L'outil de correction Oto 2 : Fonctionnalités et mise en oeuvre

Guy Tremblay et Paul Lessard

Université du Québec à Montréal (UQAM)
http://www.info2.uqam.ca/~tremblay

LATECE
(LAbo. sur les TEchnologies du Commerce Electronique)
http://www.latece.uqam.ca

Décembre 2010

Aperçu

Pourquoi un outil d'aide à la correction des travaux de programmation?

Aperçu des fonctionnalités : historique et évolution

Les langages spécifiques au domaine (DSL)

Mise en oeuvre du langage de script d'Oto 2

Conclusion

Aperçu

Quelques constats (1999!)

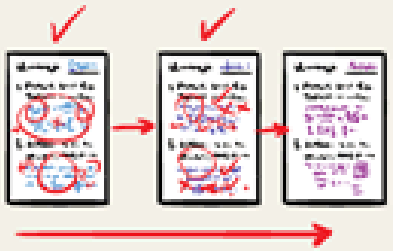
GRADING METHODS

METHOD 1

GRADE ONE PROBLEM AT A TIME FOR ALL THE PAPERS.

METHOD 2

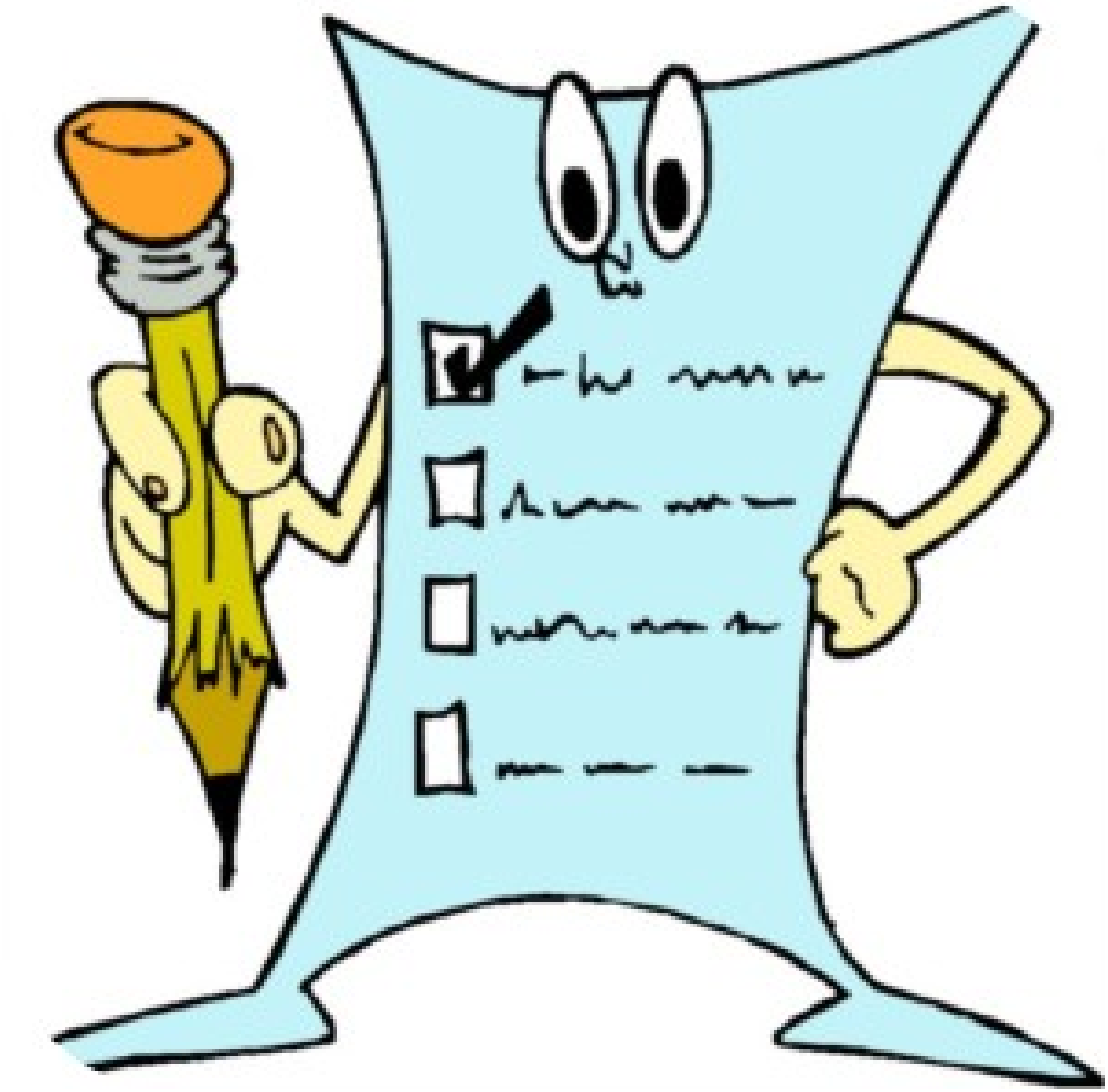
GRADE ALL THE PROBLEMS IN ONE PAPER BEFORE MOVING TO THE NEXT.



METHOD 3

START WITH METHOD 1, DESPAIR OVER HOW LONG IT TOOK TO GRADE THE FIRST PROBLEM, FORGET CONSISTENCY, SWITCH TO METHOD 2.


