# Numérotation des affaires et des documents commerciaux

Les affaires peuvent être numérotées selon un des modèles ci-après :

1. Numérotation séquentielle.
2. N° de l’année et numérotation séquentielle.
3. N° de l’année et numérotation séquentielle par année.
4. N° de client et l’une des variantes 1 à 3, avec séquence globale.
5. N° de client et l’une des variantes 1 à 3, avec séquence propre au client.

Afin de simplifier la production des numéros, on part de l’idée que toute séquence est liée à une clé. Le numéro final produit peut inclure la clé et un préfixe. Ainsi :

1. Clé vide, numéro.  
   0000**1**, 0000**2**, 0000**3**, …
2. Clé vide, préfixe « année » + numéro.  
   2010-000**1**, 2010-000**2** … 2010-000**4**, 2011-000**5**, 2011-000**6** …
3. Clé « année », préfixe « année » + numéro.  
   2010-000**1**, 2010-000**2** … 2010-000**4**, 2011-000**1**, 2011-000**2** …
4. Comme 1 à 3, avec un préfixe « client » supplémentaire.  
   1008/000**1**, 1008/000**2**, 1024/000**3**, 1008/000**4** …  
   1008/2010-000**1**, 1008/2010-000**2**, 1024/2011-000**3**, 1008/2011-000**4** …  
   1008/2010-000**1**, 1008/2010-000**2**, 1024/2011-000**1**, 1008/2011-000**2** …
5. Comme 1 à 3, avec une clé « client » et un préfixe « client » supplémentaire.  
   1008/000**1**, 1008/000**2**, 1024/000**1**, 1008/000**3** …  
   1008/2010-000**1**, 1008/2010-000**2**, 1024/2011-000**1**, 1008/2011-000**3** …  
   1008/2010-000**1**, 1008/2010-000**2**, 1024/2011-000**1**, 1008/2011-000**1** …

Détail d’implémentation : la classe Business.RefIdGenerator génère des numéros séquentiels uniques, propres au générateur défini par un nom. Celui-ci résulte de la concaténation du DRUID de l’entité (par ex. AffairEntity) et de la clé optionnelle.

Les documents commerciaux peuvent reprendre le n° d’affaire comme préfixe ou l’intégrer à leur clé. Ils peuvent aussi inclure un code qui représente le type de document considéré (offre, bon pour commande, confirmation de commande, bulletin de livraison, facture, autre), sous la forme d’un préfixe ou d’une clé.

Détail d’implémentation : si le numéro de document dépend du n° d’affaire, le nombre de documents au sein d’une séquence restera faible. Il n’est dès lors techniquement pas utile de définir un générateur, car l’unicité des numéros au sein d’une affaire peut être garantie par un verrou.

Exemples de n° de documents qui peuvent être produits avec ce schéma (10 = offre, 50 = facture) :

* aaaa.10.01, aaaa.10.02 🡪 variantes 01 et 02 de l’offre
* aaaa.50.01, aaaa.50.01.1, aaaa.50.01.2 🡪 facture initiale 01, 1er et 2ème rappels de la facture
* aaaa.5010, aaaa.5011, aaaa.5012 🡪 facture initiale 01, 1er et 2ème rappels de la facture

L’implémentation actuelle se trouve dans la méthode FormattedIdGenerator.AssignIds… Le format est géré par la classe FormatterHelper (exemples : yy, yyyy, ##, n, nn, nnn, …) et les formats avancés sont fournis au travers de classes qui implémentent IFormatTokenFormatter, par exemple #doc() est implémenté par DocumentInAffairReferenceNumberFormatter.

# Affaire

Une affaire regroupe tous les documents commerciaux liés à une *affaire* (ou un dossier). Dans une affaire, il existe au plus une **confirmation de commande**, mais il peut y avoir plusieurs offres en cours, par exemple.

Chaque document d’une affaire est associé à un identificateur de variante.

# Documents commerciaux

Un document commercial (BusinessDocument) se décompose en plusieurs blocs :

* Les métadonnées du document (type de document, date d’établissement, libellé, n°, auteur, etc.).
* Les informations générales liées au document commercial (client, adresses, comptabilisation, assujettis­sement TVA, mode de facturation HT/TTC, monnaie, etc.).
* Les lignes d’articles y compris titre, textes, sous-totaux, rabais et groupes.
* Le récapitulatif TVA (une ligne par code/taux).
* Le grand total (HT, TTC, prix arrêté[[1]](#footnote-1)).

En outre, dans le cas d’une facture, des informations de paiement (PaymentTransaction) viennent compléter le document à proprement dit (BVR, montant dû, date d’échéance, encaissement, etc.).

## Métadonnées du document

Les métadonnées du document (DocumentMetadata) stockent les données suivantes :

* Numéros de référence (A/B/C).
* Date de création et date de dernière modification.
* Si le document est lié à un fichier : nom, chemin complet et type MIME.
* Nom et description du document.
* Commentaires liés.
* Workflows actifs liés (lien sur une entité Workflow qui contient une liste de *threads*).
* Titre du document (correspond par exemple au *Concerne* d’une facture).
* Catégorie du document permettant de classifier le document lié (facture, offre, etc.).
* État du document : —, brouillon (en cours d’élaboration), actif et inactif. Les états *actif* et *inactif* correspondent à un document non modifiable.
* Lien sur le document réel.
* Collection des versions valides imprimées/conservées du document.

# Documents commerciaux, factures

Une affaire peut comporter zéro, une ou plusieurs factures.

Une facture contient des quantités facturées qui peuvent ou non correspondre aux quantités livrées. Il n’y a pas forcément de lien entre un bulletin de livraison et une facture (mais la facture peut reprendre les quantités d’un ou plusieurs bulletins de livraison).

# Documents commerciaux, demandes d’acompte

Les demandes d’acompte peuvent intervenir au moment de la commande (acompte à la commande), ou pendant toute la phase de *production*, voire après la livraison.

En principe, les modalités de paiement font partie de l’offre acceptée par le client. Elles sont spécifiées sous la forme d’un texte, par ex. « Acomptes : 20% à titre de confirmation de commande, 30% le 01.09.2011, solde à la livraison ».

## Principe comptable

Lors de la comptabilisation d’un acompte, les montants sont ventilés au prorata, article par article, dans les comptes d’acomptes liés aux articles, au moment de l’encaissement. Il peut ainsi y avoir un compte d’acompte pour les marchandises, un compte d’acompte pour les prestations, etc.

*Ceci reste à valider : faut-il comptabiliser au moment de l’émission de la facture d’acompte, puis au moment de l’encaissement. Et aussi au moment de la clôture de la facture ?*

## Facturation avec acomptes

La facturation avec acomptes est difficile à faire cohabiter avec des factures partielles, dans la mesure où le calcul de la valeur facturée pour chaque article est compliqué. Et dès qu’une facture partielle a été émise, l’émission de nouvelles demandes d’acomptes rend le tout encore plus ingérable.

Je propose que l’utilisateur choisisse le mode de facturation souhaité pour l’affaire : facturation d’acomptes puis solde, ou facturation (éventuellement partielle) selon les prestations fournies.

## Calcul des montants à facturer

La facture d’acompte contient toutes les lignes d’articles avec valeur et quantité reprises de la confirmation de commande.

De manière interne, pour la comptabilisa­tion, les montants des lignes d’articles sont calculés au prorata (un acompte de 20% implique que chaque article sera comptabilisé à 20% de sa valeur).

# Documents commerciaux, rappels

Un rappel est un document commercial similaire à une *facture*. Il fait référence à une facture de base, mais ne reprend pas le détail des articles.

Un rappel est donc une facture qui comprend les informations suivantes :

* Texte : libellé, n°, date d’émission de la facture originale.
* Texte : récapitulatif des montants facturés (date d’échéance, HT, TTC et total TVA sans détail).
* Texte : liste des encaissements enregistrés pour la facture (s’il y en a), avec date et montant.
* Optionnellement, une ligne d’article avec frais de rappels (montant HT, TTC et TVA).
* Optionnellement, des lignes avec titre et textes saisis par l’utilisateur.
* Optionnellement, un escompte accordé (faut-il prévoir un article correspondant ?).
* Le récapitulatif TVA (une ligne par code/taux) correspondant aux frais.
* Le montant encore dû qui apparaît en lieu et place du grand total.
* Les informations de paiement relatives au rappel.

La comptabilisation d’une facture de type rappel ne reprend que les lignes d’articles du rappel (dans ce cas, on ne passera en comptabilité que les frais de rappels, la TVA associée à ces frais et un éventuel escompte).

# Documents commerciaux et quantités

Le prix unitaire se calcule en fonction de la **quantité commandée**.

* L’*offre* spécifie la **quantité commandée**, éventuellement déjà des informations de disponibilité ou de date de livraison estimée, au moyen de **quantités en suspens** avec des dates échelonnées.  
  Les quantités en suspens décrivent simplement quand la quantité commandée sera livrée et elles n’interviennent pas dans les calculs.
* Le *bon pour commande* peut être traité soit comme l’offre (sans aucune information supplémentaire liée à des dates de livraison, par exemple), soit comme la confirmation de commande (cf. ci-après).
* La *confirmation de commande* spécifie la **quantité commandée**, les **quantités confirmées** (avec des dates de livraison échelonnées) et la **quantité offerte** (par ex. parce que le client commande 19 pièces et qu’un lot en compte 24, qu’on ne souhaite pas le fractionner et qu’on préfère livrer les 24, en offrant les 5 pièces).
* Le *bulletin de livraison* spécifie la **quantité livrée** et les **quantités encore en suspens** (avec des dates de livraison échelonnées, voire une mention telle que « épuisé »). Aucun prix n’est indiqué.
* La *facture* spécifie la **quantité commandée** (qui détermine le prix unitaire) et la **quantité facturée**. Elle peut rappeler les quantités encore en suspens, déjà facturées ou offertes, à titre d’information.

Pour résumer, les types de quantités suivants doivent pouvoir être différenciés :

* Quantité commandée : ArticleQuantityType.Ordered.
* Quantité facturée : ArticleQuantityType.Billed.
* Quantité confirmée et/ou en suspens : ArticleQuantityType.Delayed.
* Quantité en suspens avec date non confirmée : ArticleQuantityType.Expected.
* Quantité livrée : ArticleQuantityType.Shipped.
* Quantité livrée précédemment : ArticleQuantityType.ShippedPreviously.
* Quantité à titre d’information (par ex. quantité offerte) : ArticleQuantityType.Information.

Le passage d’un type de document à l’autre est géré par la classe AffairActions, via un workflow. Contraire­ment à Crésus Facturation 9, chaque document commercial est une copie complète et indépendante. On copie dès lors l’offre pour en faire un bon pour commande, puis on copie le bon pour commande pour produire la confirmation de commande, etc.

### Établissement de l’offre

Au moment de l’établissement de l’offre, l’interface ne doit pas présenter le détail des quantités. Il faut prévoir une saisie simplifiée directement au niveau de la ligne d’article de la quantité commandée (Ordered).

À l’avenir, il faudra aussi pouvoir spécifier une ou plusieurs quantités en suspens (Expected) avec des dates estimées. Gérer les cas suivants :

* La quantité commandée dépasse les quantités en suspens (ou il n’y a aucune quantité en suspens) : une partie de/toute la commande sera honorée immédiatement.  
  Afficher les quantités livrées de suite s’il y a des quantités en suspens.
* La quantité commandée est plus faible que les quantités en suspens : les quantités seront probablement ajustées au moment de la confirmation de commande.  
  Afficher un avertissement pour attirer l’attention sur ce fait.

La saisie des quantités en suspens pourra se faire dans une tuile séparée, détaillée.

### Édition de la confirmation de commande (éventuellement bon pour commande)

À ce stade, les articles ne peuvent plus être modifiés. Seules les quantités doivent pouvoir être modifiées : la quantité commandée (Ordered) et une ou plusieurs quantités confirmées/en suspens (Delayed), avec les dates correspondantes. S’il y avait des quantités en suspens dans l’offre (Expected), elles doivent être converties en quantités confirmées/en suspens (Delayed).

En principe, la somme des quantités confirmées/en suspens correspond à chaque quantité commandée. Dans un document simplifié, les quantités confirmées/en suspens peuvent ne pas être spécifiées du tout.

