Le *gem* `gv` : Solution au devoir #2 avec `gli` et avec le patron «*Repository*»

Guy Tremblay
Professeur

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

27 novembre 2018

Contenu

1. Introduction

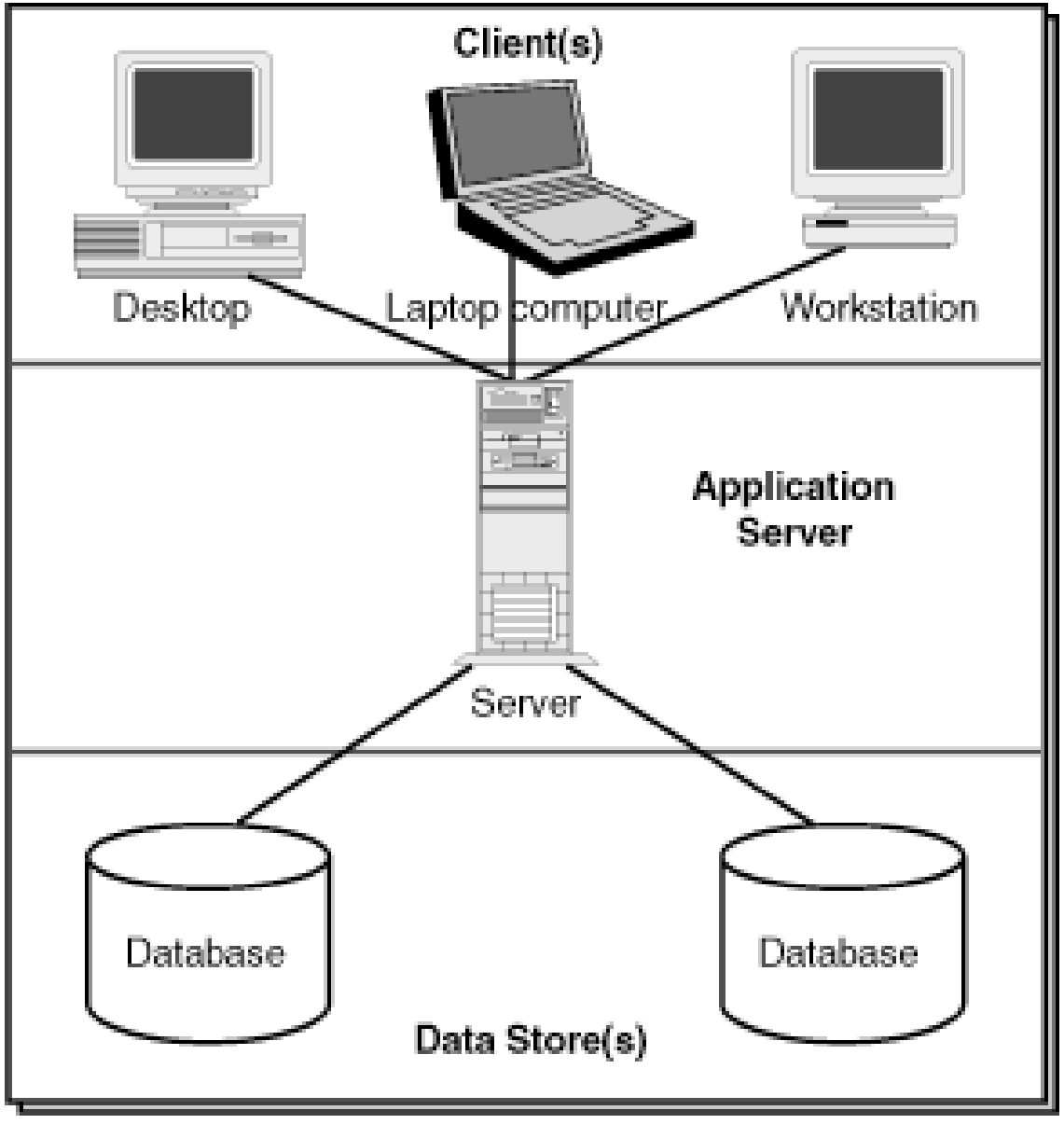
Ces diapositives. . .

- . . . présentent la solution au devoir 2, mais. . .
  - avec le code organisé selon la structure d'un *gem* ;
  - avec une architecture multi-couches ;
  - en utilisant le *gem* `gli` et son DSL pour définir l'interface personne–machine (lignes de commandes) ;

  Note : Pour simplifier la présentation (espace limité sur les diapos), le traitement des erreurs est omis !
- . . . présentent le patron «*Repository*» (entrepôt, pour manipuler des données persistantes) ⇒ pas avec le style fonctionnel !
- . . . présentent la méthode *table-driven* de traitement de cas multiples pour simplifier le code ;
- . . . pourront servir d'inspiration — de modèle, de *template* — pour l'application à développer dans le devoir 3.