WORLD'S
DEEPEST
HOLE




fotolia
fotolia
fotolia


fotolia
fotolia
fotolia
fotolia
fotolia

Aperçu

Aperçu

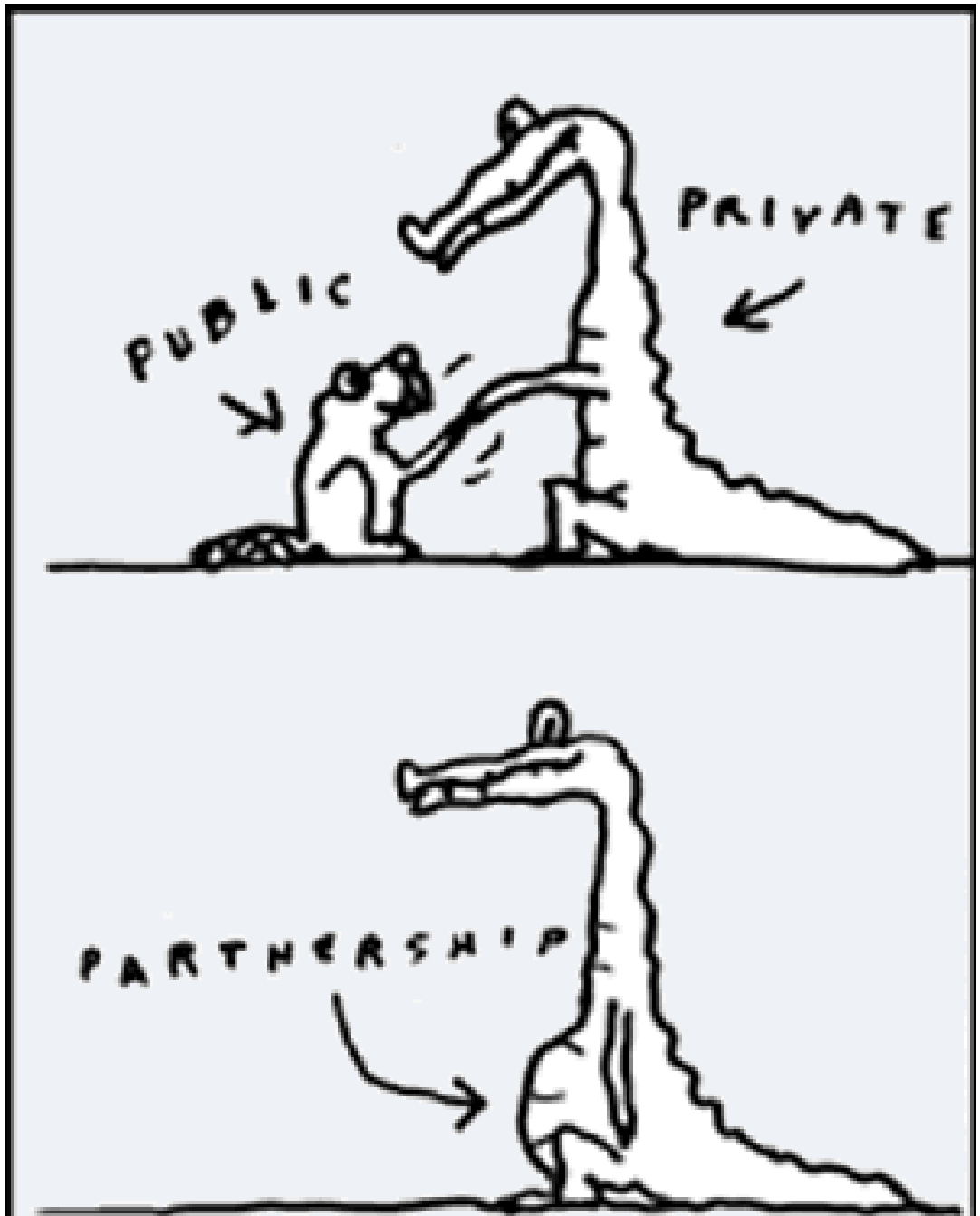
OCETJ

Éric Labonté (M.Sc. info., 2002) : “Outil de correction semi-automatique de programmes Java”.

Outil de Correction et d’Évaluation de Travaux Java

- Objectifs clés :
  - Feedback avant la remise
  - Approche objet $\Rightarrow$ Tests de classes et méthodes
- Fonctionnalités :
  - Vérification préliminaire des travaux
  - Remise électronique des travaux
  - Tests (JUnit) des travaux

Utilisation de JUnit



Faiblesses d'OCETJ

Processus fixe de correction :

1. Compilation avec `javac`
2. Correction avec `junit`

Cégep du
Vieux Montréal

Aperçu

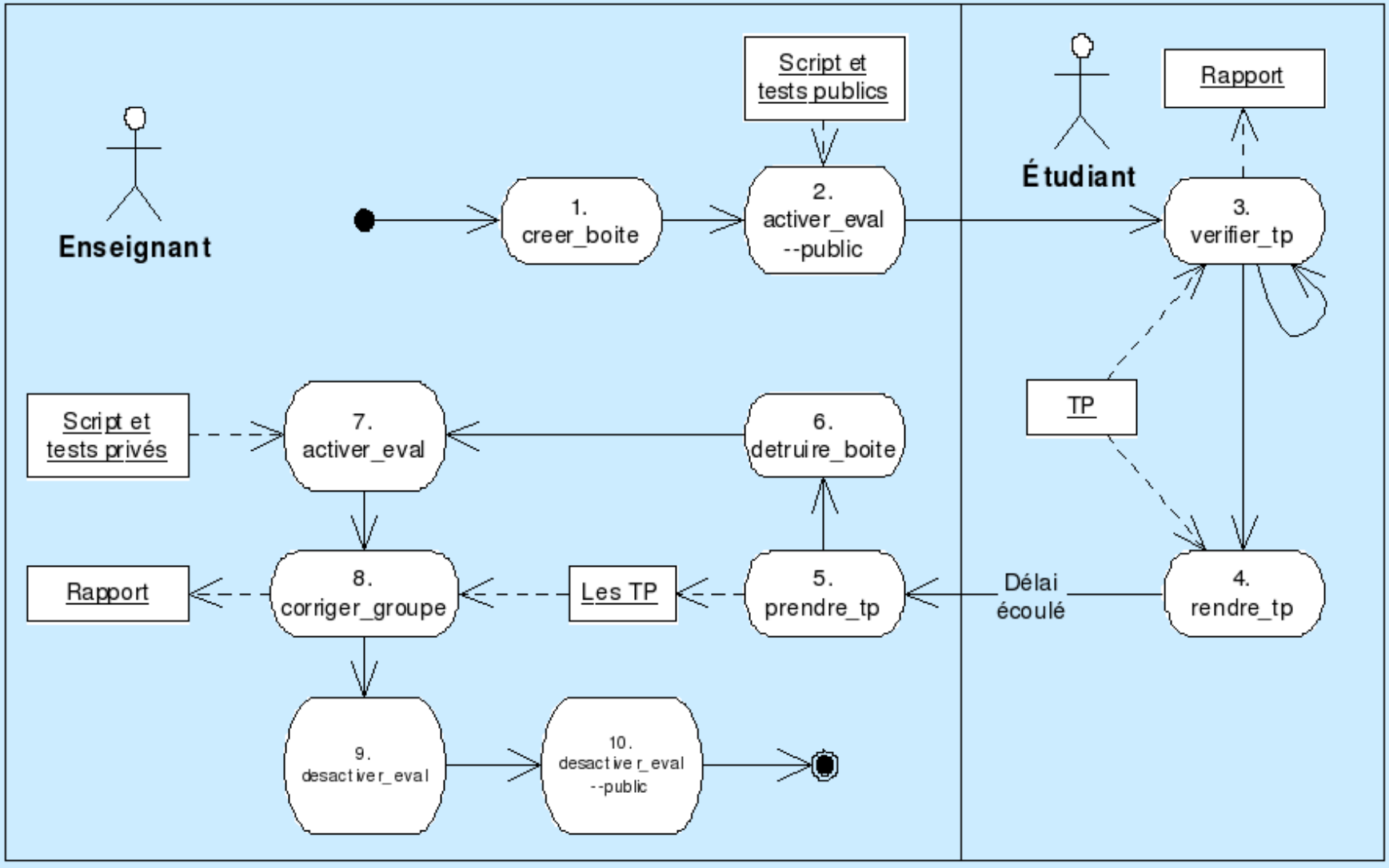
Oto

Frédéric Guérin (M.Sc. info., 2005) : “Oto, un outil générique et extensible pour corriger les travaux de programmation”.