La quantité commandée est éditée au niveau de la ligne d’article. Les autres quantités apparaissent sous la forme d’un tableau à *n* lignes. L’interface peut offrir des indications (par ex. total confirmé/en suspens pas égal au total commandé), comme cela est aussi le cas pour l’offre détaillée.

### Établissement d’un bulletin de livraison (BL)

Le bulletin de livraison doit permettre une saisie rapide des quantités livrées (Shipped), en repartant du dernier document produit. La quantité commandée (Ordered) n’est évidemment plus modifiable. Les quantités déjà livrées et décomptées sur des BL précédents (ShippedPreviously) sont maintenues à titre d’information. Les quantités en suspens correspondent au solde à livrer.

### Établissement d’une facture

La facture reprend soit ➀ l’entier des quantités de la confirmation de commande ou ➁ les quantités livrées d’un ou plusieurs bulletins de livraison. Les quantités sont soit facturées (Billed), soit en suspens (Delayed), soit présentes à titre d’information (Information). Pour le calcul du prix unitaire, la quantité commandée est reprise (Ordered), sans que cette information ne doive forcément apparaître dans le document produit.

L’interface permet de comparer aisément ce qui a été commandé, ce qui a été livré, ce qui a déjà été facturé et ce qui doit encore l’être. En particulier, si plusieurs factures doivent être émises, l’utilisateur doit conserver une vue d’ensemble pour éviter des erreurs (double facturation ou oublis).

# Documents commerciaux et lignes d’articles

Le document commercial comprend une collection ordonnée de lignes d’articles (nota bene : c’est un abus de langage, puisqu’il peut aussi s’agir d’autres éléments, tels que des textes, des sous-totaux, des rabais, etc.). Elles dérivent toutes de l’entité abstraite AbstractDocumentItem et peuvent être classées comme suit :

* Article.
* Titre ou texte.
* Sous-total.
* Rabais.
* Taxe (par ex. taxe sur le CO2, RPLP).
* Frais de rappel.
* Frais de port et d’emballage.
* Total.
* TVA.
* Grand total.

Remarques :

* Un groupe de lignes se termine toujours par un sous-total qui peut à son tour être suivi d’un ou plusieurs rabais, et d’un sous-total final.
* Les taxes et les frais sont représentés par des lignes d’articles standards, même si elles sont présentées séparément à l’utilisateur.
* Les lignes de TVA sont produites automatiquement par le calculateur de prix et viennent toujours se rajouter à la suite du grand total.
* Le grand total est unique et vient toujours après toutes les lignes que l’utilisateur peut éditer.
* Les lignes synthétisées par le calculateur de prix se reconnaissent à l’attribut AutoGenerated.

## Attributs

Chaque ligne d’article possède des attributs, définis dans l’énumération DocumentItemAttributes :

* None, pas d’attribut particulier.
* Hidden, ligne cachée (mais prise en compte par le calculateur de prix, cela peut provoquer des surprises chez l’utilisateur— voir s’il ne faut permettre de cacher que des titres, textes et totaux).
* AutoGenerated, ligne produite automatiquement (par le calculateur de prix).
* ProFormaOnly, ligne pro forma, sans incidence sur le prix (elle est ignorée dans les totaux faits par le calculateur de prix).

## Sous-totaux

Une ligne de sous-total dans le BusinessDocument produit potentiellement plusieurs lignes à l’écran ou sur le papier :

* Sous-total avant rabais (PrimaryPriceBeforeTax et PrimaryTax).
* Rabais (Discount).
* Sous-total après rabais (ResultingPriceBeforeTax et ResultingTax).

L’énumération PriceDisplayModes disponible dans le champ DisplayModes spécifie quelles informations doivent être rendues visibles :

* None, aucune information.
* PrimaryTotal, montre le sous-total avant rabais.
* ResultingTotal, montre le sous-total après rabais.
* Discount, montre le rabais.
* Tax, montre les détails de TVA.

## Groupes

Chaque ligne est liée à un groupe (identifié par son **group index**, stocké dans GroupIndex). Ceci permet une organisation visuelle et hiérarchique. L’index encode à la fois le n° du groupe et l’identité du groupe parent, en fonction du niveau d’imbrication (nommé **group level**, calculé à partir de l’index et disponible comme propriété GroupLevel dans chaque entité) :

* 00 correspond à un groupe spécial nommé **section de pied** ou **groupe racine**.
* 01, 02, 03 sont des index des groupes à la racine du document.
* 0101, 0102, 0103 sont des index des groupes contenus dans le groupe 01.
* etc.

Exemple : un index de groupe égal à 20401 (décomposé comme 02 04 01) correspond donc à l’index d’un groupe 01, contenu dans le groupe 04, contenu dans le groupe 02. Le groupe est de niveau 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° de ligne | Group level | Group index | Type | Contenu |
| 1 | 1 | 01 | Titre | … |
| 2 | 2 | 0101 | Article A | 10.– |
| 3 | 2 | 0101 | Article B | 20.– |
| 4 | 2 | 0101 | Article C | 30.– |
| 5.1 | 2 | 0101 | Sous-total groupe 01 01 | 60.– |
| 5.2 | 2 | 0101 | Rabais | 10.– |
| 5.3 | 2 | 0101 | Sous-total récapitulatif 01 01 | 50.– |
| 6 | 2 | 0201 | Titre | … |
| 7 | 3 | 010201 | Article D | 25.– |
| 8 | 3 | 010201 | Sous-total groupe 01 02 01 | 25.– |
| 9 | 3 | 020201 | Titre | … |
| 10 | 3 | 020201 | Article E | 10.– |
| 11 | 3 | 020201 | Article F | 10.– |
| 12 | 3 | 020201 | Sous-total groupe 02 02 01 | 20.– |
| 13 | 2 | 0201 | Sous-total groupe 02 01 (2 sous-groupes) | 45.– |
| 14.1 | 1 | 01 | Sous-total groupe 01 (2 sous-groupes) | 95.– |
| 14.2 | 1 | 01 | Rabais | 5.– |
| 14.3 | 1 | 01 | Sous-total récapitulatif 01 | 90.– |
| 15 | 0 | 00 | Article Frais | 10.– |
| 16 | 0 | 00 | Sous-total (total final avant TVA) | 100.– |
| 17 | 0 | 00 | TVA | 8.– |
| 18 | 0 | 00 | Grand total | 108.– |

1 Titre

2 A 10.–

3 B 20.–

4 C 30.–

5.2 Rabais -10.–

5.3 Sous-total après rabais 50.–

6 Titre

7 D 25.–

8 Sous-total 25.–

9 Titre

10 E 10.–

11 F 10.–

12 Sous-total 20.–

13 Sous-total 45.–

14.1 Sous-total 95.–

14.2 Rabais -5.–

15 Frais 10.–

16 Total HT 100.–

17 TVA @ 8.0% 8.–

**18 Grand total TTC 108.–**

Dans le cas d’une facture où les montants sont TTC dans les lignes d’articles, le pied de la facture se transforme en ce qui suit :

14.1 Sous-total 103.–

14.2 Rabais -5.–

15 Frais 10.–

**16 Total TTC 108.–**

17 TVA @ 8.0% incluse 8.–

Il suffit donc de cacher le grand total dans un tel cas, à moins qu’on ne souhaite forcer un total arrêté.

# Interaction pour la saisie d’articles

Considérations générales :

* Les groupes doivent être représentés visuellement et clairement identifiables.
* L’ajout d’un élément doit se faire en un clic unique : il faudrait des commandes explicites pour permettre d’ajouter groupe, sous-groupe, titre, texte, article, rabais, taxe et frais.
* Les éléments groupe, titre, texte, article et rabais s’ajoutent dans le groupe actif.
* L’élément groupe s’ajoute après le groupe actif.
* Les éléments de sous-total sont ajoutés automatiquement et peuvent être cachés par l’utilisateur.
* Les éléments de sous-total peuvent être ajoutés à la main pour introduire des cascades de rabais.
* Les éléments taxe et frais s’ajoutent à la fin, dans une section de pied.
* Les éléments groupe, titre, texte et article peuvent être déplacés librement, même entre différents groupes. Ils ne peuvent pas être placés dans la section de pied.
* Les éléments rabais, taxe et frais peuvent être déplacés les uns par rapport aux autres.
* Les groupes doivent pouvoir être fusionnés ou scindés.
* Quand un groupe contient un unique sous-groupe, il faudrait cacher le sous-total du sous-groupe pour éviter des doublons dans la présentation de la facture.

Illustration (ligne 5) :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° de ligne | Group level | Group index | Type | Contenu |
| 1 | 1 | 01 | Titre | … |
| 2 | 2 | 0101 | Article A | 10.– |
| 3 | 2 | 0101 | Article B | 20.– |
| 4 | 2 | 0101 | Article C | 30.– |
| 5 | 2 | 0101 | Sous-total groupe 0101 | 60.– |
| 6 | 1 | 01 | Sous-total groupe 01 | 60.– |
| 7 | 0 | 00 | Frais | 10.– |
| 8 | 0 | 00 | Total TTC | 70.– |
| 9 | 0 | 00 | TVA incluse | 5.19 |
| 10 | 0 | 00 | Grand total | 70.– |

Nota bene : dans cet exemple, les montants sont TTC, il y a donc pas lieu d’afficher le grand total et l’indication de TVA est claire sur le fait qu’il s’agit d’une taxe sur la valeur ajoutée *incluse*.

* Les éléments de TVA et le grand total se calculent automatiquement. L’utilisateur ne peut pas intervenir sur les éléments de TVA.
* Le grand total peut être manuellement forcé à un **total arrêté**.

# Documents commerciaux et calculateur de prix

Que la facture contienne des lignes d’articles HT ou TTC n’influe aucunement sur les données stockées. C’est surtout une question d’affichage, dans la mesure où tous les éléments contiennent à la fois les montants HT et TTC. Dans le cas d’un rabais (ou du total arrêté), l’utilisateur doit spécifier s’il s’agit d’un montant HT ou TTC.

Le calculateur de prix est implémenté par la classe DocumentPriceCalculator, mais il n’est jamais appelé directement. Il doit être passé en argument à PriceCalculator.UpdatePrices, et utilisé au sein d’un bloc using. Il est par exemple invoqué automatiquement lorsque BusinessDocumentEntity est modifiée, grâce à une règle de la *business logic* liée.

## Algorithme utilisé par le calculateur de prix

Le calculateur de prix travaille en plusieurs passes :

1. Il trie les lignes du document commercial.
   1. Les lignes sont regroupées par groupe et leur ordre respectif n’est pas modifié.  
      01:a/0101:b/01:c/0101:d/02:e/0201:f/03:g 🡪 01:a/0101:b/0101:d/01:c/0201:f/02:e/03:g
   2. Les sous-totaux sont placés en fin de chaque groupe.
   3. Le groupe zéro (section de pied) est trié de manière spécifique, dans l’ordre suivant :  
      sous-groupes, articles, sous-totaux, lignes de TVA, grand total puis textes éventuels.
   4. Au besoin, le calculateur rajoute ou supprime des sous-totaux, là où cela est nécessaire.  
      🡪 Un groupe contenant des articles ou des sous-groupes non vides a forcément un sous-total.  
      🡪 Un groupe ne contenant aucun montant n’a pas de sous-total.
   5. Pour terminer, le calculateur corrige la section de pied en s’assurant qu’il y ait au moins un sous-total ou en supprimant tous les sous-totaux si le document ne contient aucun article.
2. Il calcule les prix et met à jour le contenu des lignes.  
   Pour cela, il appelle la méthode Process de chaque ligne. Les lignes de type article et sous-total créent un calculateur spécifique et appellent à leur tour DocumentPriceCalculator.Process (il existe une implémentation par type de ligne). Ce processus produit un graphe de calculateurs de groupes qui produisent en fin de compte un total de TVA sur les articles avec rabais possible, un total de TVA sur les articles sans rabais possible et un grand total.
3. Il accumule la TVA et produit les lignes de taxes liées à la TVA, ainsi que le grand total, qu’il place à la fin du groupe racine, avant d’éventuels textes.
4. Il calcule les prix définitifs à utiliser pour la comptabilisation de chaque ligne d’article.

