Langages spécifiques au domaine : Une introduction (version «abrégée»)

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

INF600A
4 décembre 2018

Contenu

1 Introduction

2 Qu'est-ce qu'un «Langage Spécifique au Domaine» ?

3 Description d'objets «complexes» à l'aide de DSLs internes en Ruby (et Java) : `Document`
- Le problème = Décrire des documents avec des attributs **variés**
- Un `module DocumentBase` partagé par les autres exemples
- Diverses façons de décrire des `Document`s
- Deux API coulantes en Java pour décrire des `Document`s

4 Conclusion

1. Introduction

Avez-vous déjà utilisé… make ?


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

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

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

Avez-vous déjà utilisé… 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
```

Avez-vous déjà utilisé... HTML ?

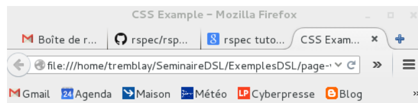

```
<HTML>
<HEAD>
<TITLE>CSS Example</TITLE>
</HEAD>

<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>
```



Title of web page

A first paragraph...

Another paragraph, with an anchor (reference)...

Si oui, alors vous avez déjà utilisé un DSL

= *Domain Specific Language*

= Langage Spécifique au Domaine

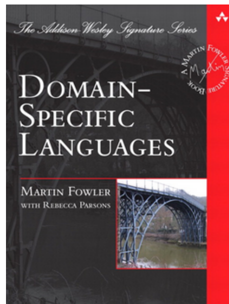
2. Qu’est-ce qu’un «Langage Spécifique au Domaine» ?

Qu’est-ce qu’un *Langage Spécifique au Domaine* (DSL) ?

Domain-specific language :

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

Source*: M. Fowler, 2011*



Quelques exemples de DSLs utilitaires

DSLs utilitaires vs. DSLs applicatif

Utility DSLs

[These] DSLs [act] simply as utilities for developers.

A developer, or a small team of developers, creates a small DSL that automates a specific, usually well-bounded aspect of software development.

Application domain DSLs

[These] DSLs describe the core business logic of an application system independent of its technical implementation. These DSLs are intended to be used by domain experts, usually non-programmers.

Source: «DSL Engineering : Designing, Implementing and Using Domain-Specific Languages», Voelter

HTML et CSS : HTML pour la structure et le contenu

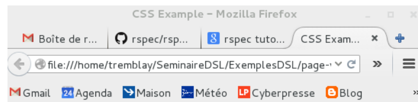
```
<!DOCTYPE HTML>
<HTML>
<HEAD>
<TITLE>CSS Example</TITLE>
</HEAD>


<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>
```



Title of web page

A first paragraph...

Another paragraph, with an anchor (reference)...

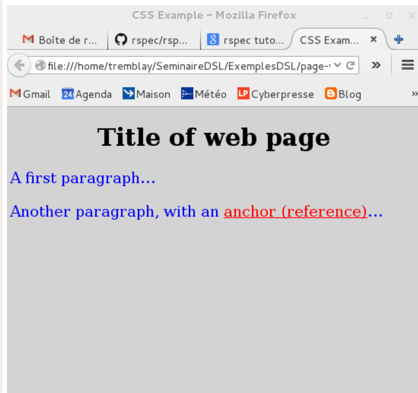
HTML et CSS : CSS pour la mise en forme et le style


```
body {
 background-color: lightgrey;
}

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

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

a {
 color: red;
}
```



`gli` : Spécification d'interfaces en mode «lignes de commandes» (à la `git`)

- `gli` = *git like interface command line parser*

- DSL pour définir *des suites de lignes de commandes*

Build Awesome Command-Line Applications in Ruby

Control Your Computer, Simplify Your Life

David Bryant Copeland

Edited by John Osborne

The Facets of Ruby Series

Make : Outil Unix d’assemblage de logiciels

Utilise une syntaxe simple et *ad hoc* — tabs $\neq$ espaces $\Rightarrow$ DSL externe

