aspect wire this subsystem instead of the current one). I think it's interesting to see how forces can be indeed aligned; everything else is more subjective.

Testing in presence of aspects

Good, *readable* automatic tests provide both a *specification* of the expected behavior and a way to check that the implementation is indeed meeting the specification. Design by Contract [Meyer1992] had a similar goal, although the proposed technique was different, centered about the idea of adding predicates (invariant, preconditions and post-conditions) around methods and classes. In a sense, while contemporary testing techniques tend to place similar statements outside the source code / artifacts under test, DbC tended to place those predicates inline[28].

Within DbC, as applied within OOP, it is well known that predicates on a base class act as constraints on derived classes. In fact, the Liskov Substitution Principle has also been formulated in terms of DbC (the original formulation was not). What about aspects? Is testing, or even the specification side of testing, affected in any way by the presence of aspects?

Consider the RibbonSet class. One could arguably add a post-condition on RibbonSet.displayOn: at the end, the ribbons member will have the same cardinality as the tracks member. That would be very reasonable, as a natural language specification for displayOn could be "create a ribbon for every track, and display that ribbon". The post-condition could be easily tested inline (using a DbC approach), while testing it from the outside would require the machinery discussed above (a shim around Path and Canvas).

That test, however, would break as soon as we mix in aspects. The Aging aspect removes dead Tracks from the TrackSet, so depending on when you calculate the cardinality it may or may not work. If you test from the outside, you will probably *assume* a specific cardinality, and the test will break. Ditto for the blurring aspect, which is creating more Ribbons than originally expected, de facto breaking the original specification.

What is the consequence of that? If we consider the original specification as a constraint on aspects, the whole idea of using aspects to implement application-level concerns falls apart. We should actually limit ourselves to spectator-like

[28] There are non-trivial consequences along the information hiding / function granularity when you move between those styles.

aspects [CG2002], which can't do much. On the other hand, if we want to test our code, we need to find some reasonable strategy. Here is my current view:

- Tests are specifications, and aspects may alter the specification.
- Therefore, some tests are valid only in the original context (no aspects) while others are valid in the aspect-informed context.
- External tests are therefore preferable to inline assertions, which cannot be easily excluded.
- An Aspect-Oriented test suite should be structured so that tests are executed within the appropriate context.
- That requires that tests are partitioned in subsets, and each subset must include only the aspects relevant for that portion of the specification.

This is of course possible using AspectJ, but requires that each subset is compiled independently[29]. An aspect-aware tool may probably help a lot here.

It is perhaps interesting to observe that contemporary testing practices are often implicitly accepting the notion that "nothing else changes", that is, that the test passes because a specific post-condition is true. So we usually don't check if the number of Tracks has been changed when displaying a Ribbon: we just check that the proper number of Tracks has been displayed.

Still, this has *always* been a rather hard conceptual problem, known in Artificial Intelligence as the Frame Problem since the 80s. It has also been discussed within the context of OOP and subclassing in early 90s (see for instance [BMR1993]) and then quickly forgotten when people moved from a specification mindset to a testing culture. However, aspects would necessarily bring this notion back to the table, as it's exactly the ability to make "something else" happen that makes aspects useful, and yet it's the same ability that makes reasoning, specifying and testing in presence of aspects difficult.

[29] I'm thinking of compile-time weaving here, because it's the only one available under Android.

Testing the aspects

I used 9 aspects in this project. Four of them (Sensing, Touching, Simulating, Zeroing) are basically one-liners. It's really hard to test those aspects in isolation, and it probably makes little sense. The fact that they're dealing with input makes it even harder. Honestly, I would test those interactively.

Tracking, Rendering and Coloring are not one-liners, but are still relatively devoid of logic. They basically wire things up, although Coloring is also providing a track index. Again, testing in isolation is hard: they depend on concrete classes (Surface, Track, RibbonSet, etc.) and breaking those dependencies would require major changes to the code (see the ideas about using generic aspects or XPIs). It is however possible to build an automatic test fixture for them (as a form of integration testing), albeit we must once again weight the cost and the benefit.

Aging and Blurring are the real issue here. They contain significant logic, it's also output logic which is hard to evaluate without looking at the screen, and they can't be easily tested in isolation. If we take the time to isolate Ribbon / RibbonSet from Path and Canvas, however, we can build an automatic test fixture for Aging and Blurring as well, although this will be once again a form of integration testing.

This, however, would raise another interesting issue: Aging and Blurring depends on a notion of time. Even if we build the infrastructure to test Ribbon and RibbonSet, having to rely on precise timing to verify the behavior of aging and blurring would definitely be a testability bloop. The "right thing" to do here is once again to move the Timing aspect away from the Aging aspect. That way, we could easily swap in a simulated clock during testing. Once again, note that we don't have to overengineer the main code to accept some sort of abstract timer, we don't need dependency injection, etc. What we need is to recognize behavior that we want to replace, make it modular and weave it using aspects.

