Manuel français de `Clir::DataManager`

Présentation

Cette classe permet de gérer facilement les données des instances (et des classes) dans les applications en ligne de commande (donc les applications tournant dans le Terminal).

Il suffit, pour une classe donnée, de définir les propriétés de ses instances en respectant quelques règles pour pouvoir ensuite afficher, créer, éditer et supprimer n'importe quelle valeur de ces instances.

Par exemple :

On définit les données de la classe `MaClasse` :

```
require 'clir/data_manager'

class MaClasse
  include ClirDataManagerConstants
  DATA_PROPERTIES = [
    {prop: :id, type: :integer, specs:REQUIRED|DISPLAYABLE}, valid_if: {uniq: true},
    {
      prop: :name,
      type: :string,
      specs:ALL_SPECS,
      valid_if: {
        not_empty:true,
        min_length: 3,
        max_length:40
      }
    }
  ]

  # On doit définir aussi la sauvegarde
  @@save_system   = :card # données sauvées dans des fiches
  @@save_format   = :yaml # au format YAML
  @@save_location = "./data" # dans le dossier 'data'
end
```

On instancie un manager de données pour la classe :

```
Clir::DataManager.new(MaClasse)
```

Cela implémente automatiquement plein de méthodes utiles pour les données et notamment :

```
MaClasse.get(id)
# Retourne l'instance d'identifiant :id

<MaClasse>#create([{data}])
# Pour créer une instance (avec ou sans les données {data}

MaClasse.items
# => liste des instances

MaClasse.table
# => table des instances avec en clé leur identifiant
```

Note : si la classe définissait ses données dans une autre constante que `DATA_PROPERTIES`, il faudrait donner cette constante en second argument de `Clir::DataManager.new`.

On peut maintenant utiliser cette méthode pour éditer les valeurs d'une instance :

```
inst = MaClasse.new

inst.create
# => Permet de créer l'instance

inst.edit
# => permet d'éditer (de modifier) l'instance

inst.display
# => Affiche les données de l'instance

inst.remove
# => Détruit l'instance

inst.new?
# => true si c'est une création d'instance (i.e. une instance qui n'a
#    pas encore été enregistrée
```

Méthodes `after_` et `before_`

On peut programmer pour l'instance des méthodes à appeler avant (`before_`) ou après (`after_`) les opérations ci-dessus. On trouve donc :

- **`before_create`** sera appelée avant la création si elle existe. Elle peut retourner les données à prendre en compte,
- **`after_create`** sera appelée (sans argument) après la création complète de l'instance,
- **`before_edit`** sera appelée (sans argument) avant l'édition si elle existe (noter que la création appelle aussi cette méthode),
- **`after_edit`** sera appelée (sans argument) après l'édition et l'enregistrement de l'instance,
- **`before_display`** sera appelée avec les options en premier argument (optionnellement) avant l'affichage. Elle retournera optionnellement les options modifiées,
- **`after_display`** sera appelée (sans argument) après que l'instance a été affichée (donc, quand elle est affichée,
- **`before_remove`** sera appelée (sans argument) avant la destruction de l'instance,
- **`after_remove`** sera appelée (sans argument) après la destruction de l'instance ("instance", ici, signifie ses données enregistrées, car l'instance en tant qu'objet informatique abstrait existe toujours)

Données des propriétés d'instance (`DATA_PROPERTIES`)

Le bon fonctionnement du ***manager de propriété*** tient principalement à la bonne définition des propriétés, généralement (mais pas exclusivement) dans la constante `DATA_PROPERTIES`. Cette définition permet de tout savoir sur les données et de savoir comment les gérer.

Liste `Array`

Les données sont définies dans une liste (`{Array}`) afin de préserver l'ordre défini.

Nom de la constante

Par défaut, le gem s'attend à trouver la constante `DATA_PROPERTIES` définie par la classe à manager. Mais ces données peuvent être définies dans toute autre constante si elle est fournie en second argument de l'instanciation du manager :

```
class MaClasse
  AUTRES_DATA = [...]
  def self.manager
    @@manager ||= Clir::DataManager.new(self, AUTRES_DATA)
  end
```

Attributs d'une propriété

Comme nous venons de le voir, chaque ligne (chaque item) de `DATA_PROPERTIES` définit une propriété de l'instance, comme on les appelle en ruby. Nous allons voir les arguments que doivent ou peuvent comporter chaque *property*.