Caractéristiques :

- Correction “sur mesure”
- Indépendant du langage
- Extensible
- Deux mécanismes de configuration et extension :
  - Scripts de correction
  - Modules d’extension

Utilisation typique d'Oto



Script Oto

```
lab         =  Lab7
pgm         =? ${lab}.java
classeTest  =  Test${lab}

compile :: javac { fichier = $pgm }

test    :: junit { classe = $classeTest }

penalite = $( 100 / $test.nbtests )

<<Note finale (sur 100)>>
note = $( 100 - $penalite * $test.nberreurs  )

sortir { note }
```

Modules d'extension

- Tâche dans script $\Rightarrow$ module d'extension

Modules d'extensions (suite)

- Frédéric Guérin (2005) :
  - `javac`
  - `junit`
- Guy Tremblay (2006) :
  - `tester_filtre`
  - `compiler_gcc`
  - `bash`
- Paul Lessard (2008) :
  - `plagiat`

Modules d'extensions (suite)

- Frédéric Guérin (2005) :
  - `compiler_javac`
  - `tester_junit`
- Guy Tremblay (2006) :
  - `tester_filtre`
  - `compiler_c`
- Paul Lessard (2008) :
  - `detecter_plagiat`
  - `produire_statistiques`
- Guy Tremblay (2010) :
  - `tester_pep8`
  - `tester_hunit`
  - `envoyer_rapports_courriel`

Faiblesses d'Oto

- Langage de script difficile à utiliser et peu expressif
- Correction d'un travail individuel à la fois
  $\Rightarrow$ détection de plagiat difficile (*hack*)

Aperçu

Interface personne–machine : Noyau initial

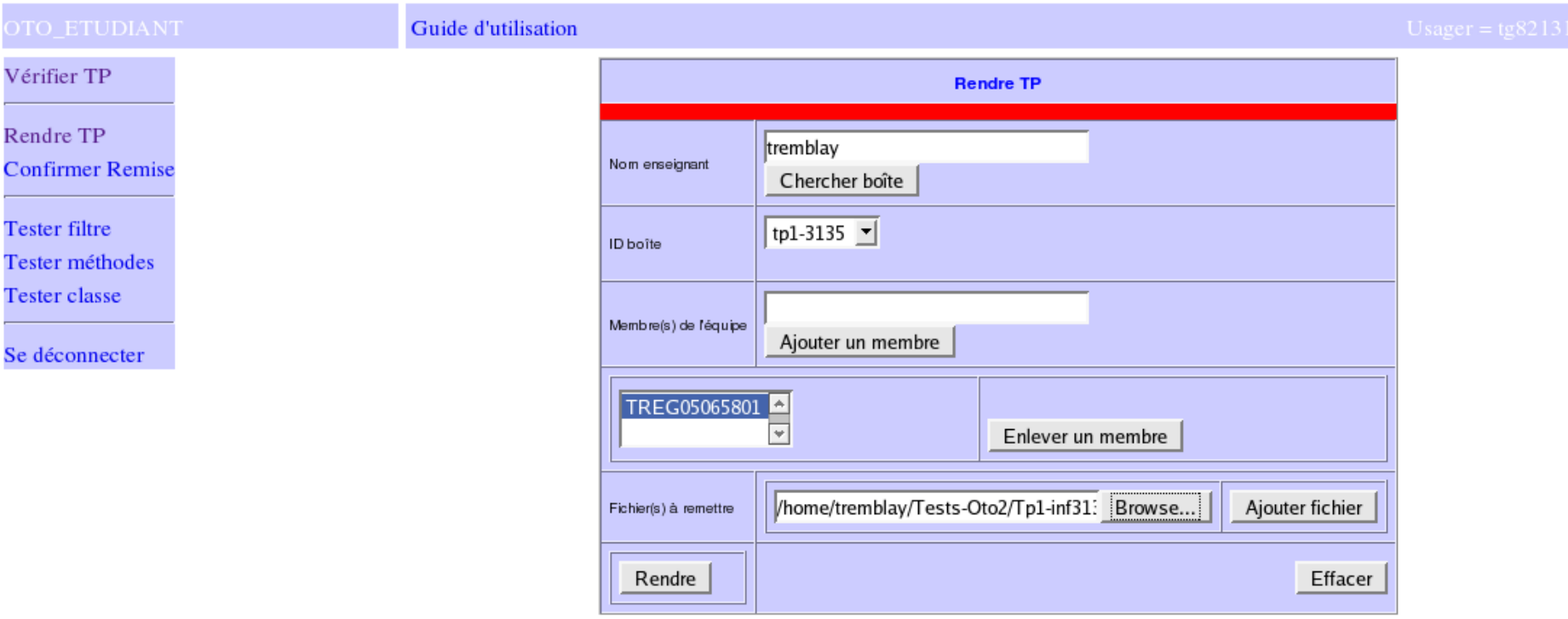
```
$ oto lister_boites tremblay_gu
TP2-INF1120
TP1-INF3135

$ oto rendre_tp tremblay_gu TP1-INF3135 TREG05065801 tp1.c

$ oto confirmer_remise tremblay_gu TP1-INF3135 TREG05065801
-- Remises effectuees dans la boite 'TP1-INF3135' de 'tremblay_
   tremblay_gu+2010.10.30.11.50.43.514109+TREG05065801:
      tp1.c 7641 Sat Oct 30 11:50:43 -0400 2010
   tremblay_gu+2010.10.30.11.51.19.191582+TREG05065801:
      tp1.c 7641 Sat Oct 30 11:51:19 -0400 2010
```

Interface personne–machine : Web

Mohamed Takim (M.Ing., génie logiciel, 2007) : “Applications Web pour l’utilisation des services de l’outil Oto”.



Aperçu

Oto 2

Paul Lessard (M.Sc. info., 2010) : “Un langage spécifique au domaine pour l’outil de correction de travaux de programmation Oto”.

Caractéristiques clés :

- Commandes :
  Inchangées
- Langage de script :
  Complètement modifié!
  $\Rightarrow$ DSL interne fragmentaire Ruby

Illustration avec script simple

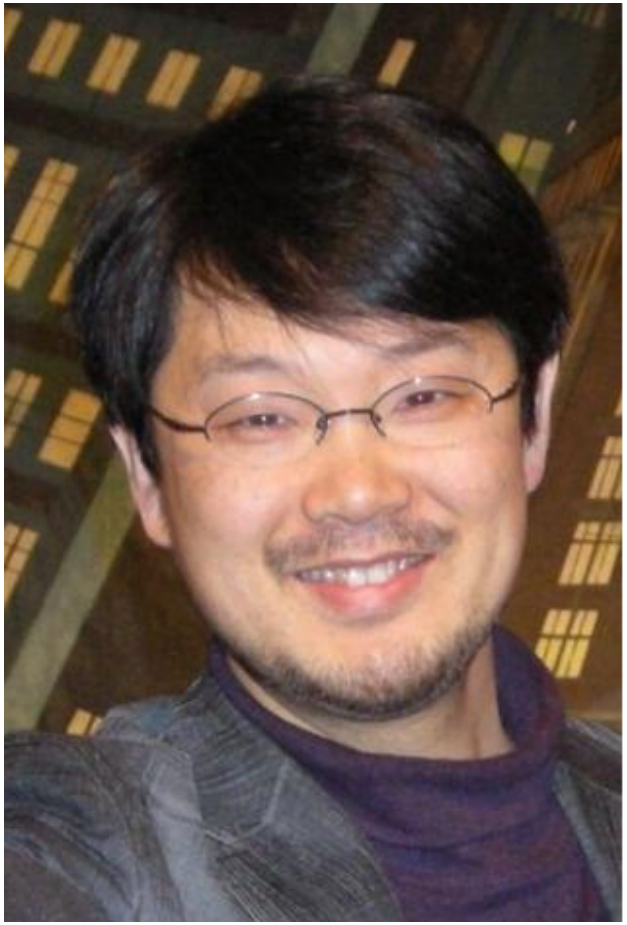


Illustration avec script simple


```
groupe.each do |tp|
  comp = compiler_javac( tp ) {
    :fichier >> "Compte.java"
  }

  assurer( comp.reussi?, comp.message_erreur_echoue )

  tests = tester_junit( tp ) {
    :classe >> "TestCompte"
  }

  tp["Nombre d'erreurs"] = tests[:nberreurs]

  tp["Details de l'execution"] = tests[:detail]
end
```

