**Analyse et Conception – Etude des contrôles dans une application**

Table des matières

[1 Objectif de l'étude 5](#_Toc453292895)

[1.1 Analyse 6](#_Toc453292896)

[1.2 Conception fonctionnelle 6](#_Toc453292897)

[1.3 Conception technique 7](#_Toc453292898)

[1.4 Tests est vérifications 7](#_Toc453292899)

[1.5 Capitalisation 7](#_Toc453292900)

[2 ANALYSE du concept Contrôle 8](#_Toc453292901)

[2.1 Dégrossir le problème (Collecter les concepts, besoins, exigences et règles de gestion) 8](#_Toc453292902)

[2.1.1 Collecter les besoins, CU, acteurs, concepts, exigences et règles de gestion (RG) liés aux contrôles auprès de la MOA 9](#_Toc453292903)

[2.1.2 Appliquer les règles de l’art 10](#_Toc453292904)

[2.1.3 Identifier quelques cas réels de contrôles 10](#_Toc453292905)

[2.1.4 Identifier les acteurs et imaginer comment l'application devra rendre compte des contrôles (rapports de contrôle, logs) 17](#_Toc453292906)

[2.2 Simuler l'utilisation des contrôles 19](#_Toc453292907)

[2.2.1 Identifier les concepts nécessaires à minima pour l'acteur le plus exigeant (Quand-Qui-Quoi-Qu'est ce - Que-Quel-Où) 19](#_Toc453292908)

[2.2.2 Identifier les concepts liés au cycle de vie d’une application 21](#_Toc453292909)

[2.2.3 Simuler le déclenchement des contrôles par un fournisseur de données 26](#_Toc453292910)

[2.2.4 Simuler la gestion des contrôles par l'Administrateur 27](#_Toc453292911)

[2.2.5 Simuler l'utilisation des rapports de contrôle (tris, requêtes, …) 29](#_Toc453292912)

[2.2.6 Lister les questions types que se posera la MOA 30](#_Toc453292913)

[2.3 Synthétiser les acteurs, concepts, exigences et RG capturés 31](#_Toc453292914)

[2.3.1 Synthèse des besoins et CU 32](#_Toc453292915)

[2.3.2 Synthèse des acteurs capturés 32](#_Toc453292916)

[2.3.3 Synthèse des concepts capturés 34](#_Toc453292917)

[2.3.4 Synthèse des exigences capturées 34](#_Toc453292918)

[2.3.5 Synthèse des RG capturées 38](#_Toc453292919)

[2.4 Affiner et Structurer les concepts 40](#_Toc453292920)

[2.4.1 Concept Date 41](#_Toc453292921)

[2.4.2 Concept User 42](#_Toc453292922)

[2.4.3 Concept Portée 43](#_Toc453292923)

[2.4.4 Concept Objet Contrôlé 45](#_Toc453292924)

[2.4.5 Concept Critère 46](#_Toc453292925)

[2.4.6 Concept Type d'Erreur 47](#_Toc453292926)

[2.4.7 Concept Lieu 48](#_Toc453292927)

[2.4.8 Concept Activité 49](#_Toc453292928)

[2.4.9 Concept Statut 50](#_Toc453292929)

[2.4.10 Concept Gravité 51](#_Toc453292930)

[2.4.11 Concept Suite A Donner 52](#_Toc453292931)

[2.4.12 Concept Fichier Résultant 53](#_Toc453292932)

[2.4.13 Concept Règle de Gestion 55](#_Toc453292933)

[2.4.14 Concept Règle de l'Art 56](#_Toc453292934)

[2.4.15 Concept Rapport de Contrôle 57](#_Toc453292935)

[2.4.16 Concept Log de Contrôle 58](#_Toc453292936)

[2.4.17 Concept Message de Contrôle 60](#_Toc453292937)

[2.5 Bâtir le modèle conceptuel des Contrôles 62](#_Toc453292938)

[2.6 Tester le modèle conceptuel 63](#_Toc453292939)