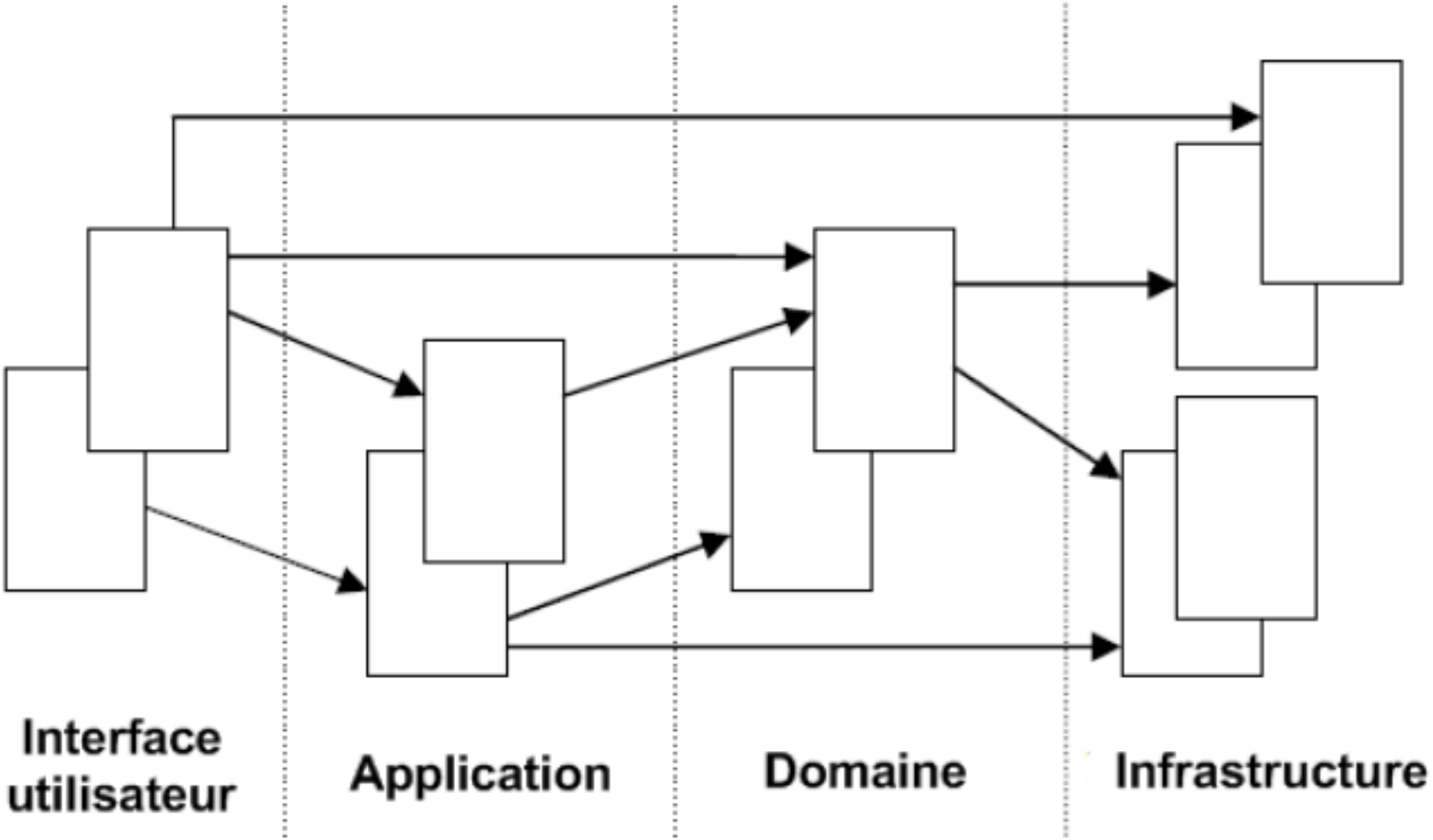
2. Les architectures multi-couches

Architectures multi-couches :
three tier architecture



Architectures multi-couches : *four tier architecture*

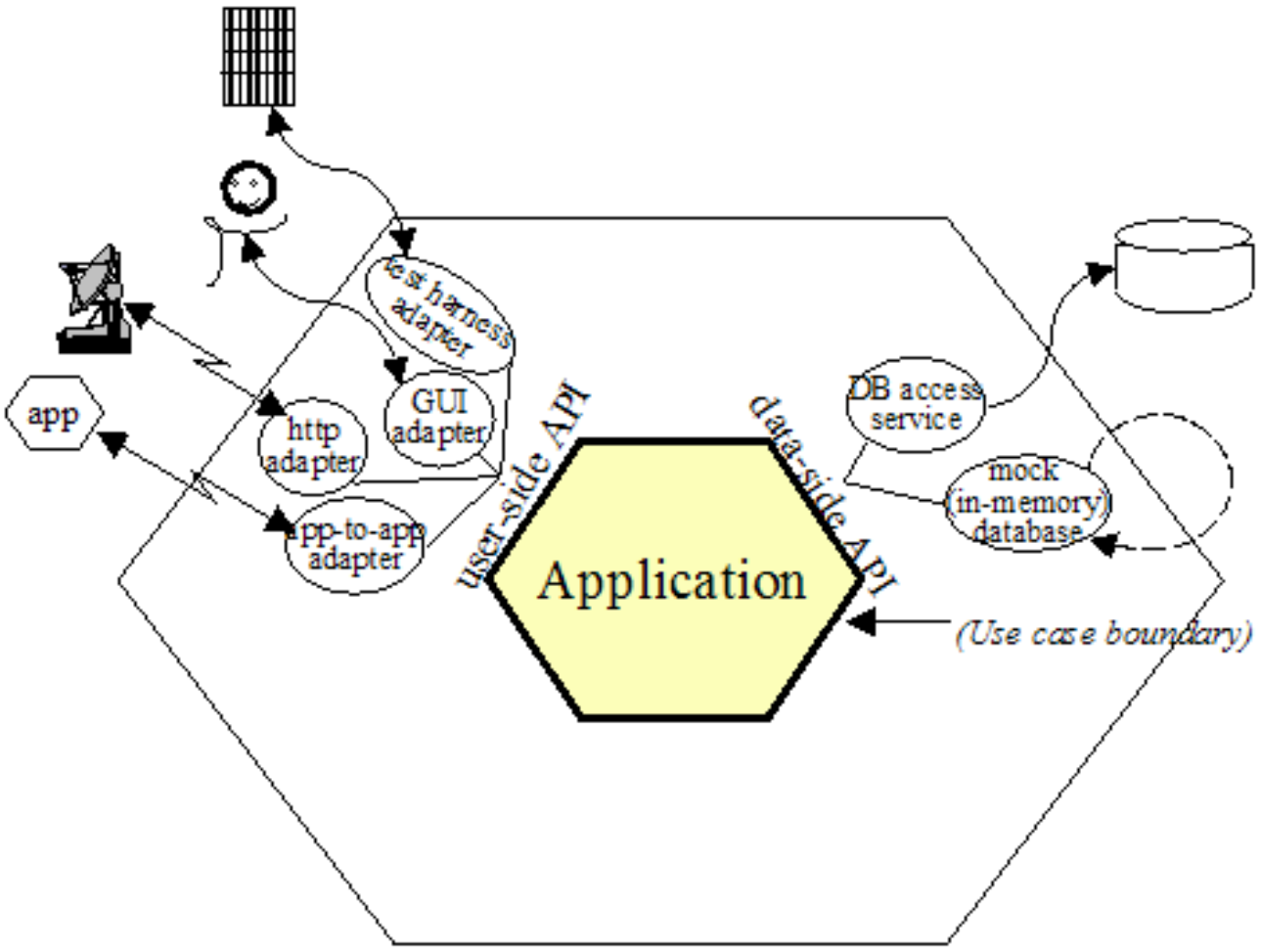
Source: `http://www.infoq.com/fr/minibooks/domain-driven-design-quickly`



Architectures multi-couches : *hexagonal (ports and adapters) architecture*

Source: Cockburn, `http://alistair.cockburn.us/Hexagonal+architecture`

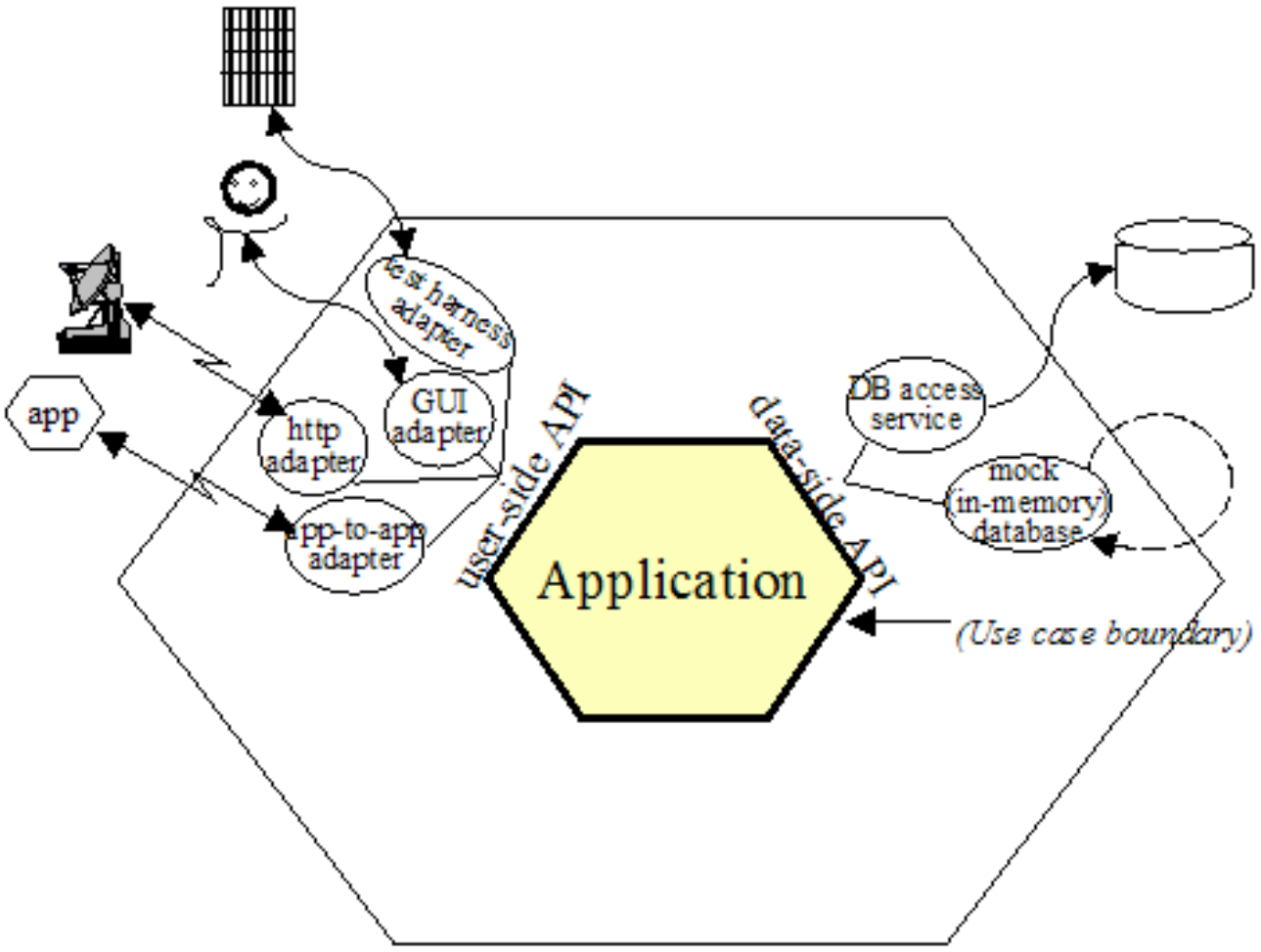
Introduite par A. Cockburn, popularisée par «DDD»



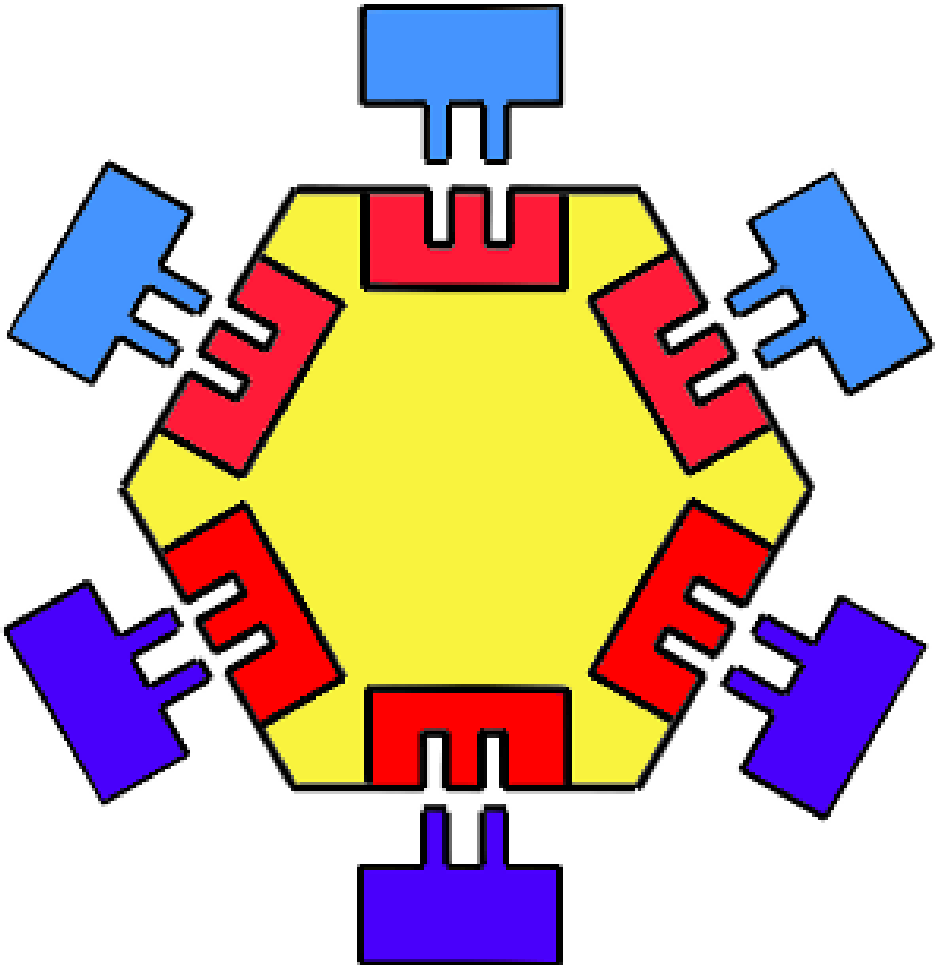
Architectures multi-couches : *hexagonal (ports and adapters) architecture*

Source: Cockburn, `http://alistair.cockburn.us/Hexagonal+architecture`

Avantage = Tester le modèle (l'application) indépendamment des «services» externes.