```
$ 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`

Ant : Outil Java d'assemblage de logiciels

Utilise une description XML des dépendances et des règles ⇒ DSL externe

```
$ 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
```

Rake : Outil Ruby d'assemblage de logiciels

Utilise du Ruby standard ⇒ DSL interne

```
$ 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`

Quelle est une différence majeure entre `make`/`ant` et `Rake` ?

Outil	Syntaxe
`make`	
`ant`	
`Rake`	

Quelle est une différence majeure entre `make`/`ant` et `Rake` ?

Outil	**Syntaxe**
`make`	*ad hoc*
`ant`	
`Rake`	

«[`make`*] 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 !*

Quelle est une différence majeure entre `make`/`ant` et `Rake` ?

Outil	Syntaxe
`make`	*ad hoc*
`ant`	XML
`Rake`	

«[`make`] 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 !*

Quelle est une différence majeure entre `make/ant` et `Rake` ?

Outil	Syntaxe
`make`	*ad hoc*
`ant`	XML
`Rake`	`Ruby`

«[`make`] 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 !*

Quelle est une différence majeure entre `make/ant` et `Rake` ?

Outil	Syntaxe
`make`	DSL externe
`ant`	DSL externe
`Rake`	DSL interne

«[`make`] 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 !*

DSL externe vs. DSL interne

DSL interne vs. DSL externe : Caractérisation de Fowler

DSL interne

An internal DSL is an idiomatic way of using a general-purpose language."

Source: *Fowler, 2009*

DSL externe

"An external DSL is a completely separate language, for which you [need] a full parser."

Source: *Fowler, 2009*

DSL interne ⇒ mise en œuvre à l'intérieur d'un langage existant

Approche avec API coulante

- Par exemple, en Ruby :
  - *Chainage de méthodes*
  - *Constructeur et bloc*

Approche générative ou avec métaprogrammation

- En Ruby :
  - *Envoi dynamique de messages*
  - *Génération dynamique de code*

Un DSL interne peut être vu comme une «API bien conçue», avec une interface coulante (*fluent*)

> *"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 & Fowler, 2005*

> *"[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*

3. Description d'objets «complexes» à l'aide de DSLs internes en Ruby (et Java) :

`Document`

Remarque : Exemples Ruby vs. Java

- Dans les diapositives qui suivent, les exemples pour illustrer la construction de documents avec Ruby sont en fond beige.
- D’autres exemples, sans lien avec la construction de documents, sont aussi présentés :
  - Autres exemples en Ruby : fond vert pâle.
  - Exemples en Java : fond rouge pâle.

3.1 Le problème = Décrire des documents avec des attributs **variés**

Les DSLs sont souvent utilisés pour la spécification ou configuration d'objets (ou systèmes) complexes

- Dans ce qui suit, on va spécifier/décrire des `Document`s.

- Ces objets — pour simplifier les diapositives — seront relativement simples : quatre (4) attributs seulement sont spécifiés et mis en œuvre (mais d'autres pourraient l'être).

Les DSLs sont souvent utilisés pour la spécification ou la configuration de systèmes ou d'objets complexes

- Mais, pour mieux comprendre la motivation de ce qui suit, imaginez qu'on ait plutôt des dizaines de propriétés — par ex., spécification d'un *gem* :