[2.6.1 Bâtir des phrases utilisant le modèle conceptuel 63](#_Toc453292940)

[3 CONCEPTION FONCTIONNELLE 64](#_Toc453292941)

[3.1 Capture des attributs et méthodes des concepts 64](#_Toc453292942)

[3.1.1 Méthode de capture des attributs et méthodes 65](#_Toc453292943)

[3.1.2 Capture des attributs et méthodes 71](#_Toc453292944)

[3.2 Dictionnaire des données d'un Contrôle 96](#_Toc453292945)

[3.3 Etudier l'enchaînement des contrôles 98](#_Toc453292946)

[3.3.1 Nécessité d'enchaîner les contrôles 98](#_Toc453292947)

[3.3.2 Importance de l'ordre des contrôles 99](#_Toc453292948)

[3.4 Bâtir le Modèle de classes de conception 99](#_Toc453292949)

[3.5 Tester le modèle de classes de conception 99](#_Toc453292950)

[4 CONCEPTION TECHNIQUE 100](#_Toc453292951)

[4.1 Concevoir des classes autonomes et réutilisables 100](#_Toc453292952)

[4.2 Factoriser au maximum le code au plus haut niveau (abstractions) 100](#_Toc453292953)

[4.3 Publier des interfaces Java 100](#_Toc453292954)

[4.4 Appliquer les règles de codage (Design Patterns) 100](#_Toc453292955)

[4.4.1 Utiliser le polymorphisme 100](#_Toc453292956)

[4.4.2 Utiliser des hooks 100](#_Toc453292957)

[4.5 Définir les objets techniques 100](#_Toc453292958)

[4.6 Bâtir le modèle de classe de codage 101](#_Toc453292959)

[4.6.1 Attribut nomClasseConcrete 101](#_Toc453292960)

[4.6.2 Attribut ordreControle 103](#_Toc453292961)

[5 TESTS ET VERIFICATIONS 105](#_Toc453292962)

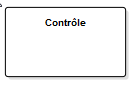
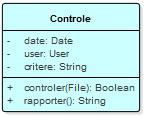
[5.1 Vérifier la Traçabilité Besoins-Exigences-RG-Classes-Attributs-Méthodes 105](#_Toc453292963)

[6 CAPITALISATION de la méthode 106](#_Toc453292964)

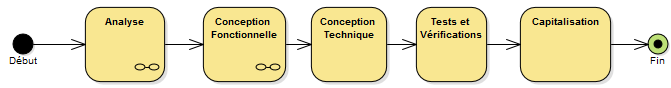
# Objectif de l'étude

L'objectif de la présente étude est d'élaborer une **méthode opérationnelle** pour **modéliser les concepts qualifiant les contrôles** dans une application informatique afin d'élaborer des classes de codage des contrôles (classes Java).

On cherche ici à définir ce que doit "**savoir**" et "**savoir faire**" un **contrôle** dans une application informatique.

La méthode proposée comporte 5 phases :



Je me suis donc directement inspiré de Khefren et des diverses méthodes que l'on trouve sur le Net.

J'ai simplement essayé de vérifier que **chaque étape**, chaque schéma proposé, … me fournissait réellement un **bénéfice**.

En particulier, les questions à se poser me semblent être :