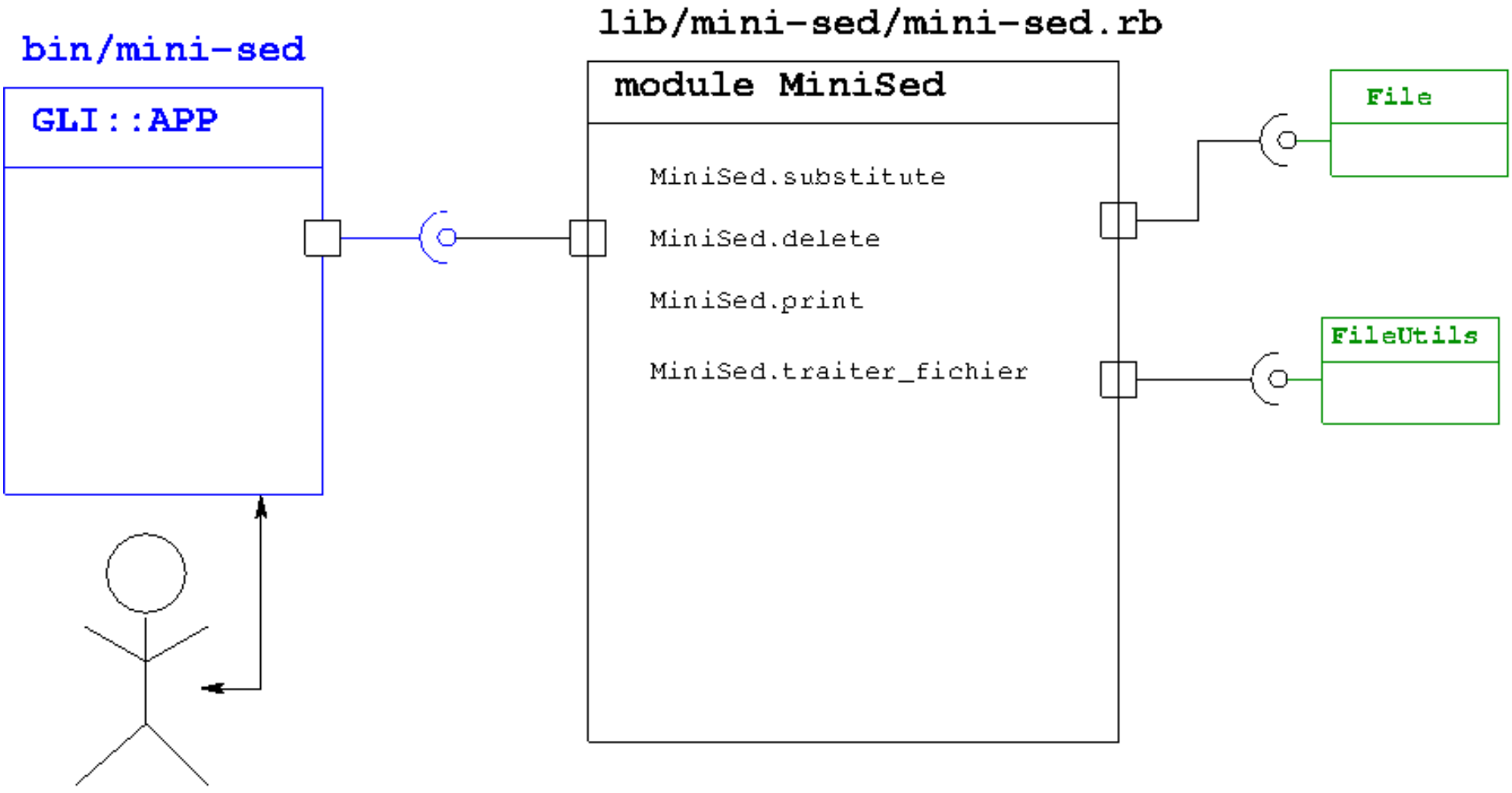


Architectures multi-couches : *hexagonal (ports and adapters) architecture* (bis)



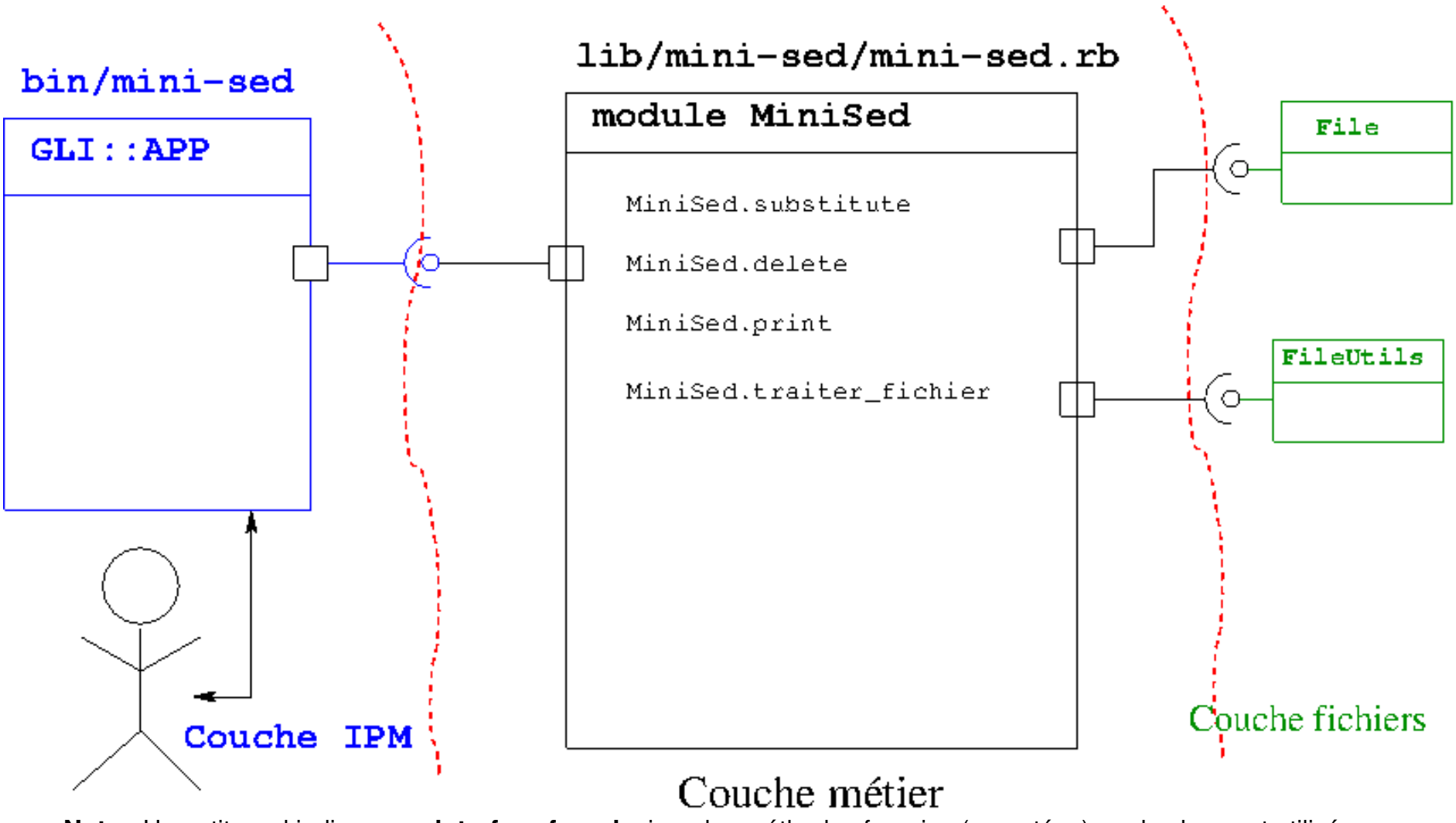
yellow: core logic
light red: primary ports
light blue: primary adapters
dark red: secondary ports
dark blue: secondary adapters

Un exemple simple : L'architecture de `minised`



Note : Un petit rond indique une **interface fournie**, i.e., des méthodes fournies (exportées) par la classe et utilisées par des clients de la classe. Le demi-cercle indique une utilisation d'une interface fournie.

Un exemple simple : L'architecture de `minised`



Note : Un petit rond indique une **interface fournie**, i.e., des méthodes fournies (exportées) par la classe et utilisées par des clients de la classe. Le demi-cercle indique une utilisation d'une interface fournie.

3. Le patron «*Repository*»

Comment peut-on manipuler des données persistantes, mais en faisant abstraction de leur représentation externe (fichiers de texte, BD relationnelle ou autre) ?

Une solution proposée par E. Evans = *Repository*

A Repository represents all objects of a certain type as a conceptual set. It acts like a collection, except with more elaborate querying capability.

Source*: E. Evans, «Domain-Driven Design—Tackling Complexity in the Heart of Software»*

Une solution proposée par E. Evans = *Repository*

A Repository represents all objects of a certain type as a conceptual set. ***It acts like a collection****, except* ***with more elaborate querying capability****.*

Source*: E. Evans, «Domain-Driven Design—Tackling Complexity in the Heart of Software»*

Une solution proposée par E. Evans = *Repository*

A Repository [. . .] [acts] like an in-memory domain object collection.
[. . .]
Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes.

Source*: M. Fowler,* `http://martinfowler.com/eaaCatalog/repository.html`

Un *repository* permet de changer facilement le mécanisme de persistance

Repository

Methods for retrieving domain objects should delegate to a specialized Repository object such that alternative storage implementations may be easily interchanged.

