On Abstraction, Information Hiding and Crosscutting Modularity

http://www.st.informatik.tu-darmstadt.de/

Mira Mezini
Darmstadt University of Technology
Where Started

Critique on Black-Box Modularity
The task of the software development team is to engineer the illusion of simplicity.
WHATEVER IS LIKELY TO CHANGE!
Kiczales: Beyond the Black-Box

Clients confront an issue that the interface claimed to hide.

An open implementation presents two interfaces
Harrison & Ossher on Subjectivity

Figure 3 - Many Subjective Views of an Object-Oriented Tree
Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.
Where we are

AOP improves software modularity
- anonymous AOP researcher

AOP is anti-modular.
- anonymous non-AOP researcher
Questions Addressed in [KiczalesMezini05]

• Does AOP improve or harm modularity?
  – in presence of crosscutting concerns (CCC) improves modularity of aspects and non-aspects
  – does not harm modularity otherwise

• If AOP is modular, what is modularity?
  – nearly the same idea and mechanisms as before
  – except for how interfaces are determined
    • aspect-aware interfaces
    • interface depends on overall system configuration
Form of Argument

• Start with
  – simple definitions of modularity and modular reasoning
  – Java and AspectJ implementations of a simple example

• For both implementations
  – analyze static modularity
  – consider interfaces for both implementations
  – analyze ability to do modular reasoning

• Discussion of *aspect-aware interfaces*
Definitions:

- Modular reasoning: make decisions about a module by studying only
  - its implementation and interface
  - interfaces of other modules referenced in the module’s implementation or interface
- Expanded modular reasoning: also study implementations of referenced modules
- Global reasoning: have to examine all the modules in the system

Code implementing a concern is \textit{modular} if:

- it is textually local
- It has a well-defined interface
- the interface is an abstraction of the implementation
- interface is enforced
- the module can be automatically composed
Example

Shape

* moveBy(int, int)

Display

Point

- getX()
- getY()
- setX(int)
- setY(int)
- moveBy(int, int)

Line

- getP1()
- getP2()
- setP1(Point)
- setP2(Point)
- moveBy(int, int)

Update Signaling

- what constitutes display state change
- signal update on display state change
class Point {
    int x, y;
    ...
    void setX(int nx) {
        x = nx;
        Display.update();
    }
}

class Line {
    Point p1, p2;
    ...
    void moveBy(int dx, int dy) {
        p1.x += dx; p1.y += dy;
        p2.x += dy; p2.y += dy;
        Display.update();
    }
}
class Point {
    int x, y;
    ...
    void setX(int nx) {
        x = nx;
    }
}

class Line {
    Point p1, p2;
    ...
    void moveBy(int dx, int dy) {
        p1.x += dx; p1.y += dy;
        p2.x += dy; p2.y += dy;
    }
}

aspect UpdateSignaling {
    pointcut change():    execution(void Point.setX(int))
                         || execution(void Point.setY(int))
                         || execution(void Shape.moveBy(int, int));
    after() returning: change() {
        Display.update();
    }
}
### Modularity Assessment

<table>
<thead>
<tr>
<th>Non AOP</th>
<th>localized</th>
<th>interface</th>
<th>abstraction</th>
<th>enforced</th>
<th>composable</th>
</tr>
</thead>
<tbody>
<tr>
<td>display updating</td>
<td>no</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Point, Line</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

```java
class Point {
    ...
    void setX(int nx) {
        x = nx;
        Display.update();
    }
}
```

```java
class Line {
    ...
    void moveBy(int dx, int dy) {
        p1.x += dx; p1.y += dy;
        p2.x += dy; p2.y += dy;
        Display.update();
    }
}
```
Modularity Assessment

<table>
<thead>
<tr>
<th></th>
<th>localized</th>
<th>interface</th>
<th>abstraction</th>
<th>enforced</th>
<th>composable</th>
</tr>
</thead>
<tbody>
<tr>
<td>n o n AOP</td>
<td>display updating</td>
<td>no</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Point, Line</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>AOP</td>
<td>display updating</td>
<td>high</td>
<td>high</td>
<td>medium</td>
<td>yes</td>
</tr>
<tr>
<td>Point, Line</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

```java
class Line {
    
    ...  

    void moveBy(int dx, int dy) {
        p1.x += dx; p1.y += dy;
        p2.x += dy; p2.y += dy;
    }

}
```

```java
class Point {
    
    ...  

    void setX(int nx) {
        x = nx;
    }

}
```
Point implements Shape
  int getX();
  int getY();
  void setX(int);
  void setY(int);
  void moveBy(int, int);