Les lignes d’articles *pro forma uniquement* ne sont pas totalisées. Le calculateur se contente de mettre à jour les prix calculés au sein de la ligne concernée, mais traite sinon l’entier de la ligne comme du texte informationnel.

## Calculateurs de prix pour les lignes en général

Les calculateurs de prix spécifiques aux lignes, dérivant de AbstractItemPriceCalculator, sont définis. Leur responsabilité est de calculer le prix définitif (ApplyFinalPriceAdjustment) :

* ArticleItemPriceCalculator 🡪 lié à une ligne d’article.
* SubTotalItemPriceCalculator 🡪 lié à un sous-total.
* GroupItemPriceCalculator 🡪 lié à un groupe.

## Calculateurs de prix pour les articles

La méthode DocumentPriceCalculator.Process(ArticleItemPriceCalculator) fait ceci, pour autant que l’article ne soit pas de type *pro forma uniquement* :

1. Si c’est le premier article rencontré, conserve la trace du calculateur du groupe en cours (s’il y en avait un), et crée un nouveau calculateur pour le groupe.
2. Ajoute dans la liste des calculateurs le calculateur d’article.
3. Calcule le prix de l’article.
4. Ajoute dans le groupe actif les éventuels calculateurs des sous-groupes accumulés précédemment.
5. Ajoute dans le groupe actif le calculateur d’article.

sinon (si c’est un article de type *pro forma uniquement*), il se contente de reculer le prix de l’article.

Le calcul du prix de l’article (ComputePrice) se déroule en plusieurs étapes :

1. Le constructeur a déjà vérifié si l’article peut ou non bénéficier d’un rabais et mis à jour en conséquence les informations liées à la ligne d’article.  
   Si l’article a des prix gelés (attribut ArticlePricesFrozen), le calculateur se contente de recalculer la TVA, mais ne touche à aucun prix stocké dans la ligne d’article.
2. Détermine le mode d’arrondi à utiliser.
3. Détermine les quantités commandée et réelle.  
   La quantité réelle correspond au nombre d’articles à prendre en compte dans le document concerné. Dans une offre ce sera la somme des articles commandés, dans une facture la somme des articles facturés.
4. Détermine quel prix d’article devra être appliqué (ArticlePriceEntity) en fonction de la quantité commandée, de la date, etc.
5. Calcule le prix unitaire HT.
   1. Si un prix unitaire a été forcé par l’utilisateur (FixedPrice/FixedUnitPrice), il fait foi (le montant est pris HT ou TTC, en fonction de l’attribut FixedPriceIncludesTaxes).
   2. Si un prix a été forcé par la logique métier (ReferenceUnitPriceBeforeTax), c’est ce prix qui sert de base.
   3. Sinon le prix il est déterminé sur la base du prix d’article (qui peut à son tour nécessiter un calcul pour un article paramétré).

Le résultat est finalement arrondi selon le mode d’arrondi en vigueur.  
Le prix unitaire TTC sera stocké dans l’entité pour éviter à avoir à le recalculer à la volée ultérieurement.

1. Calcule le prix de ligne HT (produit du prix unitaire HT et de la quantité réelle).
2. Calcule le prix de ligne HT résultant, après rabais. Les rabais sont appliqués avec ou sans TVA.
3. Calcule la TVA du prix de ligne HT et du prix de ligne HT résultant.
4. Arrondit tous les montants en fonction de la monnaie utilisée.
5. Calcule le ou les taux de TVA appliqué(s) et arrondit ceux-ci de manière à avoir 6 chiffres après la virgule lors d’un affichage en pourcents (8.000000%).

En principe, le prix unitaire devrait être forcé (ReferenceUnitPriceBeforeTax) dès que l’on copie un docu­ment commercial (par ex. une offre pour en faire un bon pour commande) ou après que le document ait été envoyé au client, pour éviter qu’une mise à jour du prix des articles ne modifie un document commercial existant.

En outre, il est possible de geler les lignes d’articles pour éviter que leurs prix ne soient altérés par le calculateur de prix au moyen d’un attribut ArticlePricesFrozen.

Les attributs suivants peuvent être définis :

* NeverApplyDiscount, n’applique jamais de rabais (l’utilisateur peut mettre/enlever cet attribut lui-même).
* ArticleNotDiscountable, la définition d’article prévoit que cet article ne peut jamais avoir de rabais.
* ArticlePricesFrozen, les prix sont gélés et le calculateur de prix n’a plus le droit d’y toucher.
* FixedUnitPriceIncludesTaxes, le prix unitaire forcé est TTC.
* FixedLinePriceIncludesTaxes, le prix de ligne forcé (avant rabais) est TTC.
* DirtyArticlePrices, les prix d’articles stockés dans la ligne ne sont plus à jour parce que l’article a potentiellement changé. On pourrait imaginer qu’un bouton pour *rafraîchir* mette cet attribut à toutes les lignes pour forcer un recalcul complet.
* DirtyArticleNotDiscountable, l’attribut ArticleNotDiscountable doit être rafraîchi.

Les prix suivants sont stockés dans une ligne d’article :

* ReferenceUnitPriceBeforeTax, le prix de référence unitaire HT imposé par la logique métier (par ex. parce que c’est une facture qui reprend le prix unitaire lié à une confirmation de commande).
* PrimaryUnitPriceBeforeTax, le prix unitaire HT, déterminé à partir de l’article ou du prix forcé.
* PrimaryLinePriceBeforeTax, PrimaryLinePriceAfterTax, le prix de ligne HT et TTC résultant du calcul quantité × prix unitaire.
* FixedPrice, le prix forcé par l’utilisateur, HT ou TTC, unitaire ou de ligne.
* ResultingLinePriceBeforeTax, ResultingLineTax1 et 2, le prix total résultant après rabais HT et la TVA associée (deux montants possibles, deux taux).
* FinalLinePriceBeforeTax, le prix final HT pour la comptabilisation des chiffres d’affaires réalisés sur l’article concerné.

## Calculateurs de prix pour les sous-totaux

La méthode DocumentPriceCalculator.Process(SubTotalItemPriceCalculator) fait ceci :

1. Ajoute dans la liste des calculateurs le calculateur de sous-total.
2. Ajoute dans le groupe actif les éventuels calculateurs des sous-groupes accumulés précédemment.
3. Calcule le sous-total en se basant sur le contenu du groupe en cours.
4. Crée un nouveau calculateur pour le groupe, ne contenant que le sous-total.  
   Note : ceci permet de cascader des sous-totaux au sein d’un même groupe, en ayant chaque fois un calculateur de groupe incluant le calculateur précédent, évitant ainsi d’additionner articles et sous-total, alors qu’il suffit de repartir du sous-total en cours.

Ainsi, le calculateur de prix produit un graphe de calculateurs spécifiques liés entre eux.

## Calculateurs de prix et TVA

La complexité des calculateurs spécifiques est liée à la gestion correcte de la TVA et de la ventilation des rabais. La TVA est représentée par la classe Tax qui stocke :

* Le total de la TVA.
* Le total HT sur lequel s’applique la TVA.
* Le détail des composants, sous forme d’une liste de TaxRateAmount, comprenant le montant de TVA et les code de TVA et taux de TVA correspondants.

En effet, un montant de TVA peut être constitué de plusieurs éléments. Par exemple, une prestation de 2000.– facturées à cheval entre 2010 et 2011 à laquelle s’ajoute la livraison pour 100.– de nourriture en 2011 produirait cette information :

* 1000.– @ TVA 7.6% 🡪 76.–
* 1000.– @ TVA 8.0% 🡪 80.–
* 100.– @ TVA 2.5% 🡪 2.50
* Total de TVA 158.50 (taux moyen de 7.5476%)

Il est primordial de conserver ces détails quand la TVA des divers articles est additionnée pour pouvoir produire un récapitulatif en pied de facture.

Quand des montants sont accumulés, il faut distinguer ceux qui peuvent être soumis à un rabais de ceux qui ne peuvent pas l’être. C’est ce que fait le GroupItemPriceCalculator.

Exemple : un groupe avec un total de 1000.– est composé de 800.– d’articles standards et de 200.– d’articles sur lesquels aucun rabais n’est possible. L’utilisateur décide d’accorder un rabais de 10% sur ce groupe. Au moment de la comptabilisation du chiffre d’affaires produit par les articles de la facture, le rabais est traité comme suit :

* 10% de rabais sur 1000.– correspond à 100.–
* Le rabais de 100.– ne s’applique qu’aux articles standards (800.–).
* Les articles standards subissent donc un rabais à hauteur de 12.5%.

Et bien sûr, si un groupe contient un groupe, et que le rabais s’applique à la facture en entier, la ventilation remonte récursivement, en ne s’appliquant que là où des rabais sont possibles. Et si le montant du rabais est spécifié TTC, il faut retrouver le rabais en %.

## TVA en fonction d’une date

La classe TaxCalculator est utilisée par le calculateur de prix des lignes d’articles, en spécifiant soit la date de la prestation, soit la durée (date de début et date de fin) de la prestation.

La méthode TaxCalculator.ComputerTax permet de calculer la TVA (plusieurs taux possibles) en fonction d’un montant HT et d’un code de TVA. Pour cela, les réglages liés à la TVA sont lus dans la base de données.

Afin d’accélérer les accès, un cache rattaché au DataContext est implémenté par le composant TaxContext, lequel conserve un tableau de toutes les définitions de TVA en provenance de la base de données.

# Syntaxe pour le formatage

Le formatage au moyen de TextFormatter reconnaît un certain nombre de commandes :

* « a », *mark*, « x », « y », « z », *clearToMarkIfEmpty* : si « z » est vide, tout ce qui suit *mark* ne sera pas inséré dans le texte.
* « a », *ignore*, « b » : l’élément *ignore* est simplement supprimé — utile uniquement pour l’algorithme interne de formatage, permettant de remplacer à la volée des éléments qui doivent être supprimés en fin de compte.
* « a », « x », *ifEmpty* : si « a » est vide, produit « x », sinon produit « a ».
* « a », « x », « y », *ifElseEmpty* : si « a » est vide, produit « x », sinon produit « a », « y ».
* « x », *format:#func* : applique la commande de formatage *func* à la valeur « x ».  
  Les commandes de formatage sont gérées par des classes implémentant IFormatTokenFormatter.

En outre, les préfixes et suffixes suivants sont reconnus, une fois que toutes les commandes ont été traitées :

* « x », « ~abc » : si « x » est vide, « abc » ne sera pas inséré dans le texte.
* « abc~ », « x » : si « x » est vide, « abc » ne sera pas inséré dans le texte.

Exemples :

* « Poids : », poids, « inconnu », « kg », *ifElseEmpty*  
  🡪 Poids : 10.5 kg  
  🡪 Poids : inconnu
* 25, *format:#string {0:0.0}%*  
  🡪25.0%  
  Le formatage est implémenté par StringFormatTokenFormatter qui accepte comme argument le même type de chaînes de formatage que la fonction string.Format.

## Énumérations

Si l’on désire obtenir le ou les textes associés à une valeur d’une énumération, décrite dans les ressources, il suffit d’écrire EnumKeyValues.GetEnumKeyValue(x). Le type de *x* doit correspondre à une enum décorée avec l’attribut [DesignerVisible].

# Ajouter une base de données dans le ruban

Pour permettre à l’utilisateur de sélectionne une nouvelle « base de données » (data set) dans le ruban, procéder comme suit :

* Pour une entité XxxEntity générant un schéma et réglée avec les attributs RAC (génère un repository, affichage individuel, création individuelle), les noms Xyz ci-après seront la version plurielle du nom Xxx.
* Créer une image avec **Pictogram**, à enregistrer dans « S:\Epsitec.Cresus\Cresus.Core\Images » avec le nom « Base.*Xyz*.icon » (par ex. « Base.Customers.icon »).  
  Les pages avec dimensions 20 × 20 et 31 × 31 doivent exister.  
  En outre, la version 31 × 31 doit exister aussi dans le style *page active*.
* Ajouter cette image au projet « Cresus.Core » : dans le dossier « Images » du Solution Explorer, ajouter le fichier existant et le configurer avec « Build Action = Embedded Resource ».
* Recompiler les assemblies concernées et relancer Designer.
* Dans Designer, définir une commande avec un nom du genre Base.ShowXyz, en prenant garde que le groupe de la commande soit bien « DatabaseSelection », puis enregistrer. Ceci est nécessaire pour que la classe DatabaseCommandHandler puisse identifier correctement la commande.
* Dans le fichier de configuration « Cresus.Core.Library.crconfig », ajouter un nœud <d n="Xxx" …> pour définir une instance de DataSetMetadata.

