Extensibility for DSL design and implementation
A case study in Common Lisp

Didier Verna
didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

DSLDI 2013 – Monday, July 1st
Taxonomy of DSLs
[Fowler, 2005, Tratt, 2008]

- DSL
  - Standalone / External
    - Full language
    - Needs parser, compiler / interpreter
    - Lex / Yacc, ANTLR etc.
  - Embedded / Internal
    - Heterogeneous
      - Program transformation outside the host language
      - Stratego, Silver etc.
    - Homogeneous
      - Extended host language
Example
Command-line options highlighting

**Usage:** advanced [-hd] [+d] [OPTIONS] cmd [OPTIONS]

Available commands: push pull.
Use 'cmd --help' to get command-specific help.

- **-h, --help**
  Print this help and exit.
- **-+(+)d, --debug[=on/off]**
  Turn debugging on or off.
  **Fallback:** on
  **Environment:** DEBUG

- Properties (bold, underline, foreground color...)
- Faces (localized property set)
- Themes (face trees)
Example
Underlying implementation

The face class

(defclass face ()
  ((name :initarg :name)
   ;; Properties:
   (foreground :initarg :foreground)
   (background :initarg :background)
   (boldp :initarg :bold)
   ;; etc.
   (subfaces :initarg :subfaces)))
Example
A DSL for theme customization

How do we go from this...

```lisp
;; default.cth --- Personal default theme for Clon

:background black
:face { option :foreground white
  :face { syntax :bold t :foreground cyan }  
  :face { usage :foreground yellow }  
  }
```

...to that?

```lisp
(setq default-theme (make-instance 'face :name 'toplevel
  :background 'black
  :subfaces (list (make-instance 'face :name 'option
    :foreground 'white
    :subfaces (list (make-instance 'face :name 'syntax
      :bold t
      :foreground 'cyan
    )
    (make-instance 'face :name 'usage
      :foreground 'yellow ))))))
```
Step 1
Hook into the Lisp parser: reader macros

- **readtable**: currently active syntax extensions table
- **macro character**: special syntactic meaning
- **reader macro**: implements macro character behavior

Let’s do it!

- Make the `{}` characters active
- Read a list of tokens until the closing brace
- Push the symbol `define-face` on top of that list

Note: RTMP (Read-Time Meta-Programming)
Step 1
Hook into the Lisp reader

This is how we go from this...

```lisp
;; default.cth --- Personal default theme for Clon

:background black
:face { option :foreground white
    :face { syntax :bold t :foreground cyan }
    :face { usage :foreground yellow }
}
```

...to that:

```lisp
;; default.cth --- Personal default theme for Clon

:background black
:face (define-face option :foreground white
    :face (define-face syntax :bold t :foreground cyan)
    :face (define-face usage :foreground yellow))
```
Step 2
Hook into the Lisp compiler: macros

- Ordinary Lisp functions
- Work on chunks of code (as data)
- Transform expressions into new expressions
- Control over evaluation

Let's make define-face a macro!

- Quoting its key arguments, except for the :face ones
- Generating a call to make-face

Note: CTMP (Compile-Time Meta-Programming)
Step 2
Hook into the Lisp compiler: macros

This is how we go from this...

```lisp
;; default.cth --- Personal default theme for Clon
:background black
:face (define-face option :foreground white
    :face (define-face syntax :bold t :foreground cyan)
    :face (define-face usage :foreground yellow))
```

...to that:

```lisp
;; default.cth --- Personal default theme for Clon
:background 'black
:face (make-face 'option :foreground 'white
    :face (make-face 'syntax :bold t :foreground 'cyan)
    :face (make-face 'usage :foreground 'yellow))
```
Step 3
A couple of wrappers

Lambda-list manipulation / 1st class functions

```lisp
(defun make-face (name &rest args &key &allow-other-keys)
  (apply #'make-instance 'face :name name args))

(defun make-theme (&rest args)
  (apply #'make-face 'toplevel args))
```

And while we’re at it...

```lisp
(defmacro define-theme (&rest args)
  '(define-face toplevel ,@args))
```
### Step 3

A couple of wrappers

This is how we go from this...

```lisp
;; default.cth --- Personal default theme for Clon

:background 'black
:face (make-face 'option :foreground 'white
    :face (make-face 'syntax :bold t :foreground 'cyan)
    :face (make-face 'usage :foreground 'yellow))
```