Rapport de correction

```
TRAVAIL:   tremblay_gu+2010.11.01.09.22.52.339050+TREG05065801

Equipe:      TREG05065801
Depot:       2010-11-01 a 09:22
Deposeur:    tremblay_gu
Nom:         Guy Tremblay
Courriel:    tremblay.guy@uqam.ca

RESULTATS:

  Nombre d'erreurs:
      0

  Details de l'execution:
      ...
      Time: 0,002

      OK (3 tests)
```

Aperçu

Aperçu

Problème : Description XML de documents

```
<document sorte="Livre">
  <titre>Domain-Specific Languages</titre>
  <auteurs>
     <auteur>M. Fowler</auteur>
     <auteur>R. Parsons</auteur>
  </auteurs>
  <editeur>Pearsons Education, Inc.</editeur>
  <annnee>2011</annee>
  <ISBN>978-9-321-71294-3</ISBN>
</document>
```

Problème : Description XML de documents

```
document sorte="Livre"
  titre Domain-Specific Languages
  auteurs
            M. Fowler
            R. Parsons

  editeur Pearsons Education, Inc.
  annnee 2011
  ISBN 978-9-321-71294-3
```

Aperçu

Les langages spécifiques au domaine (DSL)

Les langages spécifiques au domaine (DSL)

Domain-specific language *(noun): a computer programming language of limited expressiveness focused on a particular domain.*

Les langages spécifiques au domaine (DSL)

Domain-specific language *(noun): a computer programming language of limited expressiveness focused on a particular domain.*

- Langage de programmation
- Expressivité limitée
- Lié à un domaine (d'application)

Les deux principaux types de DSL

- *DSL externe*

- *DSL interne*

Les deux principaux types de DSL

- *DSL externe*
  - $\Rightarrow$ langage distinct du langage de l'application

- *DSL interne*
  - $\Rightarrow$ "sous-langage" du langage de l'application

Les deux principaux types de DSL

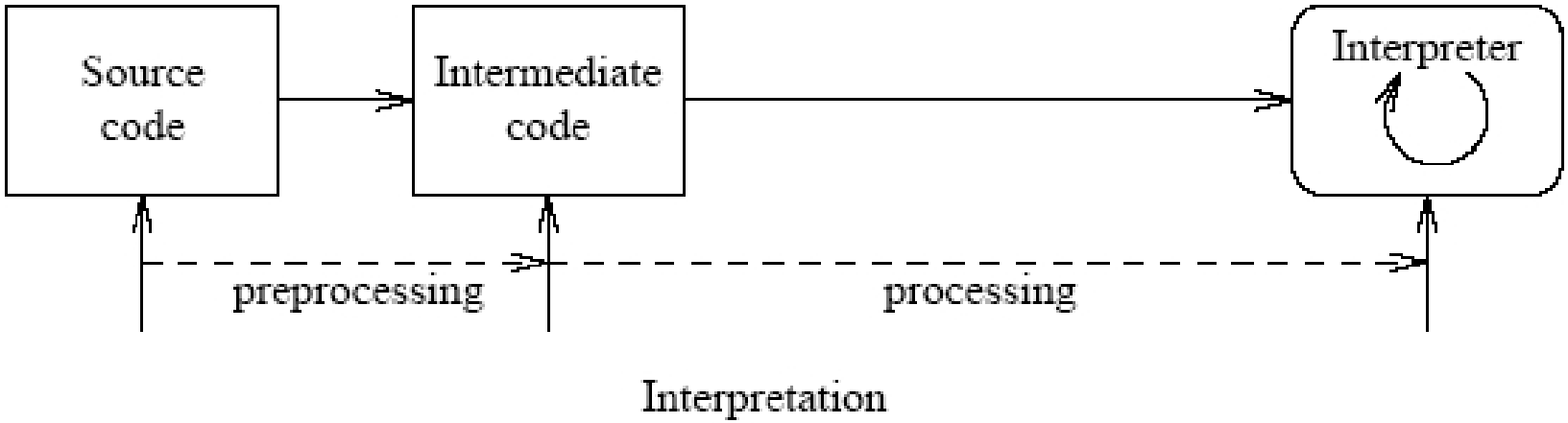
- *DSL externe*
  - $\Rightarrow$ langage distinct du langage de l'application
  - $\Rightarrow$ construction d'un analyseur et interpréteur

- *DSL interne*
  - $\Rightarrow$ "sous-langage" du langage de l'application
  - $\Rightarrow$ définition de méthodes spécifiques au DSL
  - $=$ fluent interface

> *"a fluent interface is a way of implementing an object oriented API in a way that aims to provide for more readable code."*
> *(Evans and Fowler, 2005)*

