Code Generation and DSLs in Scheme

Presented by Grant Rettke

grettke@acm.org

The Background

- Fowler's "Language Workbench" article
  - http://martinfowler.com/articles/languageWorkbench.html
- DSLs (Domain Specific Languages) are problem specific tools; language workbenches lets you implement DSLs
- Lisp is mentioned as a language workbench
- Poses a simple problem that may be solved using a DSL
  - DSLs don't have to be Turing Complete
- All code samples run on DrScheme v371 with the "Pretty Big" language level

The Problem

- We've got a set of pre-existing domain classes
- There is data within flat text files we want to use to populate our domain classes
- We need to come up with a good solution for allowing the user to specify how to get the data from those flat text files into instances of our domain classes
- We will implement a DSL to allow for the user to map the data to the classes

The Input File

- Each input line represents an event, with fields delimeted by index
- The first four digits (0-3) indicate the event type: SVCL -> Service Call, MNTC -> Maintenance, REPR -> Repair, and SLSC -> Sales Call
- The next five digits (4-8) represent the client name
- The last four digits (9-12) represent the employee id

SVCL GRANT 1723

SVCL VICKY 0007

MNTC LAZY1 4242

REPR CRLES 1024

SLSC CEOGY 0100

The Approach

- Start by implementing the solution without any code generation
- Find the solution for a single case
- Generalize the solution for any case
- Common MDSD (Model Driven Software Development) approach
- Provide only working code examples: no theory!
- Tailored examples for *object folks*
- See class.ss: Classes and Objects in PLT Scheme
  - http://download.plt-scheme.org/doc/mzlib/mzlib.html
- Next slide, code: A domain class implemented using class.ss

Domain Classes v1: Code (domain1.ss)

```
(define service-call%
  (class* object% ()
    (define field-customer-name null)
    (define/public customer-name
      (case-lambda
        [() field-customer-name]
        [(value) (set! field-customer-name value)]))
    (define field-customer-id null)
    (define/public customer-id
      (case-lambda
        [() field-customer-id]
        [(value) (set! field-customer-id value)]))
    (super-new)))
```

Domain Classes v1: Comments

- Class are more about data, less about behavior. Data entities?
- Could use structs here instead
- Lot of duplicated code, not all properties were listed here
- There is an opportunity here to simplify declaration of properties
- Recall v1
- New Goal:
  - ```
    (define service-call%
      (class* object% ()
        (property customer-name customer-id)
        (super-new)))
    ```
    ```

Syntactic Extension: How macros work

- Macros are not functions, they do not compute, think of them as "transformers"
- Functions can do nearly, but not all, that can be done using macros
- Macros extend the syntax of Scheme
- Macros work by transforming code supplied by the user into new code
- They work by pattern matching on the arguments provided to the macro
  - First match on name
  - Then match on arguments
- After finding a matching rule, a template is used to define the new code
- Next slide, code: Single property syntax

Property v1: Code (domain2.ss)

```
(define-syntax (property stx)
  (syntax-case stx ()
    ((_ name)
     #'(begin
         (define property-name null)
         (define/public name
           (case-lambda
             [() property-name]
             [(value) (set! property-name value)])))))
```

Property v1: Comments (domain2.ss)

- Takes a single argument, the name of the property
- Expands into a field and getter/setter function
- Notice that "property-name" is a private field that will be declared multiple times; but without name collisions.
  - *Hygienic macros* keep things clean!
  - Introduced variable names are *auto-magically* replaced with non-conflicting variable names
- Bug #1: Allows zero arguments
  - `(property)`
- Next slide, code: Notify user when not enough arguments are provided

Property v2: Code (domain3.ss)

```
(define-syntax (property stx)
  (syntax-case stx ()
    ((_) (raise-syntax-error
          #f
          "property requires at least one name"
          stx))
    ((_ name)
     #'(begin
         (define property-name null)
         (define/public name
           (case-lambda
             [() property-name]
             [(value) (set! property-name value)])))))
```

Property v2: Comments (domain3.ss

- The `stx` is the syntax object that gets transformed by the macro
- Using this you can do very interesting things like introduce new bindings into the users existing code
  - AKA *Non-Hygienic* macros
  - Risk stomping on existing bindings
- Use the syntax object to provide error messages that make sense to the user
- Still want to allow multiple property names
- Next slide, code: Allow multiple property names using a recursive macro

Property v3: Code (domain4.ss)

```
(define-syntax (property stx)
  (syntax-case stx ()
    ((_) (raise-syntax-error
          #f
          "property requires at least one name"
          stx))
    ((_ name)
     #'(begin
         (define property-name null)
         (define/public name
           (case-lambda
             [() property-name]
             [(value) (set! property-name value)])))))
    ((_ name names ...)
     #'(begin
         (property name)
         (property names ...)))))
```

Property v3: Comments (domain4.ss)

- Bug #2: Allows expression rather than constants
- For example
- `(property (+ 1 2))`
- Next slide, code: Disallow expressions for property names

Property v4: Code (domain8.ss)