```
spec = Gem::Specification.new do |s|
  s.name = 'minised'
  s.version = MiniSed::VERSION
  s.author = 'Your Name Here'
  s.email = 'your@email.address.com'
  s.homepage = 'http://your.website.com'
  s.platform = Gem::Platform::RUBY
  s.summary = 'Une version simplifiee de sed'
  s.description = 'Une version simplifiee de sed, avec des commandes a la git. Utilise comme
  s.files = ...
  s.require_paths << 'lib'
  s.has_rdoc = true
  s.extra_rdoc_files = ['README.rdoc','minised.rdoc']
  s.rdoc_options << '--title' << 'minised' << '--main' << 'README.rdoc' << '-ri'
  s.bindir = 'bin'
  s.executables << 'minised'
  s.add_development_dependency('rake', '~> 11.1')
  s.add_development_dependency('rdoc', '~> 0')
  s.add_development_dependency('aruba', '0.8.1')
  s.add_development_dependency('minitest', '5.4.3')
  s.add_runtime_dependency('gli','2.12.0')
end
```

Les attributs d'un `Document`

Attributs obligatoires

- `titre`
- `auteurs`
- `annee` (de publication, donc un entier)

Attribut(s) optionnel(s)

- `editeur`
- `isbn`, `serie`, `volume`, `adresse`, etc.

Contrainte de création d'un objet (invariant)

Lorsqu'on crée un nouveau `Document`, on veut assurer qu'il soit `valide?` = tous les champs obligatoires sont présents (et non vides) et l'année est un nombre

3.2 Un `module DocumentBase` partagé par les autres exemples

Module `DocumentBase`

Ce module sera utilisé comme *mixin* — avec `include` — par les différentes versions de la classe `Document` que nous verrons

```
module DocumentBase
  def to_s
    auteurs = @auteurs.join(' et ')
    editeur = @editeur ? ", #{@editeur}" : '' # Optionnel!

    "\#<'#{@titre}', #{auteurs}#{editeur}, #{@annee}>"
  end

  def valide?
    # Titre present, avec annee et au moins un auteur.
    titre && !titre.empty? &&
      annee && annee.kind_of?(Integer) &&
      auteurs && auteurs.kind_of?(Array) &&
                 !auteurs.empty?
  end
end
```

3.3 Diverses façons de décrire des `Document`s

Les trois approches pour construire des descriptions de documents que nous allons examiner

Avec API «régulière»

A. Constructeur `new` avec arguments positionnels et par mots-clés

Avec API «coulante» style *Expression Builder*

B. API coulante avec chainage de méthodes

C. API coulante avec constructeur et bloc

A. Constructeur `new` avec arguments positionnels et par mots-clés

A. Constructeur `new` avec arguments positionnels (obligatoires) et par mots-clés (optionnels) : Utilisation

Exemples d'utilisation

```
# Sans editeur.
Document.new( 'Domain-Specific Languages',
              2011,
              'M. Fowler', 'R. Parsons' )

# Avec editeur.
Document.new( 'DSLs in Action',
              2011,
              'D. Ghosh',
              editeur: 'Manning' )
```

A. Constructeur `new` avec arguments positionnels et par mots-clés : Mise en œuvre


```
class Document
  include DocumentBase

  attr_reader :titre, :annee, :auteurs, :editeur

  def initialize( titre, annee, *auteurs,
                  editeur: nil )
    @titre = titre
    @annee = annee
    @auteurs = auteurs
    @editeur = editeur

    DBC.assert valide?, "*** Erreur: Document mal constr
  end
end
```

Avantage/désavantages à utiliser `new` pour construire des objets complexes

Avantage

- Approche simple/de base

Désavantages

Avantage/désavantages à utiliser `new` pour construire des objets complexes

Avantage

- Approche simple/de base

Désavantages

- Il faut se souvenir de quels arguments sont positionnels vs. quels arguments sont par mots-clés ☹

Avantage/désavantages à utiliser `new` pour construire des objets complexes

Avantage

- Approche simple/de base

Désavantages

- Il faut se souvenir de quels arguments sont positionnels vs. quels arguments sont par mots-clés ☹
- Il faut se souvenir de l'ordre des arguments obligatoires ☹

Avantage/désavantages à utiliser `new` pour construire des objets complexes

Avantage

- Approche simple/de base

Désavantages

- Il faut se souvenir de quels arguments sont positionnels vs. quels arguments sont par mots-clés ☹
- Il faut se souvenir de l'ordre des arguments obligatoires ☹
- Si on a un grand nombre d'attributs, alors l'interface de `initialize` va devenir (très ! ?) complexe ☹

Avantage/désavantages à utiliser `new` pour construire des objets complexes

Avantage

- Approche simple/de base

Désavantages

- Il faut se souvenir de quels arguments sont positionnels vs. quels arguments sont par mots-clés ☹
- Il faut se souvenir de l'ordre des arguments obligatoires ☹
- Si on a un grand nombre d'attributs, alors l'interface de `initialize` va devenir (très ! ?) complexe ☹
- Si plusieurs attributs sont caractérisés par une ou plusieurs valeurs, alors on doit utiliser des `Array` explicites — au plus un paramètre avec nombre variable d'éléments ☹

B. API coulante avec chainage de méthodes

B. API coulante avec chainage de méthodes : Utilisation

Exemples d'utilisation