I think it's slightly amusing, for a software designer, to see all those forces pushing the Timing aspect out. In a sense, that's why I've left it in: it's easier to appreciate those forces if you can see them; had I moved the Timing aspect away from the very beginning, the need for that move would have appeared less stringent. The "amusing" part is the alignment of those forces. Decoupling Blurring from Aging requires Timing to be modularized, not because we want to replace it, but

because we want Blurring to depend on a more primitive (fine-grained) notion. Testability often pushes toward fine-grained modularization as well, because replacing some abstractions with a more deterministic counterpart helps asserting behavior. These two largely independent needs are in perfect alignment here. As usual, we must be wary of forming universal rules from a single example. Things don't always work this way. We need to grow an appreciation for forces; there is no easy substitute for that.

Integration testing vs. Unit testing

Generally speaking, and outside the usage of aspects, highly compositional software still requires that we do a significant amount of integration testing, because there is no strict guarantee that the components will interact nicely in any order and combination. Case in point: there are a couple of constants in Aging and Blurring. They determine the portion of the ribbon (in milliseconds) that we want to keep alive, and the portion of the ribbon (again in ms) that we want to blur. As we have seen, Blurring depends on Aging, but it's a byproduct of my implementation. They may simply depend on Timing. At that point Aging and Blurring may work just fine in isolation, but not in integration: just set the Aging period small enough, and you'll never see the Blurring. As usual, having done an isolated test, we have some confidence on the components, which should help finding the interaction problem. There would be a lot to say here, because this notion extends outside the realm of testing, and enters the wider space of safe composition. However, that would be definitely out of scope here.

A last remark: in line with the main idea in this booklet (using aspects for application-level concerns) I'm not dealing with the (still interesting) issue of using aspects *for* testing. I'm more concerned about testing the code inside my aspects. There is, however, a vast amount of literature on using aspects for testing, which is a very interesting field on its own.

Chapter 4: Wrap up

When I started this project, I had a much shorter booklet in mind. Turned out I had a little more to say than I expected. In fact, I had to force myself to stop writing, at the cost of leaving some material out, especially in Chapter 3. The next episodes will probably be shorter, as I can now use this as a baseline reference. They will also be my chance to get back to some notions I had to skip or barely scratch.

I have learnt a lot writing this text, to the point where I should probably archive this version, write some new code, and create a better booklet. The reason I'm not doing so is that I don't need this to be perfect. I'm more interested in sharing these ideas and learning from your feedback. There will be other times to explore new areas of the design space, and maybe to come back to the Ribbons project with a more refined sensibility.

In the end, this booklet wasn't just a about Android, or AspectJ, and not even just about Aspect Oriented Design. It was a reflective design conversation, a way to share a bit of my design approach and some of my thoughts. I hope you enjoyed reading it. If you got any new idea, or discovered new angles on aspect orientation and software design, or if you feel like you may want to try out some of this stuff, then this work succeeded.

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 booklet is the best way to say you liked it ☺.

Appendix A: Tools and Setup

All the code I've presented has been developed using Eclipse, the ADT (Android Development Tools) plugin and the AJDT (AspectJ Development Tools) plugin.

I have tested the app with both a relatively "old" version of Eclipse (Indigo) + the AJDT 2.2.0, and with the latest (as I'm writing) Eclipse ADT bundle (23.0.2) + the AJDT 2.2.3.

My suggestions if you want to write your own apps:

- Get the tools.
- Create a simple "hello world" application in plain Java, no Android.
- Convert it into an AspectJ application. You do so by right-clicking the project and then choosing Configure → Convert to AspectJ Project (at least, that's how it's done with the tools I've used).
- Add a simple aspect; for instance, a before or after advice around a non-static method, where you can print some additional text. Execute the app. Make sure it works before moving to Android.
- Create a simple "hello world" Android application.
- As above, convert it into an AspectJ app, add an aspect, deploy to your device.
- You're set!

I haven't used the Android Studio or other development environments, so I can't help you there.

Bibliography

Whenever possible, I've provided a hyperlink to a freely available version of the references. Hyperlinks were valid as of July 2014.

[Alexander2009] Christopher Alexander, *Harmony-Seeking Computations: a Science of Non-Classical Dynamics based on the Progressive Evolution of the Larger Whole*, 2009.

[Armour2004] Philip G. Armour, *The Laws of Software Process: A New Model for the Production and Management of Software*, Auerbach Publications, 2004.

[AS2004] Erik Arisholm, Dag I. K. Sjøberg, *Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software*, IEEE Transactions on Software Engineering, Vol. 30 Num. 8, August 2004

[BA2004] Lodewijk Bergmans, Mehmet Aksit, *Principles and Design Rationale of Composition Filters*, in [FECA2004]. Also available online.

[Binstock 2012] Andrew Binstock, *Interview with Alan Kay*, Dr. Dobb's Journal, July 2012.

