

# **Programming Language Support for Separation of Concerns**

Doug Orleans

Northeastern University

College of Computer and Information Science

`dougo@place.org`

# Thesis statement

The Socrates programming language unifies and generalizes object-oriented and aspect-oriented language mechanisms that support separation of concerns.

## Questions:

- What is separation of concerns?
- How do OO and AO languages support SOC?
- What is Socrates?
- How does Socrates unify and generalize OO and AO support for SOC?
- So what?

# Separation of concerns

[S]tudy in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. [...] It is what I sometimes have called "the separation of concerns"

Edsger W. Dijkstra, "On the role of scientific thought", 1974

SOC is a problem-solving principle:

- To solve a complicated problem, break it down into easier, more manageable subproblems, and solve each of these smaller problems individually.

SOC is also a program-writing principle:

- To write a complicated program, break it down into easier, more manageable subprograms, and write each of these smaller programs individually.

# OOP language support for SOC

Object-oriented programming languages support separation of concerns with inheritance and polymorphism. Different concerns can be implemented as different classes such that one is a subclass or sibling class of another.

```
abstract class File { abstract int getSize(); }
```

```
class RegularFile extends File {  
    byte[] contents;  
    int getSize() { return contents.length; }  
    byte[] getBytes(int offset, int length) {  
        byte[] bytes = new byte[length];  
        System.arraycopy(contents, offset, bytes, 0, length);  
        return bytes; }  
}  
class Directory extends File {  
    java.util.Map entries;  
    File getEntry(String name) { return (File) entries.get(name); }  
    int getSize() {  
        int size = 0; java.util.Iterator i = entries.values().iterator();  
        while (i.hasNext()) size += ((File) i.next()).getSize();  
        return size; }  
}
```

## AOP language support for SOC

Sometimes concerns cannot be separated into classes. Aspect-oriented programming languages support more flexible separation of concerns.

- AspectJ: advice and inter-type declarations
- Hyper/J: hyperslices and integration relationships
- ComposeJ: composition filters
- DemeterJ: class dictionaries and adaptive methods

# AspectJ example

```
aspect TimeStamping {
    long File.lastUsed = System.currentTimeMillis();
    before(File file):
        (call(* getBytes(..)) || call(* getEntry(..))
         || call(* getSize(..)))
        && target(file)
    {
        file.lastUsed = System.currentTimeMillis();
    }
}
```

```
aspect PermissionChecking {
    boolean File.isReadable = true;
    before(File file):
        (call(* getBytes(..)) || call(* getEntry(..)))
        && target(file)
        && if (!file.isReadable)
    {
        throw new RuntimeException("Access denied.");
    }
}
```

# Relationships between AOP and OOP

- AOP extends OOP:
  - AspectJ, Hyper/J, ComposeJ, DemeterJ all add AOP structures to an OOP language (Java)
  - Other AOP languages extend Smalltalk, C++, Ruby
- "AOP complements OOP. AOP can complement other paradigms, like procedural programming."

Gregor Kiczales, July 2003 interview with TheServerSide.com

- AspectC, AspectScheme, AspectML
- My view:
  - AOP is (almost!) a generalization of OOP
  - AOP and OOP mechanisms can be unified.

# The Socrates programming language

Socrates is implemented as a language embedded into PLT Scheme.

- Socrates is a collection of library modules containing Scheme definitions: procedures, structures, and syntax.
- The `socrates.ss` language library provides all of the libraries in the collection, plus all of the MzScheme language.
  - Socrates code and Scheme code can be freely intermixed.
  - In particular, a Socrates program will use some Scheme syntax (`define`, `let`, `lambda`) as well as some Scheme procedures (`eq?`, `cons`, `apply`, etc.)
  - Also, Socrates values are first-class Scheme values and can be manipulated by Scheme procedures.

# The Socrates programming language

- The `socrates-core.ss` library provides six basic structures along with procedures to manipulate them.
  - "static" values that make up a program: **messages, branches, fields**
  - "dynamic" values used while running a program: **decision points, objects, contexts**
- Additional libraries provide layers of abstraction (mostly just syntactic sugar) on top of the core, to emulate higher-level constructs from other languages, such as multimethods, predicate dispatching, generalized open classes, pointcuts and advice, structural pattern matching, classifiers, and composition filters.

# Messages

A **message** represents an abstract operation.

- `(make-msg)` returns a new message.
- `(msg-send msg args)` sends `msg` to the `args` list.