```
# Ordre des appels pas important -- sauf build.
Document::Builder.new
  .auteurs( 'M. Fowler', 'R. Parsons' )
  .titre( 'Domain-Specific Languages' )
  .annee( 2011 )
  .build

Document::Builder.new
  .annee( 2011 )
  .titre( 'DSLs in Action' )
  .auteurs( 'D. Ghosh' )
  .editeur( 'Manning' )
  .build
```

Le chainage de méthodes est souvent utilisé en Java

Exemple : Création de notifications pour Android

```
Intent notificationIntent = new Intent(this, MainActivity.clas

PendingIntent contentIntent =
    PendingIntent.getActivity(this, 0,
        notificationIntent, PendingIntent.FLAG_UPDATE_CURRENT)

NotificationCompat.Builder builder =
    new NotificationCompat.Builder(this)
         .setSmallIcon(R.drawable.abc)
         .setContentTitle("Notifications Example")
         .setContentText("This is a test notification")
         .setContentIntent(contentIntent);

NotificationManager manager
  = (NotificationManager) getSystemService(Context.NOTIFICATIO
      manager.notify(0, builder.build());
```

Source: `https://www.tutorialspoint.com/android/android_notifications.htm`

Le chainage de méthodes est souvent utilisé en Java

Exemple : Spécification de *mocks* avec `jMock`

```
public class TimedCacheTest extends MockObjectTestCase {
    private Mock mockLoader;

    @Override
    protected void setUp() throws Exception {
        mockLoader = mock(ObjectLoader.class);
        ...
    }

    public void testLoadsObjectThatIsNotCached() {
        mockLoader.expects(once())
                  .method("load")
                  .with(eq(KEY))
                  .will(returnValue(VALUE));
        ...
    }
}
```

http:

//searchsoftwarequality.techtarget.com/tip/Using-JMock-in-test-driven-development

Le chainage de méthodes est souvent utilisé en Java

Exemple : Création de requêtes SQL avec jOOQ

```
Author author = AUTHOR.as("author");
create
   .selectFrom(author)
   .where(exists(selectOne()
           .from(BOOK)
           .where(BOOK.STATUS.eq(BOOK_STATUS.SOLD_OUT))
           .and(BOOK.AUTHOR_ID.eq(author.ID))));
```

Source: https://en.wikipedia.org/wiki/Fluent_interface

B. API coulante avec chainage de méthodes : Mise en œuvre

Définit une classe interne pour construire les documents : `Builder`, et `new` est privé

```
class Document
  include DocumentBase

  attr_reader :titre, :annee, :auteurs, :editeur

  class Builder; end # Voir plus bas.

 # Comme dans A... mais prive!
  def initialize( titre, annee, *auteurs, editeur: nil )
    @titre = titre
    @annee = annee
    @auteurs = auteurs
    @editeur = editeur
  end

  private_class_method :new
end
```

B. API coulante avec chainage de méthodes : Mise en œuvre

Retour de `self` comme résultat ⇒ permet de chainer les appels

