Annexes pour l'examen final : Modèles UML et classes Ruby

MGL7460

Automne 2016

A Contrôleurs pour une machine à café simple

On veut définir des contrôleurs logiciels pour une machine distributrice de café simple :

- Le prix du café est nécessairement un multiple de 0.25 $.
- La machine accepte uniquement les pièces de 0.25 $ et 1.00 $.
- Si le montant total inséré est supérieur au prix du café, la machine rend la monnaie, mais toujours en pièces de 0.25 $.
- L'usager doit fournir sa propre tasse, qu'il doit préalablement insérer sous le bec verseur.

Le code Ruby sur les pages suivantes définit trois classes, qui servent à contrôler et interagir avec les composants matériels de la machine à café :

a. `ControleurPieces` : Lorsqu'une pièce de monnaie est insérée, un mécanisme *hardware* lance un appel de la méthode `piece_recue`. Cette méthode peut alors signaler l'insertion d'une pièce à un ou plusieurs autres composants. Un composant à signaler est indiqué par un bloc, transmis par un appel à `signaler_quand_piece_recue`.

   Ce contrôleur peut aussi envoyer des signaux au *hardware* pour `accepter` ou `refuser` une pièce qui vient d'être insérée, ou pour rendre la monnaie — `rendre_25_cents`.

b. `ControleurCafe` : Contrôle l'eau, le café, etc. Donc, lors d'un appel à `servir`, envoie des signaux au matériel pour chauffer l'eau, faire couler le café infusé, etc.

c. `ControleurMachineCafe` : Contrôle le comportement **global** de la machine, en fonction du `prix` spécifié pour un café (cf. constructeur), des pièces pouvant être acceptées (cf. `piece_valide?`), du total des pièces insérées par un usager, etc.

La Figure 1 ci-bas présente un diagramme de séquence UML qui illustre les interactions entre ces contrôleurs lorsqu'un usager insère une première pièce de 1.00 $ pour un café coûtant 0.75 $ — les interactions lors de l'initialisation du système ne sont pas indiquées.

Note : Les montants des pièces sont indiqués **en cents**, pour que les calculs puissent se faire de façon exacte — donc `100` = 1.00 $.