Source: `https://en.wikipedia.org/wiki/Domain-driven_design`

Pourquoi présenter ce patron avec un exemple assez détaillé ?

- Parce que ce sera utile pour comprendre ma solution au devoir #2 qui utilise le *gem* `gli`.

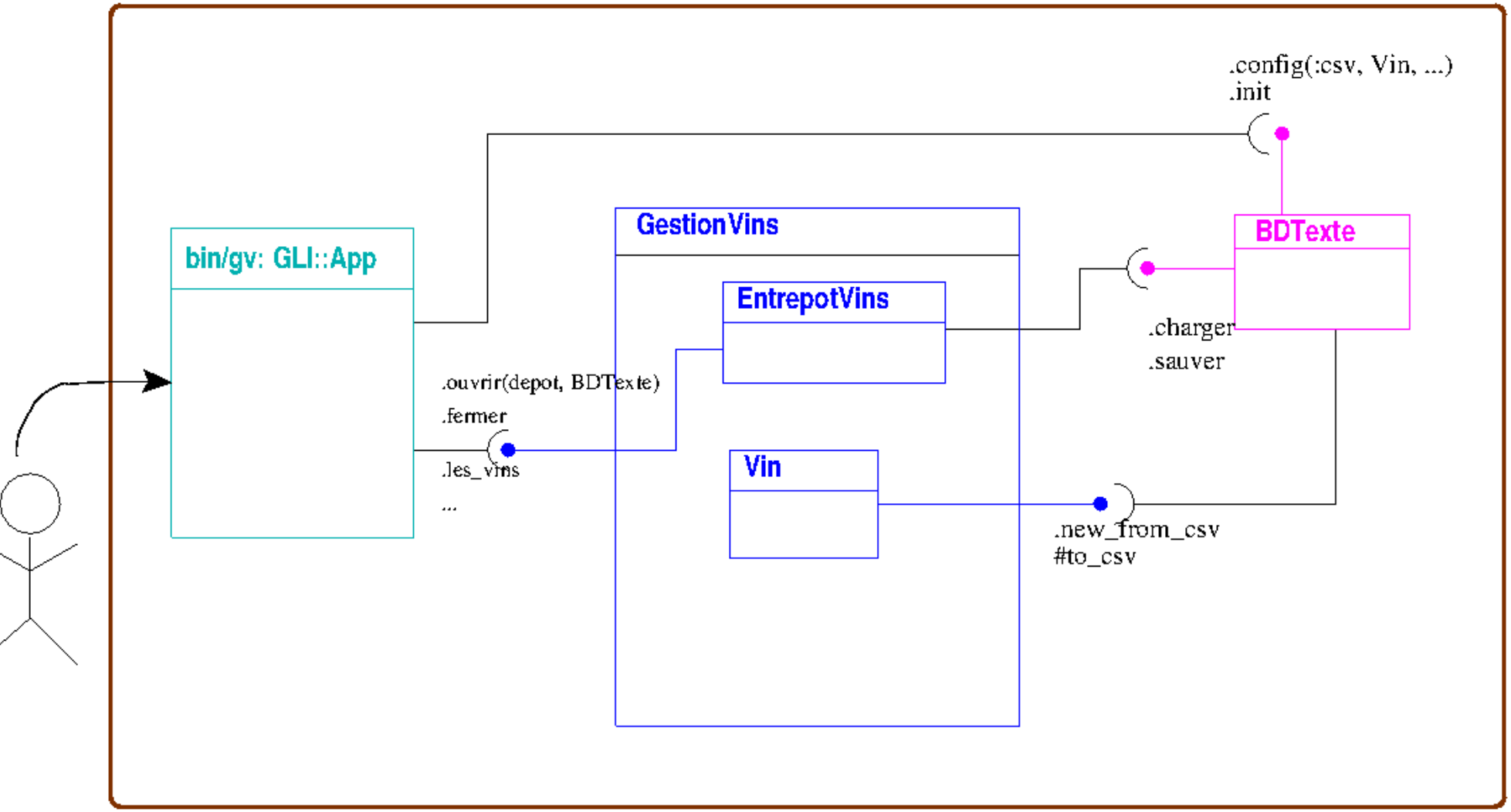
Pourquoi présenter ce patron avec un exemple assez détaillé ?

- Parce que ce sera utile pour comprendre ma solution au devoir #2 qui utilise le *gem* `gli`.

- Parce que cela pourrait vous être utile pour le devoir #3 !

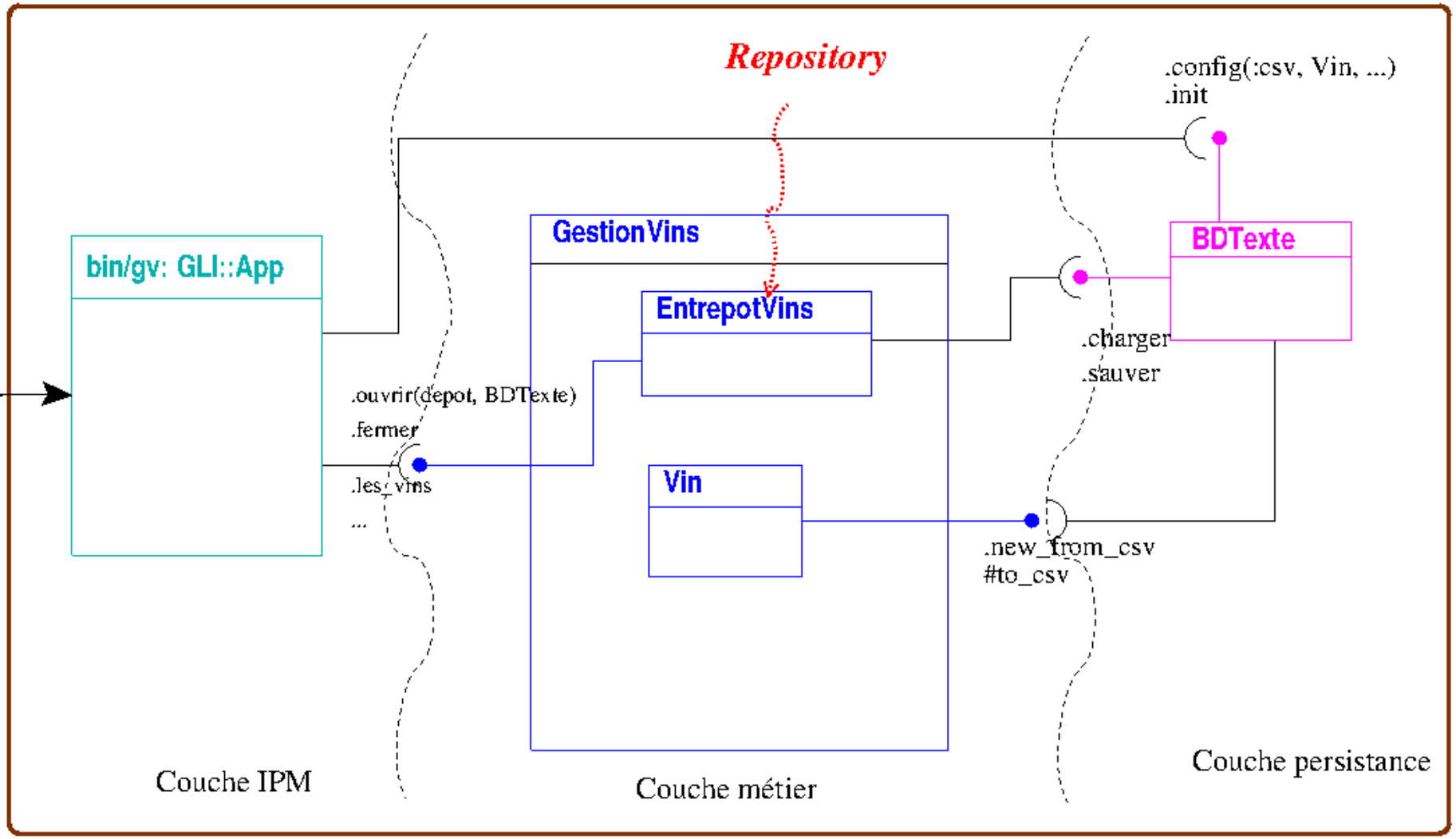
4. L'architecture de gv

L'architecture de `gv`



Note : Un petit rond indique une **interface fournie**, i.e., des méthodes fournies (exportées) par la classe et utilisées par des clients de la classe. Le demi-cercle indique une utilisation d'une interface fournie.

