What is Domain Specific Language?

Tanaka Akira
@tanaka_akr
National Institute of Advanced Industrial Science and Technology (AIST)

RubyKaigi 2019
2019-04-19
DSL – Domain Specific Language

- DSL stands for domain specific language
- DSL is implemented in two ways:
  - External DSL: individual language implementation
    make, SQL, XSLT, etc.
  - Internal DSL: implemented as a library in a host
    language
    rake, rspec, etc.
My question:
What's the difference of internal DSL and library?
The Question (Concrete Version)

- rake is considered a DSL
- But Rakefile is just a Ruby program
- Why rake is DSL?
The Question (Generalized Version)

- Internal DSL is just a library
- DSL description is just a program written in the host language
- Why internal DSL is considered a LANGUAGE?
This Question is Important

- DSL empowers Ruby programmers
- Understanding what is DSL is important to design a new DSL
My Answer:
DSL has its own semantics
DSL program is readable without Ruby semantics
This makes a program easier-to-read
External DSL is clear

- SQL is DSL for database
- XSLT is DSL for translating XML

They have own syntax and semantics
They are not general purpose language
So, they are DSL
Internal DSL is not clear

• Many ruby libraries are considered DSL
  - rake is a DSL
  - rspec is a DSL
  - etc.

• Rakefile and foo_spec.rb is written in Ruby

• They are executed in Ruby semantics
Syntax + Semantics = Language

- Syntax is a subset of string
- Semantics maps programs to semantic space
Reading a Program

- "reading a program" means "following a semantics arrow"
- We need to learn the semantics to read programs
Two Languages

- Lang. A and B have different syntax and semantics.
- Some programs, such as x, have the same meaning.
- Other programs, such as y, have different meanings.
Two Language Semantics Examples

- One program has same meaning in two languages
- One program has different meaning in two languages
One Program Has Same Meaning in Two Languages

• Ruby
  - % ruby -e 'print "hello\n"
    hello

• Perl
  - % perl -e 'print "hello\n"
    hello

'print "hello\n" has same meaning in Ruby and Perl.
One Program Has Different Meaning in Two Languages

- **Ruby**
  - `irb> 1 + 2 * 3
    7`

- **Smalltalk:**
  - `gst> 1 + 2 * 3
    9`
  - Smalltalk's binary operators have no precedence and always left-associative
    `1 + 2 * 3` is interpreted as `(1 + 2) * 3`
External DSL

- External DSL and Ruby are different languages
- Some programs, such as x, have the same meaning.
  Other programs, such as y, have different meaning.
Internal DSL

- Internal DSL Syntax is subset of host language
- However, it has own semantics and some programs can have different meaning
Internal DSL has Different Semantics?

- rspec provides English-like language for BDD
- Ruby semantics and English semantics can differ
rubyspec (mspec) Example

- "should" method is implemented with "should" meaning in English
- spec/ruby/language/and_spec.rb:
  it "evaluates to the last condition if all are true" do
    ("yes" && 1).should == 1
    (1 && "yes").should == "yes"
  end
Ruby and English

English is used everywhere in Ruby (not only DSL)

• class and method names
  matz rejects proposals until the name is appropriately
  means a feature

• English.rb
  alias $ERROR_INFO $!

• rspec

These make Ruby programs easier-to-read for English user
Internal DSL

- Internal DSL makes programs easier-to-read for domain-knowledgeable people who knows domain semantics.

- Easier-to-read doesn't mean easier-to-write. Programmers should make both meaning same. I.e. Programmers must know both languages (DSL and host language).
How to Design Good DSL
DSL Design Principle

• Respect domain convention
  This makes DSL description easier for domain-knowledgeable people

• Reduce boilerplate
  – Preamble/Postamble
  – Frequent snippet

• Hide non-domain issue
  memory-management, etc.
Several DSL Examples

- rake: DSL for build process
- erb: DSL for templates
- shell.rb: DSL for Unix-shell
Build Process

- There are many build tools using dependencies make, rake, cmake, SCons, Ant, ...

- build process = dependencies + actions

```
  a.c
    ▼
    | gcc -c a.c
    |    ▼
    |    a.o
    ▼
  b.c
    ▼
    | gcc -c b.c
    |    ▼
    |    b.o
    ▼
  gcc -o com a.o b.o
    ▼
  com
```
"make" is a famous build tool
Makefile represents the graphical structure succinctly
rake

- **Rakefile:**
  
  file "a.o" => "a.c" do
  sh "gcc -c a.c" end
  
  file "b.o" => "b.c" do
  sh "gcc -c b.c" end
  
  file "com" => ["a.o", "b.o"] do
  sh "gcc -o com a.o b.o" end

- **Graphical Structure**

  "rake" is build tool written in Ruby
  Rakefile is similar to Makefile
  Rakefile is bit more verbose than Makefile
rake without DSL