Liste des attributs

- `:prop`
- `:name`
- `:short_name`
- `:type`
- `:specs`
- `:default`
- `:values`
- `:values_filter`
- `:itransform` (transformation immédiate de la valeur entrée)
- `:mformat` (transformation de la valeur pour affichage)

`:prop` : Nom de la propriété, requise

`:prop` définit le nom de la propriété, pour savoir si c'est l'identifiant, le nom, la date de début etc.

La première propriété, dans *Data-Manager*, doit obligatoirement définir la propriété `:id` :

```
DATA_PROPERTIES = [
  {prop: :id ... }
 ]
```

Nom de propriété spéciale : `:<class min>_id`

Un nom de propriété spécial — et même magique — est constitué d'un nom minusculisé de classe connue suivi par `_id`. Quand un nom est composé de la sorte, cela signifie que la propriété concernant l'identifiant d'une autre instance dans la classe est minusculisée.

Par exemple :

```
class MaClasseManaged
  DATA_PROPERTIES = [
    # ...
    {prop: user_id  ,name:"Propriétaire"  ,type: :id ...}
  ]
end
```

... signifie que chaque instance de cette classe peut être liée à une instance de la classe `User`.

DataManager aussitôt lui-même ce genre de propriété, en :

- utilisant la méthode générale `choose` pour définir cette donnée,
- implémentant la méthode-propriété `classe_min` qui retournera l'instance en question.

Minusculisation du nom de la classe

Si le nom de la classe contient des capitales, il sera décamélisé (`DataManager` => `data_manager` => `data_manager_id`).

Si le nom de la classe est composé, les doubles doubles-points seront remplacés aussi par des tirets plats (`Edic::Produit` => `edic_produit` => `edic_produit_id`

Propriété spéciale : `<classe min>_ids`

Similaire à la classe précédente, mais pour une liste d'éléments. Le type est alors `:ids`

```
DATA_PROPERTIES = [
  # ...
  {prop: espace_nom_user_ids, name: "Partitipantes", type: :ids, values_filter: {sexe:
'F'}, ...}
]
```

Le code ci-dessus permet de choisir une liste de femmes (de “participantes”).

Espace de nom en question

Attention : si la classe relative se trouve dans un espace de nom, même si c’est le même espace de nom que la classe en question, il faut indiquer cet espace de nom dans `:prop` :

```
module EspaceNom
  class User
    # ...
  end

  class MaClasseManaged
    DATA_PROPERTIES = [
      # ...
      {prop: espace_nom_user_id , name: "Propriétaire" ,type: :id ...}
    ]
  end
end
```

De la même manière, il faut noter que la propriété-méthode qui sera créé, pour récupérer l’instance, comportera cet espace de nom. Donc, dans le programme il faudra utiliser :

```
instance.espace_nom_user
# => instance EspaceNom::User
```

Pour simplifier le code, on peut définir la classe managée de cette manière :

```
require 'clir/data-manager'

class MaClasseManaged
  DATA_PROPERTIES = [
    # ...
    {prop: espace_user_id: name: "Propriétaire", type: :id}
    ]
end #/class

Clir::DataManager.new(MaClasseManaged)

class MaClasseManaged


  alias :user :espace_user
  # et : alias :user_id :espace_user_id

end
```

De cette manière on pourra facilement utiliser :

```
item = MaClasseManaged.new
item.user
# => Instance User de l'item
```

Attribut `:name`

Nom humain de la propriété, qui sera utilisé en label pour afficher les données de l'instance. C'est forcément un `String` (TODO: pour le moment en tout cas, car on pourrait imaginer que ça soit une procédure à évaluer en fonction de l'instance).

Attribut `:short_name`

Nom humain court de la propriété, spécialement désigné pour être utilisé dans l'affichage par table.