Il faut ensuite implémenter au moins un *summary view controller* pour l’entité correspondante. Si celle-ci doit pouvoir être éditée, il faut aussi implémenter un *edition view controller*.

# Ajouter un projet dans la solution

Pour ajouter un projet lié à Crésus Core dans la solution :

* Clic droit sur la solution, « Add » et « New Project… ».
* Choisir « .Net Framework 4.0 » dans la combo et sélectionner « Class Library » dans la catégorie « Visual C# ». Nommer le projet Cresus.Core.Library.*Xyz* et valider.
* Supprimer le fichier « Class1.cs ».
* Copier les liens « AssemblyInfo.Solution.cs » et « Epsitec.Cresus.snk » d’un autre projet depuis le dossier « Properties » dans la racine du projet fraîchement créé.
* Dans les propriétés du projet :
  + « Application », changer le « Default namespace » en Epsitec.Cresus.Core.
  + « Signing », cocher « Sign the assembly » et sélectionner le fichier .snk copié précédemment.
* Ajouter les références à « Common », « Cresus.Core.Library », « Cresus.Core.Library.Data », « Cresus.DataLayer », et d’autres en fonction des besoins, supprimer les références pas utiles.
* Déplacer « AssemblyInfo.Solution.cs » et « Epsitec.Cresus.snk » dans le dossier « Properties ».
* Éditer le fichier « AssemblyInfo.cs » pour ne conserver que les attributs suivants :
  + AssemblyTitle, nom du genre « Cresus.Core.Library.*Xyz* ».
  + AssemblyDescription, texte libre (souvent vide).
  + AssemblyProduct, toujours « Crésus ».
  + AssemblyCopyright, notice de copyright standard, « Copyright © 2011, EPSITEC SA, 1400 Yverdon-les-Bains, Switzerland »
* Ajouter dans le projet les dossiers « Resources », comprenant « Cresus.Core.Library.*Xyz* » contenant à son tour « SourceCode ».
* Dans Designer, créer un nouveau module de référence[[2]](#footnote-2), avec le chemin d’accès absolu défini sur « S:\Epsitec.Cresus\Cresus.Core.Library.*Xyz*\Resources », le nom « Cresus.Core.Library.*Xyz* » et le namespace source « Epsitec.Cresus.Core », de type « Application ».
* Déverrouiller le module (Ctrl-L).
* Ouvrir les informations du module (bouton > en-dessus des boutons « Textes », « Légendes », …) et adapter le namespace pour les ressources en « «Epsitec.Cresus.Core.Library.*Xyz* » ; sélectionner l’option « Textes riches ».
* Enregistrer, quitter et relancer Designer.
* Revenir à Visual Studio.
* Ajouter les fichiers « module.info » et tous les fichiers « \*.resource » (« Add » et « Existing Item ») dans « Cresus.Core.Library.Xyz ». Vérifier que « Build Action = Content » et « Copy to Output Directory = Copy if newer ».
* Ajouter les deux fichiers « Entitites.cs » et « Res.cs » dans le sous-dossier « SourceCode » quand ils auront été générés (celui des entités n’est créé que s’il existe au moins une entité dans les ressources). Ces fichiers ont un « Build Action = compile » réglé par défaut et il n’y a donc rien de plus à faire.
* Après avoir créé des ressources dans d’autres langues, il faudra encore ajouter dans le même dossier les fichiers « \*.resource » correspondants (comme ci-dessus).

# Entités à créer

Dans les réglages de l’entreprise :

* Générateur pour les clients (…/…/Customer/A/nnnnnn/ /200000)
* Générateur pour les affaires (…/…/Affair/A/#customer(A).yyyy.nnnn/#customer(A).yyyy/1000)
* Générateur pour les documents (…/…/DocumentMetadata/A/#doc()/ /0)

Il est indispensable d’avoir des workflows définis avant de pouvoir créer un client.

# Composants

## Les composants liés à l’interface graphique

Les composants liés à l’interface graphique dérivent de ViewControllerComponent. Ils sont créés par une *factory* implémentant IViewControllerComponentFactory (pour un composant de type T, c’est en principe une classe Factory dérivant de DefaultViewControllerComponentFactory<T>). L’initialisation de ces composants est gérée par le constructeur du DataViewOrchestrator.

Les composants suivants sont actuellement implémentés :

* DataViewController
* MainViewController
* MainWindowController
* RibbonViewController

Pour retrouver un composant, il faut appeler DataViewOrchestrator.GetComponent<T>.

## Les composants en général

La classe CoreComponentFactory est responsable de trouver et créer les composants (Register­Components), puis de les initialiser (SetupComponents).

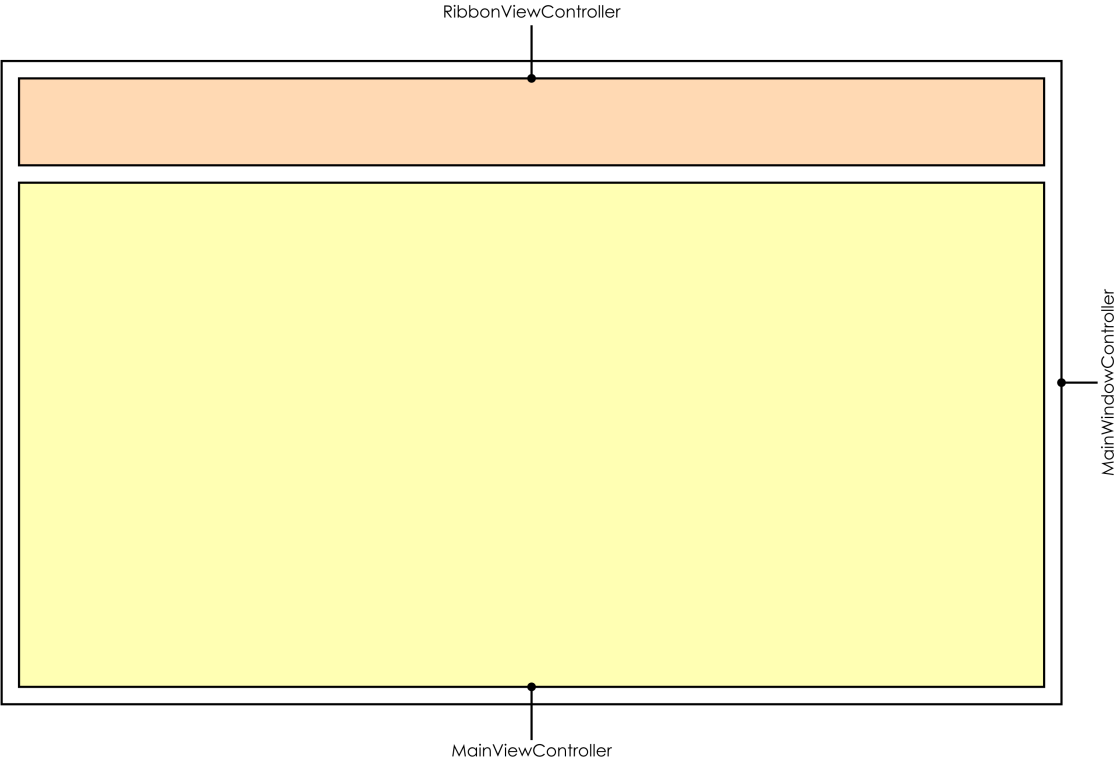
Des implémentations spécifiques de la *factory* sont utilisées pour les types de composants suivants :

* ViewControllerComponent 🡪 ViewControllerComponentFactory.
* CoreDataComponent 🡪 CoreDataComponentFactory.
* CoreAppComponent 🡪 CoreAppComponentFactory.

## Les composants de l’interface graphique

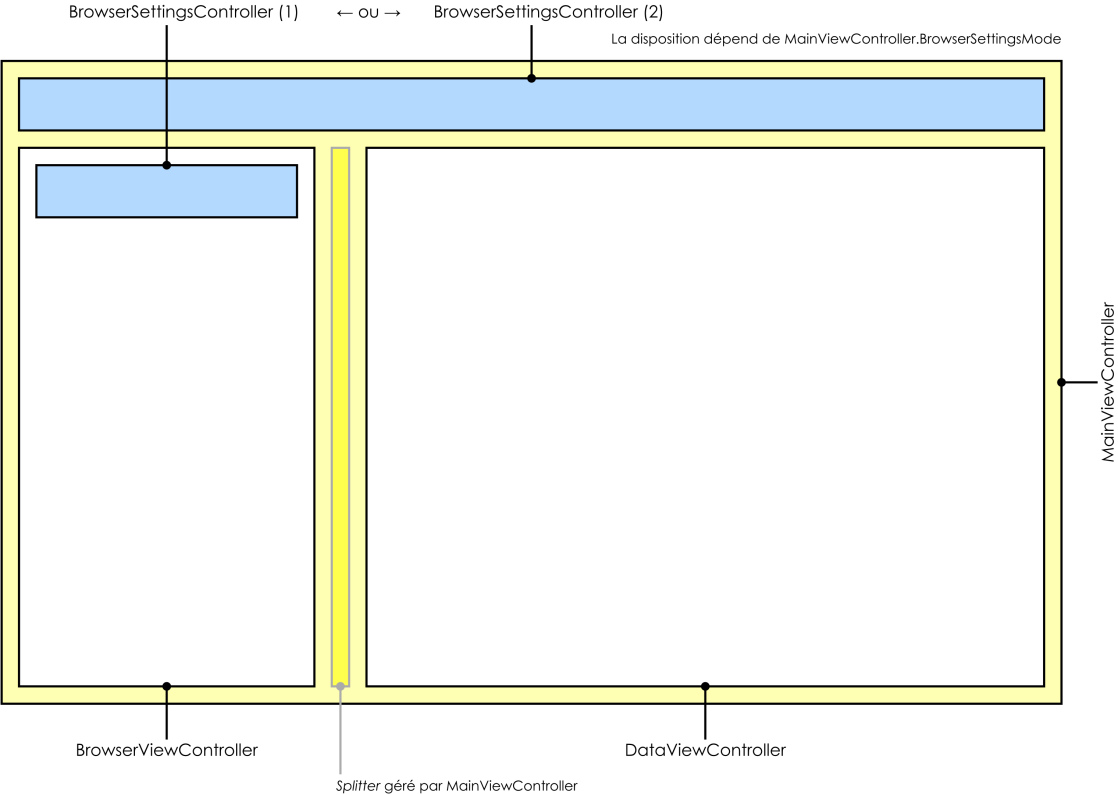
### MainWindowController, contrôleur de la fenêtre principale

La classe MainWindowController intègre dans la fenêtre de l’application un ruban (*ribbon view*) et la vue principale (*main view*).



