Programming language design and analysis

Domain-specific languages

Marius Minea

11 January 2016

based on: Martin Fowler, *Domain Specific Languages*

Defining Domain Specific Languages

DSL: a computer programming language of limited expressiveness focused on a particular domain

computer programming language

used to communicate with computer *and* between humans
should have fluency (beauty)

limited expressiveness

can't build a software system in it
contrast: data / control / abstraction structures in general PL

domain focus

makes it useful

Kinds of DSLs

external

use a different language than the application that uses them

SQL, XML, awk, regular expressions (and others in UNIX)

internal

use same general purpose programming language as application

but in a particular and limited way

LISP, Ruby

language workbenches

IDEs for building DSLs (abstract syntax, editors, generators)

more/different than usual parse/generate cycle

Why use a DSL?

improved development productivity

communication with domain experts

change in execution context
    e.g. handle definitions at runtime instead of compile time

alternative computational model
    not just imperative

What's under a DSL?

A DSL manipulates an abstraction

usually done with a *library / framework*
interfaced through an *API*

DSLs are usually a front-end to such an interface
⇒ the hard part is building the framework

DSL Patterns

appear with internal DSLs

use syntax of underlying general purpose language for visual fluency

may need:
- language with special syntactic features
- language where new syntax can be adapted / defined
- just clever use of existing syntax

Patterns: Function Sequence

```
computer();
  processor();
    cores(2);
    processorType(i386);
  disk();
    diskSize(150);
  disk();
    diskSize(75);
    diskSpeed(7200);
    diskInterface(SATA);
```

Function Sequence: Howto

usually with bare function calls (global if language allows)
⇒ but needs static parsing data (*context variables*)

```
currentObject = ...
...
currentObject.setValue(...);
```

solution: use *object scoping* for functions and parsing data

Pattern: Nested Functions

```
computer(
  processor(
    cores(2),
    Processor.Type.i386
  ),
  disk(
    size(150)
  ),
  disk(
    size(75),
    speed(7200),
    Disk.Interface.SATA
  )
);
```

Nested Functions: Howto

important property: evaluation order is inside-out
(parameters before function call)
⇒ good: evaluation returns fully-formed values/objects, usable further
⇒ awkward: textual order is opposite to natural sequencing

Useful language features:
named parameters `disk(75, 7200)` is not suggestive
optional arguments
variable number of arguments

Pattern: Method Chaining

```
computer()
  .processor()
    .cores(2)
    .i386()
  .disk()
    .size(150)
  .disk()
    .size(75)
    .speed(7200)
    .sata()
  .end();
```

Method Chaining Howto

Modifier methods return the host object
⇒ multiple modifiers can be invoked on the same object

the opposite of *command query separation*

```
HardDrive hd = new HardDrive();
hd.setCapacity(150);
hd.setExternal(true);
hd.setSpeed(7200);
```

```
new HardDrive().capacity(150).external().speed(7200);
```

Issues:
naming no longer makes clear this is a setter
problems with languages where newline is a separator
finishing problem (when to stop?), esp. with nested components

Pattern: Nested Closure

```
computer do
  processor do
    cores 2
    i386
    speed 2.2
  end
  disk do
    size 150
  end
  disk do
    size 75
    speed 7200
    sata
  end
end
```

Nested Closure Howto

Express statement sub-elements of a function call by putting them into a closure in an argument.

a single Nested Closure instead of several Nested Function arguments

Issues:
needs code to evaluate closure (vs. arguments are evaluated implicitly)
contents of closure is function sequence, still needs context variables
(but they can be created before closure / destroyed afterwards)
context variable can be explicit:

```
processor do |p|
  p.cores 2
  p.i386
end
```