Implementing Internal Domain-Specific Languages with Ruby : An Introduction

Guy Tremblay

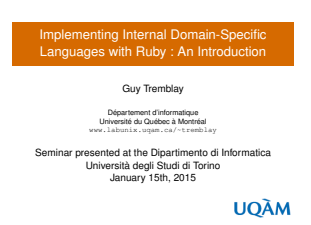
Département d'informatique
Université du Québec à Montréal
www.labunix.uqam.ca/~tremblay

Seminar presented at the Dipartimento di Informatica
Università degli Studi di Torino
January 15th, 2015

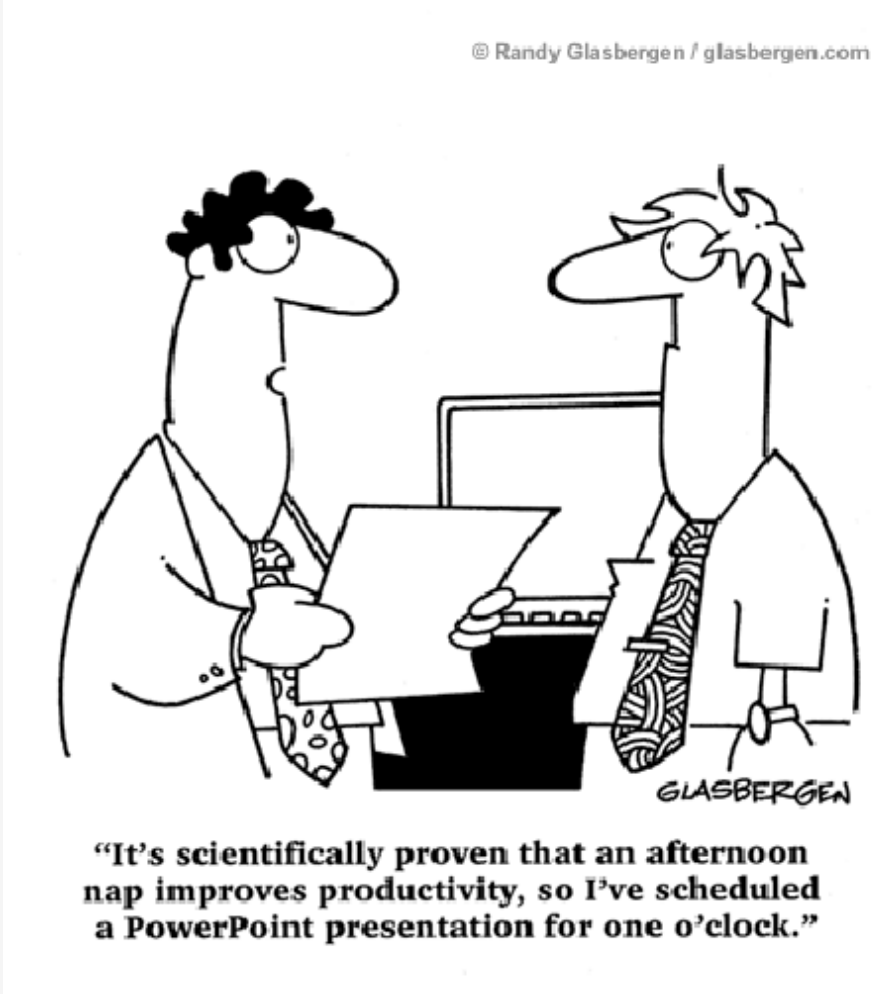
UQÀM

2015-01-16

Implementing Internal Domain-Specific Languages with Ruby : An Introduction



- Good afternoon everyone. Thanks for coming to this seminar right after lunch, which seems to be the usual time for seminars in many universities, and here is why.
- This talk is an introduction to Domain-Specific Languages—DSLs, more specifically Internal DSLs in the context of a dynamic language such as Ruby.
- But first, a few words about where I am from. As the logo on the first page says, I am from UQAM, which stands for "Université du Québec à Montréal", une université de langue française, as its name shows : there are 4 major universities in Montréal, 2 french (UQAM, Université de Montréal), 2 english (McGill, Concordia).



2015-01-16

Implementing Internal Domain-Specific Languages with Ruby : An Introduction

-

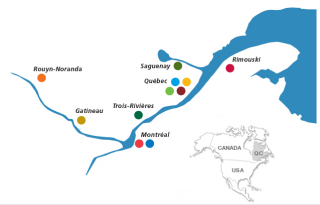
2015-01-16

Implementing Internal Domain-Specific Languages with Ruby : An Introduction

-

Le réseau de l'Université du Québec (UQ)

Le réseau de l'Université du Québec (UQ)



- UQAM belongs to the "*Réseau de l'Université du Québec*", a network of public universities throughout the province of Québec.

UQAM is a young—and large—university

- Created in 1969
- Fall 2014

Level of study	Nb. of students
Undergraduates	35 681
Masters	6 175
Ph.D.	1 718
Total	**44 017**

Full-time students	60 %
Part-time students	40 %

- 40 departments and schools

- Created in 1969
- Fall 2014

Level of study	Nb. of students
Undergraduates	35 681
Masters	6 175
Ph.D.	1 718
Total	**44 017**

Full-time students	60 %
Part-time students	40 %

- 40 departments and schools

• UQAM, as its name says, is in Montréal, the largest city in the province Québec.
• UQAM counts 40 departments and schools, but no medecine. As for engineering, there is a single program, part of our department, a very small (tiny) program in Microlectronics — less than 50 students.
• A majour characteristics of UQAM is that a lot of students are adult, part-time students, so classes are taught both during the days and during the evenings.

Le département d'informatique

39 Professors
Software engineering
Databases
Cognitive informatics
Bio-informatics
Networks and teleinformatics
Microsystems and microelectronics

Students	
Undergr. (2+3)	681
Masters (3+3)	87
Ph.D. (2)	48

Le département d'informatique

39 Professors
Software engineering
Databases
Cognitive informatics
Bio-informatics
Networks and teleinformatics
Microsystems and microelectronics
Students
Undergr. (2+3) 681
Masters (3+3) 87
Ph.D. (2) 48

- This is the main building of the sciences campus, right in downtown Montreal, in the Quartier des spectacles—the famous Festival de jazz de Montreal is just beside our building.
- Our department is quite large, with 39 professors from various areas.
- We have a large number of different programs : two bachelor programs, various "certificats", 3 masters, 2 DESS, 2 PhD programs — one in "informatique cognitive" and one, more recent (7 year old), in informatique.

2015-01-16

Implementing Internal Domain-Specific Languages with Ruby : An Introduction

A few words about Montreal. As you may know, Montreal can have a lot of snow during the winter. . . and it can be quite cold.

2015-01-16

Implementing Internal Domain-Specific Languages with Ruby : An Introduction

2015-01-16

Implementing Internal Domain-Specific Languages with Ruby : An Introduction

• One strange thing about Montreal is that even though we have a lot of snow, a typical characteristic of Montreal's architecture is that. . . the stairs are outside ! Thus, this means we have a lot of snow shoveling to do whenever there is a snow fall ☹

• As for my own background, you got some details about it in the email announcement that was sent to yoy. Let's simply say that for the last 5 years, I was the "Directeur" (Head) of my department, so I spent most of my time involved with administrative duties and problems. But this year, I am in sabbatical, which is why I am in Torino.

• One reason I wanted to come to Torino was to get away from the hard Montreal's winter, which has been quite a success so far : it was -17 with a lot of wind a week ago — compared to +17 here a few days ago !

My background

Academic background :

- B.Sc. (Math./Comp. Sci), UQAM
- M.Math. (Comp. Sci.), Univ. of Waterloo
- Ph.D. (Comp. Sci.), McGill Univ.

Professor at UQAM since 1985 :

- Teaching areas : software engineering, computer architecture, algorithmics, discrete maths and formal methods, parallel programming, *et al.*
- Research interests : parallel programming, formal methods, Web services, teaching tools
- Currently in sabbatical — after five years as Head of Department

Academic background :
- B.Sc. (Math./Comp. Sci), UQAM
- M.Math. (Comp. Sci.), Univ. of Waterloo
- Ph.D. (Comp. Sci.), McGill Univ.

Professor at UQAM since 1985 :
- Teaching areas : software engineering, computer architecture, algorithmics, discrete maths and formal methods, parallel programming, *et al.*
- Research interests : parallel programming, formal methods, Web services, teaching tools
- Currently in sabbatical — after five years as Head of Department

•

Implementazione di Linguaggi Specifici di Dominio Interni con Ruby : Un'introduzione

Guy Tremblay

Département d'informatique
Université du Québec à Montréal
www.labunix.uqam.ca/~tremblay

Seminario presentato al Dipartimento di Informatica
Università degli Studi di Torino
15 gennaio 2015

UQÀM

Implementazione di Linguaggi Specifici di Dominio Interni con Ruby : Un'introduzione

- There are a other reasons why I wanted to come to Torino :
  - I visited Italy for the first time last summer (Lago di Como) and I really enjoyed it. So I wanted to come back.
  - My thesis was in the area of parallel programming. For the last 10 years or so, I left that area aside, but I wanted to take advantage of sabbatical to come back to it. When I discovered the work being done here on Fastflow and RISC-pb^2l skeletons, I tought it was very interesting. So I contacted Marco and, thanks to him, here I am.

- For this seminar, I will not try to speak Italian—not enough practice yet. It's going to be difficult enough to speak English, a language that I have not practiced that often in the last few years.
- So, as the title says, this talk is an introduction to Domain-Specific Languages—DSLs, more specifically Internal DSLs in the context of the Ruby language.

Avete già utilizzato. . . make ?

```
$ cat Makefile
hello: hello.c
    gcc -o hello hello.c

clean:
    rm -f hello hello.o
```

```
$ make
gcc -o hello hello.c
```


make

$ cat Makefile
hello: hello.c
gcc -o hello hello.c

clean:
rm -f hello hello.o

$ make
gcc -o hello hello.c

- Used, on Unix, to automate program build tasks, i.e., to compile and link programs but also, more generally, to automate tasks.

Avete già utilizzato. . . SQL ?

```
SELECT title, price
  FROM  Book
   WHERE price > 30.00
  ORDER BY price;
```

```
Title                          Price
---------                      ---
DSLs in Action                 39.99
Domain-Specific Languages      45.99
Domain-Driven Design           49.99
```


SQL

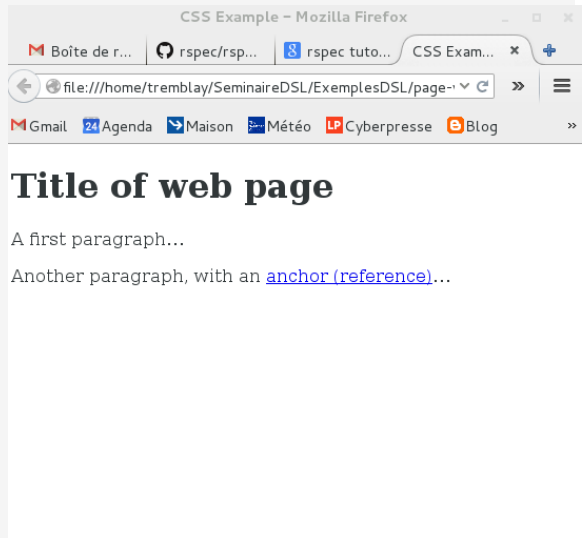
SELECT title, price
FROM Book
WHERE price > 30.00
ORDER BY price;

Title Price
DSLs in Action 39.99
Domain-Specific Languages 45.99
Domain-Driven Design 49.99

- SQL = Structured Query Language = language for managing data in relational databases.

Avete già utilizzato. . . HTML ?

```
<HTML>
<HEAD>
<TITLE>CSS Example</TITLE>
</HEAD>

<BODY>
<H1>Title of web page</H1>

<P>A first paragraph...</P>

<P>Another paragraph, with an
<A HREF=".">anchor
(reference)</A>...</P>
</BODY>

</HTML>
```



- HTML = Hypertext Meta Language, a subset/derivative of SGML for describing the contents of web pages.

Se così fosse, allora avete utilizzato Domain-Specific Languages

2015-01-16

Implementazione di Linguaggi Specifici di Dominio Interni con Ruby : Un'introduzione

Se così fosse, allora avete utilizzato Domain-Specific Languages

-

Contenuto della presentazione

- Che cosa è un “Linguaggio Specifico del Dominio” ?
- Una breve introduzione a Ruby
- Esempi di DSL in Ruby : Descrizioni XML di documenti
- Conclusione

• Che cosa è un “Linguaggio Specifico del Dominio” ?
• Una breve introduzione a Ruby
• Esempi di DSL in Ruby : Descrizioni XML di documenti
• Conclusione

•

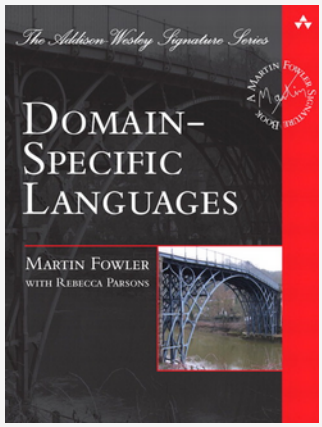
Che cosa è un “Linguaggio Specifico del Dominio” ?

