Incremental Programming with Extensible Decisions

Doug Orleans
College of Computer Science
Northeastern University
`dougo@ccs.neu.edu`

February 5, 2002

Incremental Programming

“The construction of new program components by specifying how they differ from existing components.” [Cook & Palsberg, OOPSLA 1989]

A language that supports incremental programming:

- allows good separation of concerns
- reduces code duplication
- improves extensibility and reuse

Extensible Decisions

In OOP, whenever a message is sent, a decision occurs (dynamic dispatch), but the branches of the decision are specified separately: methods that correspond to the message signature.

Incremental programming is supported because methods can override each other, but only through inheritance. Different branches of an extensible decision must involve different classes.

Design Patterns

Many behavioral design patterns [Gamma et al, 1994] are essentially workarounds for this constraint of OOP. For example, the State pattern implements state-based dispatch with one class per state.

But. . .

- Objects must delegate all state-dependent messages to their state objects: runtime overhead and a coding burden.
- The state objects must be manually kept up to date when the state condition changes: no implicit states.

A Different Solution

Change the language to make programs easier to express instead of changing your program to fit the constraints of the language.

Two approaches to allowing incremental programming without requiring inheritance:

- Aspect-Oriented Programming (AOP) [Kiczales et al, 1997]

- Predicate Dispatching [Ernst, Kaplan, & Chambers, ECOOP 1998.]

Aspect-Oriented Programming (AspectJ)

Incremental programming for crosscutting concerns: an aspect can override behavior in other classes without using inheritance.

- Each piece of advice has a **pointcut** which specifies when the advice is applicable to a message send.

- Pointcuts can be arbitrary boolean expressions.

Predicate Dispatching

A form of dynamic dispatch that unifies and generalizes the dispatch mechanisms found in many programming languages, including OO single and multiple dispatch.

- Each method implementation has a **predicate** which specifies when the method is applicable to a message send.

- Predicates can be arbitrary boolean expressions.

Advantages of AspectJ Over Predicate Dispatching

- Pointcuts can access more information about a message send than just the receiver and arguments:
  - message signature
  - control flow history
  - location of message send code

- Method combination: `before`/`after` advice, `proceed()` in `around` advice

Advantages of Predicate Dispatching Over AspectJ

- More elegant model, natural generalization of OO dispatch. Not tacked onto the side of Java.

- Method overriding is based on logical implication:

  - A method with predicate p_1 overrides a method with predicate p_2 if p_2 is true in all cases where p_1 is true.

  - Example: is-a(x, A) $\Longrightarrow$ is-a(x, B) (where A is a subclass of B)

  - I.e., subclass methods override superclass methods, just like in OOP.

Fred

A new programming language that takes the best from both worlds.

Fred's dispatch mechanism unifies those of AOP and OOP languages, and provides uniform support for incremental programming whether the concerns implemented by the components are crosscutting or not.

- Behavior is specified as **branches**, each of which has a predicate over **decision points**.

- Decision points capture message signature, message arguments, source branch, previous decision point.

Fred (continued)

- Overriding is based on logical implication of predicates, but `around` branches always override plain branches.
- `invoke-next-branch` allows method combination.
- Syntactic sugar allows more declarative syntax for common kinds of branches, to look more like multimethods or advice.

Fred Example (1/4)

```
(define-msg fact)
(define-branch (lambda (dp) (and (eq? (dp-msg dp) fact)
                                 (= (car (dp-args dp)) 1)))
               (lambda (dp) 1))
(define-branch (lambda (dp) (and (eq? (dp-msg dp) fact)
                                 (integer? (car (dp-args dp)))))
               (lambda (dp) (let ((n (car (dp-args dp))))
                              (* n (fact (- n 1))))))
```

With sugar:

```
(define-method fact ((= n 1))       1)
(define-method fact ((integer? n)) (* n (f (- n 1))))
```

Fred Example (2/4)

```
(define-around
  (lambda (dp) (and (eq? (dp-msg dp) fact)
                    (not (and (dp-previous dp)
                              (eq? (dp-msg (dp-previous dp))
                                   fact)))))
  (lambda (dp) (let ((n (car (dp-args dp))))
                 (display (list 'fact n)) (newline)
                 (invoke-next-branch))))
```

With sugar:

```
(define-before (&& (call fact) (! (cflow (call fact))))
  (with-args (n)
    (display (list 'fact n)) (newline)))
```

Fred Example (3/4)

```
(define-around
  (lambda (dp) (and (eq? (dp-msg dp) fact)
                    (dp-previous dp)
                    (eq? (dp-msg (dp-previous dp)) fact)))
  (lambda (dp) (let ((n (car (dp-args dp))))
                 (display '...) (display n) (newline)
                 (let ((fact-n (invoke-next-branch)))
                   (display fact-n) (newline)
                   fact-n))))
```

Fred Example (4/4)

With sugar:

```
(define-around (&& (call fact) (cflow (call fact)))
  (with-args (n)
    (display '...) (display n) (newline)
    (let ((fact-n (invoke-next-branch)))
      (display fact-n) (newline)
      fact-n)))
```

Fred Example Output

```
> (fact 5)
(fact 5)
...4
...3
...2
...1
1
2
6
24
120
```

Future Work

- Compare with other AOP models (composition filters, hyperslices, mixin layers, variation-oriented programming)
- Variable binding in predicates
- Customized branch overriding relationships
- Extensible predicates
- Modularization: **bundles**, based on units [Flatt & Felleisen, PLDI 1998]