Line
  <similar>
Point implements Shape

int getX();
int getY();
void setX(int): UpdateSignaling – after returning change();
void setY(int): UpdateSignaling – after returning change();
void moveBy(int, int): UpdateSignaling – after returning change();

Line

Point p1, p2;
Point getP1();
Point getP2();
void moveBy(int, int): UpdateSignaling – after returning change();

UpdateSignaling

after returning: change():
Point.setX(int), Point.setY(int), Point.moveBy(int, int),
Line.moveBy(int, int);
Interface Depends on Deployment

- Aspect cuts extended interface
  – through **Point** and **Line**

- Interface of **Point** and **Line**
  – depend on presence of aspects
  – and vice-versa

```java
class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
    }
    void setP2(Point p2) {
        this.p2 = p2;
    }
}

class Point {
    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;
    }
}

aspect UpdateSignaling {
    pointcut change(Shape shape):
        this(shape) &&
        (execution(void Shape.moveBy(int, int) ||
           execution(void Shape+.set*(*))));

    after(Shape s) returning: change(s) {
        Display.update(s);
    }
}
```
• The example has a weakness
  - x and y fields of Point are public
• The programmer decides to make x and y private.
• When doing this (s)he must ensure the system works as before.

```java
class Point {
    int x, y;
    ...
    void setX(int nx) {
        x = nx;
    }
}
```

We compare:
• reasoning with traditional interfaces about the non-AOP code against
• reasoning with AAIs about the AOP code.
Both implementations start out the same
• define accessors
• global reasoning to find references to fields
• change to use accessors
• simple change to `Line.moveBy` method

```java
void moveBy(int dx, int dy) {
    p1.x += dx;
    p1.y += dy;
    p2.x += dy;
    p2.y += dy;
}
```

```java
void moveBy(int dx, int dy) {
    p1.setX(p1.getX() + dx);
    p1.setY(p1.getY() + dy);
    p2.setX(p2.getX() + dx);
    p2.setY(p2.getY() + dy);
}
```

Is this change reasonable? Does it affect other concerns? What kind of reasoning do I need to reach a conclusion?
To discover the effect of this potential change – violation of the display updating invariant - the programmer needs to pieces of information:

- a specification of the invariant: "update the display after any top-level change of a figure element"
- structure of update signaling to infer that the invariant would be violated by the change.
• Discovering the invariant description
  – Nothing in Line is likely to describe the invariant.
  – Due to explicit call to Display.update(), the programmer might go look at the Display class.
    • We assume, optimistically, that update()’s documentation contains the invariant.
  – Expanded modular reasoning with one step leads the programmer to the invariant

• Discovering the structure of update signaling requires at least further expanded modular reasoning and in general global reasoning
• Add non-update-signaling setter methods to `Point` for the sole purpose of calling them from `Line.moveBy`?
  … maintenance nightmare

• The best I can do is probably to let `x` and `y` public… this is probably the reason why they were package public at first place!

• Information hiding is broken not by accident!
The interface of **UpdateSignaling** includes the complete structure of what method executions will signal updates.

- **modular reasoning** provides this information

Once the programmer understands that the change is invalid, the proper fix is to use **cflowbelow**:

```java
after() returning: change() && !cflowbelow(change())
{ Display.update(); }
```
Current modularity is not as good as claimed.

• With AOP,
  – its interface cuts through the classes,
  – the structure of that interface is captured declaratively,
  – the actual implementation is modularized

• Without AOP,
  – the structure is implicit
  – the actual implementation is not modular.
  – In presence of crosscutting concerns static modularity and modular reasoning are impaired
The cost: We must know the deployment setting to know the interface of a module.

- But, for CCCs we inherently have to pay the main cost of AOP.
- We have to know something about the total deployment configuration, in order to do the global reasoning required to reason about crosscutting concerns.

- By using AOP, we get modular reasoning benefits back, whereas not using AOP we do not.
- constructing aspect-aware interfaces is simple: pointcuts (or other mechanisms) can be declarative
To Hide or Not to Hide?

Agile Information Hiding

A disciplined way to establish additional interface properties without explicitly stating all of them in the interface.