```
class Builder
  # 'setters' pas definis pour Document => Immuables!
  def titre( t );       @titre = t;       self end
  def annee( a );       @annee = a;       self end
  def auteurs( *as );   @auteurs = *as;   self end
  def editeur( e );     @editeur = e;     self end

  def build
    nouveau_doc = Document.send( :new,  # Pourquoi send?
                                 @titre, @annee, *@auteurs,
                                 editeur: @editeur )
    DBC.assert nouveau_doc.valide?, "*** Erreur: Documen

    nouveau_doc
  end
end
```

B. API coulante avec chainage de méthodes : Utilisation (bis)

Exemples d’utilisation

```
d1 = Document::Builder.new
   .auteurs( 'M. Fowler', 'R. Parsons' )
   .titre( 'Domain-Specific Languages' )
   .annee( 2011 )
   .build

d1.titre # => 'Domain-Specific Languages'
d1.titre( 'Nouveau titre' ) # ??

d2 = Document::Builder.new
   .annee( 2011 )
   .titre( 'DSLs in Action' )
   .auteurs( 'D. Ghosh' )
   .editeur( 'Manning' ) # ??

d2.titre # ??
```

B. API coulante avec chainage de méthodes : Utilisation (bis)

Exemples d'utilisation

```
d1 = Document::Builder.new
   .auteurs( 'M. Fowler', 'R. Parsons' )
   .titre( 'Domain-Specific Languages' )
   .annee( 2011 )
   .build

d1.titre # => 'Domain-Specific Languages'
d1.titre( 'Nouveau titre' ) # => ArgumentError: wrong number of arguments

d2 = Document::Builder.new
   .annee( 2011 )
   .titre( 'DSLs in Action' )
   .auteurs( 'D. Ghosh' )
   .editeur( 'Manning' ) # Appel a build omis!

d2.titre # => ArgumentError: wrong number of arguments
```

B. API coulante avec chainage de méthodes

Avantages

- L'ordre des appels n'est pas important ☺
- Facile pour l'utilisateur si on ajoute de nouveaux attributs ☺
- Une méthode peut recevoir un nombre variable d'arguments ☺
- Mise en œuvre simple — en Ruby ou autres langages ☺

Désavantages

B. API coulante avec chainage de méthodes

Avantages

- L'ordre des appels n'est pas important ☺
- Facile pour l'utilisateur si on ajoute de nouveaux attributs ☺
- Une méthode peut recevoir un nombre variable d'arguments ☺
- Mise en œuvre simple — en Ruby ou autres langages ☺

Désavantages

- On doit définir une méthode `build` qui finalise la construction ☹

  Par contre, on est certain que l'objet est valide et il est immuable ☺

C. API coulante avec constructeur et bloc

C. API coulante avec constructeur et *bloc* : Utilisation

Exemples d'utilisation

```
Document.create do |d|
  d.titre = 'Domain-Specific Languages'
  d.auteurs = 'M. Fowler', 'R. Parsons' # (*)
  d.annee = 2011
end

Document.create do |d|
  d.annee = 2011
  d.titre = 'DSLs in Action'
  d.editeur = 'Manning'
  d.auteurs = 'D. Ghosh'
end
```

(*) *Implicit Array Assignment* : `https://docs.ruby-lang.org/en/2.4.0/syntax/assignment_rdoc.html`

Est-ce que cela vous rappelle quelque chose ?

L’approche avec constructeur et bloc est (très !) souvent utilisée en Ruby

Spécification d’un gemspec

```
spec = Gem::Specification.new do |s|
  s.name = 'minised'
  s.version = MiniSed::VERSION
  ...
  s.summary = 'Une version simplifiee de sed'
  ...
  s.add_development_dependency('minitest', '5.4.3')
  s.add_runtime_dependency('gli','2.12.0')
end
```

L'approche avec constructeur et bloc est (très !) souvent utilisée en Ruby

Spécification de tâches dans un `Rakefile`

```
# Cible pour l'ensemble des tests unitaires.
Rake::TestTask.new(:test) do |t|
  t.libs << "test"
  t.test_files = FileList['test/*_test.rb']
end

