Modularity: what, why and how

Stephen Kell

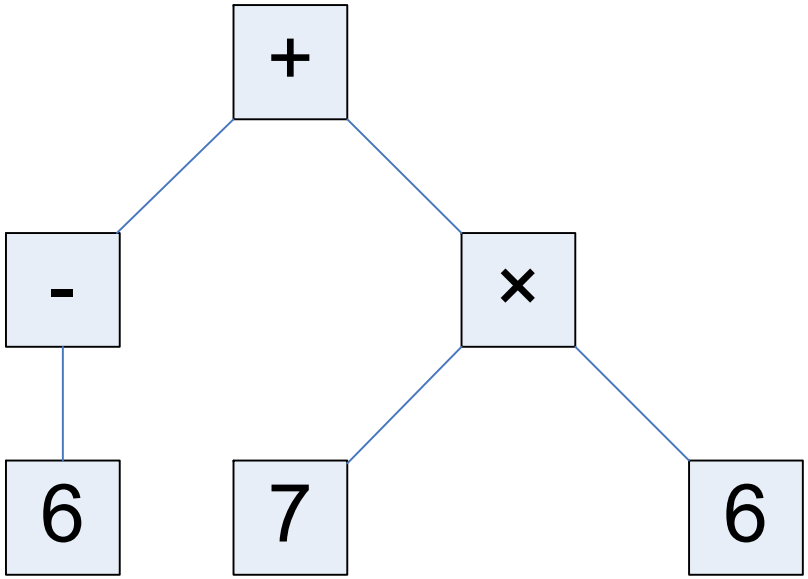
Stephen.Kell@cl.cam.ac.uk

Computer Laboratory

University of Cambridge

Some problematic code

Imagine implementing a syntax tree evaluator.



Your tree must

- support many kinds of node
- support many functions on nodes
- be *extensible* in a *modular* way

Functional programmer's solution

If you like functional programming, use an ADT:

```
data Expr = Lit  Int
          | Add Expr Expr
          | Neg Expr
          | Mul Expr Expr;
eval (Lit i) = i
eval (Add l r) = eval l + eval r
eval (Neg e) = − eval e
eval (Mul l r) = eval l ∗ eval r
```

Adding a new function is easy.

```
print (Lit i) = putStr(show i) −− etc. for other kinds of node
```

What about new kinds of node?

Object-oriented programmer's solution

If you like object-oriented programming, use interfaces:

```
interface Expr { int eval(); }
class Lit implements Expr { // field and constructor omitted ...
        int eval() { return i; } }
class Add implements Expr { // ...
        int eval() { return l.eval() + r.eval(); } }
class Neg implements Expr { // ...
        int eval() { return -e.eval(); } }
class Mul implements Expr { // ...
        int eval() { return l.eval() * r.eval(); } }
```

Adding new kinds of node is easy. New functions?

The expression problem

“A new name for an old problem.”

(Wadler, “The expression problem”,
a mail to `java-generic`ity, 1998.)

A good solution would

- keep related changes together
  - . . . rather than scattered throughout code
- avoid edits to existing code
  - . . . which might create maintenance problems

Why are these good? Something to do with *modularity*. . .

Outline of this lecture

- Expression problem intro
- **What is modularity?**
- Founding wisdom
- Openness-based approaches
- More postmodern approaches
- Notions of *module*
- Assorted research approaches

What is modularity?

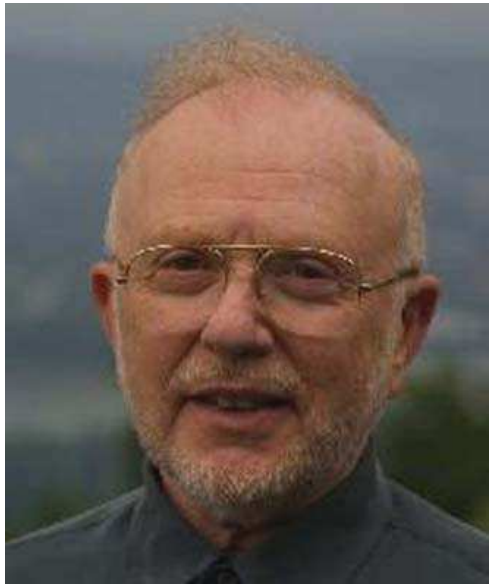
"Modularity" usually conflates a few related goals.