Che cosa è un *Domain-Specific Language* (DSL) ?

Domain-specific language :

A computer programming language of limited expressiveness focused on a particular domain.

Source: *M. Fowler, 2011*



Domain-specific language :
A computer programming language of limited expressiveness focused on a particular domain.
Source: *M. Fowler, 2011*

- Programming language ⇒ A language used by humans to program computers
- Limited expressiveness ⇒ Not *turing-complete*, not a GPL (General Purpose Language) like C, Java, Ruby, etc.
- Particular domain ⇒ Tied to a specific application domain—a specific problem area—so it (generally) uses concepts from that application domain.

Che cosa è un *Domain-Specific Language* (DSL) ?

Domain-specific language :

> *A language offering expressive power focused on a particular problem domain, such as a specific class of applications or application aspect.*

Source: *K. Czarnecki, 2005*

Domain-specific language :
A language offering expressive power focused on a particular problem domain, such as a specific class of applications or application aspect.
Source: *K. Czarnecki, 2005*

- Expressiveness is limited with respect to Turing-complete/GPL. However, in the domain for which it is defined, it is very expressive : "DSLs trade generality for expressiveness in a limited domain." [MHS05]

Che cosa è un Domain-Specific Language (DSL) ?

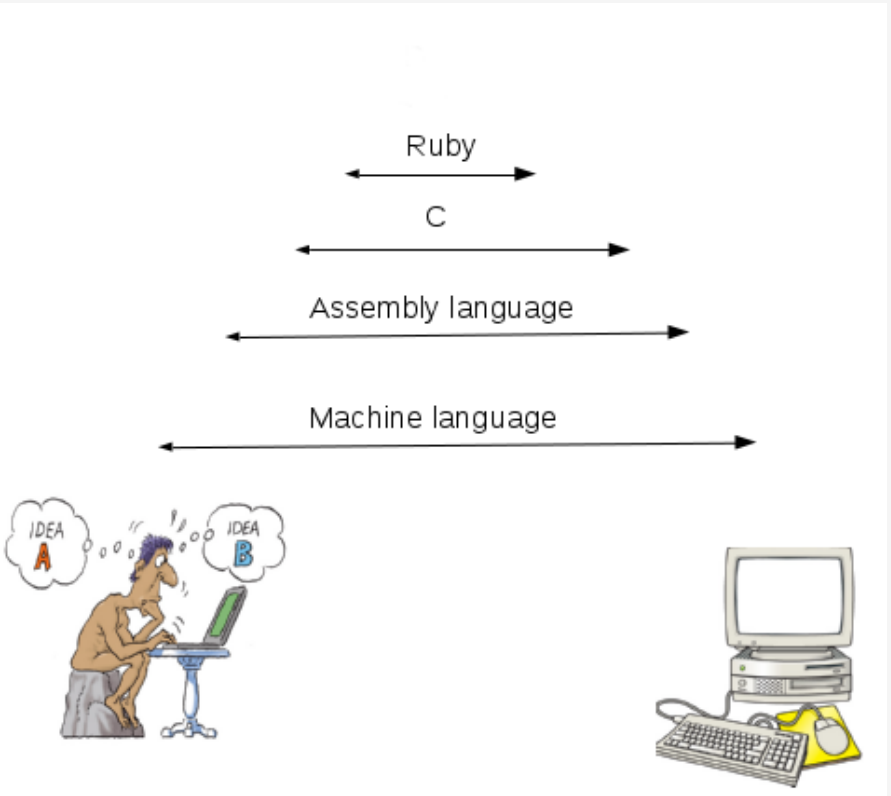
Domain-specific language :

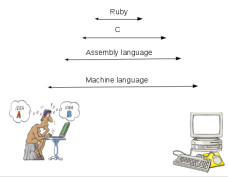
A “little language”, commonly known as a “domain-specific language” (DSL), is a language that is expressive uniquely over the specific features of programs in a given problem domain.

Source*: A. Yepez, 2010*

Domain-specific language :
A “little language”, commonly known as a “domain-specific language” (DSL), is a language that is expressive uniquely over the specific features of programs in a given problem domain.
Source*: A. Yepez, 2010*

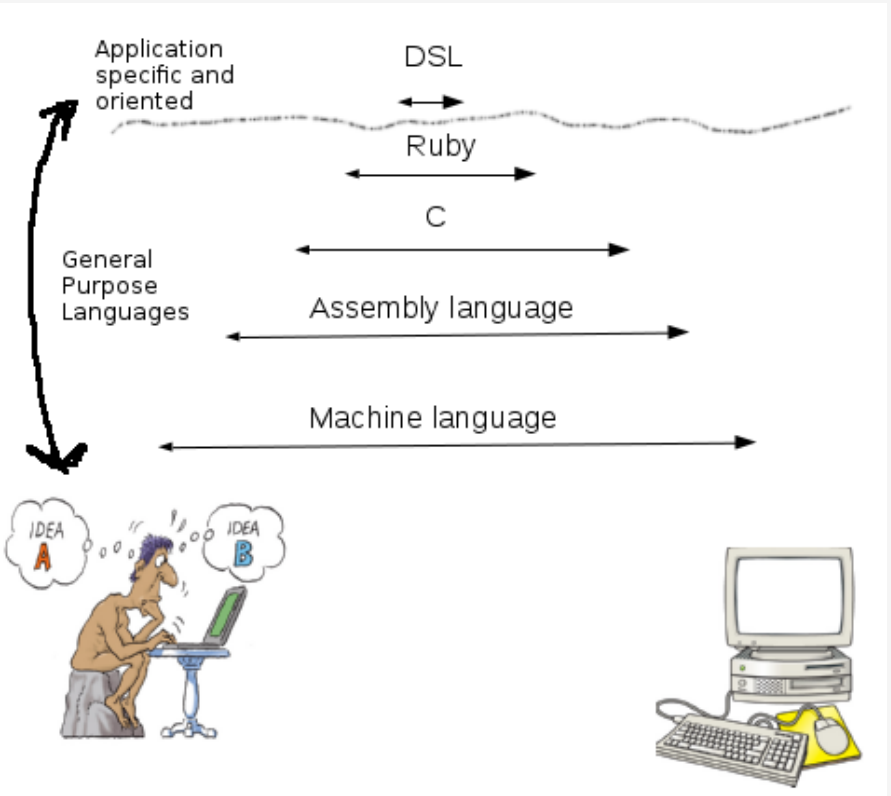
Un DSL cerca di ridurre il divario semantico tra le idee degli utenti e la loro espressione in un programma

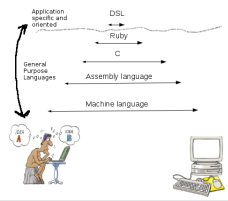




-

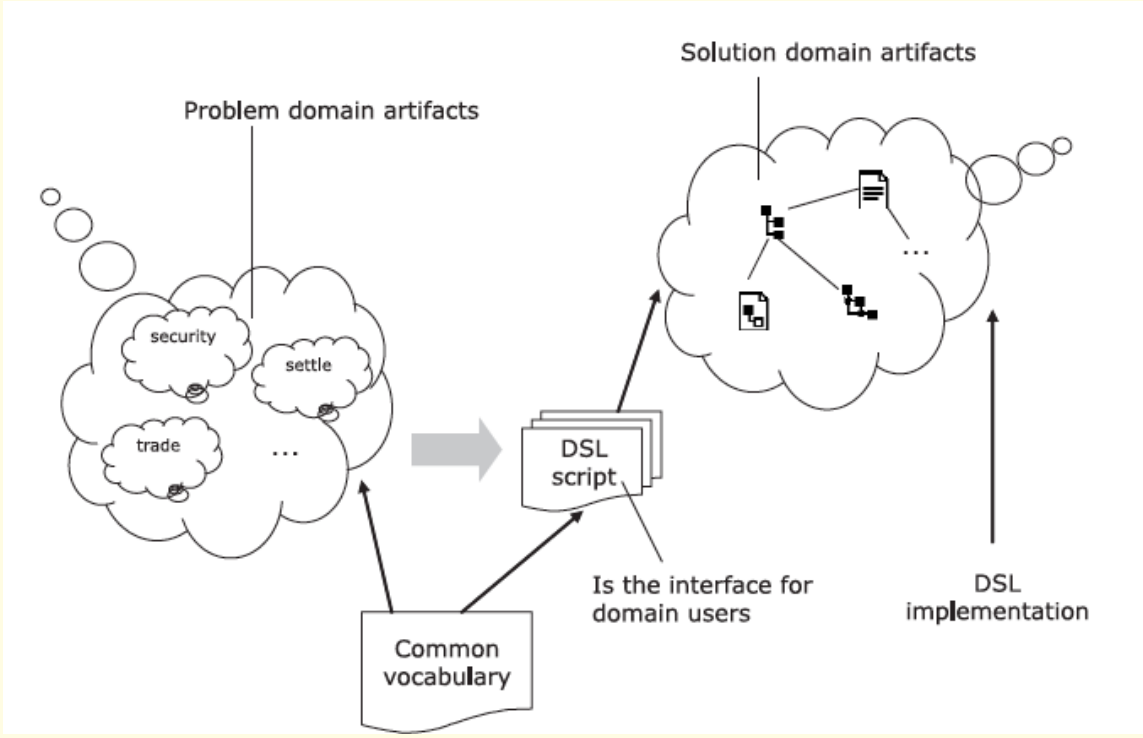
Un DSL cerca di ridurre il divario semantico tra le idee degli utenti e la loro espressione in un programma



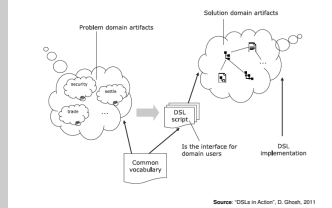


-

Un DSL utilizza il vocabolario del dominio d'applicazione



Source: "DSLs in Action", D. Ghosh, 2011



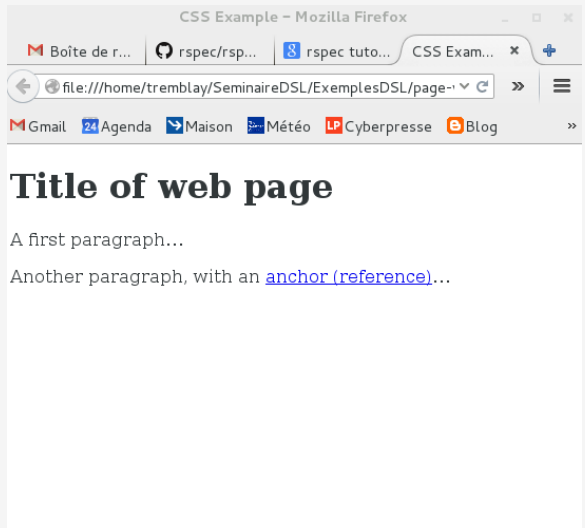
- Problem domain : Process, entities, constraints part of the business being analyzed
- Solution domain : Implementation aspects involving computer science and software engineering tools and techniques
- Problem domain[:] "all information that defines the problem and constrains the solution (the constraints being part of the problem). It includes the goals that the problem owner wishes to achieve, the context within which the problem exists, and all rules that define essential functions or other aspects of any solution product."