A message is also an applicable Scheme procedure value, so it can be sent to a list of arguments directly with an application expression. In other words, the following forms are all equivalent:

- `(msg arg ...)`
- `(apply msg (list arg ...))`
- `(msg-send msg (list arg ...))`

## Decision points

A **decision point** is constructed automatically whenever a message is sent, encapsulating some information about the dynamic context.

- `(dp-msg dp)` returns the message being sent.
- `(dp-args dp)` returns the list of arguments.
- `(dp-within dp)` returns the branch currently being followed.
- `(dp-previous dp)` returns the decision point that caused the current branch to be followed.

# Branches

A **branch** represents a concrete unit of behavior that implements parts of one or more operations. A branch specifies **what to do** (the body) and **when to do it** (the predicate).

- **(make-branch pred-proc body-proc)** returns a new branch.
  - **pred-proc** is a procedure that takes one argument, a decision point, and returns a true value if the branch is applicable, or **#f** otherwise.
  - **body-proc** is a procedure that takes one argument, the return value of the branch's predicate.

A global dispatch table stores the set of active branches.

- **(add-branch branch)** adds **branch** to the dispatch table.
- **(remove-branch branch)** removes **branch** from the dispatch table.

# Dispatching

When a message is sent, the decision point is passed as an argument to the predicate of every branch in the dispatch table.

- Among the applicable branches (those that returned true), the one with the highest precedence is determined and its body is invoked, with the value returned by its predicate as the argument.
- If no branches are applicable, the `exn:msg:not-understood` error is raised.
- If there is no single applicable branch with higher precedence than the rest, the `exn:msg:ambiguous` error is raised.

## Branch precedence

Branch precedence is determined by predicate implication analysis.

- A branch with predicate  $p_1$  has precedence over a branch with predicate  $p_2$  if  $p_1$  can be proven to logically imply  $p_2$ .
- $p_1$  logically implies  $p_2$  if  $p_2$  returns true for all decision points for which  $p_1$  returns true.

```
(define (p1 dp)
  (eq? (dp-msg dp) get-size))
```

```
(define (p2 dp)
  (and (eq? (dp-msg dp) get-size)
       (file? (car (dp-args dp)))))
```

$p_2$  implies  $p_1$ , because  $x$  is always true when  $(\text{and } x \text{ } y)$  is true.

## Branch precedence

Predicate implication is a generalization of the usual object-oriented method-overriding relation, in which subclass methods override superclass methods.

```
(define (p2 dp)
  (and (eq? (dp-msg dp) get-size)
       (file? (car (dp-args dp)))))
```

```
(define (p3 dp)
  (and (eq? (dp-msg dp) get-size)
       (regular-file? (car (dp-args dp)))))
```

$p_3$  implies  $p_2$ , because `(file? x)` is always true when `(regular-file? x)` is true.

## Branch precedence

Sometimes a different precedence relation is needed:

```
(define (p4 dp)
  (and (eq? (dp-msg dp) get-bytes)
       (regular-file? (car (dp-args dp)))))

(define (p5 dp)
  (and (eq? (dp-msg dp) get-entries)
       (directory? (car (dp-args dp)))))

(define (p6 dp)
  (and (or (eq? (dp-msg dp) get-bytes)
          (eq? (dp-msg dp) get-entries))
       (file? (car (dp-args dp)))
       (not (readable? (car (dp-args dp))))))
```

The branch for  $p_6$  needs to have higher precedence than the branches for  $p_4$  and  $p_5$  so that it can raise an error.

## Around-branches

An **around-branch** is just like a plain branch, but always has higher precedence than all plain branches:

```
(make-around p6 (lambda (_)  
                  (error "Access denied.")))
```

- Precedence between around-branches is determined by predicate implication analysis, as usual.

Around-branches are additive:

- When two around-branches are applicable and neither predicate implies the other (or they both imply each other), the more recently defined around-branch has precedence.
- In other words, around-branches never cause an ambiguous message error.

## Incremental behavior

A branch can provide incremental behavior by temporarily passing control in its body to the next most precedent applicable branch.

(**follow-next-branch** **filter-proc**) invokes the next most precedent applicable branch.

- **filter-proc** is a procedure that takes one argument, the value returned by the next branch's predicate.
- The value returned by the filter procedure is passed to the body of the next branch.

The filter procedure can be used to modify, add to, or replace the information returned by the next branch's predicate. This is similar to changing the arguments to a **super** call in Java or **proceed** in AspectJ.