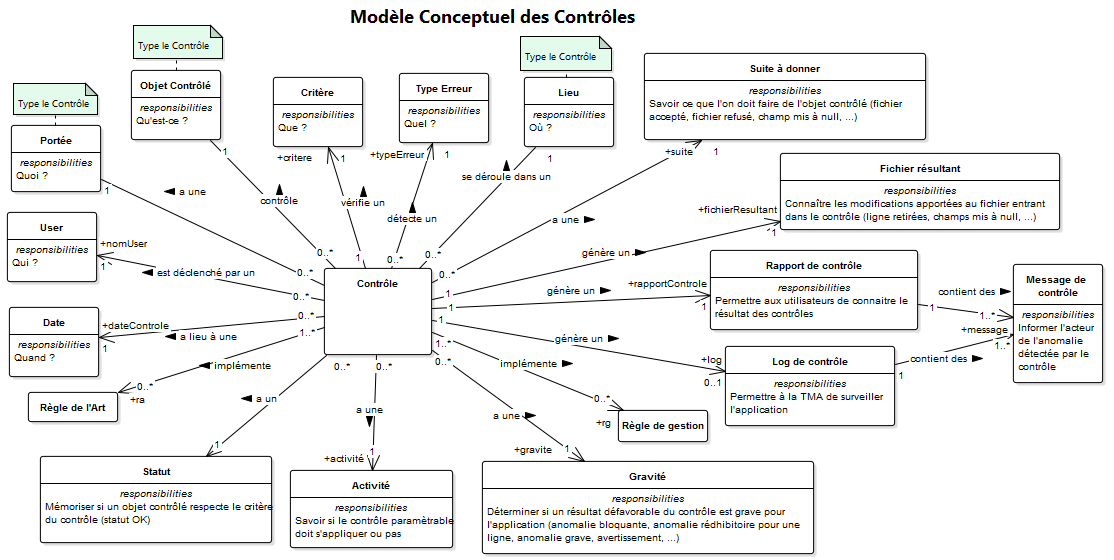
* "Aurais-je risqué d'oublier quelque chose si je n'avais pas franchi cette étape ? ".
* "Ai-je détecté des concepts que je n'aurais pas trouvés sans cette méthode ?"
* "Puis-je utiliser cette méthode pour étudier n'importe quel problème informatisable ?"

## Analyse

Phase au cours de laquelle on *capture* les **acteurs**, **concepts**, **besoins**, **exigences** et **règles de gestion** auxquels doivent satisfaire les contrôles dans une application informatique.

Cette phase correspond aux travaux réalisés lors de l'**E**tude **A**mont (EA), puis lors de l'élaboration du **D**ossier d'**A**nalyse **F**onctionnel – Partie Analyse (DAF Analyse) d'un projet Khefren.

L'Analyse consiste à **définir le périmètre** couvert par les contrôles, à **analyser** les concepts capturés en les décomposant, puis à bâtir un **modèle conceptuel** des contrôles couvrant le périmètre défini.



## Conception fonctionnelle

Phase au cours de laquelle on *capture* les **attributs** et **méthodes** dont devront disposer les classes de contrôle. On commence à "*penser Objet*" et à définir des **classes Objet** et des **hiérarchies de classes**.

Cette phase correspond aux travaux réalisés lors de l'élaboration du **D**ossier d'**A**nalyse **F**onctionnel – Partie Conception (DAF Conception) d'un projet Khefren.

On bâtit le **modèle de classes de conception** et le **dictionnaire des données**.

## Conception technique

Phase au cours de laquelle on *rajoute* des **attributs**, **méthodes**, **objets techniques** utiles aux classes de conception pour qu'elles puissent devenir des classes de code (ce que va coder le développeur Java). On est donc très près du code Java à ce stade.

On prend le modèle de classes de conception et on y applique des **règles de codage** comme des **Design Patterns**, des **bonnes pratiques informatiques**, …

Dans la mesure où l'on utilise le langage Java au Ministère, je me suis limité à l'application de recettes Java.

Cette phase correspond aux travaux réalisés lors de l'élaboration du **D**ossier de **C**onception **T**echnique (DCT) d'un projet Khefren.

On bâtit le **modèle des classes de codage**.

## Tests est vérifications

Phase au cours de laquelle on :

* Vérifie que les besoins, exigences, **R**ègles de **G**estion (RG), ont bien été prises en compte (matrices de traçabilité).
* Vérifie que les cas d'utilisation demandés par la MOA sont bien satisfaits.