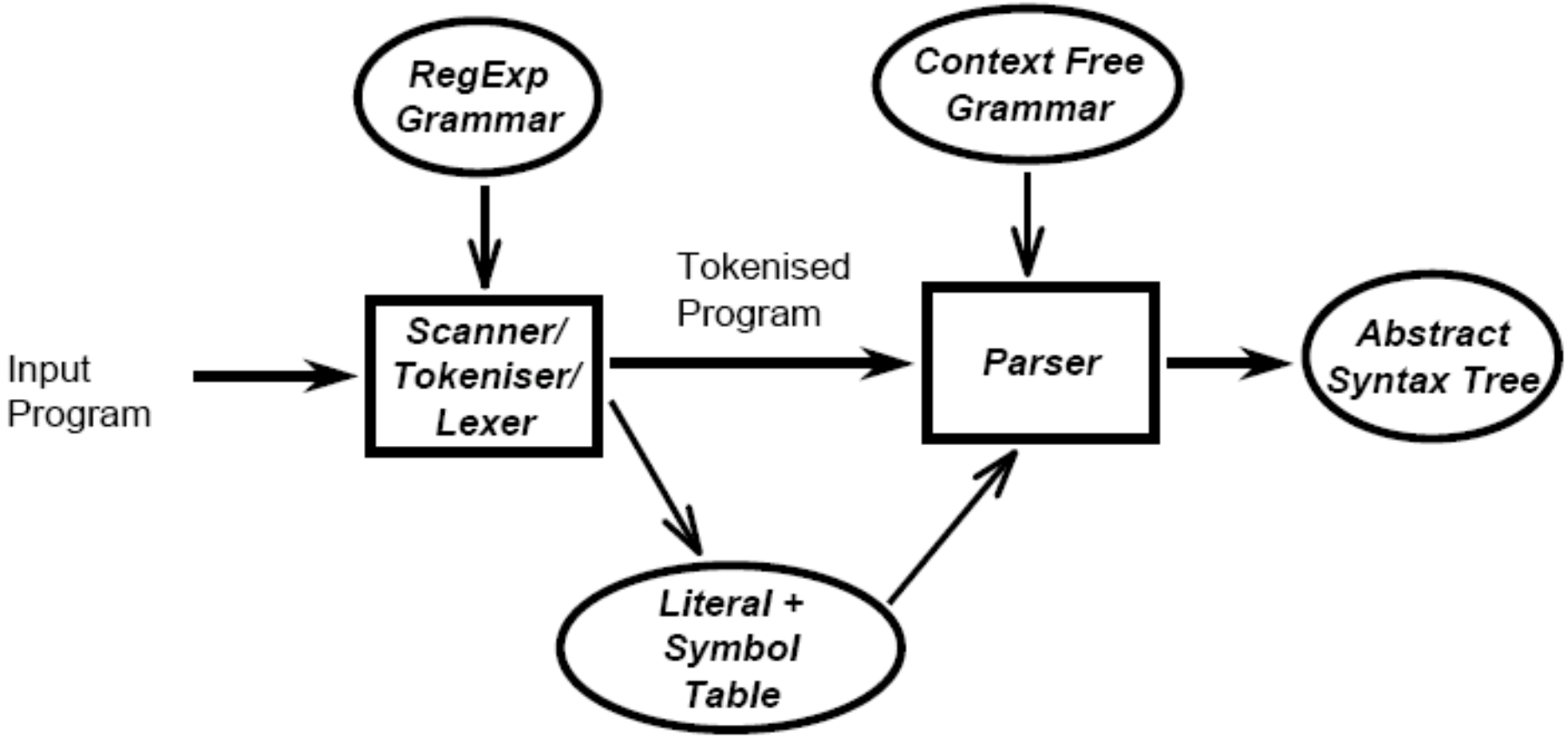
DSL externe

Vue d'ensemble du traitement



DSL externe (suite)

Analyse lexicale et syntaxique



DSL interne

Caractéristiques :

- Défini dans le langage de l'application

= *DSL enchassé* (*embedded DSL*)

$\Rightarrow$ forme/syntaxe contrainte par le langage hôte

- Quelques techniques de mise en oeuvre :
  - Chaînage de méthodes (*method chaining*)
  - Fermetures imbriquées (*nested closures*)
  - Dictionnaires littéraux (*literal collections*)
  - Réception dynamique

Aperçu

Quelques éléments de Ruby I

```
class Cours
  attr_reader :sigle

  def initialize( sigle, titre, *prealables )
    @sigle = sigle
    @titre = titre
    @prealables = prealables
  end

  def to_s
    sigles_prealables = " "
    @prealables.each do |c|
      sigles_prealables += c.sigle.to_s + " "
    end
    "< #{@sigle} '#{@titre}' (#{sigles_prealables}) >"
  end
end
```

Quelques éléments de Ruby I (suite)

```
inf1120 = Cours.new( :INF1120, 'Programmation I' )
inf1130 = Cours.new( :INF1130, 'Maths pour informaticien' )
inf2120 = Cours.new( :INF2120, 'Programmation II', inf1120 )
inf3105 = Cours.new( :INF3105, 'Str. de don.', inf1130, inf212

[inf1120, inf1130, inf2120, inf3105].each do |c|
  puts c
end

--------------------

< INF1120 'Programmation I' ( ) >
< INF1130 'Maths pour informaticien' ( ) >
< INF2120 'Programmation II' ( INF1120 ) >
< INF3105 'Str. de don.' ( INF1130 INF2120 ) >
```

Quelques éléments de Ruby I (suite)

```
inf1120 = Cours.new( :INF1120, 'Programmation I' )
inf1130 = Cours.new( :INF1130, 'Maths pour informaticien' )
inf2120 = Cours.new( :INF2120, 'Programmation II', inf1120 )
inf3105 = Cours.new( :INF3105, 'Str. de don.', inf1130, inf212

[inf1120, inf1130, inf2120, inf3105].each { |c|
  puts c
}

--------------------

< INF1120 'Programmation I' ( ) >
< INF1130 'Maths pour informaticien' ( ) >
< INF2120 'Programmation II' ( INF1120 ) >
< INF3105 'Str. de don.' ( INF1130 INF2120 ) >
```

Quelques éléments de Ruby II

```
inf1120 = Cours.new( :INF1120, 'Programmation I' )

puts inf1120.sigle

puts inf1120.instance_eval( "s" + "ig" + "le" )

puts

inf1120.instance_eval( "@sigle = :inf9999" )

puts inf1120.sigle

-----------------------------

INF1120
INF1120

inf9999
```

Quelques éléments de Ruby III

```
les_mots =  [ "abc", "def", "abc", ]
les_mots += [ "abc", "def", "a", "xxx", "x", "xx", ]


histogramme = Hash.new(0)
les_mots.each do |m|
  histogramme[m] += 1
end


puts histogramme.inspect

----------------------------

{"xx"=>1, "a"=>1, "x"=>1, "xxx"=>1, "abc"=>3, "def"=>2}
```

Quelques éléments de Ruby IV

```
class A
  def m1
    puts "Dans A#m1"
  end
end

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

B.new.m1
B.new.m2
A.new.m2

----------------------------
Dans A#m1
Dans B#method_missing( m2 )
missing.rb:15: undefined method `m2'
   for #<A:0xb7fd2728> (NoMethodError)
```

Aperçu

Trois exemples de DSL pour des documents

A. DSL externe : À la Lisp
B. DSL interne I : Avec dictionnaires et chaînage de méthodes
C. DSL interne II : Avec dictionnaires, fermetures imbriquées et réception dynamique

A. DSL externe (*à la Lisp*) : Utilisation

```
(document sorte: "Livre"
  (titre "Domain-Specific Languages")

  (auteurs
    (auteur "M. Fowler")
    (auteur "R. Parsons"))

  (editeur "Pearsons Education, Inc.")

  (annee 2011)

  (ISBN "978-9-321-71294-3")
)
```

B. DSL interne I : Utilisation

Avec dictionnaires et chaînage de méthodes

```
dsl = Document.new( :sorte => :Livre ).
      titre( "Domain-Specific Languages" ).
      auteur( "M. Fowler" ).
      auteur( "R. Parsons" ).
      isbn( "978-9-321-71294-3" ).
      annee( 2001 )
```

B. DSL interne I : Mise en oeuvre

```
class Document
  def initialize( *args )
    @auteurs = []
    @sorte = *args[0][:sorte]
  end

  def auteur( a )
    @auteurs « a
    self
  end

  ...

  def to_s
    res = "<document sorte=\"#{@sorte}\">\n"
    res += "  <titre> \"#{@titre}\" </titre>\n"
    res += "  <auteurs>\n"
    ...
    res += "</document>\n"
  end
end
```

C. DSL interne II : Utilisation

Avec dictionnaires, fermetures imbriquées et réception dynamique

```
dsl = XMLBuilder.new.document :sorte => :Livre do
  titre { "Domain-Specific Languages" }

  auteurs {
    auteur { "M. Fowler" }
    auteur { "R. Parsons" }
  }

  editeur { "Pearsons Education, Inc." }

  annee { 2011 }

  ISBN { "978-9-321-71294-3" }
end
```