# Cible pour l'ensemble des tests d'acceptation.
Rake::TestTask.new(:test_acceptation) do |t|
  t.libs << "test_acceptation"
  t.test_files = FileList['test_acceptation/*_test.rb']
  t.warning = false
end
```

C. API coulante avec constructeur et *bloc* : Mise en œuvre


```
class Document
  include DocumentBase
  attr_reader :auteurs
  attr_accessor :titre, :annee, :editeur

  def self.create
    nouveau_doc = new
    yield nouveau_doc
    DBC.assert nouveau_doc.valide?, "*** Erreur: Document mal

    nouveau_doc
  end

  def auteurs=( auteurs )
    @auteurs = auteurs.class == Array ? auteurs : [auteurs]
  end

  private_class_method :new
end
```

C. API coulante avec constructeur et *bloc* : Mise en œuvre


```
class Document
  include DocumentBase
  attr_reader :auteurs
  attr_accessor :titre, :annee, :editeur

  def self.create
    nouveau_doc = new
    yield nouveau_doc
    DBC.assert nouveau_doc.valide?, "*** Erreur: Document mal

    nouveau_doc
  end

  def auteurs=( auteurs )
    @auteurs = auteurs.class == Array ? auteurs : [auteurs]
  end

  private_class_method :new
end
```

C. API coulante avec constructeur et *bloc* : Mise en œuvre


```
class Document
  include DocumentBase
  attr_reader :auteurs
  attr_accessor :titre, :annee, :editeur

  def self.create
    nouveau_doc = new
    yield nouveau_doc
    DBC.assert nouveau_doc.valide?, "*** Erreur: Document mal

    nouveau_doc
  end

  def auteurs=( auteurs )
    @auteurs = auteurs.class == Array ? auteurs : [auteurs]
  end

  private_class_method :new
end
```

C. API coulante avec constructeur et *bloc* : Mise en œuvre (suite)

Appel «`foo = x, y`» ⇒ *Implicit array conversion* ⇒ un (1) argument = `[x, y]`

Appel du *builder*

```
Document.create do |d|
  d.titre = 'Domain-Specific Languages'
  d.auteurs = 'M. Fowler', 'R. Parsons'
  d.annee = 2011
end
```

Définition du *builder* : On évalue le bloc passé à la méthode `create`, en donnant en argument le nouvel objet créé par `new`

```
def self.create
  nouveau_doc = new

  yield nouveau_doc

  DBC.assert nouveau_doc.valide?, "*** Erreur: Documen

  nouveau_doc
end
```

C. API coulante avec constructeur et bloc

Avantages

- L'ordre des appels n'est pas important ☺
- Facile pour l'utilisateur si on ajoute de nouveaux attributs ☺
- Une méthode peut recevoir un nombre variable d'arguments ☺
- Pas besoin de méthode `build`

Désavavantages

- Nécessite l'utilisation de blocs (Ruby) ou lambda-expressions (Java : voir plus bas)

Remarque concernant l’approche utilisée par `gli`

L'approche utilisée par `gli` est une approche hybride entre les approches C et B

Spécification des commandes avec `gli`

```
command :substitute do |c|
  c.desc 'Substitution globale'
  c.switch :g

  c.action do |global_options,options,args|
    ...

  end
end
```

L'approche utilisée par `gli` pour définir une commande (options, arguments, action) utilise un constructeur avec bloc — comme C — mais les attributs sont définis avec des méthodes sans affectation — comme B

L'approche utilisée par `gli` est une approche hybride entre les approches C et B

Approche C

```
Document.create do |d|
  d.titre = '...'    # Setter explicite
  d.auteurs = '...' # Idem
end
```

Approche B

```
Document::Builder.new
   .titre( '...' )    # Setter de Document::Builder
   .auteurs( '...' ) # Idem
```

L’approche utilisée par `gli` est une approche hybride entre les approches C et B

Approche C

```
Document.create do |d|
  d.titre = '...'    # Setter explicite
  d.auteurs = '...' # Idem
end
```

Approche B