- rake can be used without DSL
- Build script:
  ```ruby
  require 'rake'
  Rake.application = Rake::Application.new
  Rake.application.init("rake", ARGV)
  Rake::FileTask.define_task("a.o" => "a.c") do system("gcc -c a.c") end
  Rake::FileTask.define_task("b.o" => "b.c") do system("gcc -c b.c") end
  Rake::FileTask.define_task("com" => ["a.o", "b.o"]) do system("gcc -o com a.o b.o") end
  Rake.application.top_level
  
  rake without DSL is not the supposed way to use rake
  "system" is used because no easy way to invoke "sh"
  
  The build script is much verbose than Rakefile
Thought Experiment: rake without DSL Improved

• Build script:
  require 'rake'
  r = Rake.new(ARGV)
  r.define_file_task("a.o" => "a.c") do r.sh("gcc -c a.c") end
  r.define_file_task("b.o" => "b.c") do r.sh("gcc -c b.c") end
  r.define_file_task("com" => ["a.o", "b.o"])
    do r.sh("gcc -o com a.o b.o") end
  r.run

• It still verbose than Rakefile
DSL Design in Rake

• Respect domain convention
  - Describe build graph using pair: target => dependencies
  - The arrow is inverse with build direction, unfortunately

• Reduce boilerplate
  - Preamble: require 'rake'; r = Rake.new
  - Frequent snippet
    "file" is shorter than "r.define_file_task"
    "sh" is shorter than "r.sh"
DSL Implementation of Rake

- `lib/rake/dsl_definition.rb`
  This filename is definite reason that Rake is DSL

- Tricks for DSL
  - global methods
    "file" and "sh" is defined to "main" object
  - singleton pattern (global variable)
    The state is maintained at Rake.application
  - dedicated command, rake
    It makes preamble/postamble implicit
ERB: template engine

- ERB source:
  ```erb
  foo
  <% 3.times do |i|
  %>
  bar<% end %>
  baz
  ``

- result:
  ```erb
  foo
  barbarbar
  baz
  ``

Template Engine can be considered as DSL for text generation
Text Generation with/without Template Engine

with ERB:

- foo
  
  `<% 3.times do |i| %>`\>bar`<% end %>`

- baz

without ERB:

- s = `""`
  
  s `<< "foo\n".freeze`

3.times do |i|
  
  s `<< "bar".freeze`

end

s `<< "\nbaz\n".freeze`

s
DSL Design in ERB

- Respect domain convention
  - Use `<% ... %>` as SGML Processing Instruction and PHP

- Reduce boilerplate
  - Preamble: `s = ""``
    Postamble: `s`
  - Frequent snippet
    - string concatenations: `s <<`
    - quotes and escapes: "...

- Hide non-domain issue
  - Destructive string concatenation (`<<`) is faster than non-destructive concatenation (`+`
  - Avoid string allocations using `.freeze`
shell.rb: Shell-like Tool in Ruby

- Bourne shell:
  cat /etc/hosts | grep localhost > /tmp/foo
  head -1 /tmp/foo

- shell.rb:
  irb> require 'shell'
  irb> Shell.new.transact {
  irb>   cat("/etc/hosts") | system("grep", "localhost") > "/tmp/foo"
  irb>   system("head", "-1", "/tmp/foo")
  irb> }
  shell(#<Th:0x000055da32f97178 run>): /bin/grep localhost
  shell(#<Th:0x000055da32f97178 run>): /bin/head -1 /tmp/foo
  => 127.0.0.1  localhost
shell.rb without Shell#transact

• shell.rb with transact:
  Shell.new.transact {
    cat("/etc/hosts") |
    system("grep", "localhost") >
    "foo"
    system("head", "-1", "foo")
  }

transact method replaces self in the block to
avoid frequent "s."
It uses instance_eval

• shell.rb without transact:
  s = Shell.new
  s.cat("/etc/hosts") |
  s.system("grep", "localhost") >
  "/tmp/foo"
  s.system("head", "-1", "foo")
DSL Design in shell.rb

- Respect domain convention
  - Use "|" for pipe, ">" for redirection
  - cat method for cat command

- Reduce boilerplate
  - Frequent snippet
    - "cat" method instead of system("cat", ...)
      def_system_command provides a way to define such methods
    - "s." is removed using instance_eval
Internal DSL
or
External DSL
Advantage of Internal DSL

Ruby and DSL can be mixed

• Ruby in DSL
  - Rake actions can be written in Ruby

• DSL in Ruby
  - generate Rake rules in Ruby loop
Disadvantage of Internal DSL

• Ruby and DSL is mixed
  – DSL description is unusable except execution
    • make -n show actions
    • rake -n doesn't show actions

• Tends to difficult to debug
  – DSL description: debugging at Ruby level, not DSL level
  – DSL implementation: dirty tricks makes debugging harder

• Tends to reach Ruby's limitation
  – The limitation may be changed by Ruby versions
Internal DSL
or
Library
Between DSL and Library

- Many techniques to respect domain knowledge and reduce boilerplate
- Some techniques are cleaner and others are more dirty
- Clean techniques (Not so DSL-ish)
  - Good names (English words)
  - Appropriate use of operators
- Dirty techniques (DSL-ish)
  - singleton pattern (global variable)
  - instance_eval
  - individual command
  - TracePoint
  - RubyVM::AbstractSyntaxTree
Readability of DSL

• Dirty techniques may improve readability
  But it cannot improve endlessly

• Readability of DSL must saturate
Writability of DSL

- Dirty techniques degrades writability (maintainability, debug) of DSL descriptions and implementations
- Disadvantage would be bigger endlessly
Readability + Writability

- readability + writability would have maximum
Dirty Techniques are Discrete

• There are not so many dirty techniques

In this image, only one dirty technique is enough
Summary

• Why internal DSL is a language?
  – It has own semantics

• Why DSL is easier-to-read?
  – Programmers (or domain experts) can use domain knowledge (domain semantics)

• How to design good DSL?
  – Moderate use of dirty techniques