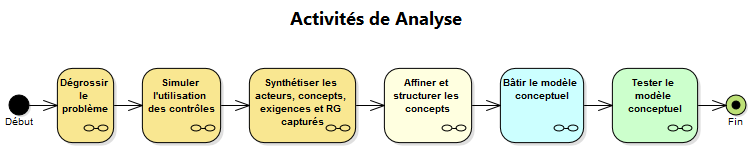
## Capitalisation

# ANALYSE du concept Contrôle

Objectif : L'objectif du chapitre "Analyse du concept Contrôle" est de **dégager les concepts essentiels relatifs à un contrôle** au vu des besoins, **C**as d'**U**tilisation (CU), exigences, règles de gestion ainsi que de l'utilisation ultérieure des contrôles par les divers acteurs.

Bénéfices attendus :

* L'analyse doit fournir les **besoins**, **Cas d'Utilisation** (CU), **acteurs**, **concepts**, **exigences**, **Règles de Gestion** (RG), **Règles de l'Art** (RA) et **éléments de contexte** nécessaires à la compréhension du problème.
* L'analyse doit **fournir un modèle conceptuel** (aussi appelé modèle du domaine) *testé* permettant de comprendre quels sont les concepts manipulés et quelles sont les relations entre concepts.

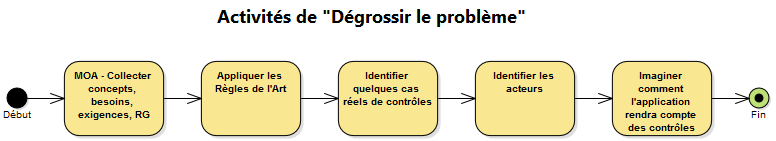


## Dégrossir le problème (Collecter les concepts, besoins, exigences et règles de gestion)

Objectif : L'objectif de cette partie "Dégrossir…" est de comprendre les premiers besoins et exigences *en matière de contrôle* dans une application informatique. Il s'agit de **déterminer le périmètre couvert** par les contrôles (équivalent de la phase d'inception dans le **R**ational **U**nified **P**rocess RUP).

Bénéfices attendus : **liste** des **besoins**, **CU**, **acteurs**, **concepts**, **exigences** et **RG** fournis par la MOA, le contexte, les règles de l'art et un premier travail d'analyse.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.



### Collecter les besoins, CU, acteurs, concepts, exigences et règles de gestion (RG) liés aux contrôles auprès de la MOA

Objectif : *capturer* les besoins, CU, acteurs, concepts, exigences et Règles de Gestion (RG) formulés par la **M**aîtrise d'**O**uvr**A**ge (MOA).

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et RG émis par le **détenteur du métier** (MOA).

Ce travail (interview de la MOA) est réalisé dès l'**E**tude **A**mont (EA) Khefren.

Toute application informatique a besoin de **contrôler** la validité des ressources qu'elle accepte en entrée ainsi que la qualité de ses livrables. La **M**aîtrise d'**O**uvr**A**ge (MOA) a donc une *exigence fonctionnelle de contrôle de validité* des fichiers entrants.

Par exemple, une application informatique qui traite des fichiers devra vérifier que les fichiers entrants sont non-null, existants, pas des répertoires, non vides, conformes au format attendu, …

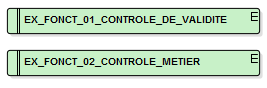
De même, la MOA a toujours une *exigence fonctionnelle de contrôle métier* des données. Il s'agit de s'assurer que les données métier gérées par l'application satisfont des **règles de gestion (RG)** définies par la MOA.

#### Concepts et Exigences mis en évidence

Concept :



Exigences :



### Appliquer les règles de l’art

Objectif : capturer les concepts, besoins, exigences et règles de gestion implicites provenant des règles de l'art.

Bénéfices attendus : **liste** des concepts, besoins, exigences et règles de gestion émis par le **contexte et les règles de l'informatique**.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

Certains besoins et exigences ne sont pas directement exprimés par la MOA lors des interviews mais ils s’imposent d’eux-mêmes à l’informaticien : le contexte et les **règles de l’art**.