Alcuni esempi di DSL

-

HTML and CSS : HTML descrive la struttura e il contenuto

```
<!DOCTYPE HTML>
<HTML>
<HEAD>
<TITLE>CSS Example</TITLE>
</HEAD>



<BODY>
<H1>Title of web page</H1>

<P>A first paragraph...</P>

<P>Another paragraph, with an
<A HREF=".">anchor
(reference)</A>...</P>
</BODY>

</HTML>
```



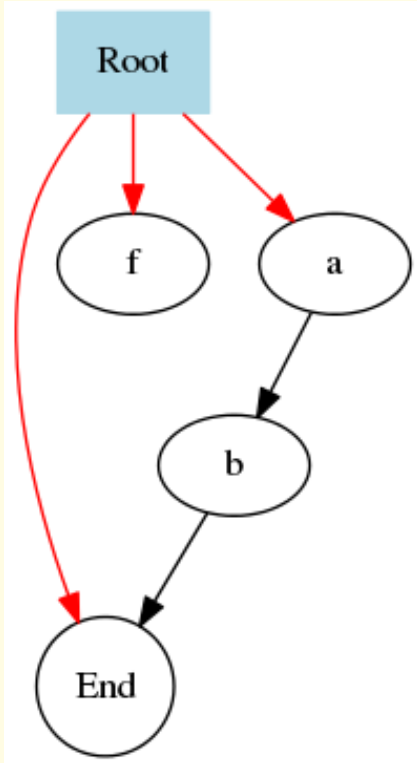
HTML and CSS : CSS descrive il *layout* e lo stile

```
bodybody {
background-color:lightgrey light-
grey;
}

h1 h1 {
  color: blackblack;
  text-align: center;
}

pp {
  font-size: 20px;
  color: blue blue;
}

aa {
  color: red red;
}
```

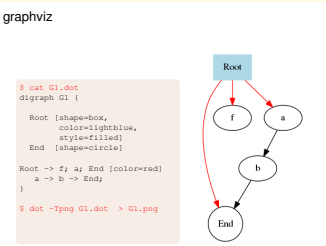
-

graphviz : Specificazione di grafici

```
$ cat G1.dot
digraph G1 {

  Root [shape=box,
        color=lightblue,
        style=filled]
  End  [shape=circle]

Root -> f; a; End [color=red]
   a -> b -> End;
}

$ dot -Tpng G1.dot  > G1.png
```



graphviz



-

Make : Unix build tool

Utilizza una sintassi semplice e *ad hoc* — tabulazioni $\neq$ spazi

```
$ cat hello.c
#include <stdio.h>

int main() {
  printf( "Hello, World!\n" );
}

$ cat Makefile
hello: hello.c
        gcc -o hello hello.c

clean:
        rm -f hello hello.o
```

```
$ make
gcc -o hello hello.c

$ ./hello
Hello, World!

$ make clean
rm -f hello hello.o
```

Source: `http://hyperpolyglot.org/build`

- Wikipedia : "a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program"
- Uses a very *ad hoc* syntax, for instance, tabs and white spaces are significant and distincts—something Stuart Feldman, the designer/author of `make` said he regretted !

Ant : Java build tool

Utilizza una descrizione XML delle dipendenze e delle regole

```
$ cat Hello.java
public class Hello {
  public static void main() {
    System.out.println(
      "Hello, World!" );
  }
}
```

```
$ cat build.xml
<project default="hello">
  <target name="hello">
    <javac srcdir="."
       includeantruntime="false"
       destdir="."/>
  </target>

  <target name="clean">
    <delete>
      <fileset dir="."
         includes="*.class"/>
    </delete>
  </target>
</project>
```

```
$ ant
Buildfile: build.xml

hello:
    [javac] Compiling 1 source
            file to [...]

BUILD SUCCESSFUL
Total time: 3 seconds

$ java Hello
Hello, World!

$ ant clean
Buildfile: build.xml

clean:

BUILD SUCCESSFUL
Total time: 0 seconds
```


- Another build tool, used mainly in the Java ecosystem.
- Properties and dependencies are described using XML—in a quite verbose fashion, as is always the case with XML ☹

Rake : Ruby build tool

Utilizza ordinario Ruby ⇒ DSL interno

```
$ cat hello.c
#include <stdio.h>

int main() {
  printf( "Hello, World!\n" );
}

$ cat Rakefile
task :default => "hello"

file "hello" => ["hello.c"] do
  sh "gcc -o hello hello.c"
end

task :clean do
     rm_f "hello hello.o"
end
```

```
$ rake
gcc -o hello hello.c

$ ./hello
Hello, World!

$ rake clean
rm -f hello hello.o
```

Source: `http://hyperpolyglot.org/build`

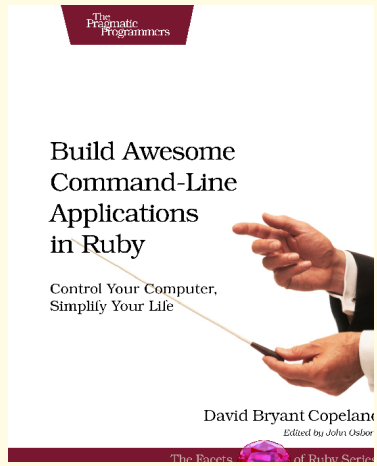
- Another build tool, from the Ruby ecosystem.
- Its major characteristic : a rake script is a regular Ruby script : no special—ad hoc or XML—syntax, just plain Ruby syntax !

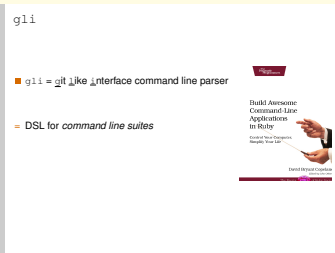
`gli` : Per specificare interfacce a riga di comando

Utilizza ordinario Ruby ⇒ DSL interno

- `gli` = `g`it `l`ike `i`nterface command line parser

= DSL for *command line suites*





- `gli` is a gem (a Ruby library) for defining command line suites.

`gli` : Per specificare interfacce a riga di comando

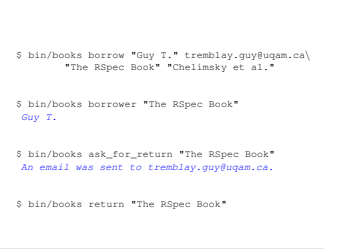
Una interfaccia a riga di comando per prestare e riportare libri

```
$ bin/books borrow "Guy T." tremblay.guy@uqam.ca\
        "The RSpec Book" "Chelimsky et al."
```

```
$ bin/books borrower "The RSpec Book"
 Guy T.
```

```
$ bin/books ask_for_return "The RSpec Book"
 An email was sent to tremblay.guy@uqam.ca.
```

```
$ bin/books return "The RSpec Book"
```

- For example, suppose we want to build a small program to manage a set of books, i.e., to know to which persons I have lent some of my books.
- Furthermore, since I always prefer to use the command-line instead of GUI, I want to be able to use commands such as those.

gli : Per specificare interfacce a riga di comando

Implementazione di bin/books con gli

```
desc 'Indicates the lending of a book'
arg_name 'name email title authors'
command :borrow do |c|
  c.action do |global_options,options,args|
    check_nb_args args, 4

     ... perform appropriate processing ...

  end
end
```


desc 'Indicates the lending of a book'
arg_name 'name email title authors'
command :borrow do |c|
  c.action do |global_options,options,args|
    check_nb_args args, 4

     ... perform appropriate processing ...

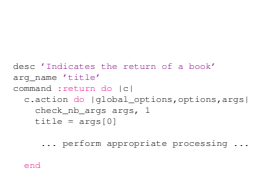
  end
end

- Using gli, each command is specified using the command method, followed by the name of the command to define, followed by a block that specifies properties and actions associated with that command, inclunding, most importantly, the action to execute when this command is to be run.

gli : Per specificare interfacce a riga di comando

Implementazione di bin/books con gli (cont.)

```
desc 'Indicates the return of a book'
arg_name 'title'
command :return do |c|
  c.action do |global_options,options,args|
    check_nb_args args, 1
    title = args[0]

     ... perform appropriate processing ...

  end
end
```

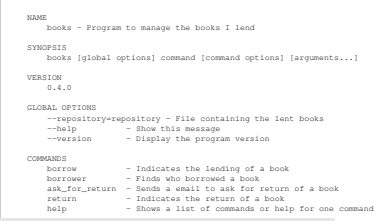



- These various commands represent, in the MVC (Model-View-Controller) terminology, the various controlers that, when run, will call the appropriate opertions from the domain—the model.

gli : Per specificare interfacce a riga di comando

Implementazione predefinita per "bin/books help"

```
NAME
    books - Program to manage the books I lend

SYNOPSIS
    books [global options] command [command options] [arguments...]

VERSION
    0.4.0

GLOBAL OPTIONS
    --repository=repository - File containing the lent books
    --help                  - Show this message
    --version               - Display the program version

COMMANDS
    borrow                  - Indicates the lending of a book
    borrower                - Finds who borrowed a book
    ask_for_return          - Sends a email to ask for return of a book
    return                  - Indicates the return of a book
    help                    - Shows a list of commands or help for one comm
```

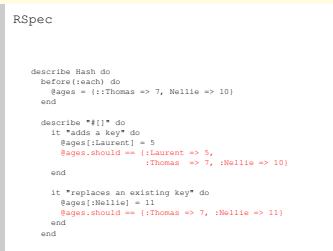



- When all the appropriate actions are specified, by default, a `help` command is automatically generated, producing the output shown above.

RSpec : A Behavior-driven test framework

Utilizza ordinario Ruby ⇒ DSL interno

```
describe Hash do
  before(:each) do
    @ages = {::Thomas => 7, Nellie => 10}
  end

  describe "#[]" do
    it "adds a key" do
      @ages[:Laurent] = 5
      @ages.should == {:Laurent => 5,
                       :Thomas  => 7, :Nellie => 10}
    end

    it "replaces an existing key" do
      @ages[:Nellie] = 11
      @ages.should == {:Thomas => 7, :Nellie => 11}
    end
  end
```

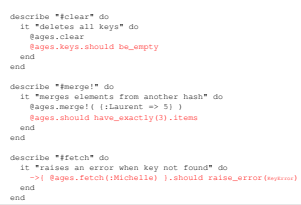



- RSpec, another Ruby gem, defines a languages for describing tests using a "behavior driven" approach. The tests are called "examples". A test is defined with the `it` method, and its name can be an arbitrary, meaningful, name. Sets of examples are organized within `describe`, which can be nested.

RSpec : A Behavior-driven test framework

Utilizza ordinario Ruby ⇒ DSL interno

```
describe "#clear" do
  it "deletes all keys" do
    @ages.clear
    @ages.keys.should be_empty
  end
end

describe "#merge!" do
  it "merges elements from another hash" do
    @ages.merge!( {:Laurent => 5} )
    @ages.should have_exactly(3).items
  end
end

describe "#fetch" do
  it "raises an error when key not found" do
    ->{ @ages.fetch(:Michelle) }.should raise_error(KeyError)
  end
end
```

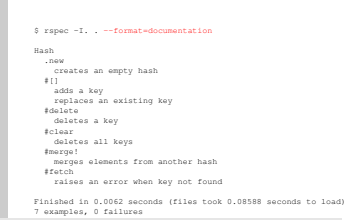



- RSpec also defines a language for describing/specifying expectations—possibly complex properties of the expected result or behavior, normal or exceptional.

RSpec : A Behavior-driven test framework

Utilizza ordinario Ruby ⇒ DSL interno

```
$ rspec -I. . --format=documentation

Hash
  .new
    creates an empty hash
  #[]
    adds a key
    replaces an existing key
  #delete
    deletes a key
  #clear
    deletes all keys
  #merge!
    merges elements from another hash
  #fetch
    raises an error when key not found

Finished in 0.0062 seconds (files took 0.08588 seconds to load)
7 examples, 0 failures
```

- When a series of tests/examples is executed with RSpec using the `documentation` format, the output, if the tests have been properly named, can be seen as an informatl specification of the expected behavior of the unit under test.

Un ADL per la distribuzione di applicazioni parallele

ADL = Architeturra Description Language

```
...
node TSHost {
  processingunits : PowerPC450[4];
  ...
}

network TSNet {
  topology : Torus;
  ...
  jitter : 10;
}

physicalconfiguration TSC {
  nodes : TSHost[350];
  network : TSNet;
}
```

Source: “Domain Specific Language for Deployment of Parallel Applications on Parallel Computing Platforms”, Arkin & Tekinerdogan, 2014

Un ADL per la distribuzione di applicazioni parallele

...
node TSHost {
processingunits : PowerPC450[4];
...
}

network TSNet {
topology : Torus;
...
jitter : 10;
}

physicalconfiguration TSC {
nodes : TSHost[350];
network : TSNet;
}

Source: “Domain Specific Language for Deployment of Parallel Applications on Parallel Computing Platforms”, Arkin & Tekinerdogan, 2014

- As mentioned earlier, DSLs can be used for specific class of application or for specific application aspect. This has already been illustrated with some of the previous examples : presentation (HTML/CSS), build (make/ant/rake), command-line controllers (gli), tests through behavior description (rspec).
- Another aspect where numerous DSLs have been proposed is architecture description. An Architecture Description Language (ADL) is a kind of DSL, whose purpose is to specify/describe the overall architecture of an application.
- This is illustrated by the following ADL, whose purpose is to describe how complex large-scale parallel applications are deployed on parallel architectures.