L'architecture de `gv`

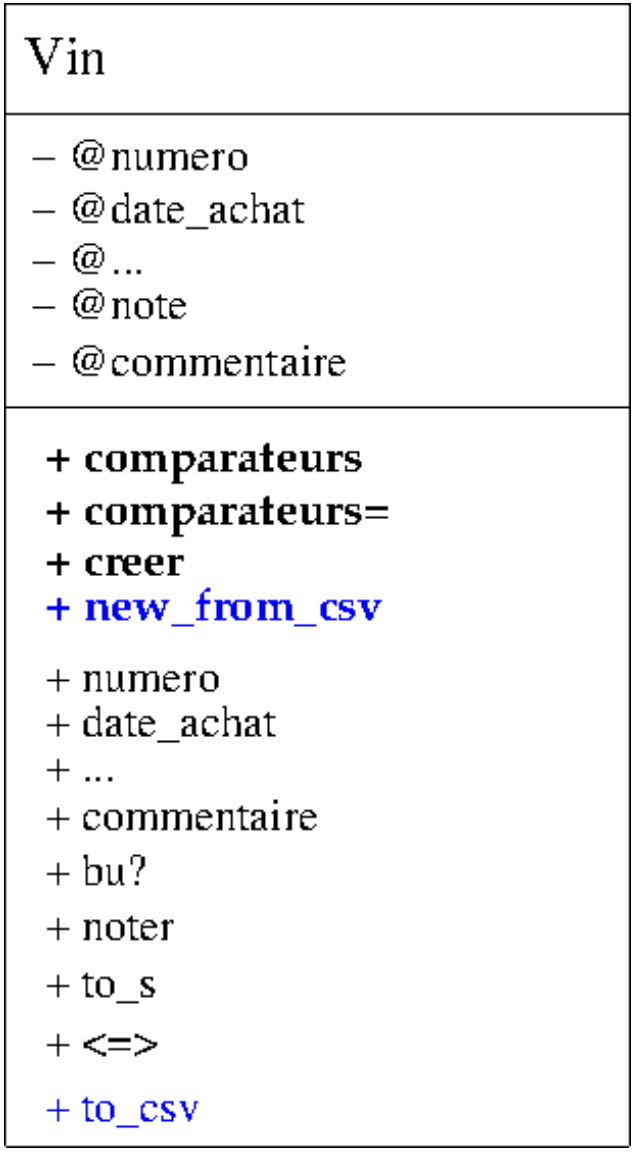


Note : Un petit rond indique une **interface fournie**, i.e., des méthodes fournies (exportées) par la classe et utilisées par des clients de la classe. Le demi-cercle indique une utilisation d'une interface fournie.

Les classes `Vin` et `EntrepotVins`

La classe `Vin`

Modélise un (1) objet `Vin` avec ses divers attributs et méthodes, dont une (ou plusieurs) paire de méthodes additionnelles : `Vin#to_X` et `Vin.new_from_X`



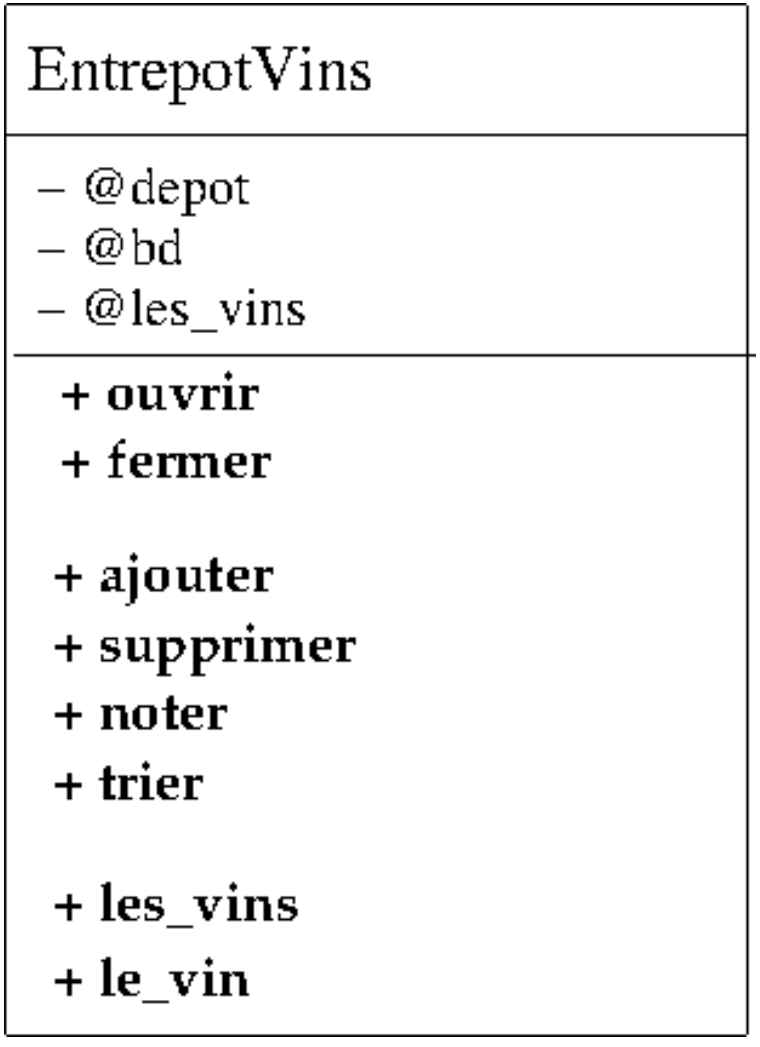
Note : Les noms de méthodes **en gras** sont des méthodes de classe (méthodes `static`).

En UML, de telles méthodes sont indiquées avec des caractères soulignées... mais ce n'était pas possible avec `xfig` ☹

```ruby
class Vin
  include Comparable

  attr_reader( *[:numero, ..., :prix] )

  def self.creer( type, appellation, millesime, nom, prix )
  def initialize( numero, date_achat, ..., commentaire = nil )

  def self.new_from_csv( ligne, separateur = ':' )

  def bu?
  def note
  def commentaire

  def noter( note, commentaire )

  def to_s( le_format = nil )

  def <=>( autre )

  def to_csv( separateur = ':' )
end
```

La classe `EntrepotVins` = le *repository*

Modélise une collection de vins — collection conservée en mémoire et chargée de/sauvegardée vers la BD externe



Note : Composée uniquement de méthodes de classes (`static`) ⇒ une seule instance, sans utilisation concurrente.

La classe `EntrepotVins` = le *repository*

Uniquement des méthodes de classe ⇒ Pas d'activations concurrentes

```
class EntrepotVins

    def self.ouvrir( depot, bd )
    def self.fermer

    def self.ajouter( type, appellation, ..., prix, qte = 1 )
    def self.supprimer( vin: nil, numero: nil )
    def self.noter( numero, note, commentaire )
    def self.trier( cles, reverse )

    def self.les_vins( bus: nil, non_bus: nil, motif: nil )
    def self.le_vin( numero )

end
```

La classe `EntrepotVins` = le *repository*

Uniquement des méthodes de classe ⇒ Pas d'activations concurrentes

```
class EntrepotVins
  class << self
    def ouvrir( depot, bd )
    def fermer

    def ajouter( type, appellation, ..., prix, qte = 1 )
    def supprimer( vin: nil, numero: nil )
    def noter( numero, note, commentaire )
    def trier( cles, reverse )

    def les_vins( bus: nil, non_bus: nil, motif: nil )
    def le_vin( numero )
  end
end
```

Un seul objet et uniquement des méthodes de classe = Autre nom ?

La classe `EntrepotVins` = le *repository*

Uniquement des méthodes de classe ⇒ Pas d'activations concurrentes

```
class EntrepotVins
  class << self
    def ouvrir( depot, bd )
    def fermer

    def ajouter( type, appellation, ..., prix, qte = 1 )
    def supprimer( vin: nil, numero: nil )
    def noter( numero, note, commentaire )
    def trier( cles, reverse )

    def les_vins( bus: nil, non_bus: nil, motif: nil )
    def le_vin( numero )
  end
end
```

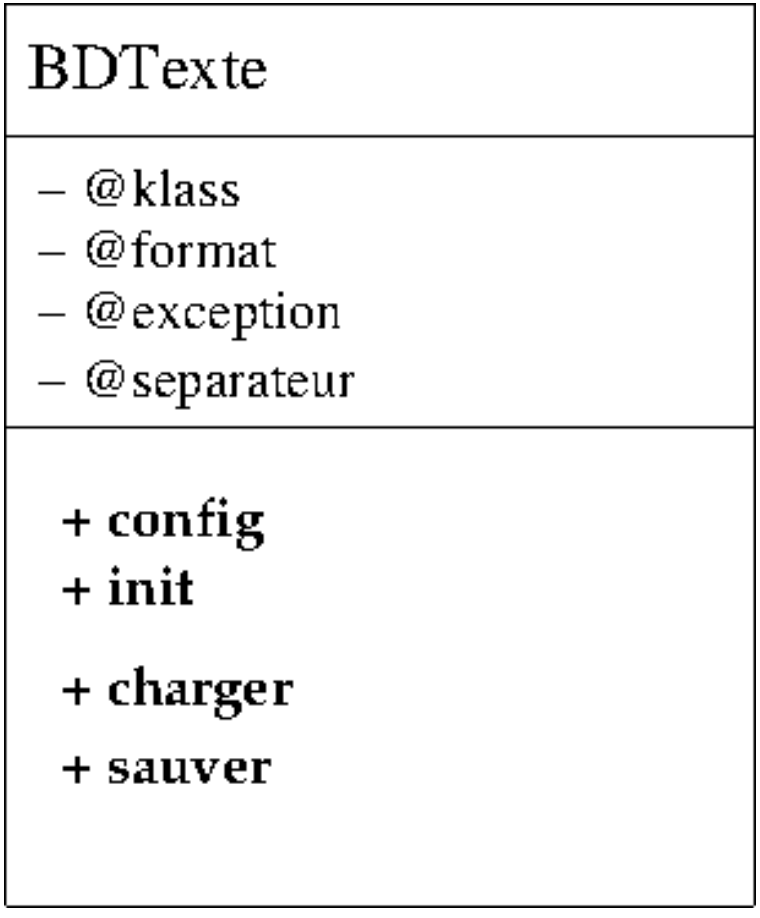
Un seul objet et uniquement des méthodes de classe =
Objet singleton : *The Singleton is a useful Design Pattern for allowing only one instance of your class*

https://www.oracle.com/technetwork/articles/java/singleton-1577166.html

La classe `BDTexte` pour les données textuelle persistantes

La classe `BDTexte`

Encapsule les détails liés au traitement et à la manipulation du fichier texte



Note : Composée uniquement de méthodes de classes (`static`) ⇒ une seule instance, sans utilisation concurrente.

