Aspect-Oriented Programming
The Fun Has Just Begun

Gregor Kiczales
University of British Columbia
Xerox PARC
gregor@cs.ubc.ca

Based on work and discussions with:

Ida Chan, Yvonne Coady, Brian de Alwis, Kris De Volder, Chris Dutchyn, Cristina Green, Stephan Gudmundson, Jan Hanneman, Erik Hilsdale, Jim Hugunin, Mik Kersten, Gail Murphy and Greg Smolyn.

We are on the verge of a fundamental change in software development. A central idea in all science and engineering – the idea of working with a system through multiple perspectives or views – is being enabled in software development in a powerful new way.

In the past, software developers have had good technology for programming at different levels of abstraction or, in other words, working with views at different levels of detail. But more recently we have learned how to program through different crosscutting views. The immediate impact of this change has been to make it possible to modularize aspects of a system's implementation that previously could not be modularized. This is leading to software that is significantly more flexible, adaptable, and in all likelihood more robust than what we could previously develop.

But the fun has just begun. Within the framework of current AOP proposals we can see significant room for further improvement and power. Experience with AOP is leading developers and researchers to ask for and develop exciting new features.

But we can also see signs of a next generation of AOP technology, that we call fluid AOP. Fluid AOP involves the ability to temporarily shift a program (or other software model) to a different structure to do some piece of work with it, and then shift it back. This is analogous to electrical engineers using the Fourier transform to make certain problems easier to solve.

We believe that the combination of traditional static structuring mechanisms like OOP, with static AOP mechanisms that support crosscutting, and fluid AOP mechanisms will enable a fundamentally more solid engineering foundation for software development than we have had to date. This is because developers will have the ability to work with programs more like other disciplines work with models. Developers will be able to write programs at greater or lesser levels of detail, and write programs that crosscut, and be able to fluidly re-structure programs for particular purposes. We believe that this kind of technical foundation for software engineering is at least as important as the process-analogy foundations that have received so much attention. In fact, we believe these technical foundations may make it possible to better integrate process and technical issues.

Models and Views

The idea of a *model* of an engineered system is familiar. Models are abstractions that capture certain aspects of the system while leaving others aside. Models can be formal, or informal.

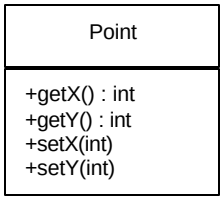
Programs Are Models That Run

Programs have much in common with models, in particular they are abstractions of a system that make certain properties explicit and hide, or abstract away, other properties. But programs have a special property that most kinds of models do not – they can automatically produce the actual computation they model. So programs are like both models and actual systems – they are abstract, formal systems that can simply be "run" to automatically produce the modeled process. We say that programs are models that run.

Models at Different Levels of Abstraction

A critical power of models is to be able to use different models to focus on different properties of a system at different times. One common way to organize such models is in different levels of detail or granularity. Schematics of a car might, for example, show a fully assembled door as an atomic unit in one view, but then show each of the door's individual components in another view. In such modeling frameworks, the individual elements of the model tend to correspond to individual components of the system.

In software development we are quite familiar with the notion of models – including programs – that differ in level of detail. For example, here are two views of a simple two-dimensional `Point` class that differ only in level of detail:




```
class Point implements FigureElement {
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) { this.x = x; }
  void setY(int y) { this.y = y; }
}
```

Models that Crosscut

Some kinds of model bear a different relationship to each other. Rather than being more or less detailed, they may capture fundamentally different aspects of a system. This can happen when the models *crosscut* each other, or in other words, carve up the system into fundamentally different units. For example, returning to the car a wiring diagram might crosscut the component model from above to some degree, and a circuit diagram could have an entirely different topology.

Aspect-Oriented Programming

The fundamental idea behind aspect-oriented programming is that it should be possible to create programs that crosscut each other – in other words to have models that run, that can differ in granularity or level of abstraction, and that can crosscut each other. Aspect-oriented programming (AOP) is motivated by careful observations that even in well-structured programs the implementation of some design concerns is difficult or even impossible to cleanly modularize.[1] The AOP bet is that these concerns could be modularized if we had the ability to write crosscutting programs. In AOP, the term *aspect* is used to refer to the modular implementation of a crosscutting concern.