```
Document::Builder.new
  .titre( '...' )    # Setter de Document::Builder
  .auteurs( '...' ) # Idem
```

Approche style `gli`

```
Document.create do |d|
  d.titre '...'
  d.auteurs '...'
end
```

Approche style `gli`

Java permet d'avoir un *getter* et un *setter* avec le même nom

```
private String titre;

void titre( String t ) { titre = t; }

String titre() { return titre; }
```

Question : Est-ce possible en `Ruby` ?

Approche style `gli`

Java permet d'avoir un *getter* et un *setter* avec le même nom

```
private String titre;

void titre( String t ) { titre = t; }

String titre() { return titre; }
```

Question : Est-ce possible en `Ruby` ?

Oui… d'une façon un peu «particulière» !

```
def titre( titre = nil )
  return @titre if titre.nil? # Getter

  @titre = titre # Setter
  self
end
```

```
doc.titre( "Pragmatic Ruby" ) # Setter
doc.titre                     # Getter
```

3.4 Deux API coulantes en Java pour décrire des `Document`s

Certaines constructions `Java` permettent de définir des API coulantes semblables à celles en Ruby

Les lambda-expressions

Construction introduite en Java 8.0 (2014)

Les classes internes anonymes et les *initializers*

Constructions pas nouvelles… mais peu connues

A. API coulante avec lambda-expression

A. API coulante avec lambda-expression : Utilisation

Donc, semblable au style utilisé avec `gli`

```
new Document( d -> {
        d.titre( "Domain-Specific Languages" );
        d.auteurs( "M. Fowler", "R. Parsons" );
        d.editeur( "Pearsons Education" );
        d.annee( 2011 );
    } )
```

A. API coulante avec lambda-expression : Mise en œuvre


```
interface DocumentBuilder { void build(Document d); }


class Document {
  private String titre, editeur
  private String[] auteurs;
  private int annee;

  Document(  DocumentBuilder b ) {
    b.build( this );
    assert estValide();
  }

  ...
```

A. API coulante avec lambda-expression : Mise en œuvre (suite)


```
// Setters
protected void titre( String t ) { titre = t; }

protected void auteurs( String... auts ) {
  auteurs = auts.clone();
}


...
// Getters
public String titre() {
  return titre;
}

public String[] auteurs() {
  return auteurs;
}
...
```

A. API coulante avec lambda-expression

Utilisation

```
new Document( d -> {
        d.titre( "Domain-Specific Languages" );
        d.auteurs( "M. Fowler", "R. Parsons" );
        d.editeur( "Pearsons Education" );
        d.annee( 2011 );
    } )
```

Mise en œuvre du constructeur `Document`

```
Document( DocumentBuilder b ) {
  b.build( this );
}
```

Donc : le constructeur `Document` appelle la lambda-expression avec l'objet nouvellement alloué comme argument.

B. API coulante avec classe interne anonyme et *initializer*

B. API coulante avec classe interne anonyme et *initializer* : Utilisation

Utilisation

```
new Document() {{
      titre( "Domain-Specific Languages" );
      auteurs( "M. Fowler", "R. Parsons" );
      editeur( "Pearsons Education" );
      annee( 2011 );
    }}
```

B. API coulante avec classe interne anonyme et *initializer* : Mise en œuvre


```
class Document {
  private String titre, editeur;
  private String[] auteurs;
  private int annee;

  Document() {}

  // Setters
  protected void titre( String t ) { titre = t; }
  protected void auteurs( String... auts ) {
    auteurs = auts.clone();
  }
  ...