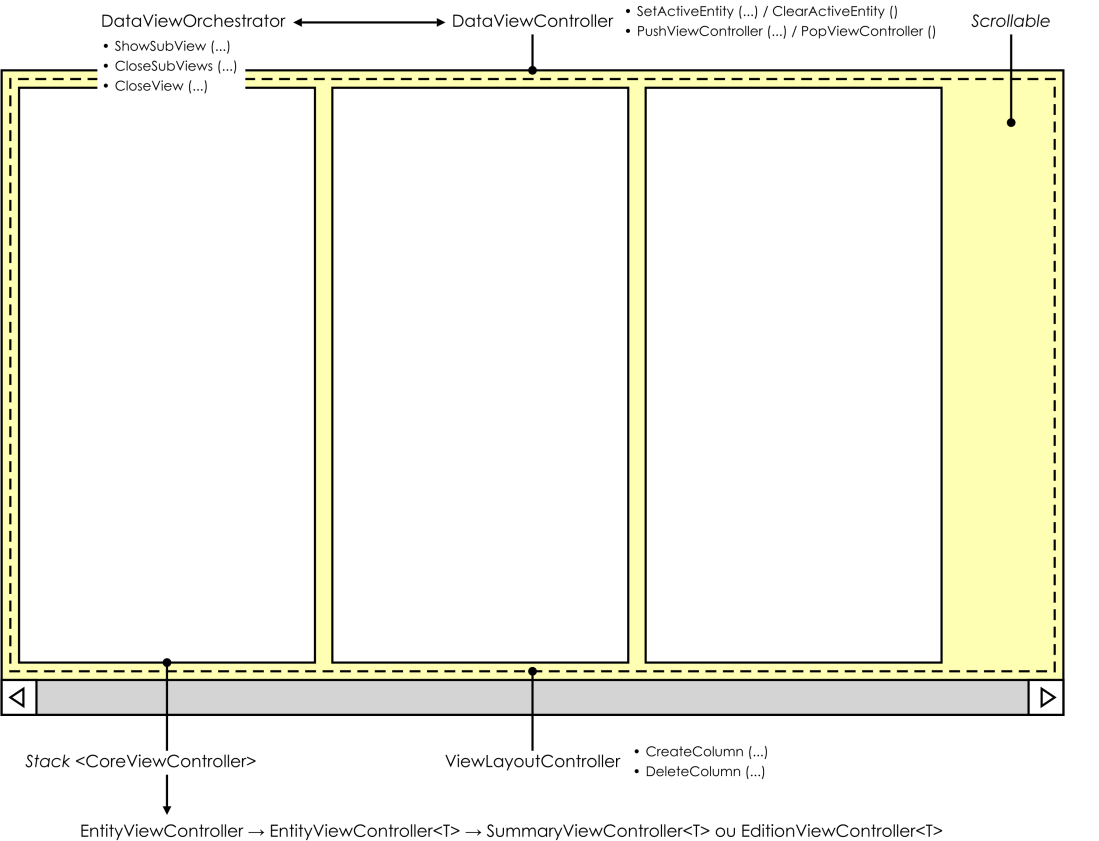
### MainViewController, contrôleur de la vue principale

La classe MainViewController gère les relations entre la liste de gauche et la vue des données (*data view*).



### DataViewController, contrôleur de la vue des données

La classe DataViewController gère la vue où les données sont affichées, en général dans une série de colonnes (d’autres vues sont possibles, par exemple pour l’éditeur de workflows).



Elle comprend :

* Une instance de Scrollable, potentiellement avec des ascenseurs.
* Une instance de FrameBox, remplissant entièrement le *view port* du *scrollable*, mis à disposition en tant que propriété Root.
* Une pile de contrôleurs dérivés de CoreViewController (empilement de gauche à droite de l’écran, colonne par colonne).
* Une instance du ViewLayoutController, responsable de peupler les colonnes de leurs contenus et de créer les instances de TileContainer correspondantes.

En appelant la méthode SetCustomUI, il est possible de forcer l’affichage d’une vue personnalisée, gérée par un contrôleur externe (comme c’est le cas pour le contrôleur d’édition des workflows).

Quand on « ouvre » une nouvelle colonne au moyen du DataViewOrchestrator, le contrôleur correspondant est ajouté au moyen de la méthode PushViewController, laquelle :

* Crée la colonne dans le ViewLayout­Controller; la colonne est représentée par TileContainer, une classe dérivée de FrameBox maintenant une référence à la fois sur le contrôleur associé, dérivé de CoreViewController, et sur le TileContainerController, responsable de la création des tuiles au bas niveau.
* Crée l’interface graphique en appelant CreateUI sur le contrôleur. Ceci va en cascade définir des briques qui permettront de construire les tuiles (cf. BridgeContext) au travers de TileContainer­Controller.
* Crée un lien entre l’entité représentée dans la nouvelle colonne et le BusinessContext, pour s’assurer que les règles métier soient bien appliquées.
* Prend note du changement de colonne dans l’historique de navigation.

### TileContainer, conteneur de tuiles

La classe TileContainer est un widget dérivé de FrameBox. Elle implémente l’interface IWidgetUpdater, ce qui lui permet de se mettre à jour visuellement quand les données représentées changent.

Elle permet de faire le lien entre les éléments suivants :

* EntityViewController.
* TileContainerController.
* La liste des objets implémentant IWidgetUpdater.

### TileContainerController, contrôleur de tuiles

La classe TileContainerController gère la création des tuiles à partir de leur description.

### ActionViewController, contrôleur des actions

La classe ActionViewController gère …

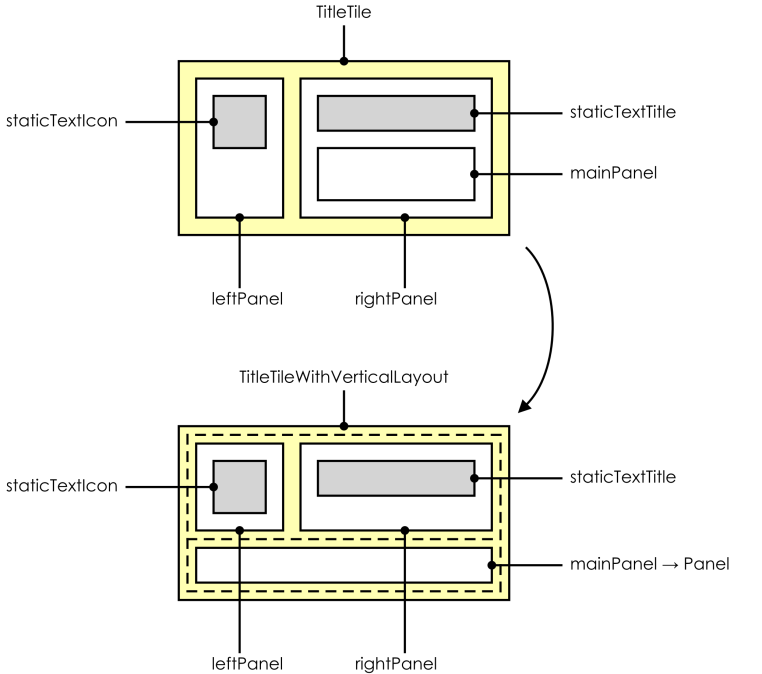
# Éléments de l’interface graphique

## Terminologie utilisée

L’écran est découpé en trois zones principales :

* Le ruban (en haut).
* Le navigateur ou « liste de gauche » (à gauche).
* La vue des données.

La vue des données est organisée en **colonnes** (parfois appelées *views* ou *subviews*). Chaque colonne résulte d’un empilement de **tuiles** (*tiles*), qui peuvent elles-mêmes être composées de sous-tuiles :



# Briques, Bridge et Wall

La classe Bridge<T> — pour un EntityViewController<T> lié à l’entité T — est responsable de la conversion des briques Brick en une collection de TileDataItem, laquelle sert à son tour de base pour créer les tuiles, éléments réels de l’interface graphique.

On distingue donc :

* Les briques (classes dérivées de Brick, dans le namespace Epsitec.Cresus.Bricks) qui décrivent des éléments d’interface graphique élémentaires, liés à un champ d’une entité, au moyen d’une série de propriétés (BrickProperty).
* Le mur de briques (BrickWall) qui est une collection de briques (avec une méthode Add et une propriété Bricks qui donne accès à l’énumération des briques contenues).
* La classe Bridge qui réalise le pont entre mur de briques et description des tuiles (TileDataItem).

Voici dans les grandes lignes le principe de fonctionnement :

1. EntityViewController<T>.CreateUI (Widget) crée une instance de BridgeContext dans un bloc using.
2. À partir de ce moment, le contexte est actif pour le thread concerné (BridgeContext.Instance).
3. La méthode CreateUI () est appelée. Par défaut, elle ne fait rien.  
   Dans de rares cas, elle génère une interface de manière traditionnelle, sans avoir recours aux briques.
4. Lorsque le bloc using se termine, la méthode BridgeContext.Dispose est invoquée.  
   Elle est responsable de la suite des opérations :
5. La méthode surchargée CreateBridgeAndBuildBricks dans EntityViewController<T> crée une instance de Bridge<T>, puis de BrickWall<T>.
6. BrickWall<T> est alors peuplé par la méthode EntityViewController<T>.CreateBricks, surchargée dans chaque contrôleur reposant sur les briques.
7. Enfin, BridgeContext.Dispose se termine en appellant sa méthode CreateTileDataItems pour remplir un TileContainerController avec les TileDataItem correspondant aux briques, puis remet à zéro le contexte actif (initialisé au point 2).

La suite des opérations est entièrement gérée par la classe TileContainerController.

## Définition des briques

Les briques ne sont jamais instanciées directement; elles sont créées au travers de la classe BrickWall<T>, où T est l’entité gérée par le contrôleur. Les briques simples sont créées ainsi :

* wall.AddBrick () 🡪 synonyme de wall.AddBrick (x => x), rattache la brique à l’entité racine.
* wall.AddBrick (x => x.OtherEntity) 🡪 rattache la brique à l’entité fille spécifiée.
* wall.AddBrick (x => x.Items) 🡪 rattache la brique à la collection d’entités spécifiée.

Les briques peuvent ensuite être configurées au moyen d’une API de type *fluent interface[[3]](#footnote-3)*.

* Template 🡪 démarre la définition d’un bloc modèle pour les briques décrivant les éléments d’une collection.
* Input 🡪 démarre la définition d’un bloc pouvant contenir une série de champs de saisie dans un élément d’édition. La présence d’un élément Input détermine que la tuile produite sera de type EditableItem.

## Traitement des définitions des tuiles

Les définitions des tuiles (TileDataItem) peuvent appartenir à différents types (TileDataType) :

* SimpleItem, tuile simple, en lecture seule.  
  Produit des widgets SummaryTile.
* EditableItem, tuile d’édition.  
  Produit des widgets EditionTile.
* CustomizedItem, tuile d’édition avec interface implémentée à la main (type actuellement pas utilisé).
* EmptyItem, tuile vide, sans entité attachée, servant de modèle pour les tuiles d’une collection.  
  Produit des widgets SummaryTile.
* CollectionItem, tuile d’une collection, créée à la volée, à partir d’un modèle vide.  
  Produit des widgets SummaryTile.

La création des tuiles à partir de leurs définitions est gérée par la classe TileContainerController.

## Inclusion

Il est possible d’inclure une interface produite par un contrôleur (de type *édition* uniquement) au moyen d’un élément Include. Celui-ci spécifie le champ qui doit être édité par inclusion de l’interface graphique. Si le champ possède déjà un élément d’interface graphique d’édition dans l’une des tuiles des contrôleurs parents, l’inclusion est simplement ignorée, car elle ferait double emploi.

Si l’inclusion se fait dans une brique pour laquelle aucune propriété n’a été définie explicitement, aucune brique supplémentaire ne sera créée; si, au contraire, la brique qui contient la directive d’inclusion définit un titre ou une autre propriété, une brique sera créée avant l’interface incluse.

# Installation de Firebird

Il faut créer manuellement les dossiers suivants sur la machine qui abrite la base de données Firebird :

C:\ProgramData\Epsitec\Firebird Databases  
C:\ProgramData\Epsitec\Firebird Databases\Snapshots

Le premier sert à stocker les bases de données à proprement parler et le second à stocker des images instantanées de la base, générées au démarrage de l’application par CoreSnapshotService.