Un ADL per la distribuzione di applicazioni parallele

ADL = Architettura Description Language

```
application TrafficSimulator {

  component Car {
    type : ParallelComponent;
    deployed on : TSC.nodes[1..150];
  }

  component Truck {
    type : ParallelComponent;
    deployed on : TSC.nodes[151..170];
  }

  component Driver {
    type : ParallelComponent;
    deployed on : TSC.nodes[171..340];
  }

  ...

   component TrafficAnalyzer {
     type : SerialComponent;
     deployed on : TSC.nodes[347].processingunits[1];
   }
}
```


-

Sistema di mediazione finanziario :
Per comperare e vendere azioni
Utilizza ordinario Scala ⇒ DSL interno

```
orders = []

ord1 = newOrder.to.buy( 100.shares.of('IBM') {
  limitPrice 300
  allOrNone true
  valueAs { qty, unitPrice -> qty * unitPrice - 500 }
}
orders << ord1

ord2 = newOrder.to.buy( 200.shares.of('GOOG') {
  limitPrice 500
  allOrNone true
  principal = 10000
  valueAs { qty, unitPrice -> qty * unitPrice - 500 }
}
orders << ord2
```

Source: "DSLs in Action", D. Ghosh, 2011

Sistema di mediazione finanziario

orders = []

ord1 = newOrder.to.buy(100.shares.of('IBM') {
limitPrice 300
allOrNone true
valueAs { qty, unitPrice -> qty * unitPrice - 500 }
}
orders << ord1

ord2 = newOrder.to.buy(200.shares.of('GOOG') {
limitPrice 500
allOrNone true
principal = 10000
valueAs { qty, unitPrice -> qty * unitPrice - 500 }
}
orders << ord2

Source: "DSLs in Action", D. Ghosh, 2011

- Of course, there are also DStravaux de programmationLs that are targeted to specific, non-CS-related, application domain. Ghost, in his 2011 book, describe one such DSL for financial brokerage operations.
- Such application domain specific DSL, for someone not acquainted with that domain (e.g., a typical software developer), are clearly not easy to understand. But, if the DSL is well-designed, it should be easy to understand for someone involved in that application domain.

Oto : Un programma per correggere i compiti di programmazione

Utilizza ordinario Ruby ⇒ DSL interno

```
$ cat marking-script.oto
group.each do |assignment|
  comp = compile_with_javac( assignment ) {
    :file >> "Assignment1.java"
  }

  tests = test_with_junit( assignment ) {
    :class >> "TestAssignment1"
  }
  nbErrs  = tests[:nberrors]
  nbTests = tests[:nbtests]

  assignment["Nb. tests"]  = nbTests
  assignment["Nb. errors"] = nbErrs
  assignment["Final mark"] = 100.0 * ( nbTests - nbErrs ) / nbTests
end

puts generate_full_report( group )
```


Oto

$ cat marking-script.oto
group.each do |assignment|
comp = compile_with_javac(assignment) {
:file >> "Assignment1.java"
}
tests = test_with_junit(assignment) {
:class >> "TestAssignment1"
}
nbErrs = tests[:nberrors]
nbTests = tests[:nbtests]
assignment["Nb. tests"] = nbTests
assignment["Nb. errors"] = nbErrs
assignment["Final mark"] = 100.0 * (nbTests - nbErrs) / nbTests
end
puts generate_full_report(group)

- The last DSL I mention is for an application that my students and I have been developing, called Oto.
- Oto is a program that helps instructors mark programming assignments. It is based on the use of tests, textual or xUnit-style
- To mark a group of assignments, the instructor specifies a "marking script"—the process and rules used to mark the assignments.
- Since version 2 (circa 2010), these marking scripts are written using a Ruby internal fragmentary DSL, i.e., Ruby code with marking specific extensions.
- Above is an example of such a marking script : for each `assignment` in the `group`, the following steps are performed : compile the program, run the appropriate tests, compute the final mark.

Oto : Un programma per correggere i compiti di programmazione

Utilizza ordinario Ruby ⇒ DSL interno

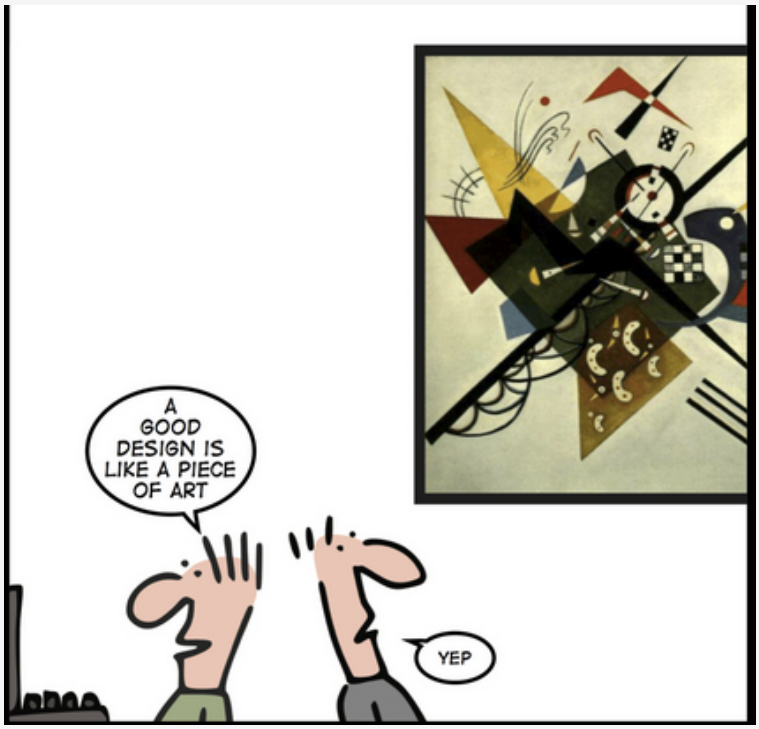
```
$ oto marking-script.oto Submitted-assignments/*.tp_oto
...

ASSIGNMENT: tremblay+2010.06.18.08.41.917949+TREG05065801.tp_oto
Team:       DURN27018400
Submission: 2010-06-18 at 08:41
Submitter:  tremblay
Name:       tremblay
Email:      ???

RESULTS:
  Nb. tests:
      4
  Nb. errors:
      0
  Final mark:
      100.0

...
```


- When the script is executed on a group of assignments submitted by students, a report such as the following is produced—the report is an excerpt that shows the result for the assignment submitted by one student/team ; various group statistics can also be included in the report.

Perché usare DSL ?

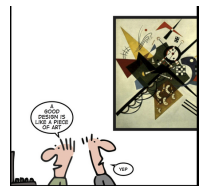
-

Un DSL fornisce astrazioni ad alto livello



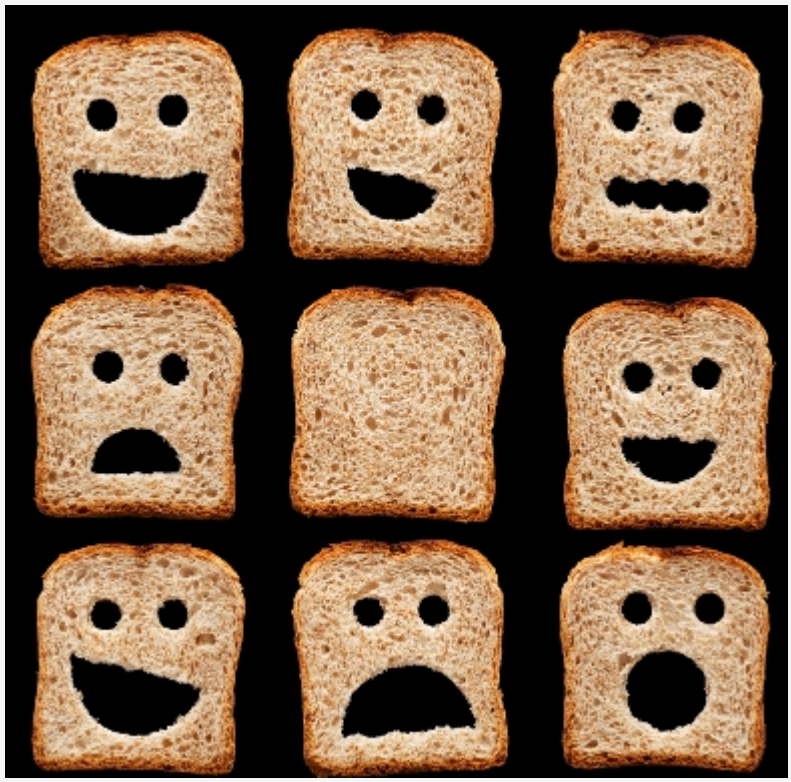
Source:

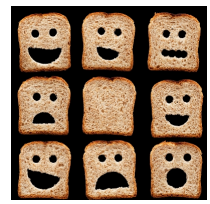
`geek-and-poke.com`



- A DSL provides high level abstractions that match the level of abstraction of the application domain.
- Having a well-designed DSL often leads to a better separation between the domain rules and the technical code implementing those rules : technical implementation details should not be present in the DSL script.

Un DSL è un linguaggio molto espressivo





- A DSL provides a very expressive language, again because it matches the concepts of the application domain
- High level abstractions combined with expressiveness lead to conciseness, thus to better productivity

Un DSL facilita la comunicazione con gli esperti

"I know nothing about the subject,
but I'm happy to give you my expert opinion."

- Expressiveness with respect to the application domain also means that a DSL script should be easier to read than a script in a regular programming language. This should thus improve the communication with the domain experts.
- In fact, a DSL allows domain experts to be involved in the specification of the business rules. Requiring those domain experts to be able to write using the DSL might be a bit "too much to ask", but at least they should be able to easily read them, and possibly (?) modify them.

Svantaggi delle DSL

- Language design is hard :
  DSL ⇒ Domain knowledge

- Another (new) language to learn

- Another software layer

• Language design is hard :
DSL ⇒ Domain knowledge
• Another (new) language to learn
• Another software layer

- Another layer ⇒ There might be performance concerns in the case of an internal DSL, since such DSL are generally implemented using scripting (non-compiled) languages. However, as for any other program and software, optimization should be a concern only when we can know if/where there is indeed a problem.
- Ass Don Knuth wrote : "We should forget about small efficiencies, say about 97% of the time : premature optimization is the root of all evil"

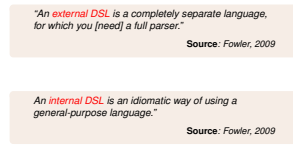
Diversi tipi di DSL

-

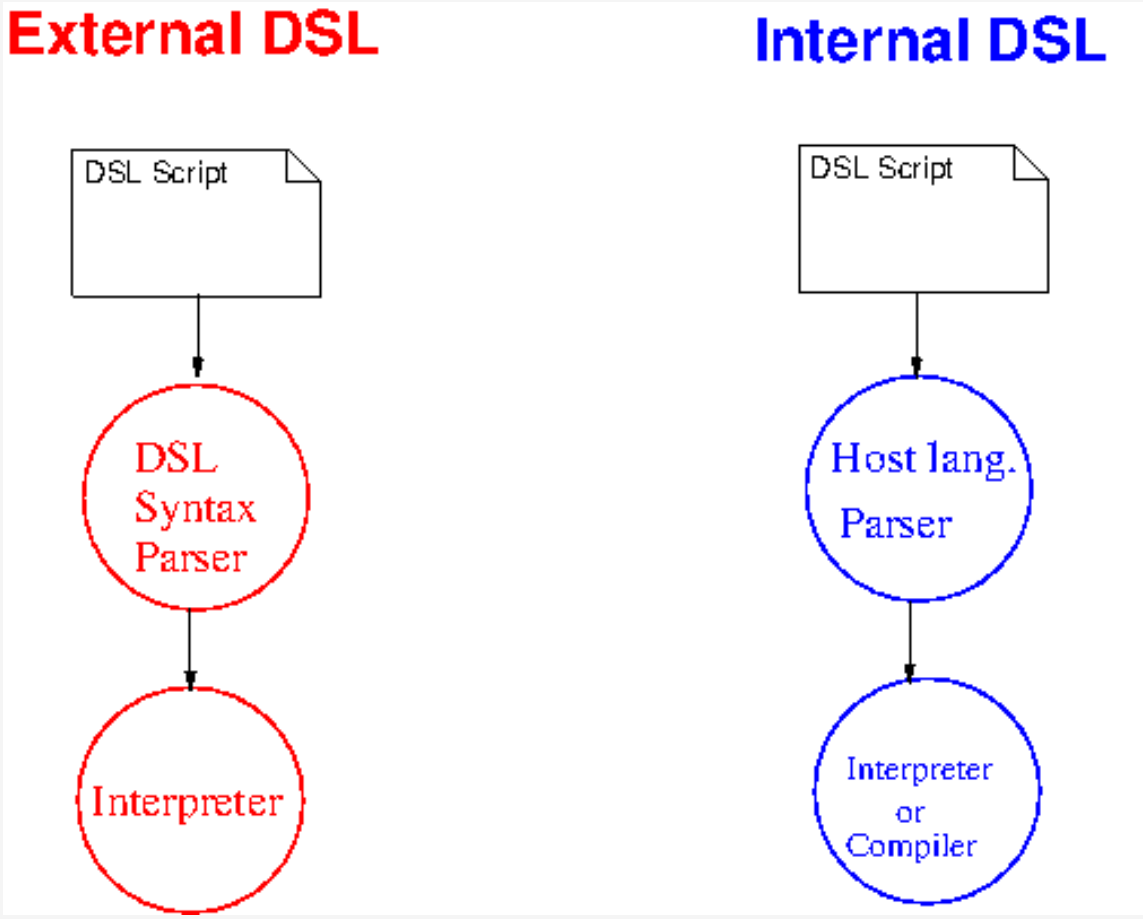
DSL esterno vs. DSL interno : Caratterizzazione di Fowler