La classe `BDTexte` = la base de données textuelle

Objet singleton

```
class BDTexte
  def self.config( format, klass,
                   separateur: nil )

  def self.init( depot, detruire: false )

  def self.charger( depot )

  def self.sauver( depot, les_elements )
end
```

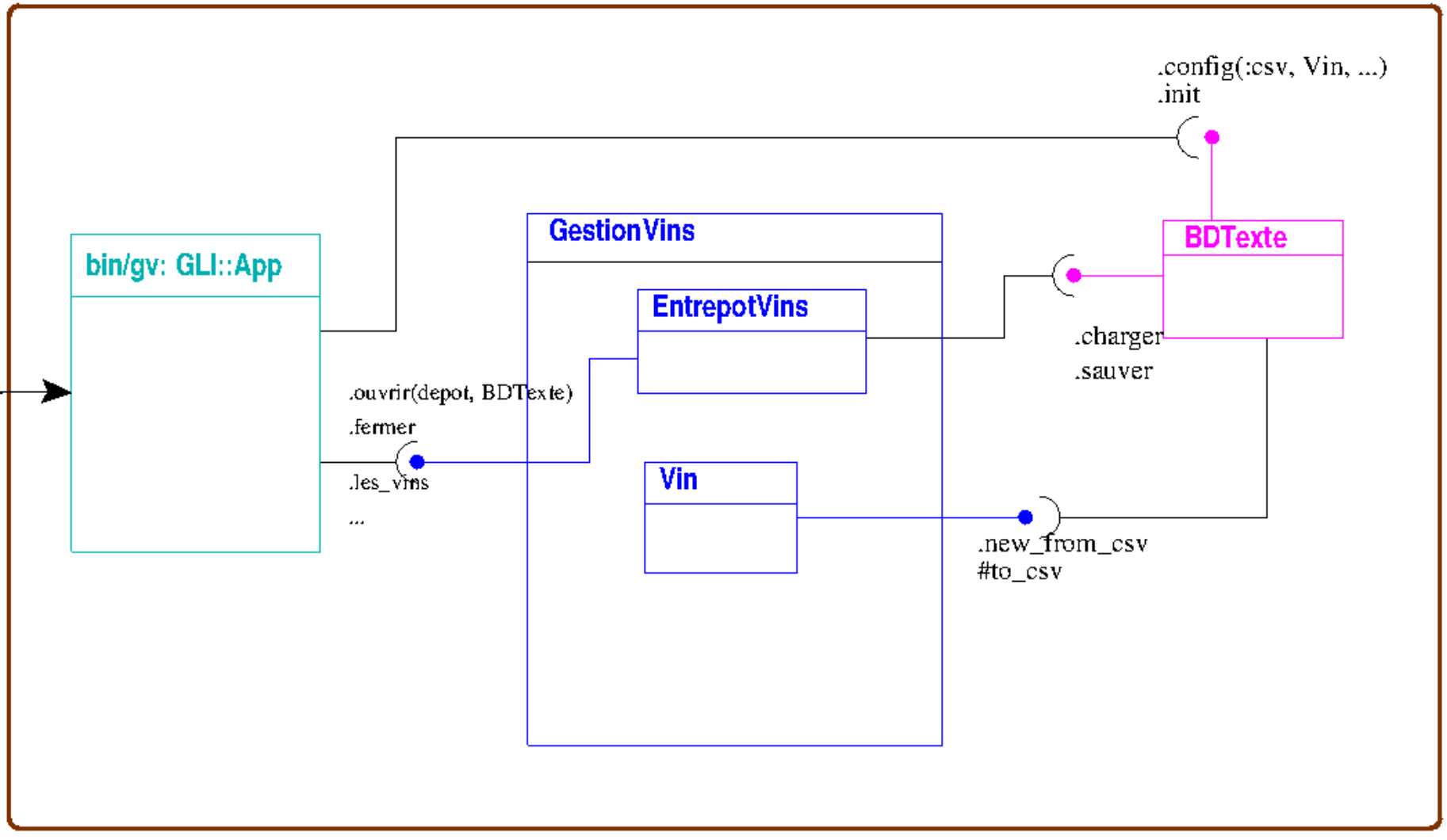
Configuration d’une instance de l’architecture

Architecture abstraite vs. instance spécifique de l'architecture pour une application

- L'architecture décrit l'organisation générale.
- Une application est une instance spécifique de l'architecture, qui doit être configurée pour connecter les divers composants entre eux.
- Pour `gv`, cette configuration se fait dans le programme principal `bin/gv` : voir plus loin pour le code.

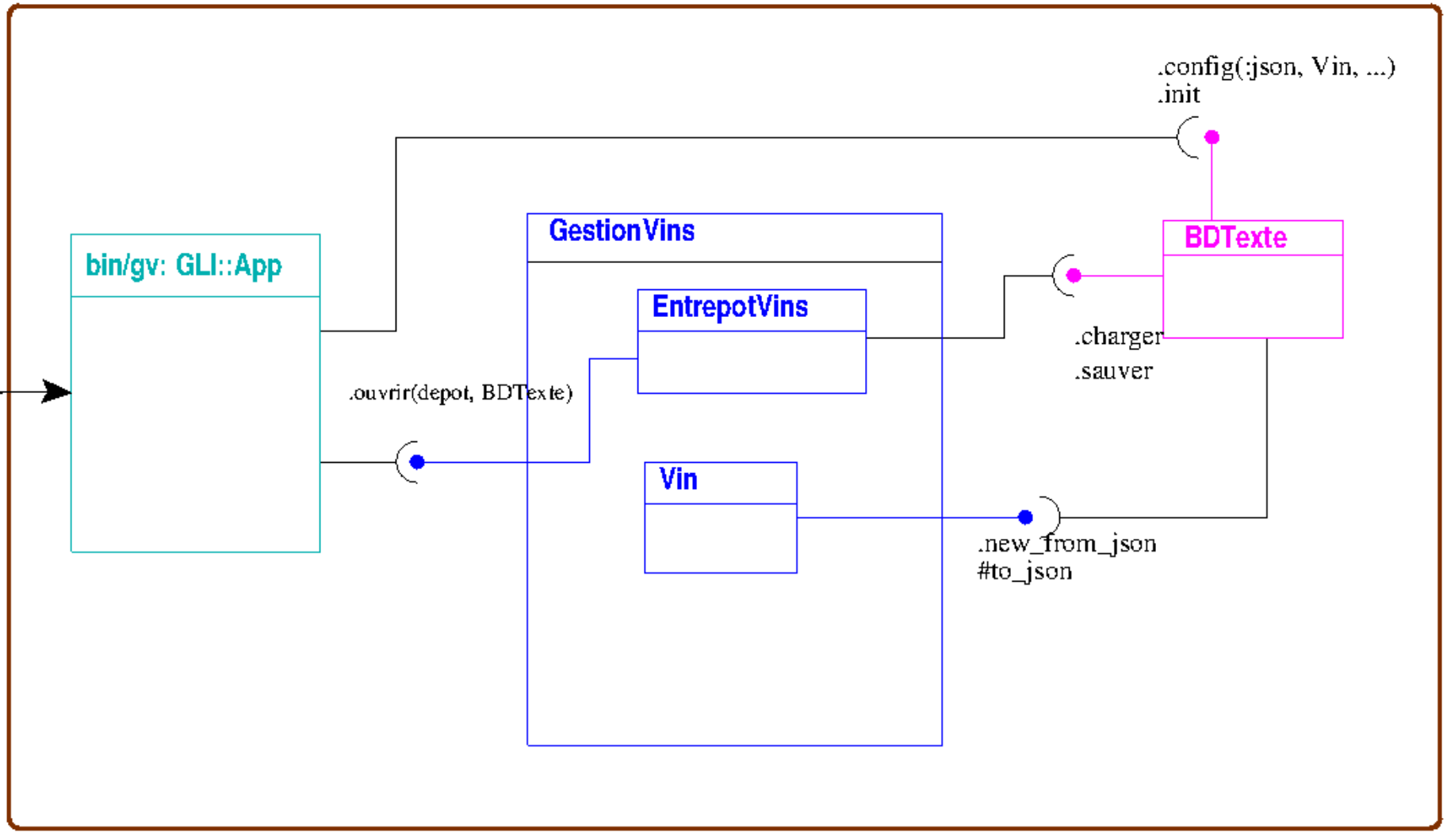
Architecture de `gv`, configurée pour utilisation d'une BD textuelle en format `:csv`