# Installation de CORE

Pour que CORE puisse fonctionner, il faut d’une part que Firebird soit installé et démarré, mais aussi qu’un certain nombre de prérequis soient installés, en particulier :

* Le redistribuable Visual C++ 2010 SP1 ().  
  L’installateur vcredist\_x86.exe est sous SVN :  
  S:\Epsitec.Cresus\External\Bootstrapper Packages\vcredist\_x86\
* L’environnement .NET 4.0 complet.  
  L’installateur dotNetFx40\_Full\_x86\_x64.exe est sous SVN :  
  S:\Build\Common\dotnet\dotnet4.0\dotNetFx40\_Full\_x86\_x64.exe

# Installation de CORE/Web

Voici les étapes décrites par Marc le 2 novembre 2011 :

* Installer nginx. L'installeur est disponible sur le web.
* Éditer le fichier de configuration de nginx [NGINX PATH]\conf\nginx.conf pour qu'il ressemble à celui dans S:\Epsitec\experimental-js\doc\nginx\nginx.conf, en particulier il faut reprendre intelligemment les chemins de fichiers.
* Copier le dossier S:\webcore quelque part sur la nouvelle machine, à l'endroit où on a fait pointer le fichier de configuration de nginx.
* Il faut accorder à nginx le droit de passer outre le firewall lors de la première exécution.
* Normalement c'est tout bon et ça devrait marcher, pour autant qu'on lance nginx et l'application crésus.core avec l'argument -server. À ce moment on a accès à l'application web à l'adresse http://localhost:4000. Attention, il semble que IE9 ait des soucis quand on lui donne le port à la fin de l'url et ça ne fonctionne pas avec IE9 mais très bien avec Firefox..

# Cresus.Core.Server et Web

## Contexte d’exécution

La classe CoreContext contient des informations permettant de déterminer si le processus s’exécute dans une application interactive (IsInteractive) ou s’il s’exécute dans le serveur web (IsServer).

# Réflexion sur la notion d’attributs d’entités

Il peut être nécessaire d’associer des informations à diverses entités (par ex. à des clients, affaires, documents, encaissements, adresses, etc.) afin de représenter des tâches en suspens ou des événements liés. Faute de mieux, nous appellerons ces informations des *attributs*, représentées par des entités EntityAttribute.

## Scénario d’utilisation pour la préparation de documents

Pour illustrer l’utilisation d’attributs dans le cas de la préparation de documents, voici un scénario typique :

* L’utilisateur prépare un document (par ex. une offre). Il peut en tout temps en imprimer des brouillons.
* Quand le document est prêt, il clique sur le bouton « Document prêt ».
* Le workflow décide alors soit ➀ de valider et de placer le document dans la file des documents à impri­mer ou ➁ de placer le document dans la file des documents à valider par un supérieur hiérarchique en vue d’un traitement ultérieur.
* L’utilisateur passe à une autre activité.

L’opération « placer le document dans la file … » revient à associer au document un *attribut* avec les informations correspondantes. Il est par exemple possible de décrire les tâches suivantes :

* « Document à imprimer automatiquement, dès que possible » : un processus s’exécutant sur le serveur imprimera le document de manière asynchrone.
* « Document à imprimer et à mettre sous pli par le secrétariat » : le document apparaîtra dans le centre d’impression d’un collaborateur du secrétariat qui en lancera explicitement l’impression.
* « Document à valider par un supérieur hiérarchique » : le document apparaîtra dans la liste des tâches du supérieur hiérarchique, avec la possibilité de l’accepter tel quel, de le modifier, de le renvoyer pour modification ou de le refuser. En fonction du choix, d’autres étapes seront produites.
* « Document à corriger » : le document apparaîtra dans la liste des tâches du collaborateur concerné.

## Lien entre attribut et entité, verrouillage

Le lien est établi entre un attribut et l’entité concernée au travers d’une référence de haut niveau, définie par l’interface IItemCodeUniversalReference, laquelle définit les champs RefCode et RefEntityId; ces champs permettent de référencer une entité implémentant IItemCode.

Le nombre d’attributs associés à un même objet n’est en soi pas limité.

Quand un attribut doit être associé à un objet, il faut au préalable **verrouiller l’objet**, afin d’éviter que des incohérences n’apparaissent dans la base de données, exactement comme lorsque des champs de l’objet concerné sont modifiés par un utilisateur.

## Informations stockées dans un attribut

L’attribut contient les informations suivantes :

* Auteur (utilisateur ayant créé l’attribut) et date de création.
* État de l’attribut : brouillon, actif, inactif.
* Dates de validité (début et fin, les deux sont optionnels).
* Entité cible, référencée au travers de son *item code* et de son *entity id*.
* Utilisateur ou groupe d’utilisateurs concerné.
* Workflow associé.
* Catégorie à laquelle appartient l’attribut.

## Lien avec les workflows

Reprenons l’exemple d’un document marqué comme « prêt » par un utilisateur présenté plus haut. Un attribut contenant les informations suivantes pourrait être créé :

* Auteur : Jean Dupont, le 25 août 2011 à 07 :12.
* Attribut dans l’état « actif », valide à partir de 07:30.
* Attribut lié à l’offre n° 2011-220-002.
* Attribut assigné au groupe d’utilisateurs « impression, mise sous pli et envoi ».
* Attribut lié au thread associé à l’offre au sein du workflow de traitement de l’affaire.
* Attribut appartenant à la catégorie « À imprimer ».

Le thread du workflow concerné est mis dans l’état « restreint » afin d’éviter que l’utilisateur ne continue à manipuler le document. La suite de l’exécution du workflow devient la responsabilité de celui qui va traiter la tâche décrite par l’attribut.

Une secrétaire appartenant au groupe d’utilisateurs « impression, mise sous pli et envoi » verra apparaître à 07:30 dans sa liste des tâches la demande d’impression de l’offre n° 2011-220-002. Elle pourra alors décider d’imprimer le document en exécutant l’étape suivante du workflow, ce qui sera possible car le système lui attribuera le workflow au moyen du champ RestrictedUserCode de WorkflowThread, jusqu’à ce que la partie qu’elle doit traiter se termine. Ensuite, le workflow pourra repasser dans l’état « actif » (à moins, bien sûr, qu’il ne se soit déjà terminé).

L’attribut passera dans l’état « inactif », indiquant qu’il ne nécessite plus aucun traitement.

# Les workflows

…

## Termination

La branche d’un workflow se termine quand le moteur d’exécution arrive sur un nœud qui n’a aucune transition en sortie. Si la branche faisait partie d’une routine, l’exécution du workflow reprendra là où le *call* avait eu lieu, sinon, le workflow sera considéré comme terminé.

## Workflow parallèle (fork)

Une branche d’un workflow peut être lancée en parallèle au moyen d’une transition de type *fork*. La transition (*edge*) s’exécuté sur le compte du nouveau *thread*, avec son propre contexte.

## Interruption prématurée d’un workflow

Il est possible d'interrompre l'exécution d'une étape au sein d'un workflow, simplement en levant une exception :

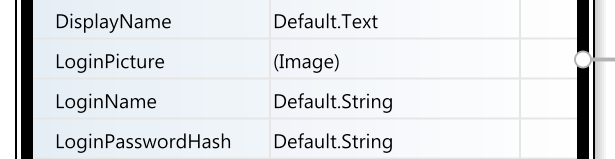
throw new WorkflowException (WorkflowCancellation.Transition);

Les modes d'annulation WorkflowCancellation suivants sont prévus :

* Action, interrompt l’action en cours (utile si c’est une action composée de plusieurs lignes) et continue l’exécution.
* Transition, interrompt la transition en cours (effectivement, ne passe pas au nœud suivant).
* Routine, interrompt la routine en cours d’exécution et rend la main à l’appelant, comme si on avait atteint la fin de la routine.
* Thread, interrompt le thread complet du workflow, si le workflow contient d’autres threads actifs, ceux-ci continuent leur exécution.
* Workflow, interrompt tout le workflow, y compris les autres threads actifs.

# Les champs de type texte

Les champs de type texte peuvent correspondre à du texte simple (type string) ou à du texte riche, potentielle­ment formaté et multilingue (type FormattedText). Le type utilisé dans la définition du champ dans le schéma est déterminant. Default.String correspond au texte simple et Default.Text au text riche :



Pour les champs de type string, le texte est stocké sans encodage, alors que pour le type FormattedText, le texte est stocké avec un encodage de type XML. Ainsi, le texte « B&B » sera stocké :

* "B&B" pour un champ de type string.
* "B&amp;B" pour un champ de type FormattedText.

Il en va de même pour les sauts de ligne (soit \n, soit <br/>, en fonction du type de champ utilisé).

En accédant aux champs via les *marshalers* (GetFormattedTextValue et SetFormatted­Text­Value), les conversions correctes ont lieu automatiquement. Ceci permet de travailler avec FormattedText au niveau de l’interface graphique, sans avoir à se soucier du format interne du texte.s

# Les convertisseurs et les validateurs

Les champs éditables de l’interface graphique sont liés aux propriétés des entités au travers de *marshalers*, responsables des opérations suivantes :

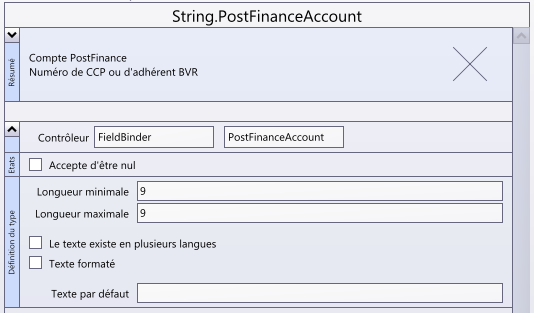
* Accès en lecture et écriture à la propriété de l’entité, au travers de *lambdas* getter et setter, qui peuvent opérer sur une valeur nullable ou non nullable.
* Conversion de la valeur de et vers une string, vérification que la conversion est possible (par ex. au moyen de convertisseurs tels que DateTimeConverter).

Quand un élément d’interface graphique doit être connecté à un *marshaler*, on utilise généralement la classe TextValueController. Ce contrôleur est utilisé par UIBuilder, au travers des méthodes Create­Text­Field, CreateTextFieldMulti, CreateAutoCompleteTextField et CreateCheckBox. La méthode Attach du contrôleur fait le lien avec le *widget* sous-jacent.

Si la propriété de l’entité est éditable au moyen d’un champ de texte, le contrôleur délègue une éventuelle adaptation de la valeur à un *field binder*. Celui-ci est obtenu au travers de la classe FieldBinderFactory, en s’appuyant sur la définition du type de propriété (si INamedProperty définit un contrôleur par défaut nommé « FieldBinder », alors la *factory* recherche une classe concrète implémentant IFieldBinder).

L’interface IFieldBinder met à disposition les méthodes de conversion ConvertToUI et ConvertFromUI, ainsi qu’une méthode de validation ValidateFromUI. La méthode FieldBinderFactory.GetValidator peut être utilisée pour obtenir un validateur compatible avec les mécanismes utilisés par Widget.

Il est ainsi possible de stocker un no de compte de chèque sous une forme compacte « 10150006 » dans l’entité et de l’avoir représenté sous sa forme humainement plus lisible, « 10-15000-6 ». C’est le cas pour les champs de type String.PostFinanceAccount, qui sont adaptés par la classe PostFinanceAccountFieldBinder :



Les valeurs numériques et les pourcentages sont aussi formatés automatiquement au moyen d’un binder spécifique implémenté par la classe NumericFieldBinder.

## Les dates et heures

Les notions de date, heure et date & heure sont représentées au moyen des types Date, Time et DateTime. Les deux premiers types n’ont pas de notion de fuseau horaire, alors que DateTime est toujours stocké dans la base sous la forme d’un temps universel (UTC).

Le convertisseur DateTimeConverter traduit de manière transparente la date et l’heure depuis le fuseau local vers le temps universel, et vice-versa.

# Logique des règles métier (Core.Business.Logic, Core.Business.Rules)

Lors des diverses étapes de vie d’une entité, un certain nombre de règles métier doivent pouvoir s’appliquer, afin de garantir que les données sont dans un état cohérent :

* RuleType.Setup, initialisation de l’entité.
* RuleType.Bind/Unbind, mise en relation d’une entité avec l’interface graphique.
* RuleType.Update, mise à jour des données de l’entité (après une modification).
* RuleType.Validate, validation des données de l’entité (au moment du SaveChanges).

L’exécution de ces règles est gérée par la classe Core.Business.Logic, laquelle maintient un contexte statique spécifique au thread en cours d’exécution. Il est par exemple possible de retrouver les composants actifs au moyen d’un appel à Logic.Current.GetComponent<T>().

C’est généralement BusinessContext qui appelle la logique lorsque des modifications sont détectées ou lorsque des entités sont créées.

Ces règles sont aussi appelées, au besoin, avant d’enregistrer les modifications avec SaveChanges :

* Une phase Update si des modifications ont eu lieu qui n’ont pas encore été vérifiées de manière asynchrone.
* Une phase Validate pour permettre de vérifier la cohérence d’ensemble des données.