---

[1] By *modularize* we mean that the implementation of the concern is: (i) localized, (ii) has a well-defined interface, and (iii) is amenable to separate development. It is also critical that such modularization be efficient. We avoid using general terms like *modular* or *modularized* to mean modular with respect to any particular programming technology, such as current type checking technology.

The past five years have seen significant advances in aspect-oriented programming. There are now four significant AOP proposals, and tens of smaller-scale proposals. At least two of these AOP systems have a significant number of users.[2]

Existing AOP systems make it possible to modularize the implementation of a wide range of crosscutting concerns, including synchronization policies, resource management, many common design patterns, design rule checking, error handling policies, distribution, replication, caching, security management and many others. Following are two canonical examples of aspects that current generation languages permit. (These examples are written in AspectJ 1.0, but most of the other AOP proposals including HyperJ, Composition Filters and Demeter support aspects like these.)

```
aspect DisplayUpdating {
  pointcut move():
    call(void Point.setX(int))   ||
    call(void Point.setY(int))   ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() returning: move() {
    Display.needsRepaint();
  }
}
```

```
aspect PublicErrorLogging {
  Log log = new Log();
  pointcut publicCall():
    call(public * com.xerox.printers.*(..));

  after() throwing (Error e): publicCall() {
    log.write(e);
  }
}
```

The effect of the `DisplayUpdating` aspect is to ensure that any time a `Point` or `Line` object moves it notifies the display to update. This aspect has a relatively small scope (2 classes), and explicitly enumerates the operations it crosscuts. The `PublicErrorLogging` aspect ensures that if any public method on any public class in the `com.xerox.printers` package throws an error to its caller, that error will be logged. This aspect has a much wider scope, and is defined in terms of properties of the methods it affects (public methods on public classes in `com.xerox.printers`).

Many people, when they first see AOP, suggest that concerns like DisplayUpdating could be modularized in other ways, including the use of patterns, reflection, or 'careful coding'. But the proposed alternatives nearly always fail to localize the crosscutting concern. They tend to involve some code that remains in the base structure, i.e. calls to `needsRepaint` in the `Point` and `Line` classes.

The power of these AOP systems comes from the fact that they support crosscutting programs. The `DisplayUpdating` aspect treats locally something that is spread out in the `Point` and `Line` classes – which is the constitutive property of crosscutting models.

There is still a great deal of potential benefit and important research to do in the space of these kinds of AOP languages. Jim Hugunin's submission to this workshop describes this in more detail, but at least four important areas are: (i) Better engineering and integration of the languages, including faster compilation, load-time weaving, integration with EJB etc.; (ii) More powerful languages to specify crosscutting, for example to be able to say, in code things like "notify the display whenever a value that is read during the display of a figure element changes"; (iii) Theoretical foundations, including exploration of whether new work in type states could provide a mechanism for type-checking the interaction of an aspect with the base structure; (iv) Tool support, including integrating with existing IDE tools and providing new tools for integrating AOP into existing development processes.

Fluid AOP

AOP proposals like AspectJ provide a kind of AOP that I want to call, in this paper, *static*, to contrast it with *fluid* AOP introduced in this section.[3] Static AOP is characterized by:

---

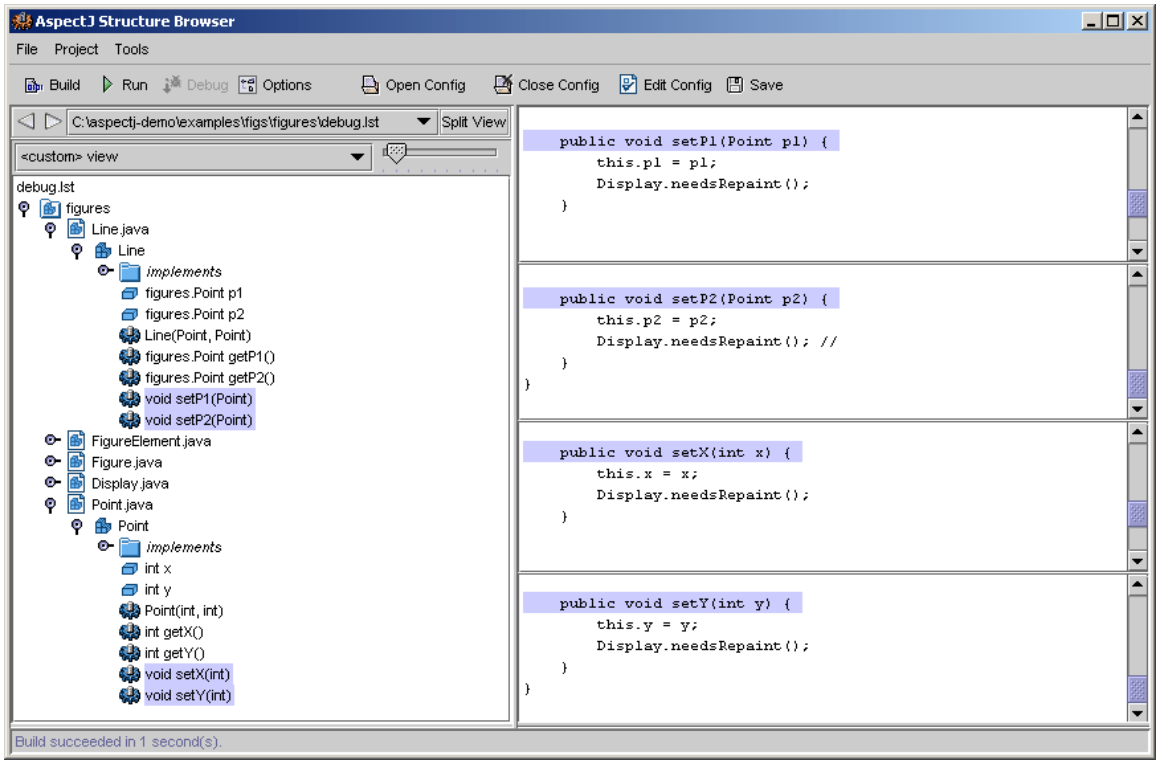
[2] As of September 2001, AspectJ compiler and development tool downloads were both over 2000 per month. More broadly, 8 AOP-related workshops were held during 2001, and a new conference, focused on AOP and the broader theme of aspect-oriented software development (AOSD) is planned for April 2002 [aosd.net]. DARPA, NSF and ATP contracts and grants have funded much of this work. (In Europe, EC already funds projects in this area, and Framework VI is expected to include an AOSD action line.)

- The existence of a primary or base structure (in AspectJ the classes),
- together with support for modular aspects that crosscut the base structure (in AspectJ the aspects).
- Arranged so that the aspect and base structures are relatively fixed or static in that changing them involves editing the program.

This supports the fundamental AOP intuition that no single view or model structure can capture all that is important about a system, but that instead crosscutting models are critical. But if that is so, then it also reasonable to expect that having a single fixed base and aspect structure should also be limiting. It might be advantageous to be able to dynamically re-form a program into a crosscutting view structure.

While this kind of idea has been bantied about for years, current AOP, by giving us a concrete implementation of static crosscutting, gives us an easy way to experiment with what fluid AOP should be like. For example, using the AspectJ structure browser tools, which know how to manipulate crosscutting program structures, we can quickly prototype tools that temporarily localize the non-modular implementation of a crosscutting concern.

The following picture shows how this can be done using technology as simple as multiple editor windows. This figure shows how the browser has found the non-modular implementation of a display updating concern, and brought that together for the programmer to examine all-at-once. Once it is so localized, the programmer can reason about it (for example to confirm that the call to `needsRepaint` always happens after the state change operation); edit it (for example to call a different method); or refactor it (for example to re-implement using AspectJ as above).