Attribut `:type`

Liste des types

```
:string     Une chaine quelconque
:number     Un nombre quelconque
:select     Valeur dans une liste (fournie par :values)
:date       Date de type JJ/MM/AAAA
:mail       Un email valide
:id         ID d'une instance connue
:ids        Liste d'IDs d'instance connues
:prix       Un prix (nombre flottant)
:people     Une ou des personnes
:url        Une URL existante
```

Attribut `:specs`

Spécificité de la propriété, permet de savoir si elle est requise (`REQUIRED`), éditable (`EDITABLE`), affichable (`DISPLAYABLE`), supprimable (`REMOVABLE`) ou toutes ces propriétés en même temps (`ALL`).

```
REQUIRED      # La propriété doit impérativement être définie
EDITABLE      # La propriété peut être éditée. L'ID par exemple,
              # ne peut pas l'être. Il est défini programmatiquement
              # et ne peut être modifié
DISPLAYABLE   # La propriété est affichée dans l'affichage complet de
              # la donnée
TABLEIZABLE   # La propriété est affichée dans l'affichage de la donnée
              # dans une table (Clir::Table). Dans ce cas, il peut être
              # intéressant de définir :short_name pour le libellé de
              # l'entête.
REMOVABLE     # La propriété peut être détruite (remplacée par nil)
ALL           # Toutes les précédentes
```

Attribut `:default`

Valeur par défaut. Elle peut être définie par :

- une valeur explicite (d'un type correspondant au type de la propriété),
- une procédure de calcul qui reçoit en premier argument l'instance courante,

  ```
  {prop: first_job_year, ... , default: Proc.new { |inst| inst.naissance + 18 }
  ```

  Noter que cette procédure peut retourner un `Symbol` qui pourra alors être considéré (ou pas) comme méthode de l'instance ou de la classe.
- un `Symbol` renvoyant à une méthode de l'instance OU une méthode de la classe de l'instance :

  ```
  {prop: first_job_year, ... , default: :year_majorite}
  # En considérant que :year_majorite est une méthode d'instance de la
  # classe, qui retourne l'année de majorité de l'individu
  ```

Attribut `:values`

S'emploie pour un type `:select`.

Attribut `:values_filter`

Filtre à utiliser pour les `:values` en fonction de l'instance.

Peut s'utiliser pour les types `:select` et les `:prop` de type `:<class min>_ids` (type `:ids`).

Cette valeur doit être une table (un dictionnaire) qui contient en clé les propriétés qui devront être filtrées et en valeur le filtrage à appliquer. Par exemple :

```
{values: clients, values_filter: {sexe: 'F'} ...}
```

... filtrera uniquement les clientes, puisqu'on cherchera dans les clients les instances dont la valeur `:sexe` est à `"F"`.

La valeur de la propriété peut être aussi un `Symbol`. C'est alors une propriété de l'instance. Un cas typique est le cas d'une facture. Une facture est donc une instance qui rassemble plusieurs achats d'un client (donc plusieurs ventes). Pour pouvoir choisir les ventes, on ne doit afficher que les achats du client. On utilise donc toutes les ressources de *DataManager* en définissant la propriété qui va consigner les identifiants des achats de cette manière :

```
class Facture

  DATA_PROPERTIES = [
    {prop: client_id, type: id, name: "Client"},
    {prop: vente_ids, type: ids: name: "Achats concernés",
      values_filter: {client_id: :client_id}
  ]

end
```

EXPLICATION

La première propriété, `:client_id`, de type `:id`, permet de choisir automatiquement un client dans la liste des clients. Elle prend comme valeur l'identifiant dudit client.

La seconde propriété, `:vente_ids`, de type `:ids`, permet de consigner les ventes concernées par la facture. Pour ne pas afficher toutes les ventes, mais seulement celles du client, le filtre `{client_id: client_id}` signifie : "ne prendre que les ventes dont le `client_id` (clé du filtre) n'est égal à la valeur `:client_id` de la propriété de la facture.

La **clé** correspond dont à une propriété de la chose concernée (ici les ventes)

La **valeur** correspond à une propriété de l'instance en édition (ici la facture).

Attribut `:valid_if`

Définition une procédure de validation de la donnée particulière. Peut être :

- une **Procédure (`Proc`)**. Elle est appelée avec en premier argument la nouvelle valeur et en second argument l'instance de l'objet traité,
- un `Symbol` peut être :
  - une méthode (prédicate) appliqué à la valeur elle-même. Par exemple `numeric?`,
  - une méthode d'instance de l'objet traité. Elle reçoit en premier argument la nouvelle valeur testée,
  - une méthode de classe de l'objet traité. Elle reçoit les mêmes argument que la procédure (nouvelle valeur, instance).