## Fields and objects

A **field** represents a set of associations between values.

- `(make-field)` returns a new field.
- `(set-field-value! field key v)` associates `v` with `key` in `field`, replacing any previous association for `key`. Keys are compared with `eq?` and are weakly held.
- `(field-value field key)` returns the value associated with `key` in `field`.

An **object** is a unique value that can be used as a key in a field.

- `(make-object class)` returns a new object. `class` is a procedure (or message) that returns true for the object.
- `(object? v)` returns `#t` if `v` is an object, `#f` otherwise.

# Contexts

A **context** is a way to pass information from a branch's predicate to its body using a set of name bindings.

- `(make-context bindings-alist)` returns a new context. `bindings-alist` is an association list mapping symbols to thunks (procedures with no arguments).
- `(context-value context sym)` returns the value resulting from invoking the thunk bound to the symbol `sym` in `context`.
- `(append-contexts context ...)` returns a new context containing all the bindings in the `context` s. If the same name is bound in more than one `context`, the leftmost binding takes precedence in the new context.

# OOP syntactic sugar

```
(module filesystem (lib "socrates.ss" "socrates")
  (provide (all-defined))
  (begin
    (define-class file? () ())
    (define-msg get-size))
  (begin
    (define-class regular-file? (file?) (contents))
    (define-method (get-size (regular-file? x))
      (bytes-length (get-contents x)))
    (define-method (get-bytes (regular-file? x)
                      (integer? offset)
                      (integer? length))
      (subbytes (get-contents x) offset (+ offset length))))
  (begin
    (define-class directory? (file?) (entries))
    (define-method (get-entry (directory? x) (string? name))
      (hash-table-get (get-entries x) name (lambda () #f)))
    (define-method (get-size (directory? x))
      (apply + (hash-table-map (get-entries x)
                              (lambda (name file)
                                (get-size file))))))
```

# AOP syntactic sugar

```
(module aspects (lib "socrates.ss" "socrates")
  (require "filesystem.ss")
  (provide (all-defined))
  (begin
    (define-field last-used file?)
    (define-after-init-method ((file? x) . _)
      (set-last-used! x (current-inexact-milliseconds)))
    (define-before (&& (call get-size get-bytes get-entry)
      (bind-args ((file? x) . _)))
      (lambda-context (x)
        (set-last-used! x (current-inexact-milliseconds))))))
(begin
  (define-field readable? file? #t)
  (define-before (&& (call get-bytes get-entry)
    (bind-args ((file? x) . _)
      (not (get-readable? x))))
    (lambda-context ()
      (error "Access denied."))))))
```

# Method and advice sugar

```
(define-method (name (type-pred var) ...)
  [ & pred-expr ]
  body-expr ...)
```

==>

```
(ensure-msg name)
(define-branch (&& (call name)
               (bind-args ((type-pred var) ...)
                           [ pred-expr ]))
(lambda-context (var ...)
  body-expr ...))
```

```
(define-before pred-proc body-proc)
```

==>

```
(define-around pred-proc
  (lambda (ctx)
    (begin (body-proc ctx)
           (follow-next-branch (lambda (ctx) ctx))))))
```

```
(ensure-msg name) ==> (define name (make-msg))
```

if `name` is not already bound

```
(define-branch p b) ==> (add-branch (make-branch p b))
```

```
(define-around p b) ==> (add-branch (make-around p b))
```

# Pointcut sugar

```
(call msg ...)
```

```
==> (lambda (dp)
      (or (eq? (dp-msg dp) msg) ...))
```

```
(bind-args ((type-pred var) ...)
 [ pred-expr ])
```

```
==> (lambda (dp)
      (match-let (((var ...) (dp-args dp)))
        (and (type-pred var) ...
              [ pred-expr ]
              (return (var var) ...))))
```

```
(cflow pred-proc) ==> (|| pred-proc (cflowbelow pred-proc))
```

```
(cflowbelow pred-proc)
```

```
==> (lambda (dp)
      (and (dp-previous dp)
           ((cflow pred-proc) (dp-previous dp))))
```

```
(&& pred-proc ...) ==> (lambda (dp) (and (pred-proc dp) ...))
```

```
(|| pred-proc ...) ==> (lambda (dp) (or (pred-proc dp) ...))
```