  // Getters
  public String titre() { return titre; }
  public String[] auteurs() { return auteurs; }
  ...
```

B. API coulante avec classe interne anonyme et *initializer*

Utilisation

```
new Document() {{
      titre( "Domain-Specific Languages" );
      auteurs( "M. Fowler", "R. Parsons" );
      editeur( "Pearsons Education" );
      annee( 2011 );
    }}
```

Mise en œuvre du constructeur `Document` (sic)

```
Document() {}
```

B. API coulante avec classe interne anonyme et *initializer*

Utilisation

```
new Document() {{
      titre( "Domain-Specific Languages" );
      auteurs( "M. Fowler", "R. Parsons" );
      editeur( "Pearsons Education" );
      annee( 2011 );
    }}
```

- Premier niveau de bloc : Classe interne anonyme
- Deuxième niveau de bloc : *instance initializer*

Une classe interne anonyme associée à l'objet peut redéfinir des méthodes de l'objet


```
class Foo {
  Foo() {}

  protected int val;
  public int val() { return val; }
  public void setVal( int v ) { val = v; }
}
---
Foo f = new Foo() {
   public void setVal( int v ) { val = 1000 * v; }
};

f.setVal( 12 );
System.out.println( f.val() ); // =>   ?
```

Une classe interne anonyme associée à l'objet peut redéfinir des méthodes de l'objet


```
class Foo {
  Foo() {}

  protected int val;
  public int val() { return val; }
  public void setVal( int v ) { val = v; }
}
---
Foo f = new Foo() {
   public void setVal( int v ) { val = 1000 * v; }
};

f.setVal( 12 );
System.out.println( f.val() ); // => 12000
```

Donc, `setVal` dans l'appel à `new Foo()` remplace (cache) l'autre !

L'initialisation d'un nouvel objet peut se faire dans un bloc d'initialisation d'instance


```
class Foo {
  Foo() {}

  protected int val;
  public int val() { return val; }
  public void setVal( int v ) { val = v; }

  { val = 999; }
}
---
Foo f = new Foo();

System.out.println( f.val() ); // => ?
```

L'initialisation d'un nouvel objet peut se faire dans un bloc d'initialisation d'instance


```
class Foo {
  Foo() {}

  protected int val;
  public int val() { return val; }
  public void setVal( int v ) { val = v; }

  { val = 999; }  // Initialisation d'instance.
}
---
Foo f = new Foo();

System.out.println( f.val() ); // => 999
```

4. Conclusion

Ruby est un langage puissant et expressif

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

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

Ruby facilite la définition de DSLs internes expressifs

Syntaxe flexible

Argument par mots-clés

Blocs et fermetures

Métaprogrammation (envoi/réception dynamique)

Mais on peut définir des DSLs internes dans n'importe quel langage… même en Java

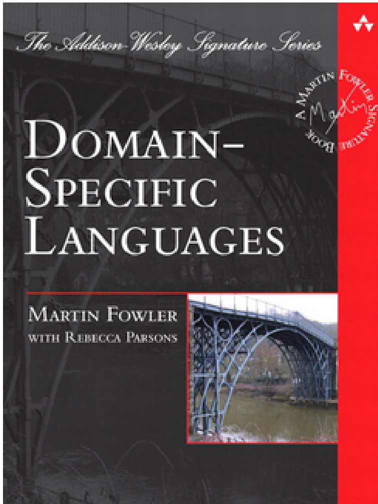
Constructeurs et chainage de méthodes

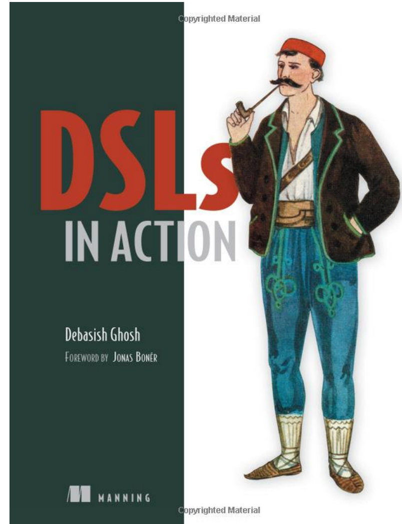
Lambda-expressions

Annotations

Static initializers

Pour des informations additionnelles sur les DSLs





Pour des informations additionnelles sur les DSLs

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

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.