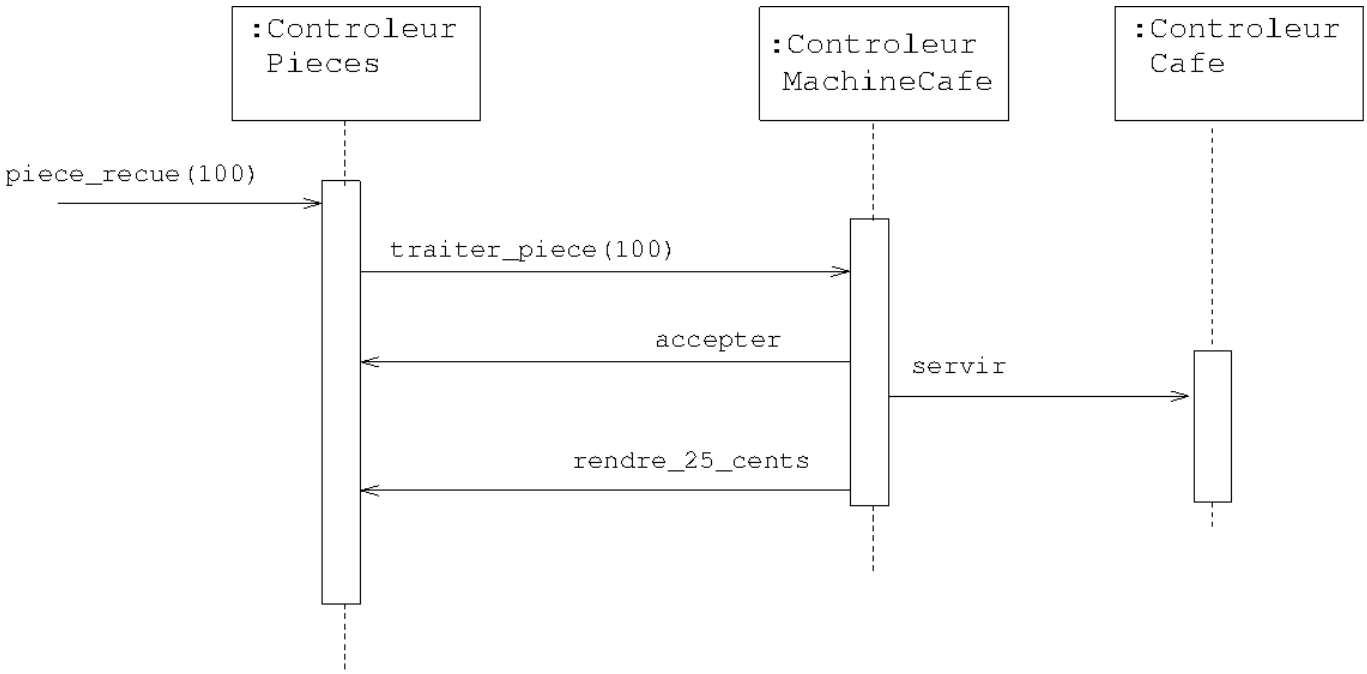

Figure 1: Diagramme de séquence UML simplifié illustrant les interactions entre les contrôleurs lors de l'insertion d'une pièce de 1.00 $ pour un café coûtant 0.75 $.

Classe Ruby 1 ControleurMachineCafe.

```ruby
class ControleurMachineCafe
  def initialize( prix_cafe )
    DBC.requires 0 < prix_cafe && prix_cafe % 25 == 0

    @controleur_pieces = ControleurPieces.new
    @controleur_pieces.signaler_quand_piece_recue do |piece|
      traiter_piece(piece)
    end
    @controleur_cafe = ControleurCafe.new

    @prix_cafe = prix_cafe
    @montant_recu = 0
  end

  def traiter_piece( piece )
    if !piece_valide?(piece)
      @controleur_pieces.refuser
      return
    end

    @controleur_pieces.accepter
    @montant_recu += piece

    if @montant_recu >= @prix_cafe
      @controleur_cafe.servir
      rendre_monnaie
    end
  end

  private

  def rendre_monnaie
    nb_25_cents_a_rendre = (@montant_recu - @prix_cafe) / 25
    nb_25_cents_a_rendre.times do
      @controleur_pieces.rendre_25_cents
    end
  end

  def piece_valide?( piece )
    [25, 100].include?(piece)
  end
end
```

Classe Ruby 2 ControleurPieces.

```ruby
class ControleurPieces
  def initialize
    @a_signaler = []
    # ...
  end

  # Pour specifier du code a executer lorsqu'une piece est inseree.
  def signaler_quand_piece_recue( &bloc )
    @a_signaler << bloc
  end

  # Methode appelee (par le "hardware") lorsqu'une piece est inseree.
  # Signale a divers autres controleurs l'insertion de cette piece via
  # le bloc indique dans signaler_quand_piece_recue.
  def piece_recue( piece )
    @a_signaler.each { |bloc| bloc.call(piece) }
  end

  # Accepte la piece inseree.
  def accepter; ...; end

  # Retourne la piece inseree.
  def refuser; ...; end

  # Rend une piece de 0.25$.
  def rendre_25_cents; ...; end
end
```

Classe Ruby 3 ControleurCafe.

```ruby
class ControleurCafe
  def initialize; ...; end

  # Sert un cafe.
  def servir; ...; end
end
```

B Modèle et classes pour des cours et programmes

La Figure 2 présente un diagramme de classes UML pour modéliser des cours et des programmes d'études. Il s'agit d'un modèle simple, puisqu'un programme d'études est composé entièrement de cours obligatoires. Ainsi, si tous les cours comportent trois (3) crédits, alors un programme d'études de 15 crédits sera alors associé à exactement cinq (5) cours.

Des classes Ruby réalisant ce modèle à l'aide de **deux formes distinctes d'API coulantes** sont ensuite présentées — Programmes Ruby 4 et 5.