This kind of temporary re-modularization of the program is what we mean by fluid AOP – we can stretch the program to bring different crosscutting aspects into focus. Many existing and past systems provide an excellent platform for pursuing this kind of work, including systems like Intentional Programming, Eclipse, VisualAge, Smalltalk and any other platform that represents code as data structures rather than ASCII text.[4]

---

[3] Together with John Lamping we used to call this *Strong AOP*, but *Fluid AOP* better captures what such software development feels like. More recently, Ossher and Tarr have called this "morphogenic code".

[4] In the mid 80s, the Interlisp-D programming environment included support for simple operations like this, including a handful of semi-automatic re-factoring patterns.

But as soon as we see this we can imagine much more. What mechanism will be used to identify the scattered structure and describe how it should be localized? In the figure above, the AspectJ construct `'call(void Display.repaint())'` was used to identify the concern, and the tool just used a simple 'juxtapose' rule for localizing it. Work mentioned above in new mechanisms for specifying crosscutting should advance this. And it should be possible to use the same mechanism that identifies the scattered calls to temporarily unify them so that editing one would mean editing all. The aspect-oriented logic meta-programming work by De Volder may have the potential to enable such operations.

While we have described Hyper/J as enabling static AOP, which it does, Hyper/J can easily be seen as an entry into the fluid AOP space. The hyperslice and dimension mechanisms in Hyper/J effectively permit pulling out a scattered concern into a single unit. But to date the Hyper/J work has not yet benefited from an interface that gives it the kind of fluid feeling we are talking about here. Instead the dimensions and slices tend to be as static in structure as aspects in AspectJ.

This is not to suggest that fluid AOP will replace static AOP. We don't envision that at all. We expect there will always be some kind of base structure that programmers may then want to fluidly re-organize for particular work they need to do on the system.

Other Work in Crosscutting Programs

The AOP community is not the only source of work with crosscutting in software development. Two other important areas are UML and object-oriented design patterns.

In UML class and interaction diagrams crosscut each other in that what is local in an interaction diagram is scattered in a class diagram. An interaction diagram localizes method calls involving multiple objects, of perhaps different classes; whereas a class diagram localizes the methods of a single class.

Object-oriented design patterns deal with crosscutting concerns in a different way. A pattern can be thought of as a "soft-overlay" of a crosscutting view of the system. For example, if two classes are involved in the subject/observer pattern, the implementation of that pattern will not be modular, it will be scattered across both classes. But by calling the scattered methods a pattern it becomes possible to at least conceptually localize that crosscutting concern. Fluid AOP will make it possible to shift back and forth between the scattered view, the temporarily localized view, or the AspectJ style modular implementation.

Other Work in Modeling of Software

One of the greatest challenges to high-level or model-based software design and development techniques has been the loss of trace-ability between the model and the code. We believe that a root cause of this has been the lack of ability to modularize crosscutting concerns in the code.

For example, high-level models are useful for things like proving end-to-end flow control properties. But without AOP the implementation of such flow-control strategies becomes scattered and trace-ability to the model is lost. Using AOP, this does not have to be the case. We are currently involved in an experiment with Boeing, Washington University and Vanderbilt University to demonstrate this in an avionics application. We expect to be able to carry the model-based proofs of flow control properties into the code, where we will use new results in type systems to check that the code fulfills the properties.

Conclusion

We believe that the combination of traditional static structuring mechanisms like OOP, with static AOP mechanisms that support crosscutting, and fluid AOP mechanisms will enable a fundamentally more solid engineering foundation for software development than we have had to date. This is because we will have the ability to work with programs more like other disciplines work with models. We will have programs at more or less levels of detail, programs that crosscut, and the ability to fluidly re-structure programs for particular purposes. We believe that this kind of technical foundation for software engineering is at least as important as the process-analogy foundations that have received so much attention. In fact, we believe these technical foundations may make it possible to better integrate process and technical issues.