Les règles sont implémentées dans des classes dérivées de GenericBusinessRule<T> et décorées au moyen de l’attribut [BusinessRule]. Si une entité implémente une interface (par exemple IItemCode), alors les règles liées à celle-ci sont aussi automatiquement appliquées.

# Liste d’entités pour l’utilisateur (liste de gauche, browser)

La **liste de gauche** présente un extrait partiel ou complet d’une collection d’entités. D’autres éléments d’interface doivent aussi savoir afficher une liste pour permettre de choisir une entité avec un champ de filtrage.

La liste de gauche est articulée autour des classes suivantes :

* BrowserViewController gère la liste de gauche au plus haut niveau. Il est réalisé sous la forme d’un composant rattaché à CoreApp, créé au démarrage de l’application par le MainViewController.
* BrowserListItem représente une ligne de la liste de gauche (entité et texte court associé).
* BrowserViewController.ItemCreator gère la création d’entités (après avoir passé par un éventuel contrôleur de création propre aux entités sélectionnées).
* BrowserViewController.BrowserNavigationPathElement représente un élément de chemin pour l’historique de navigation. La méthode Navigate permet de réactiver la liste de gauche dans un état mémorisé.

Les réglages et boutons d’action de la liste de gauche sont gérés par le contrôleur Browser­Settings­Controller, aussi créé par MainViewController. Le conteneur dans lequel les éléments d’interface sont construits est sous le contrôle de MainViewController et peut changer de taille et d’emplacement en fonction de BrowserSettingsMode.

## BrowserViewController

La classe BrowserViewController met à disposition les fonctions suivantes :

* SelectDataSet : choix du *data set* à afficher. Ceci sélectionnera une collection d’entités d’un même type. À partir du nom du data set (par ex. "Customer"), le type des entités à représenter (Customer­Entity) est déterminé en faisant appel à la classe DataSetGetter, puis la liste est rafraîchie.
* SelectEntity : sélectionne une entité de la liste afin d’afficher ses détails.
* AddNewEntity : crée une nouvelle entité, avec le type lié au *data set*.
* DeleteActiveEntity : supprime l’entité sélectionnée.

Les événements suivants sont produits :

* CurrentChanging et CurrentChanged, suivi de l’élément actif dans la liste, lorsque l’utilisateur clique sur un élément.
* DataSetSelected, un nouveau *data set* a été sélectionné.

La liste contient un certain nombre d’éléments d’interface graphique qui permettent de créer ou supprimer une entité, ou d’effectuer des réglages avancés (éléments gérés par le BrowserSettingsController). En mode compact, ces éléments d’action sont placés dans un *frame* accessible au travers de la propriété TopPanel.

La gestion de la liste (affichage, remplissage, navigation) est déléguée au BrowserListController. En fait, l’affichage proprement dit est géré par la classe ItemScroll­List du projet *big list*.

## BrowserListController

La classe BrowserListController gère la logique associée à l’affichage et à la sélection des éléments de la liste. Les données sont extraites de la base de données par le BrowserListItemProvider, puis préparées par le Browser­ListItemMapper et enfin affichées par le BrowserListItemRenderer, au travers d’appels à BrowserListItem.GetDisplayText().

Les détails de la gestion de l’affichage et du cache sont gérés par le projet *big list*.

De manière interne, l’entité sélectionnée est identifiée au moyen d’un EntityKey, ce qui garantit que l’on reste indépendant du *data context* utilisé. Il est donc théoriquement possible de remplacer le *data context* pendant la durée de vie du contrôleur.

## Accès aux données de la base

L’accès aux données de la base passe par un accesseur de bas niveau, le DataSetAccessor, lequel construit les requêtes pour le *data layer* (RequestView) et permet de récupérer une liste triée des clés d’entités, laquelle permet de ensuite de produire les instances BrowserListItem.

Le DataSetAccessor implémente les méthodes suivantes :

* GetItemCount, pour obtenir le nombre total d’entités.
* IndexOf, pour récupérer l’index d’une clé primaire dans la liste triée.
* GetItemKeys, pour obtenir une tranche de la liste, sous forme de tableau de clés primaires.

L’accès à la base se fait dans un environnement isolé (IsolatedTransaction encapsule une transaction de longue durée, utilisée en lecture seule), ce qui garantit qu’une lecture séquentielle reste cohérente, même si la base de données est modifiée pendant l’accès (qui peut durer potentiellement des heures : tant que l’utilisateur ne modifie rien, ni ne change de base visible, l’accesseur peut conserver sa transaction).

### DataContext utilisés

Deux contextes totalement isolés sont utilisés en relation avec la liste de gauche :

1. BrowserListController utilise un DataContext isolé pour établir la correspondance entre EntityKey et entité.
2. DataSetAccessor utilise un DataContext isolé en relation avec le RequestView.

Dans la mesure où ces contextes sont isolés, toute modification ou mise à jour d’une entité dans l’application ne sera pas répercutée sur le contenu de la liste de gauche. Un *refresh* complet sera alors nécessaire, de toute manière, dans la mesure où la modification peut avoir des répercussions sur le tri de la liste.

## Extraction et tri selon EntityData… [obsolète]

Quand il faut trier des entités selon le contenu de leurs champs, le système d’extraction transforme d’abord toutes les données intéressantes des entités vers string (et si possible aussi vers long). Après cette trans­formation, le tri devient très simple, puisqu’il n’y a plus à se soucier des types de données réels.

Pour définir les colonnes utiles au tri, il faut procéder comme suit :

* Créer une instance de EntityDataMetadataRecorder<T> pour l’entité de type T.
* Au moyen de rec.Column(), spécifier l’accesseur pour chaque colonne.
* Appeler rec.GetMetadata() pour obtenir un EntityDataMetadata, qui est juste un *wrapper* autour d’une collection de EntityDataColumn.
* Créer une instance de EntityDataExtractor à partir de l’instance EntityDataMetadata.
* Lui demander de créer une collection EntityDataCollection avec tri par défaut.

On pourra ensuite appeler EntityDataExtractor.Fill() pour peupler une collection de lignes avec les données en provenance d’entités, dans l’ordre défini par le tri appliqué. Ces lignes sont représentées au moyen de EntityDataRow.

Les collection EntityDataCollection ne peuvent être modifiées que par l’entremise de l’extracteur, lequel conserve une liste de toutes les collections actives. Les insertions de données par la suite se font tout de suite à la bonne position (recherche par bissection).

Pour l’instant, EntityDataRow a toujours une référence sur l’entité correspondante, mais on pourrait imaginer qu’à terme, avec l’introduction de caches, l’on ne stocke qu’une clé d’entité en lieu et place, et que la résolution vers l’entité réelle se fasse uniquement à la demande.

## Détails d’implémentation [obsolète]

### Transformation des champs en données de tri

La classe EntityDataConverter permet de produire des fonctions de conversion à partir d’une *lambda* définissant l’accès à un champ d’une entité. La méthode GetFieldConverter retourne ainsi pour chaque champ une instance de EntityDataColumnConverter, sur laquelle on peut ensuite appeler GetText ou GetNumber pour récupérer la valeur du champ sous forme textuelle ou numérique.

# Système d’impression pour les entités

Chaque entité est imprimée par une classe dérivée de Epsitec.Cresus.Core.Print.AbstractPrinter. Ces classes sont nommées des *entity printers* et sont des moteurs d’impression. Voici quelques exemples :

* MailContactPrinter, dans le projet Cresus.Core.Library.Address, opère sur MailContact.
* RelationPrinter, dans le projet Cresus.Core.Library.Business, opère sur Relation et Customer.
* BusinessDocumentPrinter, dans le projet Cresus.Core.Library.Business, opère sur toutes les instances de DocumentMetadata liées à un BusinessDocument.

Les *entity printers* sont déterminés par réflexion au travers du EntityPrinterFactoryResolver. Chaque classe dérivée de AbstractPrinter contient une classe Factory implémentant IEntityPrinterFactory, qui permet d’obtenir des informations sur les types de documents supportés, les options requises, les types de pages requis, etc.

# Accesseurs pour les lignes d’un document

Un document commercial contient des lignes d’articles. Lorsqu’il faut les représenter (dans un éditeur ou pour l’impression), on passe par des accesseurs, implémentés par DocumentItemAccessor. L’accesseur contient la logique nécessaire pour produire des lignes *physiques* à partir des lignes logiques du document de départ (un sous-total peut ainsi produire 3 lignes physiques : prix avant rabais, rabais et prix après rabais).

L’appelant interroge chaque accesseur pour obtenir les informations utiles :

* RowsCount, nombre de lignes physiques pour la ligne de document concernée.
* GroupIndex, index de groupe pour ces lignes.
* GetContent(row, column) 🡪 texte formaté représentant une cellule précise, identifiée par l’index de la ligne physique (0..RowsCount-1) et la colonne DocumentItemAccessorColumn.

Les utilisateurs de ces accesseurs sont actuellement :

* BusinessDocumentLinesController, contrôleur spécifique qui représente un tableau de lignes et gère l’édition de leur contenu; il est instancié par le SpecialBusinessDocumentLinesController.
* BusinessDocumentPrinter, moteur d’impression pour les documents commerciaux.

# Éditeur de lignes d’un document

L’éditeur de lignes d’un document commercial est composé de plusieurs contrôleurs. Au niveau le plus haut se trouve le SpecialBusinessDocumentLinesController, instancié dans une colonne (plus large que les autres) par le système de briques.

Il délègue toute l’édition à proprement dite à BusinessDocumentLinesController qui vit dans son propre namespace, BusinessDocumentControllers et s’appuie sur les contrôleurs suivants :

* LineToolbarController, le contrôleur pour la barre d’outils spécifique à l’éditeur (cette barre d’outils n’est pas nécessairement visible, à ne pas confondre avec les commandes du ruban).
* LineTableController, le contrôleur qui gère et représente la table (CellTable) des lignes.
* LineEditionPanelController, le contrôleur qui abrite le panneau d’édition dans lequel l’utilisateur va pouvoir modifier le contenu de la ligne sélectionnée.

C’est LineEditionPanelController.UpdateUI, appelé quand une ligne est sélectionnée, qui va peupler le panneau d’édition au moyen de l’éditeur approprié (article, quantité, sous-total avec rabais, etc.). L’éditeur est implémenté dans le namespace BusinessDocumentControllers.LineEditorControllers.

# Options de la ligne de commande

Cresus CORE peut être démarré avec les options suivantes :

* -db-maintenance, relit (et écrase) les pays et les localités suisses.
* -start, démarre sans afficher de splash, utile si le splash a été affiché par ailleurs.

# Représentation de grandes listes — BigList

Sous l’appellation BigList se retrouvent toute une série de classes qui permettent de gérer des grandes quantités de données destinées à être affichées dans une liste, par ex. dans la liste de gauche de l’application. L’architecture de BigList table sur un découpage très granulaire, afin de permettre une réutilisabilité maximale. Elle s’articule autour de trois types de composants : visuels, représentant les données et implémentant des compor­tements.

## Composants de haut niveau

### Classes

* Widgets.ItemScrollList est un widget qui affiche une liste de lignes, au moyen d’une ou de deux *ouvertures* distinctes (implémentées au moyen de VSplitView). Il repose sur des composants visuels plus simples, décrits ci-après.

La liste gère un ascenseur vertical et un en-tête pour identifier les colonnes.  
Propriétés :

* + ActiveIndex, index de la ligne active.
  + Features, caractéristiques/réglages de la liste.
  + Header, en-tête de la liste.
  + Selection, sélection des lignes de la liste.

Événements :

* + ActiveIndexChanged, changement de l’index de la ligne active.
  + SelectionChanged, changement au sein de la sélection.

Les autres informations sont encapsulées de manière cachée et en principe pas nécessaires pour l’utilisation de haut niveau.