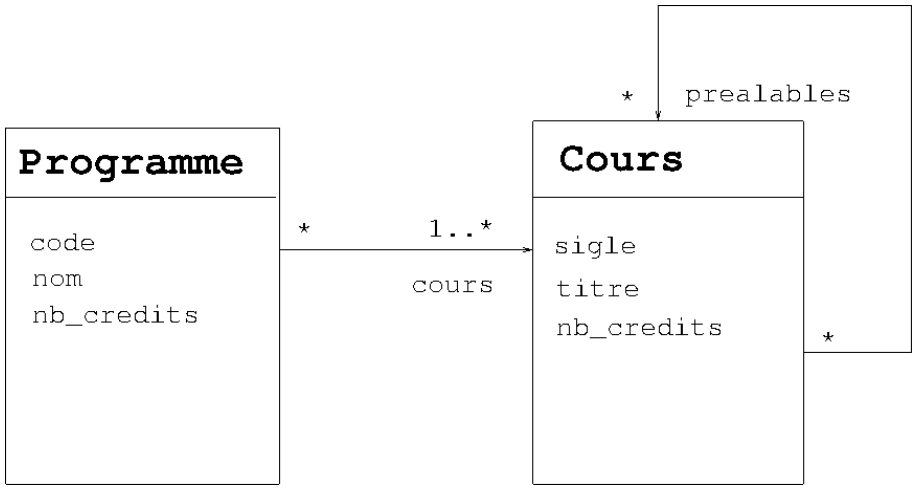

Figure 2: Modèle UML simple pour des cours et des programmes d'études.
Note : L'annotation «`*`» signifie 0, 1 ou plusieurs, alors que «`1..*`» signifie 1 ou plusieurs.

Classe Ruby 4 Cours.

```ruby
class Cours
  attr_reader :sigle

  def initialize( sigle )
    @sigle = sigle
    @prealables = []
  end

  def titre( le_titre = nil )
    return @titre unless le_titre

    @titre = le_titre
    self
  end

  def nb_credits( nb = nil )
    return @nb_credits unless nb

    @nb_credits = nb
    self
  end

  def prealables( *cours )
    return @prealables if cours.empty?

    @prealables += cours
    self
  end

  def done
    valider
    self
  end
```

```
  private

  def sigle_valide?( sigle )
    sigle && sigle.kind_of?(String) && sigle =~ /[A-Z]{3}\d{4}/
  end

  def titre_valide?( titre )
    titre && titre.kind_of?(String)
  end

  def nb_credits_valides?( nb_credits )
    nb_credits && nb_credits.kind_of?(Fixnum) && 1 <= nb_credits
  end

  def prealables_valides?( prealables )
    prealables.all? { |prealable| prealable.kind_of?(Cours) }
  end

  def valider
    fail "sigle invalide" unless sigle_valide?(sigle)
    fail "titre incorrect" unless titre_valide?(titre)
    fail "nb_credits invalide" unless nb_credits_valides?(nb_credits)
    fail "prealables invalide" unless prealables_valides?(prealables)
  end
end
```

Classe Ruby 5 Programme.

```ruby
class Programme
  attr_reader :code
  attr_accessor :nom, :nb_credits, :cours

  def Programme.create( code )
    nouveau_programme = new(code)
    yield nouveau_programme
    nouveau_programme.valider

    nouveau_programme
  end

  def initialize( code )
    @code = code
    @cours = []
  end
  private_class_method :new

  def valider
    fail "code invalide" unless code_valide?(code)
    fail "nom invalide" unless nom_valide?(nom)
    fail "nb_credits invalides" unless nb_credits_valides?(nb_credits)
  end

  private

  def code_valide?( code )
    code && code.kind_of?(Fixnum)
  end

  def nom_valide?( nom )
    nom && nom.kind_of?(String)
  end

  def nb_credits_valides?( nb_credits )
    nb_credits && nb_credits.kind_of?(Fixnum) &&
      0 < nb_credits && nb_credits_de_cours == nb_credits
  end

  def nb_credits_de_cours
    @cours.reduce(0) { |tot, c| tot + c.nb_credits }
  end
end
```