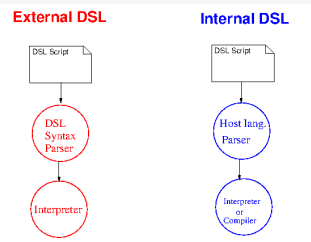
> *"An external DSL is a completely separate language, for which you [need] a full parser."*
>
> **Source**: *Fowler, 2009*

> *An internal DSL is an idiomatic way of using a general-purpose language."*
>
> **Source**: *Fowler, 2009*

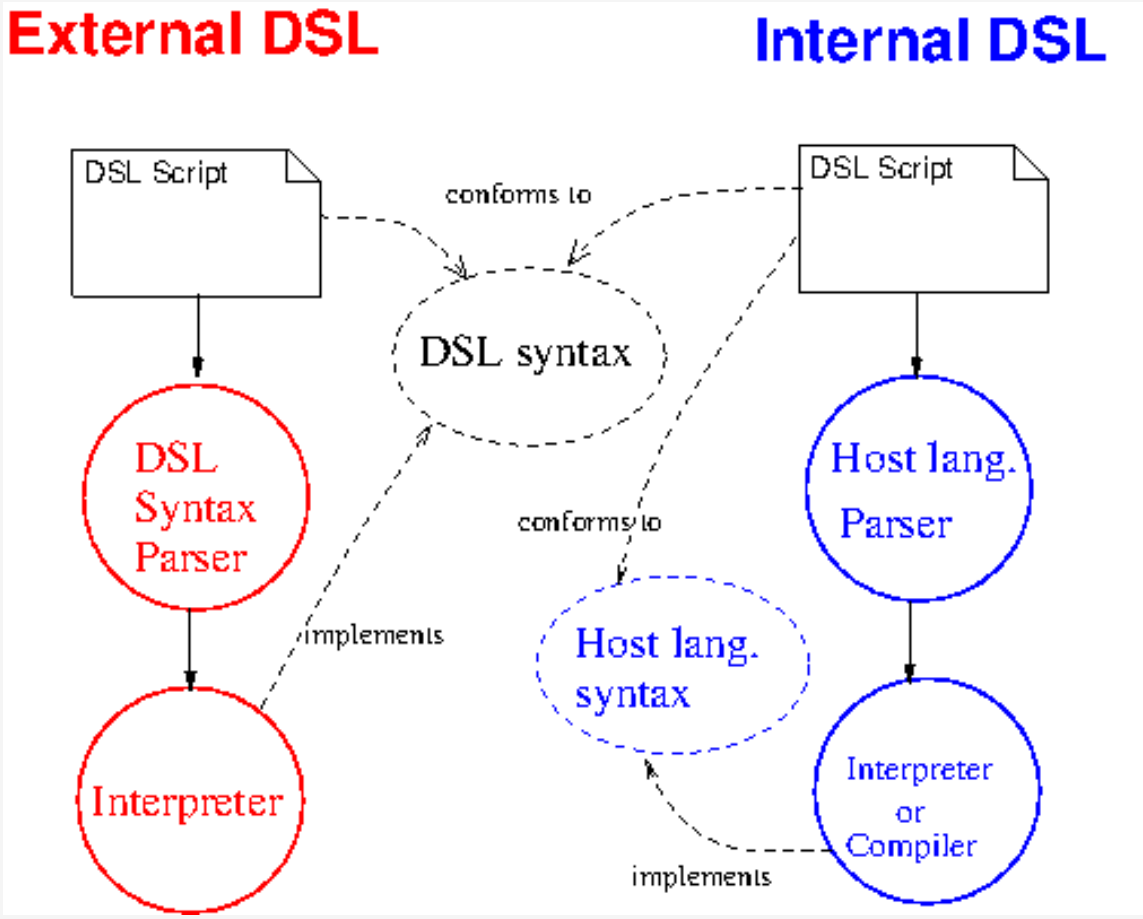


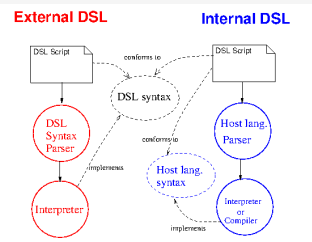
-





- External vs. internal :
  - External : the language is distinct from the language used to develop the application, so an independent parser/interpreter must be built—using the usual/typical tools (e.g., lexer/parser generator such as lex/yacc, flex/bison, antlr, etc.)
  - Internal : is a "sublanguage" of the language used to develop the application, i.e., it "uses" the infrastructure of an existing programming language (called the host language). In other words, it is implemented "on top of" an existing programming language, so it must obey the constraints (syntax, semantics) of the host language.


External DSL
Internal DSL
DSL Script
conforms to
DSL Script
DSL syntax
DSL
Syntax
Parser
Host lang.
Parser
conforms to
implements
Host lang.
syntax
Interpreter
Interpreter
or
Compiler
implements


External DSL
Internal DSL
DSL Script
conforms to
DSL Script
DSL syntax
DSL
Syntax
Parser
Host lang.
Parser
conforms to
implements
Host lang.
syntax
Interpreter
Interpreter
or
Compiler
implements

-

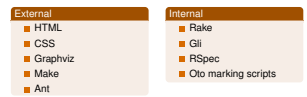
DSL esterno vs. DSL interno : Esempi

External

- HTML
- CSS
- Graphviz
- Make
- Ant

Internal

- Rake
- Gli
- RSpec
- Oto marking scripts



-

DSL indipendente vs. DSL frammentato : Caratterizzazione di Fowler

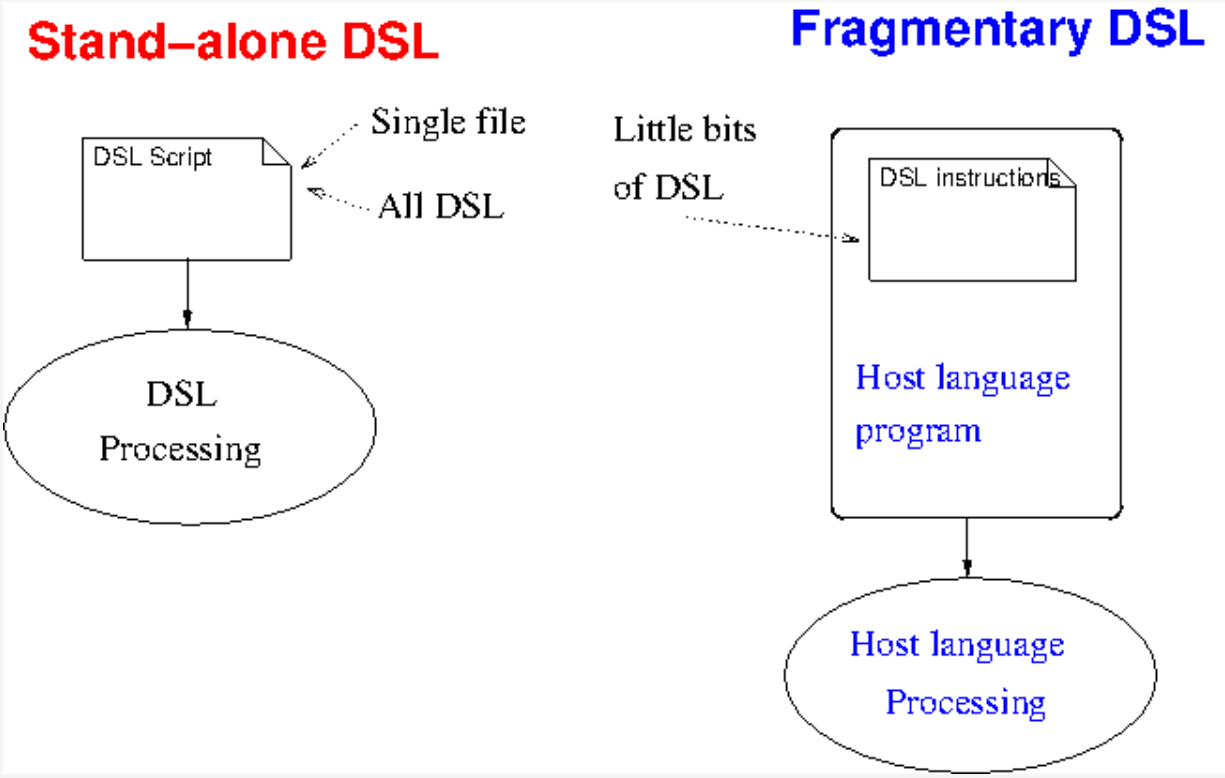
> *"[A stand-alone DSL is used through a] DSL script, typically in a single file, and it is all DSL."*
>
> **Source**: *Fowler, 2009*

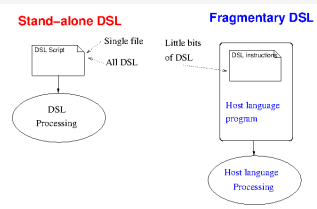
> *"[With a fragmentary DSL,] little bits of DSL are used inside the host language code. You can think of them as enhancing the host language with additionnal features."*
>
> **Source**: *Fowler, 2009*

"[A stand-alone DSL is used through a] DSL script, typically in a single file, and it is all DSL."
Source: Fowler, 2009

"[With a fragmentary DSL,] little bits of DSL are used inside the host language code. You can think of them as enhancing the host language with additionnal features."
Source: Fowler, 2009

-





- Stand-alone :
  - "[Used through a] DSL script, typically in a single file, and it is all DSL."
  - "In this case, you could/should follow what the DSL is doing without understanding the host language."
- Fragmentary :
  - "Little bits of DSL are used inside the host language code. You can think of them as enhancing the host language with additionnal features."
  - "In this case, you can't really follow what the DSL is doing without understanding the host language."

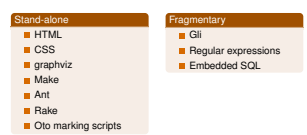
DSL indipendente vs. DSL frammentato : Esempi

Stand-alone

- HTML
- CSS
- graphviz
- Make
- Ant
- Rake
- Oto marking scripts

Fragmentary

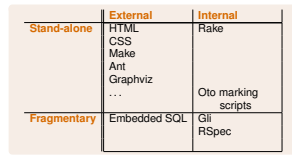
- Gli
- Regular expressions
- Embedded SQL



-

Le varie categorie : Esempi

	External	Internal
Stand-alone	HTML CSS Make Ant Graphviz . . .	Rake Oto marking scripts
Fragmentary	Embedded SQL	Gli RSpec



-

Una breve introduzione a Ruby

Caratteristiche generali di Ruby

-

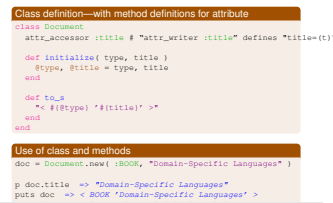
Ruby è un linguaggio “*fully object-oriented*”

Class definition—with method definitions for attribute

```
class Document
  attr_accessor :title # “attr_writer :title” defines "title=(t)"

  def initialize( type, title )
    @type, @title = type, title
  end

  def to_s
    "< #{@type} ’#{title}’ >"
  end
end
```

Use of class and methods

```
doc = Document.new( :BOOK, "Domain-Specific Languages" )

p doc.title  => "Domain-Specific Languages"
puts doc  => < BOOK ’Domain-Specific Languages’ >
```

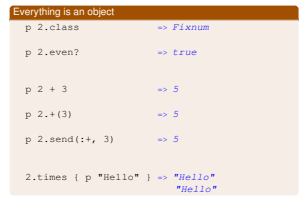



- Ruby was designed, in the mid-90s, by Yukihiro Matsumoto, a Japanese software developer.
- Ruby’s designer was inspired by a number of programming languages, notably, Perl (text and regular processing), Smalltalk (pure OO approach with full reflection) and CLU (iterators).
- Ruby is a dynamic, scripting language, in the style of Perl or Python — but, particularly with respect to Perl, much nicer and cleaner.
- Scripting dynamic language means, among other things, that is easy to call other programs, in any other language, from within a Ruby script.
- Ruby is “A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.” `https://www.ruby-lang.org/en/`
- Ruby allows for simple (implicit) definition of methods for attributes (`reader` and/or `writer`), as shown in the example.

Ruby è un linguaggio “*fully object-oriented*”

Everything is an object

```
p 2.class                  => Fixnum

p 2.even?                  => true


p 2 + 3                    => 5

p 2.+(3)                   => 5

p 2.send(:+, 3)            => 5


2.times { p "Hello" } => "Hello"
                         "Hello"
```

-

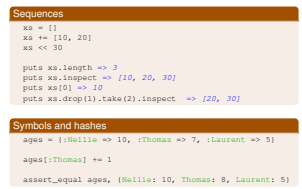
Ruby ha strutture dati ad alto livello

Sequences

```
xs = []
xs += [10, 20]
xs << 30

puts xs.length => 3
puts xs.inspect => [10, 20, 30]
puts xs[0] => 10
puts xs.drop(1).take(2).inspect  => [20, 30]
```

Symbols and hashes

```
ages = {:Nellie => 10, :Thomas => 7, :Laurent => 5}

ages[:Thomas] += 1

assert_equal ages, {Nellie: 10, Thomas: 8, Laurent: 5}
```

- Hashes : In other languages, sometimes called "dictionaries" or "maps".