"cut an interface through there and program to it"

"there is a well-defined interface" versus "has a well-defined interface"

```java
class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
    }
    void setP2(Point p2) {
        this.p2 = p2;
    }
}

class Point {
    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;
    }
}

aspect UpdateSignaling {
    pointcut change(Shape shape):
        this(shape) &&
        (execution(void Shape.moveBy(int, int) || execution(void Shape+.set(*(*))));

    after(Shape s) returning: change(s) {
        Display.update(s);
    }
}
```
What Else Have We Done

A Quick Tour on “my” AOP
The Caesar Story
Critique on AspectJ-like Languages

- Physical separation of aspect from base code
- Aspect described in terms of base application
- Unfair description: “Aspect = specification of how to patch the code such that an aspect is supported”
- Difficult:
  - reusable aspects
  - assignment of domain experts to aspects
- Aspects are “tangled”: use names from base application
- Physical separation is not enough!
The Goal by Analogy

- Layout
- Circuit structure
- Power consumption
- Heat emission

http://www.st.informatik.tu-darmstadt.de/
The Goal by Analogy
Crosscutting Models in CaesarJ

model superimposition by structural and behavioral mapping

Model 1

Model 2

if execution at certain joinpoints

Call certain method
The ALPHA Story
Critique on AspectJ-like Pointcuts

pointcut change():
    call(Point.setX(int))
    || call(void Point.setY(int))
    || call(void Shape+.moveBy(int, int));

instead of specifying WHAT the crosscutting structure is,

this pointcut describes HOW it appears in the concrete syntax of the program
"after data changes that was previously read during the most recent draw of a display, update that display"

Robust.
Minimal knowledge about implementation details of figures.

Precise.
Avoids unnecessary updates, e.g., after calls to `setX` modifying an `x` not read in control flow of `draw`. 
Can we express something like this in AspectJ?

Yes: Aspect constructs an automaton making extensive use of reflection less dependent on names, but … complex

"after data changes that was previously read during the most recent draw of a display, update that display"
Problems with Current Pointcuts ...

Management of observer lists for each field of each figure.

A list of the fields per instance observerd by each display.
This is necessary for the reset in the after advice for
pointcut `displayDraw`.

Denotes the action of drawing a display completely, i.e.,
each figure it knows.

Captures read accesses to any field in the
FigureElement hierarchy. The excluded field
`observersForFields` is introduced by the aspect.

Captures write access to any field in the
FigureElement hierarchy.

Removes d from observers of all the fields it observes.

Adds d to the observers of the read field of the figure f.

Calls `draw(f)` on all observers of the changed field of
figure f.

<table>
<thead>
<tr>
<th>&lt;&lt;aspect&gt;&gt;</th>
<th>DisplayUpdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introductions</td>
<td></td>
</tr>
<tr>
<td>FigureElement.getObserversForField(String) : List;</td>
<td></td>
</tr>
<tr>
<td>FigureElement.observersForFields : HasTable;</td>
<td></td>
</tr>
<tr>
<td>Display.getObserved() : List;</td>
<td></td>
</tr>
<tr>
<td>Display.observed : List;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>pointcuts</td>
<td></td>
</tr>
<tr>
<td>displayDraw(Display d):</td>
<td></td>
</tr>
<tr>
<td>call(void Display.drawAll()) &amp;&amp; target(d);</td>
<td></td>
</tr>
<tr>
<td>reads(Display d, FigureElement f):</td>
<td></td>
</tr>
<tr>
<td>cflow(displayDraw(d)) &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>get(* FigureElement+.*) &amp;&amp; target(f) &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>!get(java.util.HasTable FigureElement.observersForFields);</td>
<td></td>
</tr>
<tr>
<td>change(FigureElement f):</td>
<td></td>
</tr>
<tr>
<td>set(* FigureElement+.*) &amp;&amp; target(f)</td>
<td></td>
</tr>
<tr>
<td>advice</td>
<td></td>
</tr>
<tr>
<td>before(Display d): displayDraw(d)</td>
<td></td>
</tr>
<tr>
<td>after(Display d, FigureElement f) : reads(d,f)</td>
<td></td>
</tr>
<tr>
<td>after(FigureElement f): change(f)</td>
<td></td>
</tr>
</tbody>
</table>

Don't try to read this!
"after data changes that was previously read during the most recent draw of a display, update that display"

Challenges

Need knowledge about the execution: “previously read”, “most recent draw”…

Need powerful abstraction mechanisms similar to functional abstraction
The Programming Model of Alpha

 encode pointcuts as logic queries; pointcut “fires” if query has non-empty result

 high-Level user-defined pointcuts / 3rd party pointcut libraries

 ... uses/imports ... 

 pointcut abstraction via inference rules 

 low-level user-defined pointcuts / 3rd party pointcut libraries

 Store facts about program execution in an extensible list of logic DBs

 AST  Heap  Trace  Static typing ...
Pointcuts in ALPHA

```
class Main {
    display d;

    before set (P, F, _),
    get (T1, _, P, F, _),
    calls (T2, _, @this.d, draw, _),
    cflow(T1, T2),
    reachable (P, d)
    { ... }
    ...
}
```

Object specific pointcut really “talks” about itself … about “its slice of the execution.