- keep related things together
  - don't repeat yourself
  - if you have to, keep the repetitions close together
- keep unrelated things separate
  - avoid tangling concerns
  - avoid embedding change-prone assumptions

Why?

- fewer chances to make a mistake
- change confined to one place (ditto)
- maximise compositionality

On the criteria. . .



"It is almost always incorrect
to begin the decomposition of a system
into modules on the basis of a flowchart.

"We propose instead
that one begins with a list of . . . design
decisions which are *likely to change*. . ."

(Parnas, "On the criteria to be used in decomposing systems into modules", CACM 15(12), 1972.)

Use modularity for *change-robustness*.
Information hiding → data abstraction → CLU →

- . . . → private, protected etc.

Modular decomposition in our example (1)

The modular decomposition in Haskell looks like this.

ADT	eval	print	someOther
Lit	pattern	pattern	pattern
Neg	pattern	pattern	pattern
Add	pattern	pattern	pattern
Mul	pattern	pattern	pattern

Each outlined box denotes a “closed” definition in the language.

Modular decomposition in our example (2)

The modular decomposition in Java looks like this.

interface	eval	print	someOther
Lit	method	method	method
Neg	method	method	method
Add	method	method	method
Mul	method	method	method

Here, decomposition is along a different *dimension*.

Structured design

"Mr Constantine
has observed that programs that were the easiest to implement and change were those composed of simple, independent modules.

"Problem solving
is hardest when all aspects of the problem must be considered simultaneously."

(Stevens, Myers & Constantine,"Structured design", IBM R&D Journal vol 13, 1974.)

This introduced the ideas of *coupling* and *cohesion*.

Coupling and cohesion

Coupling is bad: minimise it.

- the extent to which *changes* in one module…
- … entail changes in another

Cohesion is good: maximise it.

- the extent to which the various contents of a single module…
- … are related to one another

Problems with coupling and cohesion

Cohesion is underspecified: *what* should cohere?

- coherence occurs along different dimensions…
- … as shown by expression problem

Low coupling is trickier than it sounds.

- *any* vocabulary (for data, functions) entails coupling…
- … but can't avoid choosing *some* vocabulary!

Both trade off against a hidden enemy: too many modules

An incremental solution to the expression problem

Languages offering “open” constructs enable “no editing”.

```
-- Don't use an ADT (closed); use a "type class" (open)
class Exp x
-- Declare each kind of node as its own ADT (one constructor only)
data (Exp x, Exp y) => Add x y = Add x y
-- ...
-- Declare each such ADT to be an instance of the type class
instance (Exp x, Exp y) => Exp (Add x y)
-- ...
-- Define a type class for each function
class Exp x => Eval x where eval :: x -> Int
-- Code is defined per-function, per-kind-of-node
instance (Eval x, Eval y) => Eval (Add x y)
where eval (Add x y) = eval x + eval y
```

The finer-grained decomposition

Exp typeclass	Eval	Print	SomeOther
Lit	instance	instance	instance
Neg	instance	instance	instance
Add	instance	instance	instance
Mul	instance	instance	instance

Incrementality: locality lost

These solutions work by

- breaking previously “closed” *complete* definitions…
- …into “open” *partial* definitions;

A corollary: what was once together is now separate.

- we have cohesion only at a small scale
  - imaging binning each instance declaration in the Haskell code…
  - …into one of a smaller set of source files
  - would you decompose by node kind, or by operation?

Separation of concerns

In the same year as Stevens, Dijkstra wrote:

“To study in depth an
aspect of one’s subject matter in isolation
… all the time knowing that [it is] only
one of the aspects… is what I sometimes
have called ‘the separation of concerns’.”

(Dijkstra, “On the role of scientific thought”, EWD447, 1974.)

Locality regained?