C. DSL interne II : Mise en oeuvre des attributs

```
class XMLBuilder

  def method_missing( sym, *args, &block ) # Version 1
    @value += "<#{sym.to_s}"

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

    @value += ">"
  end

  ...
```

C. DSL interne II : Mise en oeuvre des enfants

```
def method_missing( sym, *args, &block ) # Version 2
  @value += "<#{sym.to_s}"

  if block.nil?
    @value += "/>"     # Pas de d'enfants.
  else
    @value += ">"      # Avec enfants.

    # On genere les enfants.
    builder = XMLBuilder.new
    block_value = builder.instance_eval(&block).to_s
    @value += block_value

    # On genere la balise de fermeture.
    @value += "</#{sym.to_s}>"
  end
end

...
```

Aperçu

Aperçu

Exemple de script

```
groupe.each do |tp|
  comp = compiler_javac( tp ) {
    :fichier >> "Compte.java"
  }

  assurer( comp.reussi?, comp.message_erreur_echoue )

  tests = tester_junit( tp ) {
    :classe >> "TestCompte"
  }

  tp["Nombre d'erreurs"] = tests[:nberreurs]
end

puts produire_rapport_complet( groupe )
```

Appel à `oto` avec le script

```
$ oto script.oto TestCompte.class *.tp_oto
```

Résultats : Premier travail correct

```
TRAVAIL:   tremblay_gu+2010.11.01.09.22.52.339050+TREG05065801

Equipe:      TREG05065801
Depot:       2010-11-01 a 09:22
Deposeur:    tremblay_gu
Nom:         Guy Tremblay
Courriel:    tremblay.guy@uqam.ca

RESULTATS:

  Nombre d'erreurs:
      0
```

Résultats : Deuxième travail incorrect

```
TRAVAIL:   lessard_p+2010.11.01.09.22.52.339050+TREG05065801.tj

Equipe:      LESP07068001
Depot:       2010-11-01 a 09:22
Deposeur:    lessard_p
Nom:         Paul Lessard
Courriel:    lessard.paul@courrier.uqam.ca


RESULTATS:

  Assertion non satisfaite:
      Erreur de compilation : Compte.java:3: <identifier> expe
        rivate int bal;
              ^
      Compte.java:3: <identifier> expected
        rivate int bal;
                    ^
      2 errors
```