Ruby permette lo stile funzionale e lo stile imperativo

Functional style

```
docs = [ Document.new( :BOOK, "Domain-Specific Languages" ),
         Document.new( :BOOK, "DSLs in Action" ),
         Document.new( :BOOK, "Domain-Driven Design" ) ]

words = docs.
          map { |d| d.title.split(/[\s-]/) }.
          flatten.
          select { |w| ('A'..'Z').cover? w[0] }


puts words.join(", ")
=> Domain, Specific, Languages, DSLs, Action,
   Domain, Driven, Design
```


- In this example, `map`, `flatten`, `select` and `join` are all pure functions — e.g., as in Haskell, although not with lazy evaluation.
- In this example, we want to find all the words that appear in the titles of the various books.

Ruby permette lo stile funzionale e lo stile imperativo

Imperative style

```
puts words.inspect
=> ["Domain", "Specific", "Languages", "DSLs",
    "Action", "Domain", "Driven", "Design"]


nb_occurrences = Hash.new(0)
words.each do |w|
  nb_occurrences[w] += 1    # Hash indexing
end

puts nb_occurrences
=> {"Domain"=>2, "Specific"=>1, "Languages"=>1,
   "DSL"=>1, "Action"=>1, "Driven"=>1, "Design"=>1}
```


- The `each` method with the `do`/`end` block is not a language-defined control structure : it is a libray method defined on `Array` objects — `each` is usually defined by all type of data structure objects.
- In this example, we want to find the number of times each word appears in the list of words.

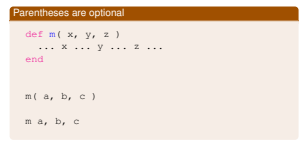
Caratteristiche di Ruby che facilitano l'implementazione di DSL interni

-

Ruby ha una sintassi flessibile

Parentheses are optional

```
def m( x, y, z )
  ... x ... y ... z ...
end


m( a, b, c )

m a, b, c
```

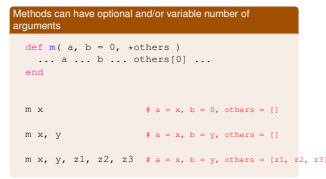



- Flexible syntax means the parser is not strict, is very lenient and tolerant : semi-colons are optional (unless you want multiple statements on the same line), parentheses are optional, etc.

Ruby ha una sintassi flessibile

Methods can have optional and/or variable number of arguments

```
def m( a, b = 0, *others )
  ... a ... b ... others[0] ...
end


m x                     # a = x, b = 0, others = []

m x, y                  # a = x, b = y, others = []

m x, y, z1, z2, z3      # a = x, b = y, others = [z1, z2, z3]
```

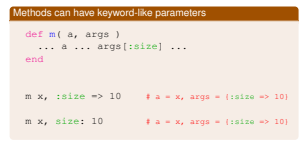



-

Ruby ha una sintassi flessibile

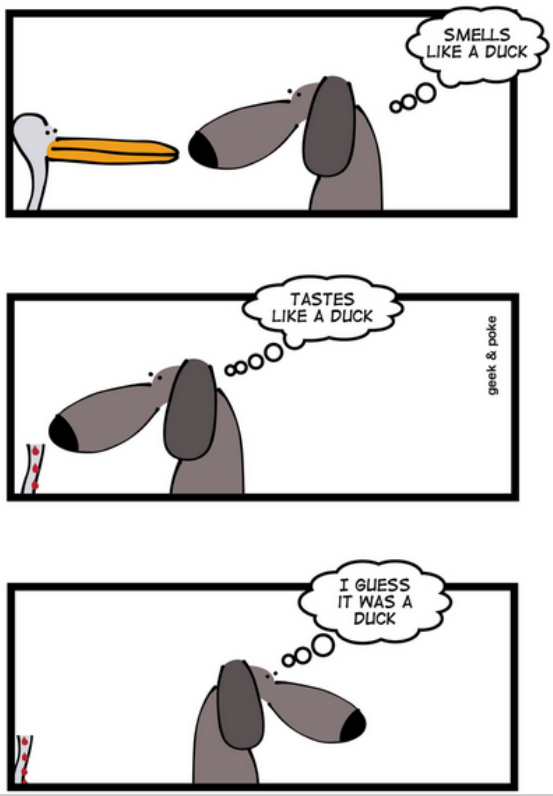
Methods can have keyword-like parameters

```ruby
def m( a, args )
  ... a ... args[:size] ...
end


m x, :size => 10      # a = x, args = {:size => 10}

m x, size: 10         # a = x, args = {:size => 10}
```

- This looks a little like in Smalltalk, with its keyword parameters.

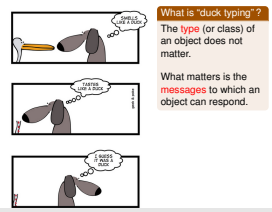
Ruby utilizza “duck typing” polimorfismo



What is “duck typing” ?

The type (or class) of an object does not matter.

What matters is the messages to which an object can respond.



- Duck typing is a form of dynamic typing, that provides support for polymorphism : The type or class of an object does not matter ; what matters is the messages the object can respond to !
- Duck typing = “If it walks like a duck, and swims like a duck and quacks like a duck, it must be a duck !”

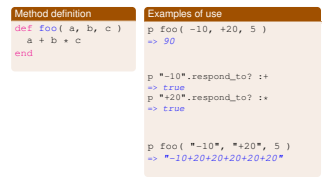
Ruby utilizza “duck typing” polimorfismo

Method definition

```
def foo( a, b, c )
  a + b * c
end
```

Examples of use

```
p foo( -10, +20, 5 )
=> 90


p "-10".respond_to? :+
=> true
p "+20".respond_to? :*
=> true


p foo( "-10", "+20", 5 )
=> "-10+20+20+20+20+20"
```

-

Ruby utilizza “duck typing”

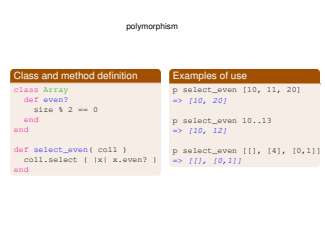
Duck typing provides support for polymorphism

Class and method definition

```ruby
class Array
  def even?
    size % 2 == 0
  end
end

def select_even( coll )
  coll.select { |x| x.even? }
end
```

Examples of use

```ruby
p select_even [10, 11, 20]
=> [10, 20]

p select_even 10..13
=> [10, 12]

p select_even [[], [4], [0,1]]
=> [[], [0,1]]
```

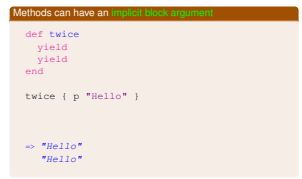
- Note : “`2.even? == true`”

Ruby facilita la programmazione di nuove strutture di controllo

Methods can have an implicit block argument

```
def twice
  yield
  yield
end

twice { p "Hello" }
```

```
=> "Hello"
   "Hello"
```

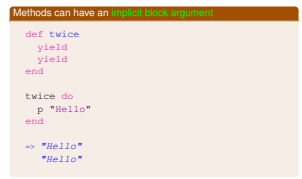



- The use of `yield` with the implicit block argument makes it possible to define new control structures. In fact, most of what "looks like" control structures in Ruby are defined in the library.

Ruby facilita la programmazione di nuove strutture di controllo

Methods can have an implicit block argument

```
def twice
  yield
  yield
end

twice do
  p "Hello"
end

=> "Hello"
   "Hello"
```

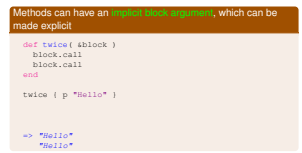



- The `yield` operator was initially introduced in the CLU language, circa 1974, so it is not something really new, although it was not well known until reintroduced by Ruby.
- The curly brackets are synonymous to `do`/`end` (except for precedence/priority in certain contexts). However, there is a convention in Ruby that curly brackets are used for single line piece of code, whereas `do`/`end` are used for multi-line segments of code.

Ruby facilita la programmazione di nuove strutture di controllo

Methods can have an implicit block argument, which can be made explicit

```
def twice( &block )
  block.call
  block.call
end

twice { p "Hello" }



=> "Hello"
   "Hello"
```

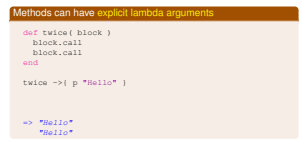



- The implicit block can be manipulated explicitly by indicating "`&block`" (or whatever other name, but with "`&`") as the last argument.

Ruby facilita la programmazione di nuove strutture di controllo

Methods can have explicit lambda arguments

```
def twice( block )
  block.call
  block.call
end

twice ->{ p "Hello" }



=> "Hello"
   "Hello"
```

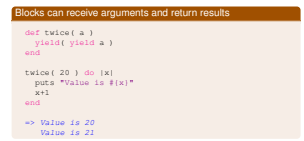



- The keyword "`lambda`" can also be used in place of "`->`".

Ruby facilita la programmazione di nuove strutture di controllo

Blocks can receive arguments and return results

```
def twice( a )
  yield( yield a )
end

twice( 20 ) do |x|
  puts "Value is #{x}"
  x+1
end

=> Value is 20
   Value is 21
```

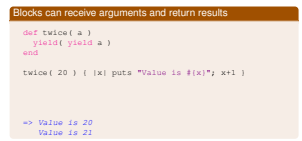
-

Ruby facilita la programmazione di nuove strutture di controllo

Blocks can receive arguments and return results

```
def twice( a )
  yield( yield a )
end

twice( 20 ) { |x| puts "Value is #{x}"; x+1 }
```

```
=> Value is 20
   Value is 21
```

- A block can be delimited by "{...}" or by "`do...end`".
- The form with "{...}" is generally used when it fits on a single line, whereas the "`do...end`" is used for blocks with multiple lines.

Ruby permette di estendere le classi esistenti

Class for `MyTime`

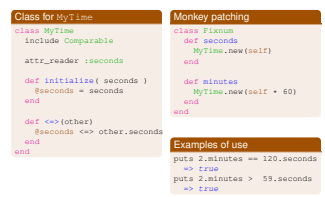
```
class MyTime
  include Comparable

  attr_reader :seconds

  def initialize( seconds )
    @seconds = seconds
  end

  def <=>(other)
    @seconds <=> other.seconds
  end
end
```

Monkey patching

```
class Fixnum
  def seconds
    MyTime.new(self)
  end

  def minutes
    MyTime.new(self * 60)
  end
end
```

Examples of use

```
puts 2.minutes == 120.seconds
  => true
puts 2.minutes >  59.seconds
  => true
```

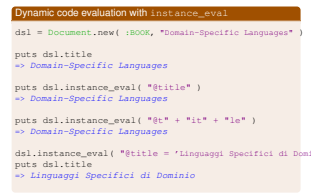



-

Ruby fornisce il supporto per metaprogrammazione dinamica

Dynamic code evaluation with `instance_eval`

```
dsl = Document.new( :BOOK, "Domain-Specific Languages" )

puts dsl.title
=> Domain-Specific Languages

puts dsl.instance_eval( "@title" )
=> Domain-Specific Languages

puts dsl.instance_eval( "@t" + "it" + "le" )
=> Domain-Specific Languages

dsl.instance_eval( "@title = 'Linguaggi Specifici di Dominio'" )
puts dsl.title
=> Linguaggi Specifici di Dominio
```

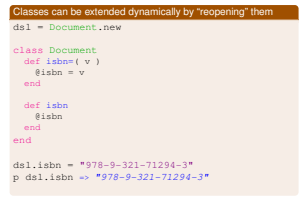



- Dynamic meta-programming means modifying or extending the behavior of the program at run-time.
- Such extensions can be done in various ways, among them : `instance_eval`, `class_eval`, `method_missing`.
- The `instance_eval` method takes as argument a string—a string that represents some code—and evaluates that string/code in the context of the receiving object.

Ruby fornisce il supporto per metaprogrammazione dinamica

Classes can be extended dynamically by “reopening” them

```ruby
dsl = Document.new

class Document
  def isbn=( v )
    @isbn = v
  end

  def isbn
    @isbn
  end
end

dsl.isbn = "978-9-321-71294-3"
p dsl.isbn => "978-9-321-71294-3"
```

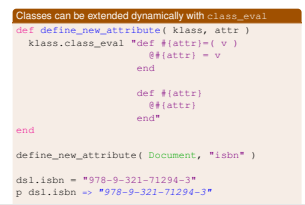



- In Ruby, it is possible to extend `at run-time` the behavior of a class.
- This can be done, for example, by reopening the class definition.