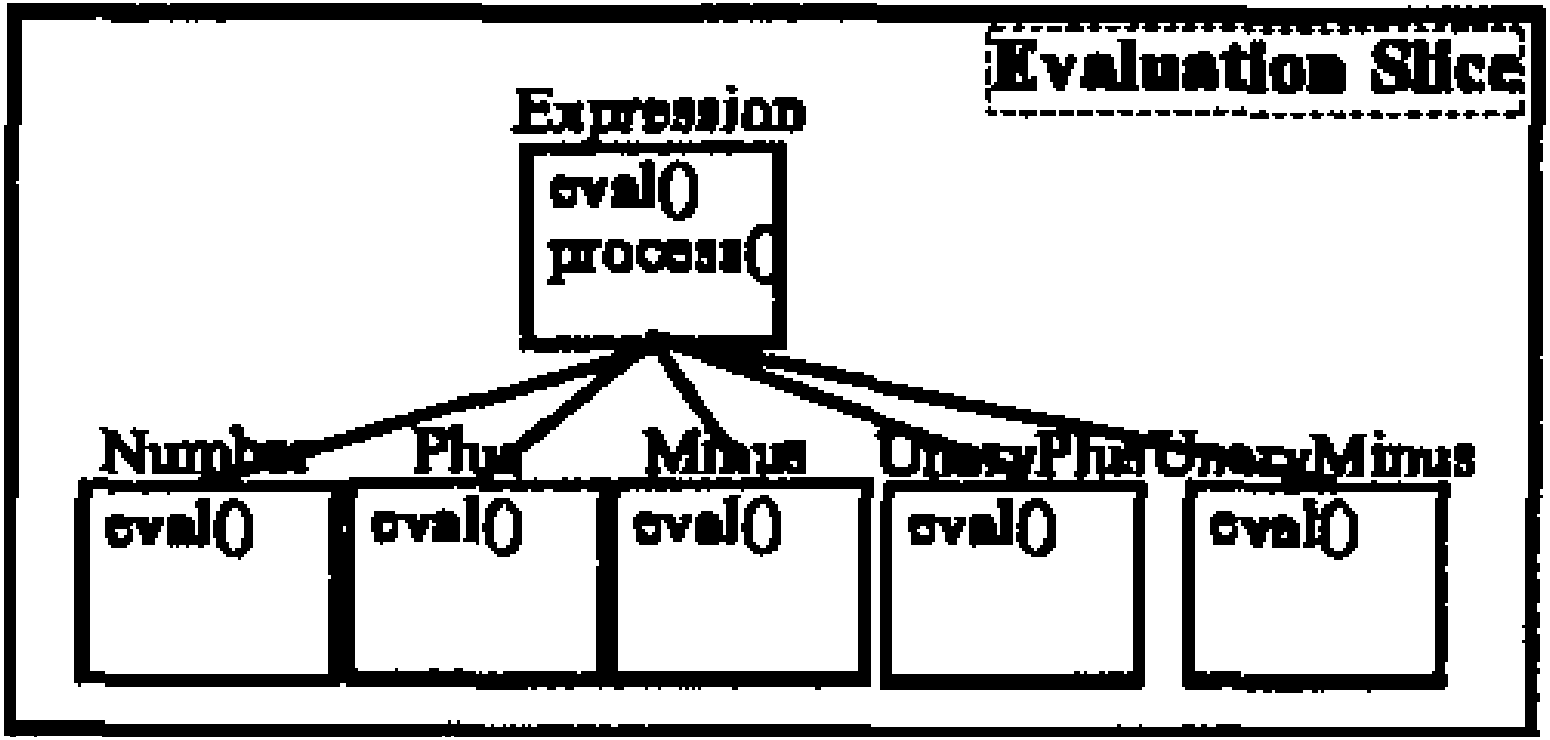
Why need there be *one true decomposition* of a system? Instead,

- describe different parts of a system. . .
- . . . using different decompositions!
- compose in a separate step

This is the basis of *multi-dimensional separation of concerns* (Tarr et al., ICSE 1999).