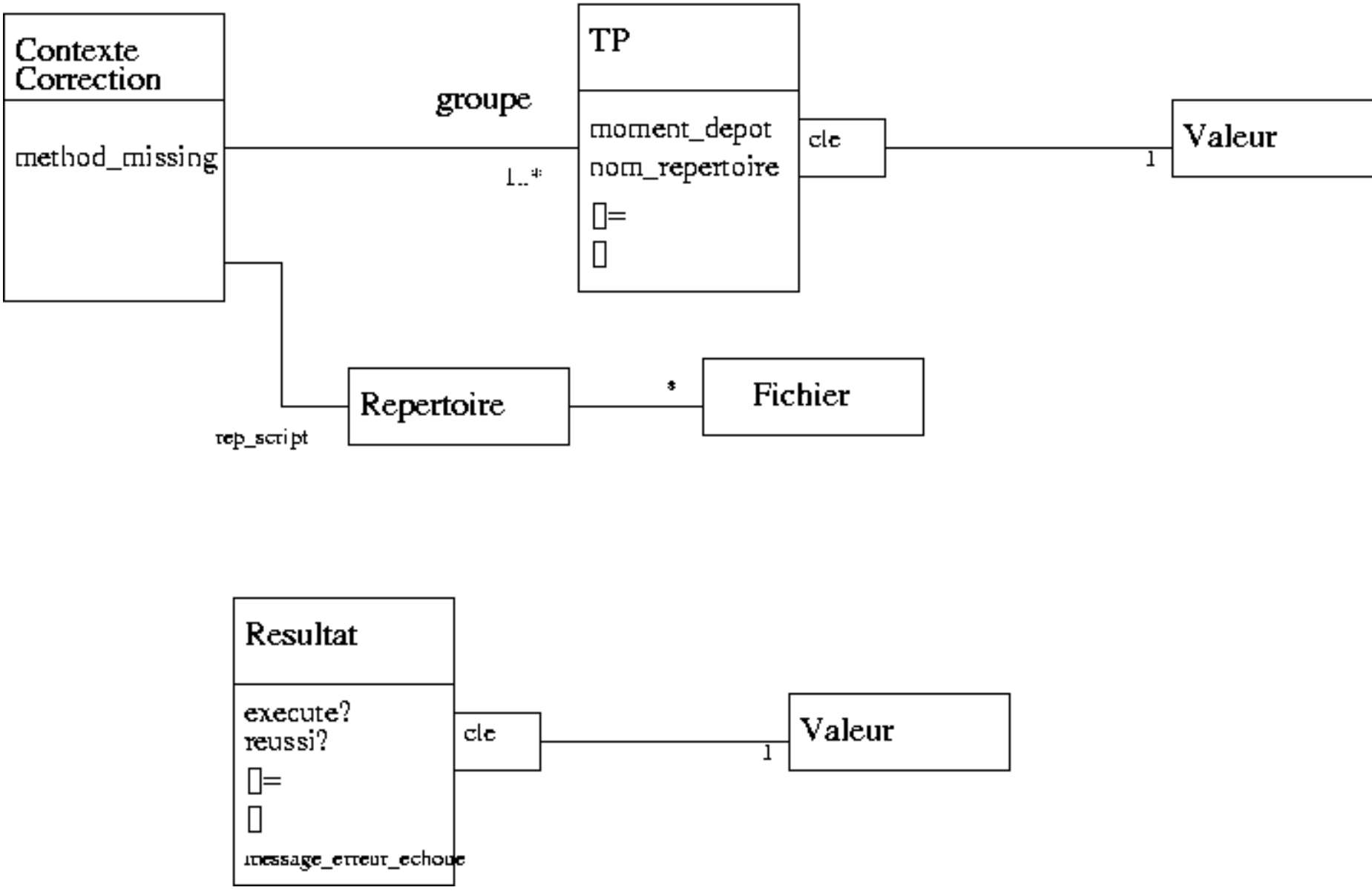
Aperçu

Fragmentary Internal DSL

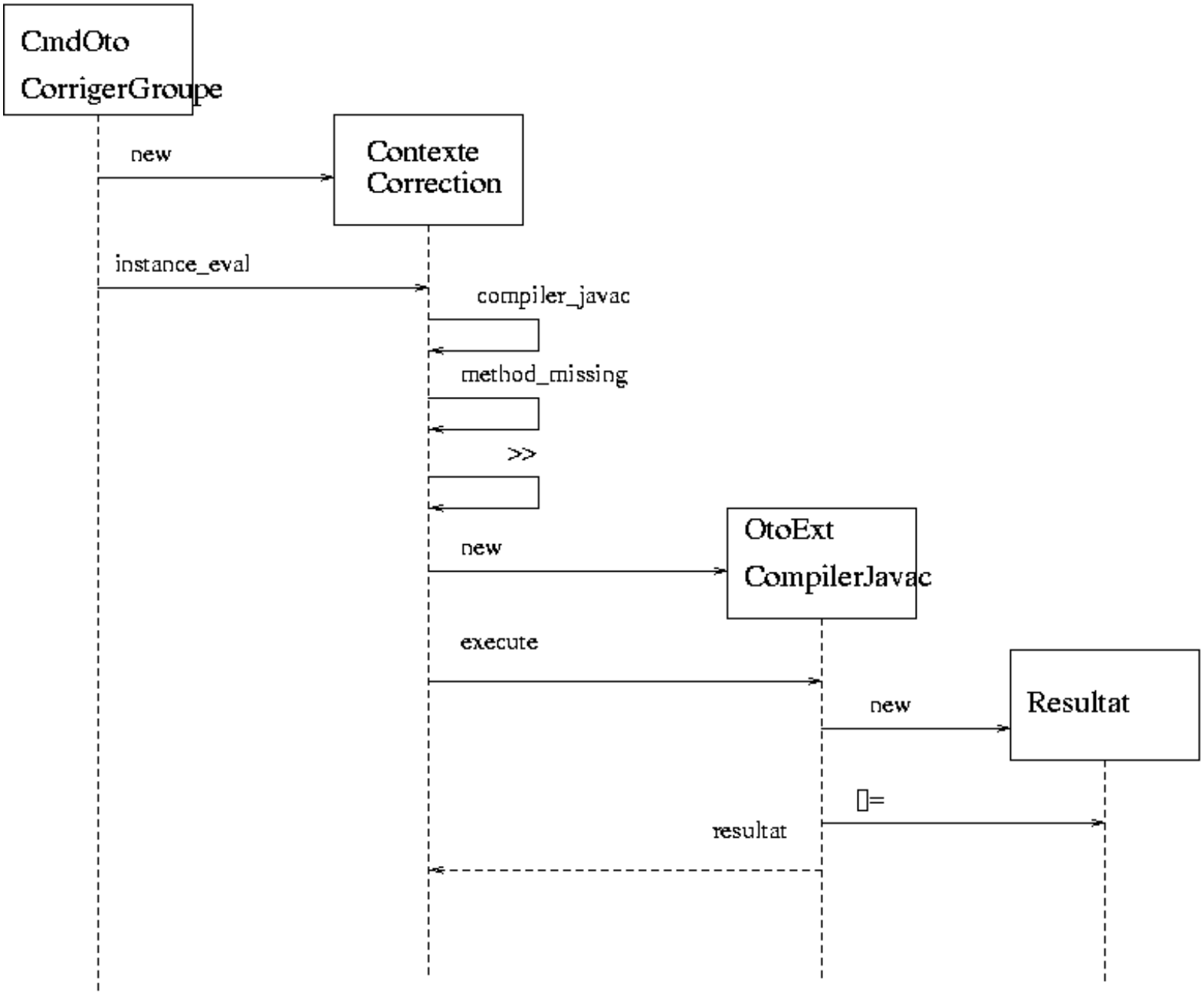
Encore Fowler :

> *Another way DSLs appear is in a fragmentary form. In this form, 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.*

Modèle UML des concepts clés



Aperçu des interactions



Correction d'un groupe de travaux

```
$ oto script.oto TestCompte.class *.tp_oto
```

```
def CmdOtoCorrigerGroupe.run( script_oto, fichiers, tps )
  script ← Lire le script_oto
  rep_script ← Créer un répertoire temporaire de travail
  Copier les fichiers auxiliaires (et le script) dans rep_script
  Copier les tps dans rep_script
  cc ← ContexteCorrection.new( script, fichiers, tps )
  cc.instance_eval( script )
  Faire le ménage
end
```

Exécution du script : Accès aux travaux

```
groupe.each do |tp|
  ...
end
```

Attribut de `ContexteCorrection` :

```
  attr_reader :groupe
      # De type Array => avec méthode each
```

Documentation de `each` :

```
Array#each
  arr.each { |item| block } -> arr

  Calls block once for each element in arr,
    passing that element as a parameter.
```

Exécution du script : Compilation d'un travail

```
comp = compiler_javac( tp ) {
  :fichier >> "Compte.java"
}
```

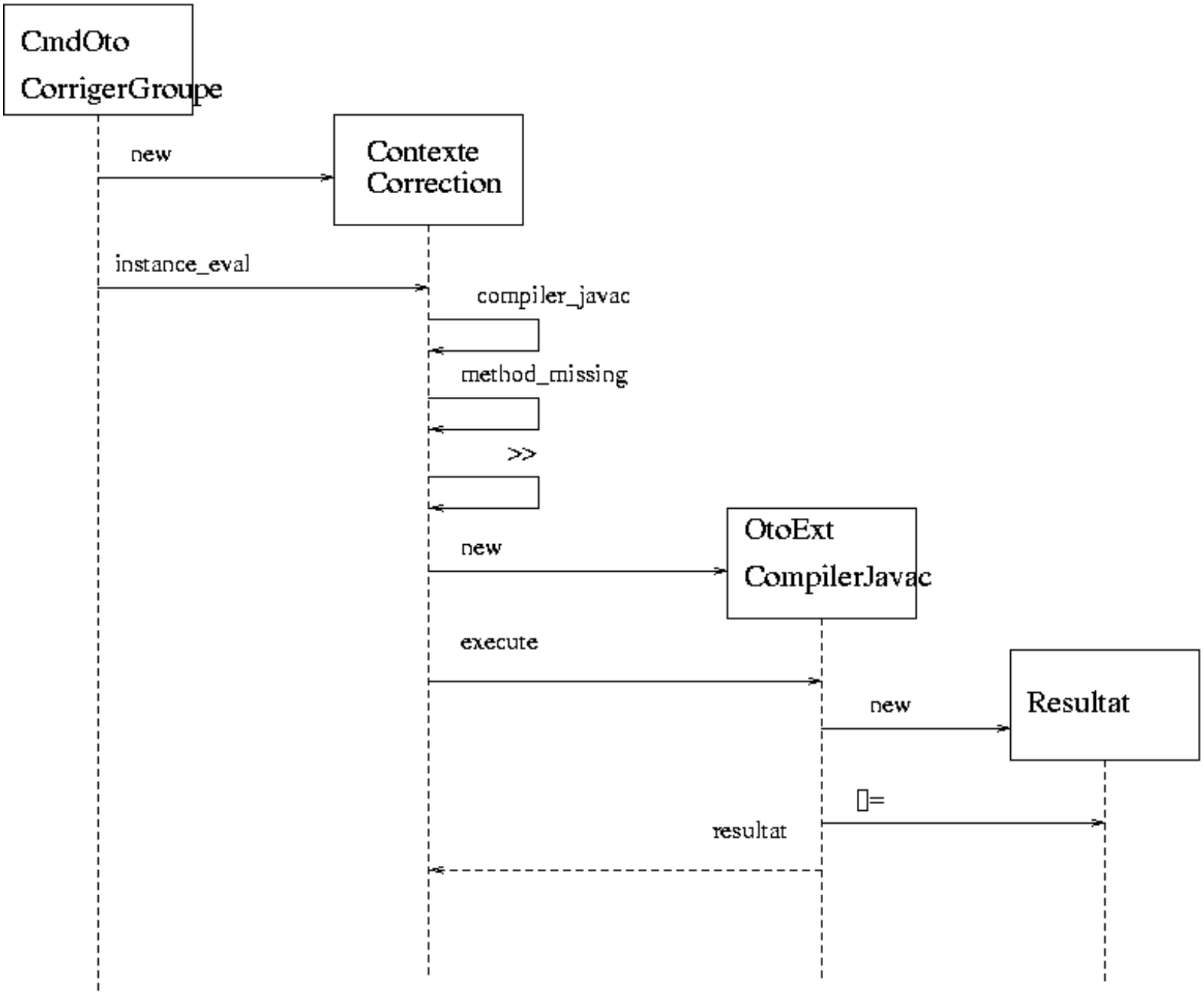
Méthodes de `ContexteCorrection` :