```
((_ name)
 (begin
   (if (not (identifier? #'name))
       (raise-syntax-error
        #f
        "property names must be identifiers, not expre
        #'name))
   #'(begin
       (define property-name null)
       (define/public name
         (case-lambda
           [() property-name]
           [(value) (set! property-name value)])))))
```

Property v4: Comments (domain8.ss)

- Bug #3: Allows duplicate property names.
- For example
- `(property (fnord fnord fnord))`
- Not a problem
- Underlying class system does not allow duplicates
- User sees a sensible error message

How easy can we make it to create data entities?

- These are basically data entities
- They should be easy to define and use
- This may be a bit contrived, oh well, it is another macro
- Let creating data entities look like this
-
```
(define-entity service-call%
  (customer-name customer-id))
```
- Next slide, code: A macro for defining data entities

define-entity V1: Code (domain5.ss)

```
(define-syntax (define-entity stx)
  (syntax-case stx ()
    ((_ type (p ps ...))
     #'(define type
         (class* object% ()
           (property p)
           (property ps ...)
           (super-new))))))
```

Populating the data entities: The Approach

- We've already got domain classes
- There is data within flat text files we will use to populate our domain classes
- For each domain class, we will implement a function that given a line of text will create an instance of the class and populate it using the data formatted within that line of text
  - Responsible for knowing how to parse the data
- Another function will iterate over the lines of text
  - For each line of text it will take the first four digits of text (the class type) and determine if a function exists that can turn this line of text into a class via a lookup table (hash-map)
- This is not generalized code, it only handles one case
- Next slide, code: Raw code that populates a service-class object

Code: populating objects (domain6.ss)

```
(define (parse-svcl-line line)
  (let ([instance (make-object service-call%)]
        [name (substring line 5 14)]
        [id (substring line 14 24)])
    (send instance customer-name name)
    (send instance customer-id id)
    instance))
(define mappings (make-hash-table 'equal))
(hash-table-put! mappings "SVCL" parse-svcl-line)
(define (process-line line)
  (let* ([type (substring line 0 4)]
         [handler (hash-table-get mappings type null
    (if (not (null? handler))
        (handler line)
        #f)))
```

Populating the data entities: The New Approach

- Re-factor code with end goal of generalization in mind
- Use two lookup tables
  - **`prefix->type`** use 4 digit prefix to find class type
  - **`type->populator`** use a class type to find a function that populates it
- Will use the class type to create a generalized population method
- Will use field mappings to create generalized calls to the object's setter methods
- Next slide, code: A revised object population approach with generalization in mind

Revise existing code (domain6.ss)

```
(define prefix->type (make-hash-table 'equal))
(define type->populator (make-hash-table 'equal))
(hash-table-put! prefix->type
                 "SVCL" service-call%)
(hash-table-put! type->populator
                 service-call% parse-svcl-line)
(define (process-line line)
  (let* ([prefix (substring line 0 4)]
         [type
          (hash-table-get prefix->type prefix null)])
    (unless (null? type)
      (let
          ([populator
            (hash-table-get type->populator type null)])
        (unless (null? populator)
          (populator line)
          #f))
        #f)))
```

Responsibility the DSL

- Implement the (populator) function to populate the class using line data
- Specify the (4 digit text) data to class type mapping
- Specify the class name to populator function mapping
- How it will look
  - ```
    (define-mapping
      service-call%
      "SVCL"
      (4 9 customer-name)
      (9 13 customer-id))
    (define-mapping
      sales-call%
      "SLCL"
      (4 9 customer-name)
      (9 13 customer-id))
    ```
    ```

Ultimate Goal

- ```
  (define-mapping
    service-call% "SVCL"
    (5 14 customer-name) (14 24 customer-id))
  ```
- Next slide, code: The DSL macro specification
  ```

define-mapping: Code (domain8.ss)

```
(define-syntax (define-mapping stx)
  (syntax-case stx ()
    ((_ type prefix
        (start stop field) ...)
     #'(begin
         (define (populator line)
           (let ([instance (make-object type)])
             (begin
               (send
                instance
                field
                (substring line start stop))
                                          ...
               instance)))
         (hash-table-put! prefix->type prefix type)
         (hash-table-put! type->populator
                          type populator)))))
```

Conclusion

- Meta programming is *generalizing and generating* what *we already know* how to do
- Scheme uses pattern matching and templating to generate code
  - Remarkably different from *walking the tree*! (No ASTs)
  - Leverage what you know: the entire DSL tool-chain is in Scheme
  - Can embed Scheme code within the DSL itself!
- Typical problems solved by meta programming: System configuration, Boilerplate code, Little languages
- These solutions are usually called "frameworks", regardless of the type of meta programming used
- Where might you apply meta programming were you given the chance?