“after data changes that was read during the most recent draw of d, update d”

Control flows in the past

http://www.st.informatik.tu-darmstadt.de/
two modules in A&B crosscut when projections of the modules into X intersect & neither is a subset of the other
The EScala Story
EScala in a Nutshell

http://www.st.informatik.tu-darmstadt.de/
EScala in a Nutshell

**ESScala**
- EBS
  - decoupled producer/consumer
- FRP
  - streams
  - data-driven programming
- OOP
  - Abstraction
  - Encapsulation
  - Modular compilation/loading
  - Dynamic structure
- AOP
  - Obliviousness
  - Implicit events
  - Global quantification
- Larger-scale object modules
  - A la Newspeak
• “modularity = information hiding” point of view is rooted in classical logic.
• Well-known limitations of classical logic as a representation formalism for human knowledge.
• Yet, information hiding is a undisputed dogma in programming.

• Programmers use non-classical reasoning in meaningful ways, they are humans too 😊
• Classical information hiding has its limitations
• Generally, it might be worth investigating notions of modularity based on non-classical logic.

• Some notions of modularity that escape classical modularity can be understood based on non-classical logics:
  – **AOP and default logic**
    – State, mutation, aliasing and separation logic
    – Error handling and para-consistent logics
• One can **reason by default** that the semantics of a method call is to execute the corresponding method body,

• **Aspects** that intercept such method calls are considered **exceptions to** that **default rule**.

• In this setting, one can - using defaults - reason locally about the program behavior.

• In case one learns later that the default assumption turns out to be wrong, there is a controlled process of updating the conclusions one has drawn from the invalid default assumption.
Each module - statement, expression, function, object - is a little "black box" - relates to the rest through a well-defined I/O interface (IO-wires).

Intuition underlying communication between modules:
- “sending pulses down a wire” - passing messages
- “single-point sampling of the world at the end of the wire” by algorithmic protocols

Lanier: “world as a planet of the help desks in which human race will be largely engaged in maintaining very large software systems …”
• Programmers forced to stream intentions into sequential steps aligned with this pipeline view of the world

• Complex algorithmic protocols needed to give meaning to sequences of pulses
  – accidental complexity!

• Pure hierarchical structuring
  – hard to accommodate different perspectives into pure hierarchical systems (crosscutting concerns)
• Components probe “measurable fundamental” properties of program execution and take decisions based on some evolving model of the world

– components connected by “surfaces” sampled at several points in parallel instead of “wires sampled at single points”

– pattern classification and automatic maintenance of implicit confirmatory and predictive models instead of sampling algorithmic protocols
There always exist different (hierarchical) logical sub-trees of origination, each of which is reigned by a principle (=archae) that cannot be subsumed under the guiding principles of the other trees.

Diversity of organizing principles is the basis of adaptability. In addition, adaptability if promoted by the organization of diversity.

“[T]he sphere of complexity is that of organized diversity, of the organization of diversity.”
Arthur Koestler. The Art of Creation

Looking at problems from different frames of references is argued to be at the core of the creativity process.
static void encodeStream(InputStream in, OutputStream out) {
    int readindex = 0;

    byte[] buff = new byte[N];

    while ( (readindex = in.read(buff)) == N) {
        out.write( Encoder.encodeDuration(buff) );
    }

    if (readindex > 0) {
        for (int i = readindex; i < N; i++) buff[i] = 0;
        out.write( Encoder.encodeDuration(buff) );
    }
}
static void encodeStream(InputStream in, OutputStream out) {
    int readindex = 0;

    byte[] buff = new byte[N];

    while ( (readindex = in.read(buff)) == N) {
        out.write( Encoder.encodeDuration(buff) );
    }

    if (readindex > 0) {
        for (int i = readindex; i < N; i++) buff[i] = 0;
        out.write( Encoder.encodeDuration(buff) );
    }
}

The problem aggravated if one has to write things like
“after data changes that was read during the
most recent draw of a display, update that display”
/**
 * encodeStream converts stream of bytes into sounds.
 * @param in stream of bytes to encode
 * @param out stream of audio samples representing input
 */

encodeStream(InputStream input, OutputStream output) {

    while there is data in input: read N bytes from it, perform encodeDuration on those bytes, and write result into output

    if, however, after reading the input, the number of bytes read is less than N, then, before continuing with writing out, patch it with zeros.

}