`BDTexte.config( :cvs, Vin, separateur: ':' )`



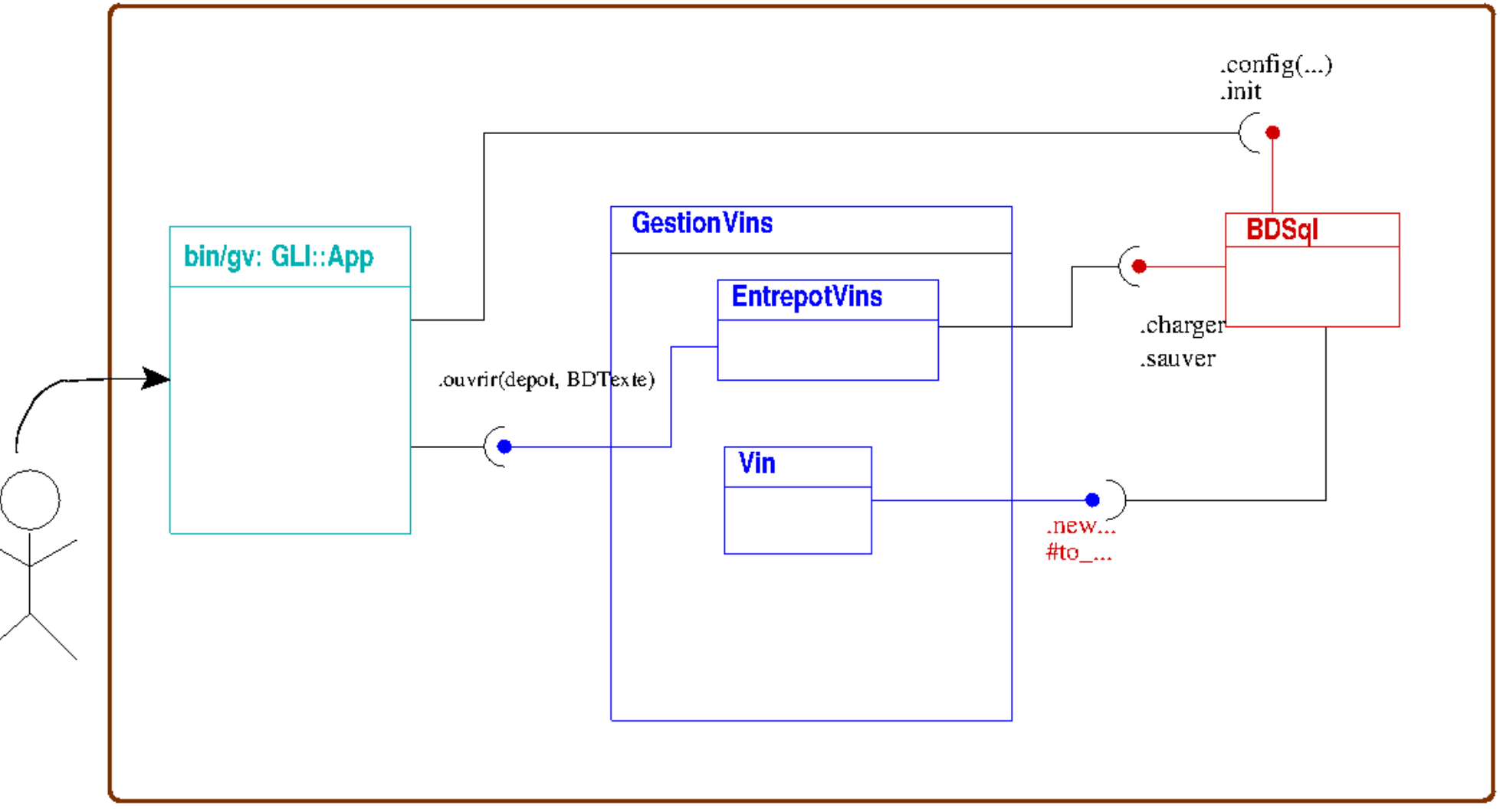
Architecture de `gv`, configurée pour utilisation d'une BD textuelle en format `:json`

`BDTexte.config( :json, Vin )`



Architecture de `gv`, configurée pour utilisation d'une BD... SQL

Aucun impact sur `EntrepotVins`, mais ajout de méthodes à `Vin`



5. La mise en œuvre de gv

Le programme principal `bin/gv`

Fichier `bin/gv` : Configuration de l'architecture


```
# Alias pour <<simplifier>> le code
GV = GestionVins
ENTREPOT = GestionVins::EntrepotVins

DEPOT_DEFAUT = '.vins.txt'

# Injection des dependances pour la BD textuelle.
BDTexte.config( :cvs , GV::Vin,
                separateur: ':' )
```

Fichier `bin/gv` : Démarrage du programme

Rappel : Pour simplifier, les diapos ne contiennent aucun traitement d'erreurs !

```
# Bloc de code execute au demarrage du programme
pre do |global_options,_command,_options,_args|
  depot = global_options[:stdin] ? '-' :
                                   global_options[:depot]

  ENTREPOT.ouvrir( depot, BD )

  true
end

# Bloc de code execute a la fin du programme
post do |_global_options,_command,_options,_args|
  ENTREPOT.fermer
end
```

Fichier `bin/gv` : Spécification de la commande `selectionner`


```
desc 'Selectionne des vins'
arg_name 'motif'
skips_post
command :selectionner do |selectionner|
  selectionner.desc 'Selectionne les vins bus'
  selectionner.switch :bus, :negatable => false
  selectionner.desc 'Selectionne les vins non-bus'
  selectionner.switch :'non-bus', :negatable => false
  selectionner.desc 'Selectionne tous les vins'
  selectionner.switch :tous, :negatable => false

  selectionner.action do |_global_options,options,args|
    motif = args.shift
    bus, non_bus = options[:bus], options[:'non-bus']
    bus = non_bus = true if options[:tous] || !bus && !non_bus

    emettre_sur_stdout( ENTREPOT.les_vins(bus: bus,
                          non_bus: non_bus, motif: motif) )
  end
end
```

Fichier `bin/gv` : Spécification de la commande `noter`


```
desc 'Note un vin'
arg_name 'numero_vin note commentaire'
command :noter do |c|
  c.action do |_global_options,_options,args|
    numero_vin = args.shift.to_i
    note = args.shift.to_i
    commentaire = args.shift

    ENTREPOT.noter(numero_vin, note, commentaire)
  end
end
```

L'action fait simplement analyser les arguments, puis appele une méthode appropriée du modèle, ici, de l'entrepôt de données.

Fichier `bin/gv` : Spécification de la commande `lister`

Utilise la «requête» `les_vins` de l'entrepôt de données et `Vin#to_s`

```
desc 'Liste l\'ensemble des vins de la cave a vins'
skips_post
command [:lister, :ls] do |lister|
  lister.desc 'Affiche au format court'; lister.switch :court
  lister.desc 'Affiche au format long'; lister.switch :long
  lister.desc 'Affiche avec un format specifique'; lister.flag

  lister.action do |_global_options,options,_args|
    option = [:court, :long, :format]
                .select { |opt| options[opt] }
    option = [:long] if option.empty?
    le_format = case option.shift
                when :court  then FORMAT_COURT
                when :long   then FORMAT_LONG
                when :format then options[:format]
                end

    print ENTREPOT
            .les_vins.map { |e| e.to_s(le_format) << "\n" }
            .join
  end
```

Fichier `bin/gv` : Spécification de la commande `supprimer`

Idem : Simple appel à la méthode de l'entrepôt

```
desc 'Supprime un vin'
arg_name 'numero_vin'
command :supprimer do |c|
  c.action do |_global_options,_options,args|
    numeros_a_supprimer(args).each do |numero|
      ENTREPOT.supprimer(numero: numero)
    end
  end
end
```

Fichier `bin/gv` : Méthodes auxiliaires utilisées pour obtenir les numéros à supprimer


```
def numeros_a_supprimer( args )
  if args.empty?
    STDIN
      .readlines
      .reject { |ligne| ligne.strip.empty? }
      .flat_map { |ligne| ligne
                  .split
                  .map { |chaine| num_sur_ligne(chaine.chomp)
  else
    [num_sur_ligne(args.shift)]
  end
end

def num_sur_ligne( ligne )
  m = /\b#{GV::Motifs::NUM_VIN}\b/.match( ligne.strip )

  m[0].to_i
end
```

L’entrepôt de données

`EntrepotVins`

La classe `EntrepotVins` : Ouverture (démarrage du programme) et fermeture (fin du programme)

Délègue à la base de données la lecture/écriture effective (`charger/sauver`)

```
class EntrepotVins
  def self.ouvrir( depot, bd )
    @depot = depot
    @bd = bd
    @les_vins = @bd.charger( depot )
  end

  def self.fermer
    @bd.sauver( @depot, @les_vins )
  end
```

La classe `EntrepotVins` : Méthodes qui modifient l'état

Au sens de B. Meyer (*command/query separation*), ce sont des *commandes*

```
def self.ajouter( type, appellation, millesime,
                  nom, prix, qte = 1 )
  qte.times do
    @les_vins << Vin.creer( type, appellation, millesime,
                            nom, prix )
  end
end

def self.noter( numero, note, commentaire )
  vin = GV::EntrepotVins.le_vin(numero)

  vin.noter(note, commentaire)
end
```

La classe `EntrepotVins` : Méthodes qui observent l'état sans le modifier

Au sens de B. Meyer (*command/query separation*), ce sont des *requêtes*

```ruby
def self.le_vin( numero )
  @les_vins.find { |v| v.numero == numero }
end

def self.les_vins( bus: nil, non_bus: nil, motif: nil )
  tous_les_vins = (bus.nil? && non_bus.nil?) ||
                  (bus && non_bus)

  @les_vins
    .select do |v|
      (tous_les_vins || (bus && v.bu?) || (non_bus && !v.bu?
      (motif.nil? || /#{motif}/i =~ v.to_s) &&
      (block_given? ? yield(v) : v)
  end
end
```

La méthode `les_vins` permet des requêtes complexes :

- En fonction des options `bus` vs. `non-bus` ;
- En fonction d'un `motif` ;
- En fonction d'un prédicat (bloc) — mais ce n'est pas utilisé par `bin/gv`.

La base de données textuelle

`BDTexte`

La classe `BDTexte` : Configuration


```
class BDTexte
  def self.config( format,
                   klass,
                   separateur: nil )
    @klass = klass
    @format = format
    @separateur = separateur
  end
```

Donc, on spécifie :

- La classe (`klass`) des objets qui devront être créés lors du chargement d'un fichier texte.
- Le `format` de lecture/écriture, ce qui déterminera quelles méthodes devront être appelées : `klass.new_from_format` et `klass#to_format`
- Le `separateur` à utiliser si nécessaire pour le `format` (par ex., CSV).