## Composants visuels

### Classes

* ItemListVerticalContentView est un widget qui affiche une liste de lignes. Les éléments sont stockés dans un ItemList. Le rendu des lignes est délégué à un IItemDataRenderer et le rendu des marques est délégué à un IItemMarkRenderer. Le comportement de la liste se configure au moyen d’objets dérivés de la classe EventProcessorPolicy.
* ItemListColumnHeaderView est un widget qui affiche l’en-tête d’une liste. Les colonnes sont stockées dans un ItemListColumnCollection (qui est une version spécialisée d’une liste de colonnes définies par ItemListColumn). Le comportement de l’en-tête se configure au moyen d’objets dérivés de la classe EventProcessorPolicy.
* …

### Interfaces liées

* IItemDataRenderer implémente une méthode Render qui reçoit les arguments suivants :
  + ItemData, donnée à représenter.
  + ItemState, état de la ligne (sélectionnée, active, marquée, etc.).
  + ItemListRow, définition de la ligne (index de la ligne, offset depuis le haut de la partie visible, hauteur et marges).
  + Graphics et Rectangle, le port graphique et la zone de clipping.
* IItemMarkRenderer implémente une méthode Render qui reçoit les arguments suivants :
  + ItemListMark, définition de la marque (point d’attachement, index de la ligne concernée et épaisseur).
  + ItemListMarkOffset, l’offset depuis le haut de la partie visible.
  + Graphics et Rectangle, le port graphique et la surface à peindre.
* …

## Composants représentant les données

### Classes principales

* ItemList<TData> encapsule la liste de données. La classe gère les notions de sélection, d’élément actif, d’élément qui a le focus et de la plage d’éléments visible. Les éléments sont référencés au moyen d’un index de 0 à n. Le scrolling de la plage d’éléments visible se fait en points et est découplé de l’index du premier élément visible.

La classe a les propriétés suivantes :

* + Features, réglages spécifiques.
  + Marks, liste de marques liées aux lignes.
  + Count, nombre total de lignes.
  + Selection, gestion de la sélection des lignes.
  + VisibleFrame, gestion de la visibilité du contenu.
  + ActiveIndex, FocusedIndex, index de la ligne active et de la ligne qui a le focus. Ce n’est pas forcément la même (par ex. si on se déplace avec Ctrl-flèche haut/bas, la ligne qui a le focus se déplace, mais la ligne active reste inchangée).
  + Cache, le cache des lignes qui se peuple au fur et à mesure de l’utilisation de la liste.

Les méthodes GetItemState et SetItemState permettent de manipuler l’état d’une ligne de manière ciblée, par exemple pour la cacher (Hidden), la mettre en évidence (Hilite1 et Hilite2), voire lui associer des données complémentaires.

* ItemListCollection<TData> encapsule plusieurs instances ItemList sous un même toit et s’assure qu’elles soient toutes synchronisées (elles partagent le même cache, la même collection de marques et la même sélection). Cette classe est utile pour ItemScrollList, quand deux vues sont actives en même temps pour une seule source de données.
* ItemListColumn définit une colonne avec les propriétés principales suivantes :
  + Layout, largeur et élasticité.
  + Title, le texte (formaté) à afficher.
  + SortOrder et SortIndex, l’ordre de tri de la colonne, si la colonne intervient dans le tri.

L’ordre du tri est géré par la méthode ItemListColumnCollection.SpecifySort, responsable de renuméroter les SortIndex. Il est en effet important que les colonnes puissent être listées de manière cohérente, puisqu’elles interviennent en principe toutes sur le tri final d’une ligne.

* ItemListMark, définit une marque avec les propriétés suivantes :
  + Attachment, marque attachée avant ou après.
  + Index, ligne à laquelle la marque doit être rattachée.
  + Breadth, épaisseur de la marque.
* ItemListSelection, définit la sélection des lignes d’une liste :
  + SelectedItemCount, nombre de lignes sélectionnées.

Les méthodes Select et IsSelected permettent de manipuler la sélection des lignes. En fonction du mode de sélection défini par ItemListFeatures.SelectionMode, le comportement de la sélection sera différent (toujours une ligne sélectionnée, plusieurs, etc.).

* ItemListVisibleFrame, définit la partie visible d’une liste :
  + VisibleIndex, VisibleOffset, VisibleHeight, VisibleCount, VisibleRows…
  + L’événement VisibleContentChanged.

Les méthodes associées permettent de modifier la plage visible, par ex. pour implémenter le scrolling relatif (en pixels) ou la navigation ligne par ligne.

### Classes de bas niveau

* ItemCache<TData, TState> donne accès aux données à représenter dans une liste. Les données sont stockées dans le cache, ainsi que l’état de chaque ligne de la liste. Une représentation ultra-compacte permet de n’utiliser que 16 bits par ligne, pour toutes les lignes qui n’ont encore jamais été affichées (codage de la hauteur de la ligne et de son état : sélectionné/caché/mis en évidence/…).

Les Features sont partagées avec la classe ItemList<TData>.

Les données arrivent dans le cache au travers des interfaces attachées aux propriétés DataProvider et DataMapper (voir plus loin).

* ItemData<TData> encapsule une donnée brute (de type TData, par ex. une entité en provenance de la base de données) et un état associé, représenté par ItemState.
* ItemState définit l’état d’un élément, ainsi que la place nécessaire (hauteur, marges, padding). Dans les cas simples, cet état peut se comprimer et être représenté par un mot de 16 bits dans le cache.
* ItemListRow représente la position d’une ligne dans la liste, à savoir : son index, son offset et sa hauteur totale, ainsi que des indications supplémentaires (première ligne, dernière ligne, ligne paire ou impaire).

### Interfaces

* IItemList, interface d’accès à une liste d’éléments. Les propriétés et méthodes de gestion de la ligne active et de la ligne qui a le focus sont implémentées dans AbstractItemList qui sert de base à ItemList, qui à son tour est la base de ItemList<T>.
* IItemDataProvider<TData> implémente une propriété Count et une méthode Resolve. Celle-ci permet à partir d’un index, d’obtenir la valeur correspondante dans la source de données (par ex. une entité depuis la base de données).
* IItemDataMapper<TData> implémente une méthode Map qui permet de convertir une valeur brute de la source de données en une instance de ItemData<TData>. C’est au moment de cette conversion que l’état de la ligne est déterminé (calcul de la hauteur nécessaire, par ex.).

## Événements

La gestion des événements clavier et souris n’est pas implémentée directement dans les widgets de BigList, mais reportée dans des classes de traitement d’événements (ViewEventProcessor), dont le but est de faire le lien entre les processeurs spécialisés (par ex. BigList.Processors.MouseDownProcessor) et la vue manipulée.

### Classes de traitement d’événements

Ces classes implémentent toutes au minimum IEventProcessorHost. Elles implémentent les interfaces I…Processor en fonction de leurs capacités.

* ItemListColumnHeaderView.ViewEventProcessor gère les événements pour un en-tête de liste.
* ItemListVerticalContentView.ViewEventProcessor gère les événements pour une liste verticale. La gestion de la roulette est faite directement; les autres événements sont transmis à des processeurs spécialisés.

### Classes des processeurs spécialisés

* EventProcessor est une classe abstraite dont dérivent tous les processeurs spécialisés. Elle implémente IEventProcessor.
* KeyDownProcessor traite les touches pressées, par ex. pour provoquer un défilement (touches Home/End, flèches, Page Up/Down, avec ou sans Ctrl, Shift).
* MouseDownProcessor traite les boutons pressés, par ex. pour sélectionner une plage de lignes, avec scroll automatique en extrémité de zone visible.
* MouseDragProcessor traite les opérations de drag & drop de manière générique.

### Interface implémentée par les processeurs spécialisés

* IEventProcessor implémente deux méthodes :
  + ProcessMessage, traite un événement utilisateur et signale s’il a été consommé.
  + PaintOverlay, peint un feed-back visuel, au besoin.

### Interfaces à disposition des processeurs spécialisés

* IEventProcessorHost lie un ViewEventProcessor avec les processeurs spécifiques. Un hôte implémente les méthodes Add/Remove, une propriété EventProcessors qui permet d’obtenir la liste de tous les processeurs actifs et une méthode GetPolicy qui permet d’obtenir l’EventProcessor­Policy du type spécifié.
* IDetectionProcessor implémente la méthode Detect qui à partir d’un point retourne l’index de l’objet détecté, ou -1.
* IDraggingProcessor implémente deux méthodes :
  + DetectDrag, détermine la liste des cadres (MouseDragFrame) qui correspondent à la position donnée. Si l’on vise par ex. entre deux en-têtes de colonnes, il y aura deux cadres à considérer.
  + ApplyDrag, déplace un cadre de sa position originelle à sa position courante.
* IScrollingProcessor implémente la méthode Scroll qui permet de faire défiler le contenu avec une amplitude [x, y], exprimée en pixels/lignes/pages/taille totale de document. Le mode de défile­ment détermine ce qui doit être déplacé (la partie visible, le focus, la ligne active, ou la ligne active et sélectionnée).
* ISelectionProcessor implémente deux méthodes :
  + IsSelected, détermine si pour un index donné, l’objet est sélectionné ou non.
  + Select, change l’état de sélection d’un objet déterminé par son index (le mode de sélection spécifie s’il faut sélectionner/désélectionner/inverser la sélection ou activer/mettre le focus sur l’élément considéré).

### Classes de bas niveau

* MouseDragFrame représente un cadre lors d’une opération de drag & drop :
  + Index, index de l’objet considéré.
  + Grip, poignée du cadre (par ex. bord gauche).
  + Bounds, géométrie du cadre.
  + Direction, direction du drag (par ex. horizontal).
  + Constraint, contrainte de déplacement de la poignée considérée.
  + Elasticity, élément élastique, ou non.
* …

### Classes de configuration

* EventProcessorPolicy est la classe abstraite.
* KeyDownProcessorPolicy spécifie comment se comporte le *scroll* *passif* (Ctrl-flèches).
* MouseDownProcessorPolicy spécifie les propriétés suivantes :
  + AutoFollow, la sélection suit la souris d’item en item en cas de bouton pressé (sinon, la sélection n’est active que tant que la souris survole le premier item).
  + AutoScroll, quand la souris survole un élément à une extrémité de la zone visible, démarre un défilement automatique pour pouvoir accéder aux éléments suivant/précédent.
  + AutoScrollDelay et AutoScrollRepeat, délai et taux de défilement automatique.
  + SelectionPolicy, sélection au moment du bouton enfoncé ou relâché.
* MouseDragProcessorPolicy spécifie les détails du drag & drop :
  + ResizePolicy, redimensionne ou non les cadres quand on clique sur leurs bordures, en mode indépendant ou couplé.

La méthode Filter permet de filtrer une liste de cadres selon le *mode indépendant* où en principe seul un cadre est redimensionné (celui qui n’est pas élastique) ou selon le *mode couplé* où les deux cadres adjacents seront considérés.

## Utilisation de BigList

Pour afficher une liste dans une interface graphique, il faut :

* Définir un IItemDataProvider et un IItemDataMapper.
* Définir un IItemDataRenderer et au besoin aussi un IItemMarkRenderer.
* Créer un widget ItemScrollList et le configurer au moyen de SetupItemList, en lui passant le *provider*, le *mapper*, les *renderers* et d’éventuels autres paramètres optionnels.

En général, IItemDataProvider et IItemDataMapper sont implémentés soi-même, en fonction de la source de données. Il est par contre possible d’utiliser des IItemDataRenderer standards définis dans le namespace BigList.Renderers.

### Interrogations

1. Faut-il que BigList propose des *widgets* de haut niveau, par ex. incluant un ascenseur horizontal ?
2. Faut-il que BigList offre des facilités pour les tris, opération de filtrage interactif, etc. ?

1. Le prix arrêté n’intervient pas dans le calcul du prix et dans la comptabilisation. Il sert uniquement de base à l’établissement des informations de paiement. À l’encaissement, la différence entre le total et le prix arrêté sera traité comme un escompte. [↑](#footnote-ref-1)
2. Pour pouvoir créer un module, il faut que le service Module.RepositoryServer.exe sur svn.opac.ch soit actif, et que l’utilisateur Windows du PC client soit reconnu par le serveur (par ex. « Daniel » avec le mot de passe « …123 »). [↑](#footnote-ref-2)
3. Voir la définition <http://en.wikipedia.org/wiki/Fluent_interface>. [↑](#footnote-ref-3)