- `compiler_javac` :
  extensibilité via chargement dynamique $\Rightarrow$ `:(`
- `method_missing` :
  `:)`

Appel à `method_missing` :

```
method_missing( :compiler_javac,
                [tp],
                { :fichier >> "Compte.java" } )
# sym = :compiler_javac
# args = [ tp ]
# &block = { :fichier >> "Compte.java" }
```

Aperçu des interactions

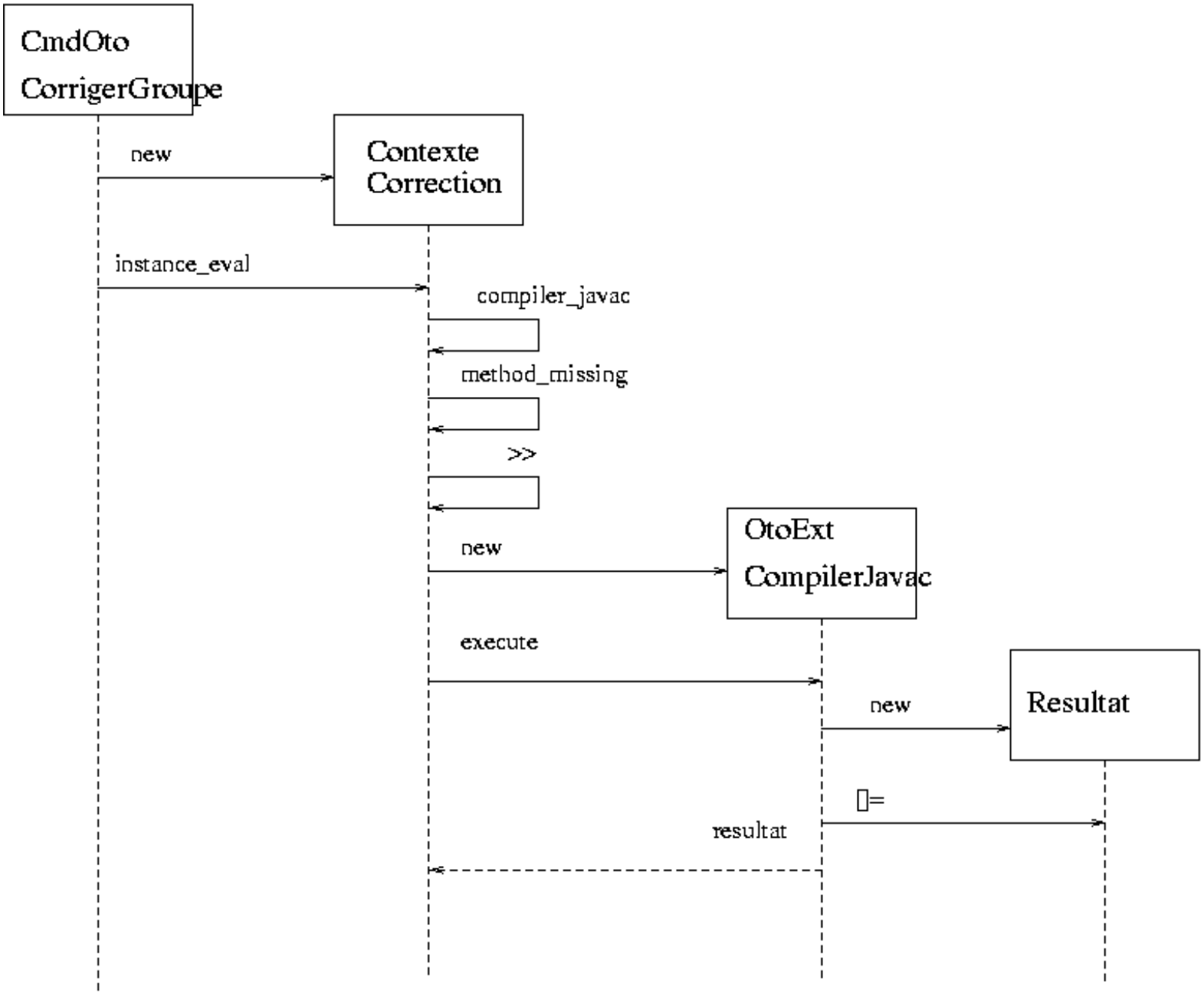


Exécution du script : Compilation (suite)

```
def method_missing( sym, *args, &block )
  SI il existe un module d'extension nommé sym ALORS
    tps ← args[0]
    parametres ← Évaluer les paramètres via block
    ext ← Créer un objet de la classe de l'extension sym
    res ← ext.executer( tp, parametres, self )
    Retourner res
  SINON SI il existe un module de rapport nommé sym ALORS
    ... générer rapport ...
  SINON
    ... interpréter sym comme une commande Unix
  FIN
end
```

```
def OtoExtCompilerJavac.executer( tp, parametres, contexte )
  .
  .
  .
end
```

Aperçu des interactions



Exécution du script : Compilation (suite)

```
def OtoExtCompilerJavac.executer( tp, parametres, contexte )
  Identifier les diverses options via parametres et contexte

  cmd ← "javac #{options} #{tp}"
  stdout, stderr ← Exécuter cmd au niveau du shell

  res ← Créer un objet Resultat approprié

  Analyser stdout/stderr et, entre autres:
      res.reussi? ← aucune erreur fatale rencontrée
      res.message_erreur_echoue ← Information de stderr (si présente)
      res[:detail] ← Information émise sur stdout

  Retourner res
end
```

Exécution du script : Évaluation de l'assertion

```
assurer( comp.reussi?, comp.message_erreur_echoue )
```

Attributs et méthodes de `Resultat` :

```
attr_accessor :message_erreur_echoue

def reussi?
  @reussi
end

def []=( cle, valeur )
  @resultats[cle] = valeur
end
```

Exécution du script : Informations sur travail

```
tp["Nombre d'erreurs"] = tests[:nberreurs]
```

Méthode de `Resultat` :

```
def [](cle)
  @resultats[cle]
end
```

Méthode de `TP` :

```
def []=(cle, valeur)
  @resultats[cle] = valeur
end
```

Aperçu

Conclusion

Le langage de script d'Oto 2.0 est

- beaucoup plus puissant et expressif
- beaucoup plus simple à utiliser

Désavantage (avantage ;) découlant de ce nouveau langage :

- Il faut, minimalement, connaître Ruby

Travaux futurs

- Ajout de nouvelles constructions au langage de script
- Développement de modules pour l'évaluation de la qualité — style, métriques, etc.
- Réécriture des fonctionnalités pour les spécifications simples de tests (*c-shell script*)
- Développement d'une mise en oeuvre REST de l'application Web

Pour plus d'informations :

- `http://oto.uqam.ca`