Ruby fornisce il supporto per metaprogrammazione dinamica

Classes can be extended dynamically with `class_eval`

```
def define_new_attribute( klass, attr )
  klass.class_eval "def #{attr}=( v )
                      @#{attr} = v
                    end

                    def #{attr}
                      @#{attr}
                    end"
end

define_new_attribute( Document, "isbn" )

dsl.isbn = "978-9-321-71294-3"
p dsl.isbn => "978-9-321-71294-3"
```

- A class can also be modified/extended using the `class_eval` method.
- The `class_eval` method is similar to `instance_eval` in that it receives a string that is going to be evaluated. The difference is that the string of code is evaluated in the context of the receiving class.

Ruby fornisce il supporto per metaprogrammazione dinamica

Classes can be extended dynamically with `class_eval` (bis)

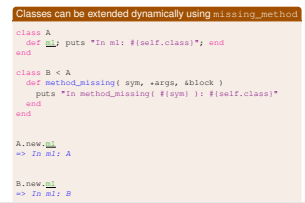
```ruby
def define_new_attribute( klass, attr )
  klass.class_eval do
    # setter
    define_method "#{attr}=" do |v|
      instance_variable_set( "@#{attr}", v )
    end

    # getter
    define_method attr do
      instance_variable_get "@#{attr}"
    end
  end
end

define_new_attribute( Document, "isbn" )
dsl.isbn = "978-9-321-71294-3"
p dsl.isbn => "978-9-321-71294-3"
```


- This would also be equivalent to the above.

Ruby fornisce il supporto per metaprogrammazione dinamica

Classes can be extended dynamically using `missing_method`

```ruby
class A
  def m1; puts "In m1: #{self.class}"; end
end

class B < A
  def method_missing( sym, *args, &block )
    puts "In method_missing( #{sym} ): #{self.class}"
  end
end


A.new.m1
=> In m1: A


B.new.m1
=> In m1: B
```

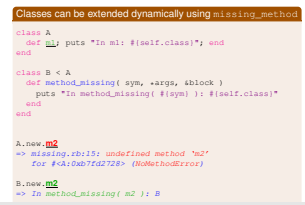



- Since `B` is a subclass of `A`, the `m1` method is implicitly defined for `B` objects, through inheritance.
- This is a feature that was taken from Smalltalk, where a class could redefine the `doesNotUnderstand:` method inherited from `Object`.

Ruby fornisce il supporto per metaprogrammazione dinamica

Classes can be extended dynamically using `missing_method`

```
class A
  def m1; puts "In m1: #{self.class}"; end
end

class B < A
  def method_missing( sym, *args, &block )
    puts "In method_missing( #{sym} ): #{self.class}"
  end
end


A.new.m2
=> missing.rb:15: undefined method `m2'
    for #<A:0xb7fd2728> (NoMethodError)

B.new.m2
=> In method_missing( m2 ): B
```

- The situation is different for `m2`, which is defined neither in `A` nor in `B`.
- When called on an `A`, an exception is raised, because no `m2` method is found.
- When called on a `B`, no `m2` method is found, but a `method_missing` method is defined, and it is this method that handles the call to `m2`.

Esempi di DSL in Ruby : Descrizioni XML di documenti

Problema : Vogliamo costruire descrizioni XML per documenti

```
<document type="BOOK">
  <title>Domain-Specific Languages</title>
  <authors>
     <author>M. Fowler</author>
     <author>R. Parsons</author>
  </authors>
  <year>2011</year>
</document>
```


- Suppose we want/need to write/build XML description for documents.
- Fact : Reading/writing XML is painful !
  - There is a lot of noise : keywords, opening vs. closing
  - There are lots of brackets—even worse than Lisp, which is full of parentheses !
  - The real information, which are to be associated with the key tags in red, is lost in a sea of useless stuff ☹
- So, we want a way to describe such documents using XML structures without all the noise and redudancy caused by the XML syntax.

Problema : Vogliamo costruire descrizioni XML per documenti

```
document type="BOOK"
  title Domain-Specific Languages
  authors
     author M. Fowler
     author R. Parsons

  year 2011
```


document type="BOOK"
  title Domain-Specific Languages
  authors
     author M. Fowler
     author R. Parsons

  year 2011

- So,


Say No to XML
Say Yes to DSL

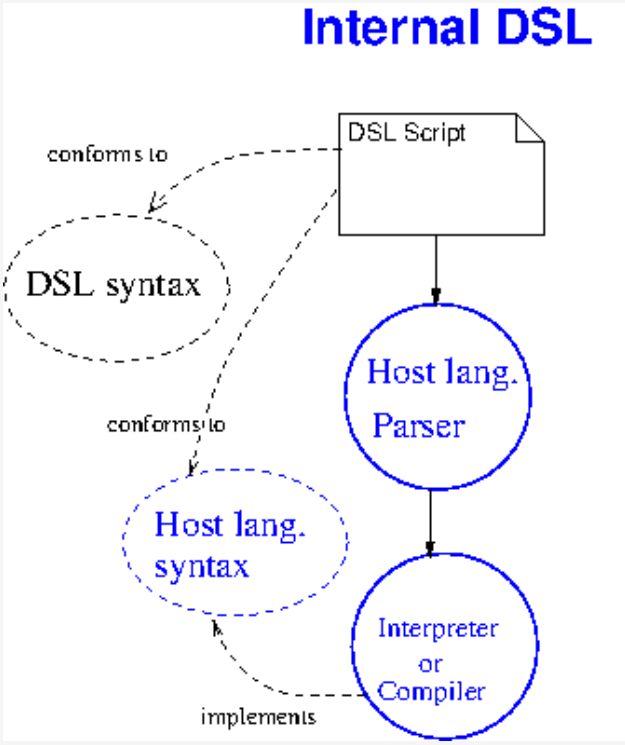

Say No to XML
Say Yes to DSL

-

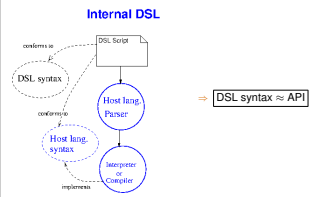
Due classi di tecniche per realizzare DSL interni con Ruby

-

Un DSL interno è come un “*well-designed*” API



$\Rightarrow$ DSL syntax $\approx$ API



- An internal DSL is defined within the language used for the application. It is thus fully constrained by the host language’s syntax.
- In that sense, an internal DSL can simply be seen as an API, except. . .

Un DSL interno è come un "*well-designed*" API

A DSL should define a fluent (application program) interface

"A fluent interface is a way of implementing an object oriented API in a way that aims to provide for more readable code."

Source: *Evans and Fowler, 2005*

Regular API vs. Internal DSL

"[By contrast with a regular API], An internal DSL should have the feel of putting together whole sentences, rather than a sequence of disconnected commands."

Source: *Fowler, 2011*

A DSL should define a fluent (application program) interface
"A fluent interface is a way of implementing an object oriented API in a way that aims to provide for more readable code."
Source: Evans and Fowler, 2005
Regular API vs. Internal DSL
"[By contrast with a regular API], An internal DSL should have the feel of putting together whole sentences, rather than a sequence of disconnected commands."
Source: Fowler, 2011

- Except that the API is much more "fluent"
- More fluent in the sense that the instructions one writes sound/look like whole sentences instead of disconnected commands.

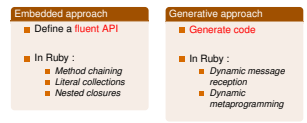
Due classi di tecniche per realizzare DSL interni

Embedded approach

- Define a fluent API
- In Ruby :
  - *Method chaining*
  - *Literal collections*
  - *Nested closures*

Generative approach

- Generate code
- In Ruby :
  - *Dynamic message reception*
  - *Dynamic metaprogramming*



- With the embedded approach, we use "regular" language constructs to define an API. However, we try to make the API readable and expressive—we try to define a "fluent API".
- With the generative approach, we use language constructs that generate code.
- In other languages, the generative approach can be obtained using macros (Lisp, clojure, Elixir), pre-processors (C), static meta-programming (Scala), etc.
- First, I want to explain a bit more the idea of "fluent interface".

Quattro DSL interni per costruire descrizioni XML per documenti

Quattro DSL interni per costruire descrizioni XML per documenti

-

DSL interni per costruire descrizioni XML per documenti

Fluent API

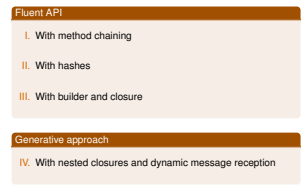
I. With method chaining

II. With hashes

III. With builder and closure

Generative approach

IV. With nested closures and dynamic message reception



-

I. Un DSL interno con “*method chaining*” : Uso

An example of use

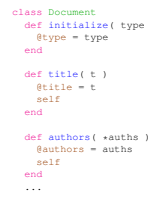
```
dsl = Document.new( :BOOK ).
  title( "Domain-Specific Languages" ).
  authors( "M. Fowler", "R. Parsons" ).
  year( 2011 )
```

À la Java !

- Typical implementation strategy for interal DSL in pure Java

I. Un DSL interno con “*method chaining*” : Implementazione

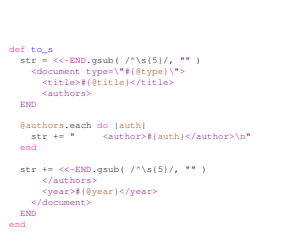
```ruby
class Document
  def initialize( type )
    @type = type
  end

  def title( t )
    @title = t
    self
  end

  def authors( *auths )
    @authors = auths
    self
  end
  ...
```

- Method chaining : each method returns the object operated on (`self` in Ruby, `this` in Java), so that the object can be used in a subsequent method call. The various method calls can then “chained” together.
- I must stress that in all these examples, I am defining a very simple and naïve implementation, with no validation and error handling. This is to simplify the presentation.
- Furthermore, I am also simply generating a string that represents the XML structure. Of course, generating an appropriate data structure is quite possible but would be a bit more complex in this introductory presentation.
- By returning `self`, the return of the method call can then be used as the target for the next method call.

I. Un DSL interno con “*method chaining*” : Implementazione (cont.)

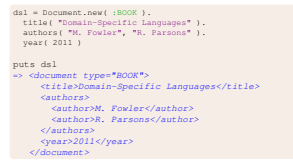
```ruby
def to_s
  str = <<-END.gsub( /^\s{5}/, "" )
    <document type=\"#{@type}\">
      <title>#{@title}</title>
      <authors>
  END

  @authors.each do |auth|
    str += "      <author>#{auth}</author>\n"
  end

  str += <<-END.gsub( /^\s{5}/, "" )
      </authors>
      <year>#{@year}</year>
    </document>
  END
end
```

- Implementation of `to_s`, shown for completeness purpose, but not that interesting in regards with discussed methods : first three approaches use exactly the same `to_s` method.

I. Un DSL interno con “*method chaining*” : Uso

```
dsl = Document.new( :BOOK ).
  title( "Domain-Specific Languages" ).
  authors( "M. Fowler", "R. Parsons" ).
  year( 2011 )

puts dsl
=> <document type="BOOK">
     <title>Domain-Specific Languages</title>
     <authors>
       <author>M. Fowler</author>
       <author>R. Parsons</author>
     </authors>
     <year>2011</year>
   </document>
```

-

II. Un DSL interno con “*hashes*” : Uso

An example of use

```
dsl = Document.new type: :BOOK,
  title: "Domain-Specific Languages",
  authors: ["M. Fowler", "R. Parsons"],
  year: 2011
```


II. Un DSL interno con “*hashes*” : Uso

An example of use

```
dsl = Document.new type: :BOOK,
  title: "Domain-Specific Languages",
  authors: ["M. Fowler", "R. Parsons"],
  year: 2011
```

- The various fields are specified using keyword-attributes, in this case using the new Ruby syntax for hashes : “`X: val`” can be used instead of “`:X => val`”

II. Un DSL interno con “*hashes*” : Implementazione

```ruby
class Document

  def initialize( *args )
    @type = args[0][:type]
    @title = args[0][:title]
    @year = args[0][:year]
    if args[0][:authors]
      @authors = args[0][:authors]
    else
      @authors = []
    end
  end
  ...
```


- Constructing the object simply requires fetching the appropriate key in the arguments hash.
- The `to_s` method is not shown, as it is identical to the previous one.

III. Un DSL interno con costruttore e “*closure*” : Uso

An example of use

```
dsl = Document.create( :BOOK ) do |doc|
  doc.title = "Domain-Specific Languages"
  doc.authors = "M. Fowler", "R. Parsons"
  doc.year = 2011
end
```


III. Un DSL interno con costruttore e “*closure*” : Uso

An example of use

```
dsl = Document.create( :BOOK ) do |doc|
  doc.title = "Domain-Specific Languages"
  doc.authors = "M. Fowler", "R. Parsons"
  doc.year = 2011
end
```

- `Document.create` will return a partially constructed object, that will be passed as argument to the block, where it will be bound to `doc`, so it can then used to call the various setters to finish its construction.
- Typical implementation for Ruby, using in various DSL.
- Provides a more specific context for building the indicated object, using the argument passed to the builder block.
- The attributes can be set using the Ruby setter style, i.e., `doc.title=` is a call to the method `def title=(t)`, explicitly (with `def`) or implicitly (with `attr_writer`) defined.