- “hyperslice” notion of decomposition
- tool support for *on-demand remodlarization*
- Hyper/J implementation for Java

Multi-dimensional separation of concerns (1)

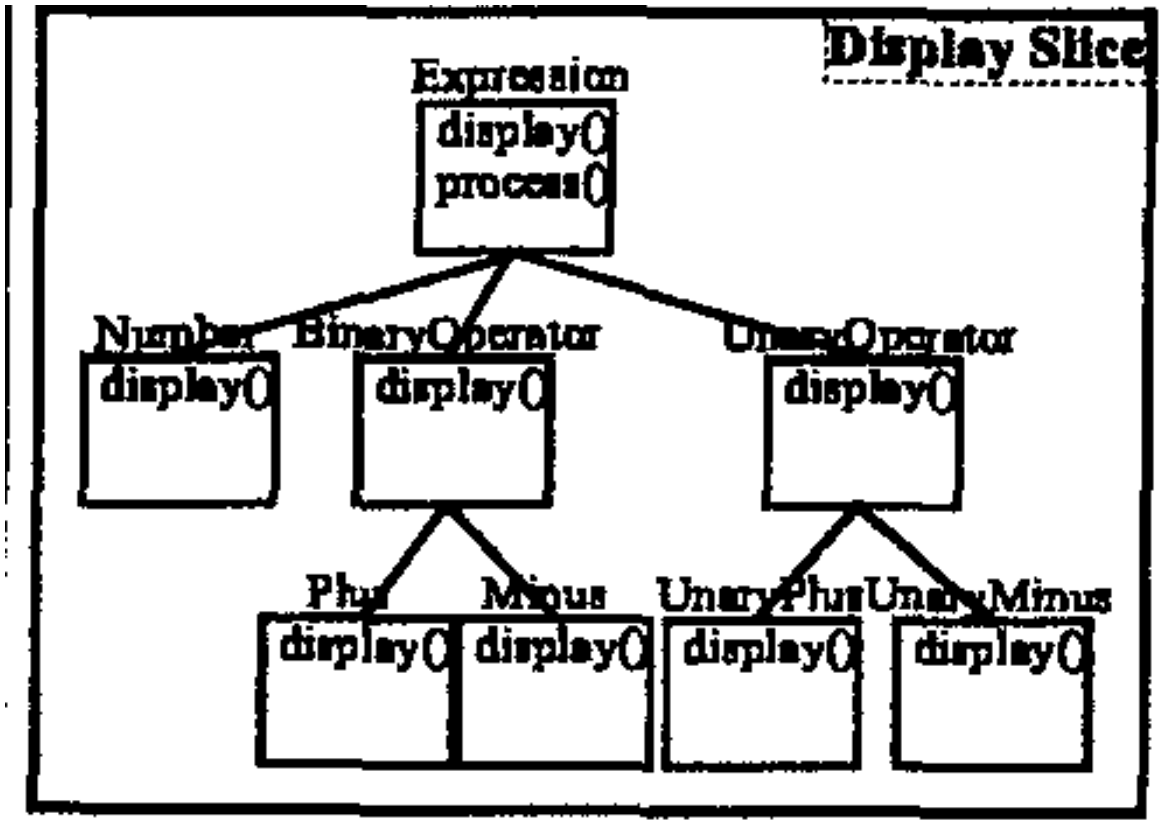


This is a by-node decomposition

- good for adding new kinds of node

Could also have sliced by method…

Multi-dimensional separation of concerns (2)



Slices are more general than “by function” or “by class”:

- Here we slice some common code out of display()
- . . . to isolate code shared between nodes of given arity

Aspect-oriented programming (1)

The expression problem is hard because

- no *one* separation of the two dimensions is sufficient
- … MDSoC's power is the ability to pick and choose

Often our problem is simpler:

- a system can be factored into "base" and "extension"
- … where the extensions are mostly independent…
- … but would ordinarily require "scattered" code

AspectJ (2)

AspectJ exploits the “base” versus “extension” distinction.

- idea: describe extensions separately, then . . .
- . . . splice the code in at compile time
- “pointcuts” are expressions defining points in execution
  - e.g. call(void Point.setX(int))
  - matches just before any call to Point.setX(int))
  - a sort of query over events at run time
- “advice” is code that is spliced in (“woven”) at these points
  - e.g. before(): System.out.println(”about to move”);

Uses of AOP

Applications of AOP:

- canonical, dull example: adding logging
- real, exciting example: prefetching in BSD 3.3!
  - Coady, ESEC/FSE 2001
- any “extension”-style feature or extrafunctional change. . .
- . . .