...to that:

```lisp
;; default.cth --- Personal default theme for Clon

:background 'black
:face (make-instance 'face :name 'option :foreground 'white
    :face (make-instance 'face :name 'syntax :bold t :foreground 'cyan)
    :face (make-instance 'face :name 'usage :foreground 'yellow))
```
The CLOS Meta-Object Protocol (MOP)

- **CLOS itself is object-oriented**
  - The CLOS MOP: a *de facto* implementation standard
  - The CLOS components (classes, methods etc.) are (meta-)objects of some (meta-)classes

**Generic functions, methods**

```lisp
(defun func ((arg1 class1) arg2 ...) body)
```

Methods are *outside* the classes (ordinary function calls)
Multiple dispatch (multi-methods)
Step 4
Hook into the object system: the MOP

- Object instantiation \texttt{(make-instance)} is a protocol
- Slot initialization \texttt{(initialize-instance)} is a generic function

Let's extend it!

- Provide our own method for the \texttt{face} class
- Collect all \texttt{:face} arguments
- call the next (standard) method with a new \texttt{:subfaces initarg}
Step 4
Hook into the object system: the MOP

This is how we go from this...

```lisp
;; default.cth —— Personal default theme for Clon
:background 'black
:face (make-instance 'face :name 'option :foreground 'white
       :face (make-instance 'face :name 'syntax :bold t :foreground 'cyan)
       :face (make-instance 'face :name 'usage :foreground 'yellow))
```

...to that:

```lisp
;; default.cth —— Personal default theme for Clon
:background 'black
:subfaces (list (make-instance 'face :name 'option :foreground 'white
    :subfaces (list (make-instance 'face
        :name 'syntax :bold t :foreground 'cyan)
    (make-instance 'face
        :name 'usage :foreground 'yellow))))
```
Mostly a matter of **read, compile etc.**

```lisp
(defun read-user-theme ()
  (with-open-file (stream (merge-pathnames "./faces" (user-homedir-pathname)))
    (read (make-concatenated-stream (make-string-input-stream "(define-theme_"
                                      stream
                                      (make-string-input-stream ")")))))

(defun make-user-theme (&optional compile)
  (if compile
      `(funcall (compile nil (lambda () ,(read-user-theme))))
      (read-user-theme)))
```
Mostly a matter of... Just-Do-It™

```lisp
(setq default-theme
  (define-theme
    :background black
    :face { option :foreground white
      :face { syntax :bold t :foreground cyan }
      :face { usage :foreground yellow }
    }
  )
)
```
Conclusion

- Impact of GPL on DSL design and implementation
- Key GPL aspect: extensibility
- Embedded homogeneous approach
  - A single language
  - DSL infrastructure smaller
  - DSL both internal and external
- Common Lisp
  - Functional, Imperative, Object-Oriented
  - MOP
  - CTMP (macros)
  - RTMP (reader macros)
  - read, eval, compile
Internal vs External DSLs
[Kamin, 1998, Czarnecki et al., 2004]

- Sub-optimal syntax **ok but...**
  - **Not ok:**
    - [Fowler, 2010]: “external DSLs have their own custom syntax and you write a full parser to process them”
    - [Kamin, 1998, Czarnecki et al., 2004]: “a prerequisite for embedding is that the syntax for the new language be a subset of the syntax for the host language”
    - BTW, same disagreement at the semantic level (MOP)

- Poor error reporting
  - Research: [Tratt, 2008]
  - Lisp: ? (but Cf. condition system & restarts)
Controversial aspects of extensibility

- Dynamic typing
  - pros: end-user friendly
  - cons: run-time type errors / checking
  - Research: [Taha and Sheard, 1997]
  - Hybrid languages (Cf. Racket)

- Lazy Evaluation
  - pros: infinite data structures, new control primitives *etc.*
  - cons: pure functional languages only
  - Lisp: laziness through macros (not as straightforward), but side-effects for free, and still functional.
The root of (Lisp) extensibility

- Reflection
  - Introspection
  - Intercession

- Implementation
  - By API

  - Structural Reflection (program)
  - Behavioral Reflection (language)