L’informaticien doit par exemple toujours s’assurer que n’importe quel type de fichier ne peut être entré dans l’application même si la MOA n’a pas spécifié ce point lors des interviews.

La MOA a donc toujours une *exigence fonctionnelle* implicite *d’application des règles de l’art*.

#### Concepts et exigences mis en évidence





### Identifier quelques cas réels de contrôles

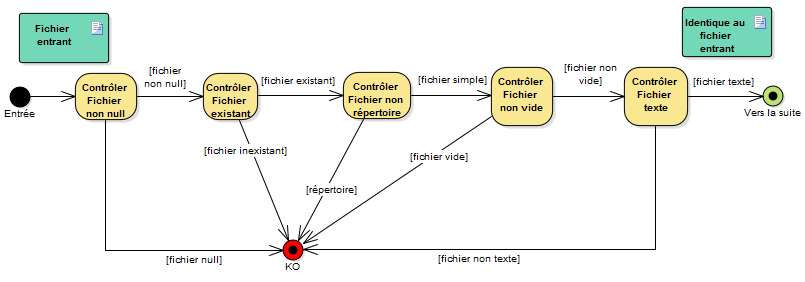
Objectif : capturer les concepts, besoins, exigences et règles de gestion provenant de l'étude de quelques cas réels.

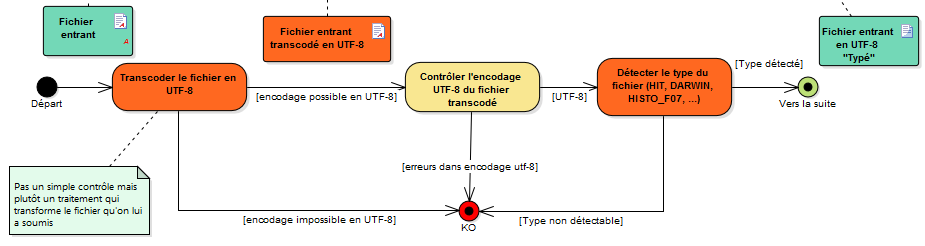
Bénéfices attendus : **liste** des concepts, besoins, exigences et règles de gestion obtenus par une première **analyse de quelques cas réels**.

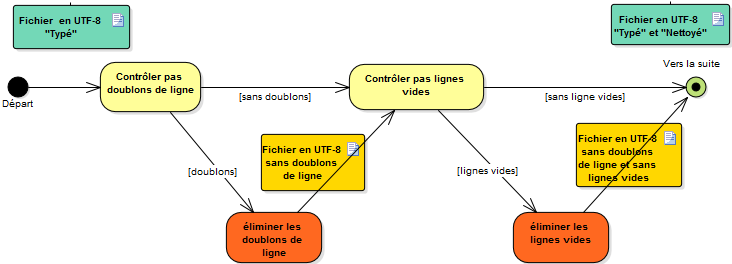
Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

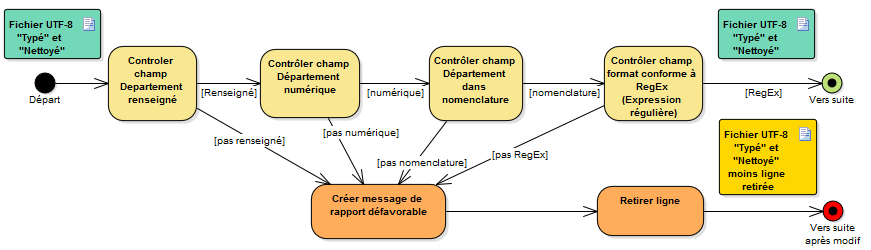
**Détecter quelques contrôles** et traitements que l'application devra obligatoirement exécuter. L'idée est de dégrossir le problème et de se faire une idée de ce qu'est le concept de contrôle. On applique notamment des **règles de l’art** (comme par exemple, il est toujours inutile d’accepter en entrée d’une application informatique un fichier vide).