Quantification and obliviousness

Aspects have some useful modularity properties

- quantification
  - "for all join points matching P, do this…"
  - enables locality
    - gather in one place related logic that applies to many
- obliviousness
  - implies non-invasiveness
  - also implies *unanticipatedness* (stronger)
    - original code needn't be *designed for* extension
      - cf. the Haskell typeclass expression example
- see Filman & Friedman, 2000

The “paradox” (trade-off) of AOP

AOP is controversial.

- e.g. Steimann, “The paradoxical success of AOP” (Onward! ’06)
- to gain some modularity (less scattering of feature code)…
- AOP trades off some other (strong coupling between aspect and class)
- This can sometimes be a good-value trade… not always.

It prompts us to investigate notions of *module*.

Overview and interval

- Expression problem intro
- What is modularity?
- Founding wisdom
- Openness-based approaches
- More postmodern approaches
- **Notions of *module***
- Assorted research approaches

Information hiding again

Spot the difference:

```
before()  call(void Point.setX(int)):
{ System.out.println("about to move"); }
```

```
--- a/Pos.java      2001-01-26 23:30:52.000000000 +0000
+++ b/Pos.java      2008-05-09 15:05:39.396998000 +0100
@@ -36,7 +36,12 @@
        // update our position
+       System.out.println("about to move");
        myPoint.setX(42);
```

- patches are a kind of module. . .
  - . . . if a bad one—very brittle!
- patches and aspects have something in common. . .

Information hiding lost

Aspects are an improvement on patches. . .

- . . . better localised, more abstract, less syntactic
- but still, pointcuts can range over *any* code internals
  - double-edge: enough power to blow off both feet

Call this a *white-box* approach

- cf. black-box, where [some] internals are *hidden*
- (reality: many shades of grey)

White- versus black-box composition

White-box examples:

- patches
- slices (incl. hyperslices)
- aspects
- superimpositions (Apel, 2009)

Black-box examples:

- abstract data types
- most PLs' "module" (and sim.) constructs
- mixins, features, virtual classes, processes, actors, . . .
- adapters (see Yellin & Strom, OOPSLA 1994)

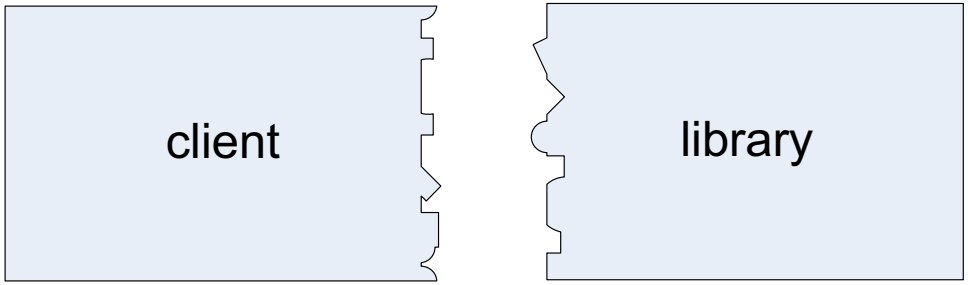
Cake (1): separating the concern of integration

Modularity is hard.
PLs make simplifying assumptions.

- code in ground-up order
- components fit perfectly…
- & are homogeneous (wrt lang)
- interfaces don’t change
- components are never replaced

Reality: none of the above!

Cake (2): interface relations



Don’t write wrappers (tedious); describe *correspondences*.

```
client ↔ library
{
  // initialization
  mpeg2_init() → { avcodec_init();
                   av_register_all(); }
  // data structure representing an open stream
  values FILE ↔ AVFormatContext {};
  // ...
}
```

Conclusions

Modularity is a deceptively subtle problem which:

- balance many different goals
  - locality (avoid repetition; high cohesion)
  - change-robustness (compositionality; low coupling)
  - other aspects: obliviousness / anticipation…
  - hidden issues: performance, complexity
- influences many aspects of language and tool design
  - open versus closed abstractions
  - white-box versus black-box composition
  - special-purpose languages …

Bibliography and acknowledgements

Papers to read:

- Parnas, 1972
- Stevens, 1974
- Harrison, 1992
- Kiczales, 1997
- Tarr, 1999
- Filman & Friedman, 2000
- Steimann, 2006
- Kell 2009a, 2009b

Acknowledgements:
Ostermann and Laemmel paper
photo attributions