Toutes ces méthodes doivent retourner `nil` si la nouvelle donnée est valide, ou le message d'erreur en cas de problème.

Attribut `:itransform`

Transformation immédiate de la valeur entrée par l'utilisateur. Peut-être une `Proc` ou un `Symbol`.

Cf. Formatage des données.

Attribut `:mformat`

Définition de la méthode de transformation de la donnée à l'affichage. Peut-être une `Proc` ou un `Symbol`.

Répond aux mêmes critères que `itransform` mais s'applique seulement à l'affichage de la donnée.

Cf. Formatage des données.

Attribut `:if`

L'attribut `:if` permet de déterminer si la propriété doit appartenir à l'instance, doit être défini. Par exemple, pour un certain produit il peut exister des propriétés spéciales qui n'appartiennent qu'à lui.

Peut être :

- un procédure (`Proc`),
- une méthode de l'instance (`Symbol`),
- une valeur booléenne explicite.

Cet attribut peut être une `Proc`édure. Cette procédure reçoit en argument l'instance et doit retourner `true` ou `false`. Par exemple :

```
{prop: :cover, ... , if: Proc.new { |inst| inst.type == 'livre' }
```

Dans le cas ci-dessus, la propriété `:cover` définissant la couverture ne sera défini (éditable, affichable) que si la propriété `type` du produit est 'livre'.

L'attribut peut être également une méthode (prédicate certainement) de l'instance. C'est alors un `{Symbol}`.

```
{prop: :cover, ... , if: :livre?}
# => Seulement si <instance>.livre? retourne true
```

Nom humain utilisé pour l'affichage de l'instance

Il y a 5 façons différents pour définir le nom (`:name` dans `Tty-prompt`) qui sera utilisé pour désigner l'instance :

- **valeur par défaut**. Par défaut, il s'agit de la **deuxième propriété définie** (entendu que la première est l'identifiant),
- **valeur générale**. Dans la classe, si on définit la **méthode d'instance #name4tty**, c'est cette méthode qui sera appelée et devra retourner le nom à afficher. Typiquement, si la classe est `People`, qui définit les propriétés `:nom` et `:prenom`, on pourra avoir une méthode :

  ```
  class MaClasseManaged
    def name4tty
      "#{prenom} #{nom}".strip
    end
  end
  ```

- **nom de méthode**. `#name4tty` peut également définir la méthode à utiliser, ce sera alors un `Symbol` :

  ```
  class MaClasseManaged
    def name4tty
      :patronyme
    end

    def patronyme
      @patronyme ||= "#{prenom} #{nom}".strip
    end
  end
  ```

  Noter qu'on peut aussi le faire par alias :

  ```
  class MaClasseManaged
    alias :name4tty :patronyme
  end
  ```

- surclassement. Il est possible de redéfinir la méthode (de classe) utilisée pour définir l'attribut `:name4tty` dans les options utilisées. Par exemple avec la méthode de classe `::choose` :

  ```
  class MaClasseManaged
    def self.choose
      data_manager.choose({name4tty: :patronyme})
    end
  end
  ```

  De cette manière, chaque fois que `MaClasseManaged.choose` sera invoquée, cette méthode sera utilisée, redéfinissant la propriété à utiliser.

  Il est également possible de définir une procédure plutôt qu'un symbol. Elle sera alors évaluée sur l'instance (noter, ci-dessous, également, comment on peut garder des options qui seraient passées) :

  ```
  class MaClasseManaged
    def self.choose(options = nil)
      options ||= {}
      options.merge!({name4tty: Proc.new { |ins| "#{inst.prenom} #{inst.nom}".strip
  })
      data_manager.choose(options)
    end
  end
  ```

- **attribut d'option**. Enfin, il est possible, ponctuellement de définir la méthode qui doit être utilisée en renseignant les `:options` qui accompagnent souvent un affichage. Par exemple, pour la méthode de classe `::choose`, qui permet de choisir une instance, on peut utiliser :

  ```
  class MaClasseManaged
    def self.envoyer_carte
      destinataire = choose({name4tty: :patronyme})
    end
  end
  ```

  Ici aussi on peut avoir en valeur soit un `Symbol`, pour une méthode, soit une procédure qui prendra en premier argument l'instance traitée.