[BMR1993] Alex Borgida, John Mylopoulos, Raymond Reiter,*... And Nothing Else Changes: The Frame Problem in Procedure Specifications*, Proceedings of the 15th International Conference on Software Engineering, May 1993.

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerlad, Michael Stal, *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns*, Wiley, 1996.

[Brooks1986] Fred P. Brooks, *No Silver Bullet - Essence and Accident in Software Engineering*, Proceedings of the IFIP Tenth World Computing Conference, 1986. Reprinted in other Brook's works, including "The Mythical Man Month".

[CBML2002] Richard Cardone, Adam Brown, Sean McDirmid, Calvin Lin, *Using Mixins to Build Flexible Widgets*, Proceedings of the 1st international conference on Aspect-oriented software development, ACM, 2002.

[CCHW2004] Adrian Colyer, Andy Clement, George Harley, Matthew Webster, *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison Wesley Professional, December 2004.

[CE2000] Krzysztof Czarnecki, Ulrich W. Eisenecker, *Aspect-Oriented Decomposition and Composition*, in: Generative programming: methods, tools, and applications, ACM Press / Addison-Wesley, 2000. Draft also available online.

[CG2002] Curtis Clifton , Gary T. Leavens, *Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning*, TR #02-10, October 2002.

[CK1994] Shyam R. Chidamber, Chris F. Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering, Vol. 20 No. 6, June 1994. Early draft available at MIT.

[CN1993] Peter Coad, Jill Nicola, *Object-Oriented Programming*, Prentice-Hall, 1993.

[FBBO1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[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.

[Hatton2009] Les Hatton, *Power-Law Distributions of Component Size in General Software Systems*, IEEE Transactions on Software Engineering, Vol. 35 No. 4, July/August 2009.

[JVHG1994] Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Kay1993] Alan C. Kay, *The Early History of Smalltalk*, ACM SIGPLAN Notices, Volume 28 No. 3, March 1993.

[Kendall1999] Elizabeth A. Kendall, Role model designs and implementations with aspect-oriented programming, ACM SIGPLAN Notices, Vol. 34 No. 10, October 1999. Also available online.

[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.

[LS1986] Stanley Letovsky, Elliot Soloway, *Delocalized plans and program comprehension*, IEEE Software, Vol. 3 No. 3, May 1986.

[Meyer1992] Bertrand Meyer, *Applying "Design by Contract"*, IEEE Computer, Vol. 25 No. 10, October 1992.

[Miles2004] Russell Miles, *AspectJ Cookbook*, O'Reilly, December 2004.

[MMD2005] Marius Marin, Leon Moonen, Arie van Deursen, *A Classification of Crosscutting Concerns*, Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005.

[NSPB2007] James Noble, Arno Schmidmeier, David J. Pearce, Andrew P. Black, *Patterns of Aspect-Oriented Design*, Proceeding of EuroPLoP 2007.

[Parnas1979] David L. Parnas, *Designing software for ease of extension and contraction*, IEEE Transactions on Software Engineering, Vol. 5 No. 2, March, 1979.

[PC1986] David L. Parnas, Paul C. Clements, *A rational design process: How and why to fake it*, IEEE Transactions on Software Engineering, Vol. 12 No. 2, February 1986.

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

[Pescio2008] Carlo Pescio, *Can AOP inform OOP (toward SOA, too? :-) [part 1]*, published on carlopescio.com, April 2008.

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

[Pescio2011a] Carlo Pescio, *Is Software Design Literature Dead?*, published on carlopescio.com, February 2011.

[Pescio2011b] Carlo Pescio, *Your coding conventions are hurting you*, published on carlopescio.com, April 2011.

[Pescio2012a] Carlo Pescio, *Notes on Software Design, Chapter 16: Learning to see the Forcefield*, published on carlopescio.com, May 2012.

[Pescio2012b] Carlo Pescio, *Don't do it*, published on carlopescio.com, October 2012.

[Pescio2012c] Carlo Pescio, *Ask not what an object is, but...*, published on carlopescio.com, December 2012.

[SJSJ2005] Neeraj Sangal, Ev Jordan, Vineet Sinha, Daniel Jackson, *Using Dependency Models to Manage Complex Software Architecture*, Proceedings of OOPSLA 2005, October 2005.

[SR2002] K. Chandra Sekharaiah, D. Janaki Ram, *Object Schizophrenia Problem in Modeling Is-role-of Inheritance*, in ECOOP 2002 "Inheritance Workshop", Springer Verlag, 2002. Also available online.

[TC2005] Paolo Tonella, Mariano Ceccato, *Refactoring the Aspectizable Interfaces: An Empirical Assessment,* IEEE Transactions on Software Engineering, Vol. 31 No. 10, October 2005.

[VideiraLopes2004] Cristina Videira Lopes, *AOP: A Historical Perspective*, in [FECA2004]. Earlier draft available as ISR technical report.

[Wampler2004] Dean Wampler, *Noninvasiveness and Aspect-Oriented Design: Lessons from Object-Oriented Design Principles*, 2004.

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.