La classe `BDTexte` : Chargement du fichier


```ruby
def self.charger( depot )
  new_from_format = "new_from_#{@format}".to_sym

  (depot == '-' ? STDIN.readlines : IO.readlines(depot))
    .map do |ligne|
    if @separateur
      @klass.send( new_from_format, ligne, @separateur )
    else
      @klass.send( new_from_format, ligne )
    end
  end
end
```

La classe `BDTexte` : Sauvegarde du fichier


```
def self.sauver( depot, les_elements )
  FileUtils.cp depot, "#{depot}.bak" # Copie de sauvegarde.

  to_format = "to_#{@format}".to_sym
  arguments = @separateur ? [@separateur] : []

  File.open( depot, "w" ) do |fich|
    les_elements.each do |v|
      fich.puts v.send( to_format, *arguments )
    end
  end
end
```

La classe `Vin`

La classe `Vin` : Création d'un nouvel objet (`ajouter`) et instantiation d'un objet (`new`)


```
READERS = [:numero, :date_achat, :type, :appellation, :mille
ACCESSORS = [:note, :commentaire]
...
def self.creer( type, appellation, millesime, nom, prix )
  numero = Vin.numero_max.nil? ? 0 : Vin.numero_max + 1
  date_achat = Time.now.to_date
  new( numero, date_achat, type.to_sym, appellation,
       millesime.to_i, nom, prix.to_f )
end


def initialize( numero, date_achat,
                type, appellation, millesime, nom, prix,
                note = nil, commentaire = nil )
  (READERS + ACCESSORS).each do |var|
    instance_variable_set "@#{var}",
                          (binding.local_variable_get var)
  end
  Vin.numero_max = Vin.numero_max ? [Vin.numero_max, numero]
end
```

La classe `Vin` : Comparaison

Solution purement fonctionnelle avec `reduce`


```
def <=>( autre )
  Vin.comparateurs.reduce(0) do |r, champ|
    r.nonzero? ? r : send(champ) <=> autre.send(champ)
  end
end
```

La classe `Vin` : Conversion vers une ligne de texte style CSV


```
def to_csv( separateur = ':' )
  [numero.to_s,
   date_achat.strftime("%d/%m/%y"),
   type.to_s,
   appellation,
   millesime.to_s,
   nom,
   sprintf( "%.2f", prix ),
   note? ? note.to_s : '',
   note? ? commentaire : ''
  ].join(separateur)
end
```

La classe `Vin` : Création à partir d'une ligne de texte style CSV


```
def self.new_from_csv( ligne, separateur = ':' )
  num_vin, date_achat, type, appellation, millesime, nom,
           prix, note, commentaire =
    ligne.chomp.split(separateur, 9)

  j, m, a = date_achat.split("/")

  new( num_vin.to_i,
       Date.new( 2000 + a.to_i, m.to_i, j.to_i ),
       type.to_sym,
       appellation,
       millesime.to_i,
       nom,
       prix.to_f,
       note.empty? ? nil : note.to_i,
       commentaire.empty? ? nil : commentaire )
end
```

La classe `Vin` : Conversion vers un élément JSON


```ruby
# Rappel
READERS = [:numero, :date_achat, :type, :appellation, :mille
ACCESSORS = [:note, :commentaire]

...

require 'json'

...

def to_json
  (READERS + ACCESSORS)
    .map { |c|  [c, instance_variable_get("@#{c}")] }
    .to_h
    .to_json
end
```

La classe `Vin` : Création à partir d'un élément JSON


```ruby
def self.new_from_json( json_hash )
  hash = JSON.parse( json_hash )

  # On "corrige" le type de certains des champs.
  a, m, j = hash["date_achat"].split("-").map(&:to_i)
  hash["date_achat"] = Date.new( a.to_i, m.to_i, j.to_i )
  hash["type"] = hash["type"].to_sym

  new(*(READERS+ACCESSORS).map(&:to_s).map { |k| hash[k] })
end
```

La classe `Vin` : Conversion en chaine de caractères

Méthode `to_s`

```
def to_s( le_format = nil )
  le_format ||= '%I [%T - %.2P$]: %A %M, %N (%D) => %n {%c}'
  vrai_format, args = generer_format_et_args(le_format)

  format( vrai_format, *args )
end
```

La classe `Vin` : Conversion en chaine de caractères (suite)

Méthode `generer_format_et_args`

```
def generer_format_et_args( le_format )
  format_a_traiter = le_format.dup
  format_final = ''

  indicateurs = "IDTAMNPnc"
  motif = /^(?<prefixe>[^%]*)
           %
           (?<largeur>[-]?\d*[.]?\d*)
           (?<car_champ>[#{indicateurs}])
          /x

... suite page suivante ...
```

Note : Pour un `MatchData`, on peut donner un nom (+ significatif) à un groupe capturé avec « `(?<...>)` ».
Donc, au lieu d'utiliser `m[1]`, `m[2]` ou `m[3]`, on utilisera donc `m[:prefixe]`, `m[:largeur]` et `m[:car_champ]`.

```
  les_args = []
  while m = motif.match(format_a_traiter) do
    case m[:car_champ]
    when 'I'
      car_vrai_format = 'd'; val = numero
    when 'D'
      car_vrai_format = 's'; val = date_achat.strftime("%d/%m/%y")
    when 'T'
      car_vrai_format = 's'; val = type
    when 'A'
      car_vrai_format = 's'; val = appellation
    when 'M'
      car_vrai_format = 'd'; val = millesime
    when 'N'
      car_vrai_format = 's'; val = nom
    when 'P'
      car_vrai_format = 'f'; val = prix
    when 'n'
      if note? car_vrai_format = 'd'; val = note else car_vrai_format = 's'; val = '' end
    when 'c'
      car_vrai_format = 's'; val = note? ? commentaire : ''
    end

    format_final << m[:prefixe] << '%' << m[:largeur] << car_vrai_format
    les_args << val

    format_a_traiter = m.post_match
  end
  format_final << format_a_traiter

  [format_final.gsub(/\\n/, "\n"), les_args]
end
```

Q : Est-ce d'un bon style ?

```
  les_args = []
  while m = motif.match(format_a_traiter) do
    case m[:car_champ]
    when 'I'
      car_vrai_format = 'd'; val = numero
    when 'D'
      car_vrai_format = 's'; val = date_achat.strftime("%d/%m/%y")
    when 'T'
      car_vrai_format = 's'; val = type
    when 'A'
      car_vrai_format = 's'; val = appellation
    when 'M'
      car_vrai_format = 'd'; val = millesime
    when 'N'
      car_vrai_format = 's'; val = nom
    when 'P'
      car_vrai_format = 'f'; val = prix
    when 'n'
      if note? car_vrai_format = 'd'; val = note else car_vrai_format = 's'; val = '' end
    when 'c'
      car_vrai_format = 's'; val = note? ? commentaire : ''
    end

    format_final << m[:prefixe] << '%' << m[:largeur] << car_vrai_format
    les_args << val

    format_a_traiter = m.post_match
  end
  format_final << format_a_traiter

  [format_final.gsub(/\\n/, "\n"), les_args]
end
```

Q : Est-ce d'un bon style ? R : Non, pas très bon, car répétitif (pas DRY) ☹

Table-driven methods

Table-driven methods are schemes that allow you to look up information in a table rather than using logic statements (i.e. case, if). In simple cases, it's quicker and easier to use logic statements, but as the logic chain becomes more complex, table-driven code is simpler than complicated logic, easier to modify and more efficient.

Source*: S. McConnell,* «Code Complete, Second Edition»

Table-driven methods

Table-driven methods are schemes that allow you to look up information in a table rather than using logic statements (i.e. case, if). In simple cases, it's quicker and easier to use logic statements, but as the logic chain becomes more complex, table-driven code is simpler than complicated logic, easier to modify and more efficient.

Source*: S. McConnell,* «Code Complete, Second Edition»

Méthode `generer_format_et_arg` — version améliorée (*table-driven*)


```
def generer_format_et_args( le_format )
  format_a_traiter = le_format.dup
  format_final = ''

  format_et_valeur = vrai_format_et_valeur # Methode auxiliaire
  indicateurs = format_et_valeur.keys.join
  motif = /^(?<pref>[^%]*)%(?<largeur>[-]?\d*[.]?\d*)
            (?<car_champ>[#{indicateurs}])/x

  les_args = []
  while m = motif.match(format_a_traiter) do
    car_vrai_format, val = format_et_valeur[m[:car_champ]]
    format_final << m[:pref] << '%' << m[:largeur] << car_vrai
    les_args << val
  end
  format_final << format_a_traiter

  [format_final.gsub(/\\n/, "\n"), les_args]
end
```

Méthode `vrai_format_et_valeur` — la table (`Hash`) pour l'approche *table-driven*


```
def vrai_format_et_valeur
  {
    'I' => ['d', numero],
    'D' => ['s', date_achat.strftime("%d/%m/%y")],
    'T' => ['s', type],
    'A' => ['s', appellation],
    'M' => ['d', millesime],
    'N' => ['s', nom],
    'P' => ['f', prix],
    'n' => (note? ? ['d', note] : ['s', '']),
    'c' => ['s', note? ? commentaire : ''],
  }
end
```

- Facile à lire
- Facile d'ajouter un nouveau format

6. Conclusion

Les avantages d'une architecture multi-couches

- On peut changer/modifier l'IPM de façon indépendante du modèle — indépendante de la «logique métier».

- On peut changer/modifier la représentation des données persistantes, là aussi de façon indépendante du modèle.

Donc, pour le devoir 3. . .

- Vous pouvez vous inspirer de `gv` pour structurer/organiser votre application, votre `gem`.

Donc, pour le devoir 3. . .

- Vous pouvez vous inspirer de `gv` pour structurer/organiser votre application, votre `gem`.

- Vous pouvez utiliser, tel quel ou en le modifiant, le code de `gv` :

```
git clone http://www.labunix.uqam.ca/~tremblay_gu/git/gv.git
```

Donc, pour le devoir 3. . .

- Vous pouvez vous inspirer de `gv` pour structurer/organiser votre application, votre `gem`.

- Vous pouvez utiliser, tel quel ou en le modifiant, le code de `gv` :

```
git clone http://www.labunix.uqam.ca/~tremblay_gu/git/gv.git
```

Il suffit d'indiquer la source, dans un commentaire au début du fichier.