Notez que ces méthodes sont présentées par ordre inverse de précédence. C'est-à-dire que c'est la dernière qui sera utilisée prioritairement et la première qui sera utilisée en dernier recours. Donc, dans l'ordre, l'application fera :

1. attribut direct dans les options envoyées (définissant soit une méthode — `Symbol` — soit une procédure — `Proc`),
2. redéfinition de la méthode générale de classe (par exemple `::choose`),
3. à égalité : méthode d'instance `#name4tty` (retournant un string, la valeur à afficher, ou bien un symbol, la méthode à utiliser)

4. la deuxième propriété définie dans les données des propriétés.

Propriété conditionnelle

Cf. l'attribut `:if`

Formatage de l'affichage des données

La classe managée peut définir des méthodes de formatage d'affichage particulières pour chaque donnée. Par convention, ce nom doit être `f_<property name>`. Par exemple, si la propriété est **`name`** alors le nom de la méthode de mise en forme de sa valeur doit être par défaut **`f_name`**.

Par exemple :

```
class MaClasseManaged

  def age
    @age ||= data[:age]
  end

  def f_age
    "#{age} ans"
  end

end
```

On peut cependant définir un nom de méthode propre en définissant la propriété **`:mformate`** dans les données des propriétés.

> Note : toutes les valeurs commençant par "`:m` dans la définition des propriétés concernent des méthodes.

Par exemple :

```
class MaClasse
  DATA_PROPERTIES = [
    {prop: :name,   name:"Patronyme", type: :string, specs:ALL_SPECS, mformate:
:formate_nom}
    {prop: :sexe,   name:'Sexe', type: :string, specs:ALL_SPECS}
  ]

  def name; @name ||= data[:name] end
  def sexe; @sexe ||= data[:sexe] end
  def formate_nom
    "#{sexe == 'F' ? "Madame" : "Monsieur"} #{name}"
  end
end #/class
```

Transformation de la donnée en entrée ou en sortie

Quand on attend un nom (patronyme), on peut vouloir systématiquement le passer en capitale, quelle que soit l'entrée de l'utilisateur.

Pour ce faire, dans la donnée de la propriété dans `DATA_PROPERTIES`, on ajoute l'attribut `:itransform` (qui signifie "input transform" ou "transformation de la donnée entrée") qui transforme la donnée entrée (donc la donnée qui sera enregistrée) ou l'attribut `:mformat` qui transforme la valeur enregistrée seulement à l'affichage.

Ces attributs peuvent avoir différents types de valeur :

- **une procédure** qui reçoit en premier argument l'instance et en second argument la valeur entrée
- **un symbol**. qui définit :
  - soit une **méthode de la valeur** (p.e. `:upcase` pour pour un `String` ou `:round` pour un nombre)
  - soit une **méthode de l'instance** — qui reçoit en premier argument la valeur.

Atteindre le manager de données (`#data_manager`)

Depuis la classe, on peut faire appel au manager de données à l'aide de `data_manager`. Par exemple :

```
MaClasseManaged.data_manager.save_format
# Retourne :yaml si le format défini (dans @@save_format) est :yaml
```

La seule méthode du data manager qui est exposée publiquement d'office, c'est la propriété `save_location` qui retourne le chemin d'accès soit au fichier de données (si `@@save_system = :file`) soit au dossier des fiches (si `@@save_system = :card`). On peut l'atteindre par :

```
MaClasseManaged.save_location
# => /path/to/folder/de/sauvegarde/
```

Choisir un item

Parmi les méthodes les plus pratiques du manage de données, il y a la méthode `choose` (implémentée dans la classe) qui permet de choisir un élément parmi tous ceux existant, avec ou non un filtre :

```
MaClasseManaged.choose(options = nil)
# => retourne l'instance choisie
```