# Context sugar

```
(return (var expr) ...)
```

```
==> (make-context  
      (list (cons 'var (lambda () expr))  
            ...))
```

```
(lambda-context (var ...)   
  body-expr ...)
```

```
==> (lambda (ctx)  
      (let ((var (context-value ctx 'var))  
            ...))  
      body-expr ...))
```

```
(follow-next-branch (var expr) ...)
```

```
==> (follow-next-branch*  
      (lambda (ctx)  
        (append-contexts  
          (return (var expr) ...)   
          ctx)))
```

# Class and field sugar

```
(define-class class-name (superclass) (field-name ...))
```

```
(define-method (class-name x) #f)
(define-method (class-name (object? x))
  & (eq? (object-class x) class-name)
  #t)
==> (define-method (superclass (object? x))
  & (eq? (object-class x) class-name)
  #t)
(define-field field-name class-name)
...
```

```
(define-field field-name type-pred)
```

```
(ensure-msg get-field-name)
(ensure-msg set-field-name!)
(let ((field (make-field)))
  (define-method (get-field-name (type-pred key))
    (field-value field key))
  (define-method (set-field-name! (type-pred key) value)
    (set-field-value! field key value)))
==>
```



# Law of Demeter for Socrates

The Law of Demeter is a style rule for object-oriented design. In order to reduce dependencies between classes (or modules), a method should only send messages to a restricted set of closely-related objects; the motto is "don't talk to strangers".

## **The Law of Demeter for Socrates** (informal definition)

In the body of a branch, messages may only be sent to:

- arguments of the current decision point
- values associated with the arguments
- objects created while following the branch
- free variables

## Law of Demeter checker

A Law of Demeter checker can be implemented in Socrates as a set of branches that dynamically detect and report violations.

```
(define (lod-violation? dp)
  (let* ((prev (dp-previous dp))
        (potential-preferred-suppliers
         (append (dp-args prev)
                 (get-associated-values prev)
                 (get-created-objects prev)
                 (free-var-values (branch-body (dp-within dp))))))
        (illegal-suppliers
         (lset-difference eq? (filter object? (dp-args dp))
                          potential-preferred-suppliers)))
    (and (not (null? illegal-suppliers))
         (return (dp dp) (ill illegal-suppliers)))))

(define-before
  (&& (cflowbelow ?)
    (! (args dp? ..))
    (! (cflow (call report-lod-violation)))
    lod-violation?)
  (lambda-context (dp ill))
```

## Checking the domino puzzle GUI

An experiment: run the Law of Demeter checker on the domino puzzle GUI. `check`

- Good stress test of the Socrates implementation; shook out a lot of bugs, mostly in the predicate implication analysis code.
  - some unimplemented features that hadn't been needed before
  - some infinite recursion bugs due to the very reflective nature
  - some efficiency problems due to exponential blowups
- It reports many, many violations. A lot are false positives, mostly due to mixing Scheme and Socrates, but it can be difficult to tell. A few are actual violations.
- There may also be false negatives, but I haven't looked for them yet.

## An actual violation

```
(define-predicate (has-neighbor? x dir)
  (let ((pos (offset (get-position x) (get-offset dir)))
        (polyomino (get-polyomino x)))
    (and (in-range? pos (get-dimensions polyomino))
         (get-tile polyomino pos))))
```

Law of Demeter violation!

Branch: (method-call (draw-border (dc? dc) (tile? tile)))

Previous: (has-neighbor? #<object:tile?> north)

Message: (in-range? #f #<object:dimensions?>)

Illegal supplier: #<object:dimensions?>

- `x` is a preferred supplier
- `(get-polyomino x)` is a preferred supplier
- `(get-dimensions (get-polyomino x))` is not.

Fixed by breaking into two predicate methods.

## A false positive

```
(define-predicate (has-neighbor? x dir)
  (let ((pos (offset (get-position x) (get-offset dir)))
        (polyomino (get-polyomino x)))
    (has-tile? polyomino pos)))
(define-predicate (has-tile? polyomino pos)
  (and (in-range? pos (get-dimensions polyomino))
       (get-tile polyomino pos)))
```

Law of Demeter violation!

Branch: (method-call (draw-border (dc? dc) (tile? tile)))

Previous: (has-neighbor? #<object:tile?> north)

Message: (offset #<object:position?> #<object:offset?>)

Illegal supplier: #<object:offset?>

The implementation only keeps track of associations between objects, but `dir` is a symbol.

Keeping track of associations for non-objects is slower, more memory-intensive, and seems to cause different false positives.

# Predicate implication analysis

# Contributions

# Conclusion

# Future research