III. Un DSL interno con costruttore e “*closure*” : Implementazione

```ruby
class Document
  attr_accessor :title, :authors, :year

  def initialize( type )
    @type = type
  end

  def Document.create( type )
    new_doc = Document.new( type )
    yield new_doc
    new_doc
  end

  ...
end
```


- The various setters are defined with the `attr_accessor` methods. For each indicated name, two methods will be defined, for example :
  - `def title; @title; end`
  - `def title=(t); @title = t; end`
- The block passed to the `create` method is evaluated with the `yield` instruction, passing as argument to that block the newly created `new_doc` object.

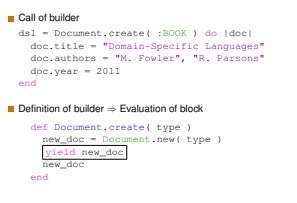
III. Un DSL interno con costruttore e “*closure*” : Implementazione (cont.)

- Call of builder

```
dsl = Document.create( :BOOK ) do |doc|
  doc.title = "Domain-Specific Languages"
  doc.authors = "M. Fowler", "R. Parsons"
  doc.year = 2011
end
```

- Definition of builder ⇒ Evaluation of block

```
def Document.create( type )
  new_doc = Document.new( type )
  yield new_doc
  new_doc
end
```

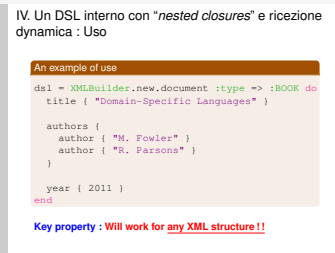
• The block passed to the `create` method is evaluated with the `yield` instruction, passing as argument to that block the newly created `new_doc` object.

IV. Un DSL interno con “*nested closures*” e ricezione dynamica : Uso

An example of use

```
dsl = XMLBuilder.new.document :type => :BOOK do
  title { "Domain-Specific Languages" }

  authors {
    author { "M. Fowler" }
    author { "R. Parsons" }
  }

  year { 2011 }
end
```

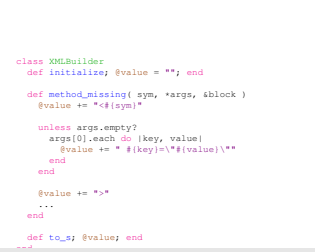
Key property : Will work for any XML structure ! !



- This DSL requires a little more details in the interface—e.g., explicit `:type`, explicit children structure with `authors/author`—but is is much more general : contrary to the previous DSLs, it would work for any kind of XML structure, not only for our documents !
- Use of this DSL looks pretty much, in terms of structure, as the XML structure itself — with the required hierarchy. The difference is that it is much less noisy, with fewer brackets, matching closing keywords, etc.

IV. Un DSL interno con “*nested closures*” e ricezione dynamica : Implementazione degli attributes

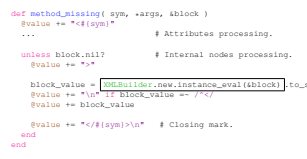
```ruby
class XMLBuilder
  def initialize; @value = ""; end

  def method_missing( sym, *args, &block )
    @value += "<#{sym}"

    unless args.empty?
      args[0].each do |key, value|
        @value += " #{key}=\"#{value}\""
      end
    end

    @value += ">"
    ...
  end

  def to_s; @value; end
end
```

- Uses the `method_missing` method shown earlier : when a symbol is encountered, this method does not exist, so it leads to a method_missing call. An appropriate XML tag is thus created, with the proper attributes, but also for the children when embedded closures calls are encountered.
- We don’t know beforehand which node will be required. It is when such a method call is performed that an appropriate XML node representation will be created.

IV. Un DSL interno con “*nested closures*” e ricezione dynamica : Implementazione dei “*children nodes*”

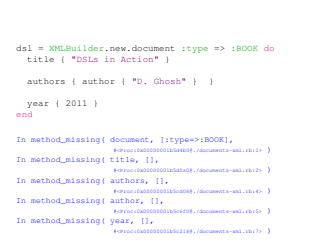
```ruby
def method_missing( sym, *args, &block )
  @value += "<#{sym}"
  ...                              # Attributes processing.

  unless block.nil?                # Internal nodes processing.
    @value += ">"

    block_value = XMLBuilder.new.instance_eval(&block).to_s
    @value += "\n" if block_value =~ /^</
    @value += block_value

    @value += "</#{sym}>\n"    # Closing mark.
  end
end
```

- Furthermore, because of the hierarchical nature of the nodes, we encounter a recursive call of the `XMLBuilder` within the processing of `missing_method`.

Un esempio che mostra le chiamate di metodo `method_missing`

```
dsl = XMLBuilder.new.document :type => :BOOK do
  title { "DSLs in Action" }

  authors { author { "D. Ghosh" }  }

  year { 2011 }
end


In method_missing( document, [:type=>:BOOK],
                        #<Proc:0x00000001b5d4b0@./documents-xml.rb:1> )
In method_missing( title, [],
                        #<Proc:0x00000001b5d0a0@./documents-xml.rb:2> )
In method_missing( authors, [],
                        #<Proc:0x00000001b5cd08@./documents-xml.rb:4> )
In method_missing( author, [],
                        #<Proc:0x00000001b5c6f0@./documents-xml.rb:5> )
In method_missing( year, [],
                        #<Proc:0x00000001b5c218@./documents-xml.rb:7> )
```

- In this simple implementation, we simply generate the appropriate string representing the XML description of the document. However, in a less naïve implementation, we would create an appropriate data structure and we would also define new methods on the fly, to allow further processing of that data structure.

Conclusione

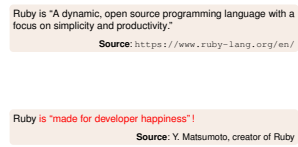
Ruby è un linguaggio di programmazione molto interessante, potente ed espressivo

Ruby is “A dynamic, open source programming language with a focus on simplicity and productivity.”

Source: `https://www.ruby-lang.org/en/`

Ruby is “made for developer happiness” !

Source: Y. Matsumoto, creator of Ruby

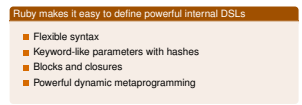


- As for developer happiness, that is how I felt when I discovered Ruby. I did a couple of years of Lisp programming (during my PhD thesis). Then, I did a couple of years of programming in Perl. I really enjoyed how powerful Perl was, with its dynamic typing and powerful string processing, but the programs were awful, ugly, difficult to read once they had been written.
- And of course, compared to Java, I really enjoyed how much more concise Ruby programs were.

Ruby è un linguaggio di programmazione molto interessante, potente ed espressivo

Ruby makes it easy to define powerful internal DSLs

- Flexible syntax
- Keyword-like parameters with hashes
- Blocks and closures
- Powerful dynamic metaprogramming



-

Alcune cose che voglio esplorare a Torino

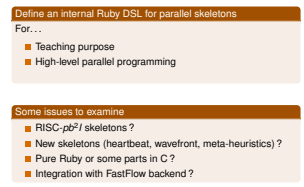
Define an internal Ruby DSL for parallel skeletons

For. . .

- Teaching purpose
- High-level parallel programming

Some issues to examine

- RISC-pb^2l skeletons ?
- New skeletons (heartbeat, wavefront, meta-heuristics) ?
- Pure Ruby or some parts in C ?
- Integration with FastFlow backend ?



• So, when I read the papers on parallel skeletons, especially the RISC-pb^2l skeletons, I thought it might be interesting to express these in Ruby.
• Could be interesting to teach high-level parallel programming strategies and patterns in (very) high-level language.
• Ruby is not, yet ?, really targeted for high-performance parallel programming. However, it might be interesting as a language in which to define a high-level DSL for parallel programming.
• That is, an interface with an appropriate backend could be defined, in any language, given the scripting nature of Ruby and the ease with which other programs can be called.

Alcune cose che voglio esplorare a Torino

-

Alcune cose che voglio esplorare a Torino

- And, very important, too...

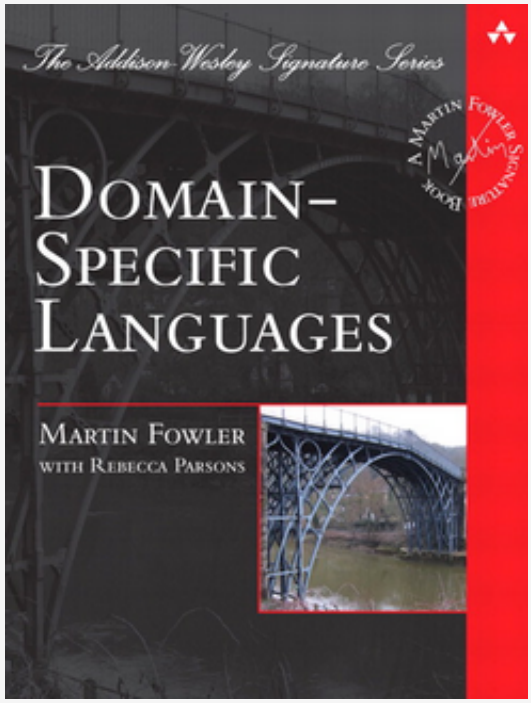
Alcune cose che voglio esplorare a Torino

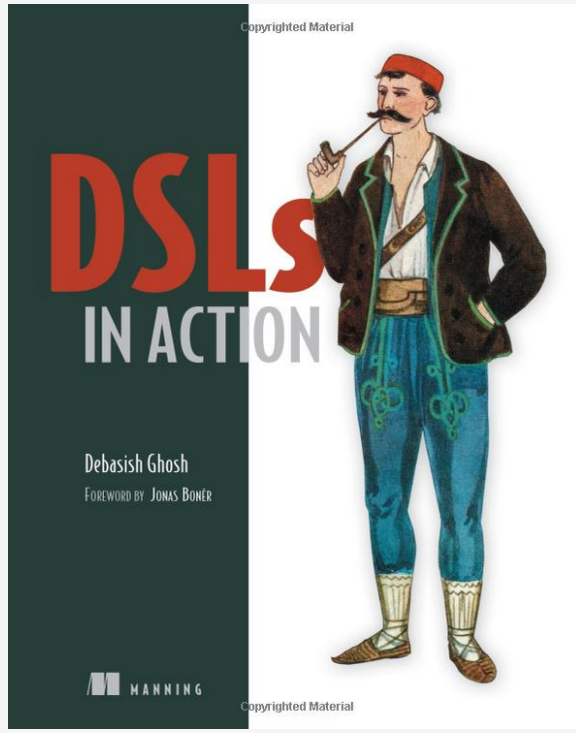
- Barolo, Barbera d'Asti, Barbaresco, Gattinara

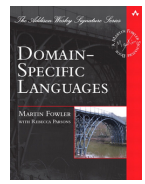
Per ulteriori informazioni sulle DSL

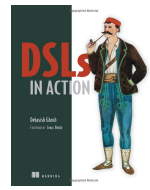
-

Per ulteriori informazioni sulle DSL









- The book by Fowler is organized in terms of patterns for defining DSL—internal vs. external ones.
- The book by Ghosh is organized more in terms of language-specific techniques—Ruby, Groovy, Scala, closure.

Per ulteriori informazioni sulle DSL

Today's slides with additional material, examples & notes :

`www.labunix.uqam.ca/~tremblay/dsl.pdf`

Today's slides with additional material, examples & notes :
www.labunix.uqam.ca/~tremblay/dsl.pdf

-

Per ulteriori informazioni sulle DSL

E. Arkin and B. Tekinerdogan.
Domain specific language for deployment of parallel applications on parallel computing platforms.
In *Proc. of the 2014 European Conf. on Software Architecture Workshops*, ECSAW '14, pages 16 :1–16 :8. ACM, 2014.

J. Bentley.
Programming pearls : little languages.
Commun. ACM, 29 :711–721, August 1986.

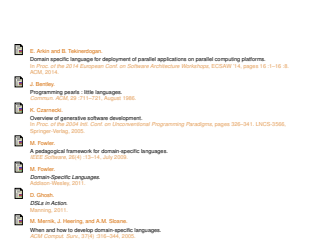
K. Czarnecki.
Overview of generative software development.
In *Proc. of the 2004 Intl. Conf. on Unconventional Programming Paradigms*, pages 326–341. LNCS-3566, Springer-Verlag, 2005.

M. Fowler.
A pedagogical framework for domain-specific languages.
IEEE Software, 26(4) :13–14, July 2009.

M. Fowler.
Domain-Specific Languages.
Addison-Wesley, 2011.

D. Ghosh.
DSLs in Action.
Manning, 2011.

M. Mernik, J. Heering, and A.M. Sloane.
When and how to develop domain-specific languages.
ACM Comput. Surv., 37(4) :316–344, 2005.



Domande ?

Domande ?

-