Les options sont les suivantes :

```
:question     La question à poser
:multi        [Boolean] Si true, on peut choisir plusieurs items
:create       [Boolean] Si true, on peut créer un nouvel item
:complete     [Boolean] Si true, on ajoute un menu "Finir" pour… finir
              Utile lorsque le choix est appelé en boucle quand on ne
              veut pas (ou ne peut pas) utiliser :multi (ci-dessus)
:finir        [Boolean] idem
:filter       [Hash] Le filtre des éléments (cf. ci-dessous)
:exclude      [Array<Ids>] Liste des identifiants à exclure de la liste
:default      [Array<Ids>] Quand :multi est true, liste des ids qui
              doivent être sélectionnés par défaut.
:sort_key     Propriété qui doit servir de clé de classement
:sort_dir     Direction du classement, 'asc' (défaut) ou 'desc'
```

Filtre pour `choose`

La propriété `:filter` permet à la méthode `#choose` de choisir seulement un type d'élément. C'est un table avec des clés qui sont les propriétés des items.

Il y a quelques valeurs spéciales :

```
:periode  [Periode] Définit la période dans laquelle l'item doit se trouver
          Pour juger de la place de l'item, le filtre se sert d'une propriété
          :date, :created_at ou :time qui doit donc exister.
```

Grouper les éléments pour `choose`

On peut aussi grouper les éléments pour les choisir en deux temps. Cf. ci-après.

Groupement des items (`@@group_by`)

Lorsqu'il y a beaucoup d'items, il peut être plus clair de les grouper pour voir les choisir plus facilement. Ou par exemple, ce peut être simplement une histoire de cohérence sémantique. Par exemple, si l'on veut choisir un livre, on peut vouloir dans un premier temps le choisir simplement par le titre, sans avoir pour chaque livre les versions de support (papier, numérique, …) ou les langues.

Pour ce faire, on définit la variable générale `@@group_by` qui déterminera que la classe sera toujours groupée par cette donnée. Elle peut se définir au même niveau que les variables `@@save_location` etc. Sa valeur est une clés définie dans `DATA_PROPERTIES`.

```
@@group_by = :<props>
```

Par exemple :

```
class MaClasseManaged
  DATA_PROPERTIES = [
    {prop: :id, name: "ID", ....}
    {prop: :livre, name: "Livre", ...}
    {prop: :name,  name: "Nom", ....}
    {prop: :cate,  name: "Catégorie", ...}
  ]

  @@save_location = "path/to/the/file"

  @@group_by = :cate

end
```

Si la propriété est une propriété de type *identifiant* (par exemple `livree_id`) alors les items seront rassemblé sous l'instance correspondante et le nom de cette instance sera utilisée pour l'affichage.

Si la propriété choisie est une propriété quelconque, alors c'est sa valeur qui est utilisée dans la liste affichée. Par exemple, s'il existe les catégorie (`:cate`) "Boisson", "Entrée" et "Fromage", alors tous les éléments dans la catégorie "Boisson" seront rassemblés sous ce nom unique, tous les éléments de catégorie "Fromage" seront rassemblés sous ce nom, etc.

Les petits plus

Le fait de travailler avec `Clir::DataManager` offre de nombreux avantages, comme on a pu le voir. Il existe cependant quelques petites astuces à connaitre.

Messages féminisés

Pour obtenir des messages adaptés au genre d'une classe, on peut définir la méthode de classe `::feminine?` qui reverra true dans le cas d'une classe féminine. Par exemple :

```
class PaireDeLunettes
  def self.feminine?; true end
end

# Produira par exemple le message suivant à la création d'une nouvelle
# instance : "Nouvelle PaireDeLunettes créée avec succès !"

class SacAMain
  def self.feminine?; false end # <= mais inutile, car par défaut
end

# Produira par exemple le message suivant à la création d'une nouvelle
# instance : "Nouveau SacAMain créé avec succès !"
```

Filtrer la liste des propriétés

Quand la liste des propriétés de l'instance est affichée, par exemple pour l'éditer (aka la modifier), on peut atteindre très rapidement la propriété à modifier en tapant ses premières lettres (ou ses lettres caractéristiques). Cela filtre la liste des propriétés et n'affiche que les propriétés correspondant au filtre. Si la liste des propriétés est longue, on peut énormément se simplifier la vie avec cette astuce.