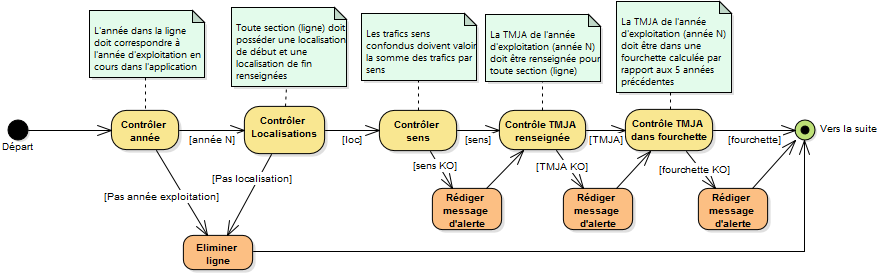
Par exemple, en analysant le début du traitement d'un fichier dans l'application TraficWeb :











#### Résumé des contrôles sous forme tabulaire

| **ordre** | **Lieu** | **Niveau**  **(Portée)** | **Type** | **META-NOM du contrôle** |
| --- | --- | --- | --- | --- |
| 1 | Surface | Fichier | Nullité | Contrôle de surface de non-nullité d'un fichier |
| 2 | Surface | Fichier | Existence de fichier | Contrôle de surface d'existence d'un fichier |
| 3 | Surface | Fichier | Non-répertoire | Contrôle de surface de simplicité (non-répertoire) d'un fichier |
| 4 | Surface | Fichier | Contenu (non vide) | Contrôle de surface de contenu (non vide) d'un fichier |
| 5 | Surface | Fichier | Textuel | Contrôle de surface du caractère textuel d'un fichier |
| 6 | Surface | Fichier | Transcodage\_UTF8 | Transcodage de surface du fichier en UTF-8 |
| 7 | Surface | Fichier | Encodage\_UTF8 | Contrôle de surface de l'encodage en UTF-8 du fichier |
| 8 | Surface | Fichier | Détection\_Type\_HIT | Détection de surface du type HIT d'un fichier |
| 9 | Surface | Fichier | Détection\_Type\_HISTO\_F07 | Détection de surface du type HISTO\_F07 d'un fichier |
| 10 | Surface | Fichier | Détection\_Type\_HISTO\_F08 | Détection de surface du type HISTO\_F08 d'un fichier |
| 11 | Surface | Fichier | Détection\_Type\_DARWIN | Détection de surface du type DARWIN csv d'un fichier |
| 12 | Surface | Fichier | Détection\_Type\_FEOR\_XML | Détection de surface du type FEOR XML d'un fichier |
| 13 | Surface | Inter-Lignes | Détection de doublons de lignes | Détection-Elimination de surface des doublons de lignes |
| 14 | Surface | Ligne | Nullité | Contrôle de surface de nullité d'une ligne |
| 15 | Surface | Ligne | Renseigné | Contrôle de surface de renseignement d'une ligne (non null ou vide) |
| 16 | Surface | Champ | Renseigné | Contrôle de surface de renseignement d'un champ (non null ou vide) |
| 17 | Surface | Champ | Format Numérique | Contrôle de surface de format numérique d'un champ |
| 18 | Surface | Champ | Dans Nomenclature | Contrôle de surface de nomenclature d'un champ |
| 19 | Surface | Champ | Regex (expression régulière) | Contrôle de surface d'expression régulière d'un champ |
| 20 | Métier | Champ | Règle de gestion | Contrôle métier sur l'année qui doit valoir l'année d'exploitation |
| 21 | Métier | Inter-champs | Règle de gestion | Contrôle métier sur Les localisations de début et de fin d'une section qui doivent être renseignées. |
| 22 | Métier | Inter-Lignes | Règle de gestion | Contrôle métier sur les sens des trafics |

#### Concepts, exigences et RG mis en évidence

L'examen des diagrammes d'activité des contrôles précédents permet de dégager de nouveaux concepts, exigences et règles de gestion.

Par exemple, lorsqu'un fichier entrant est inexistant ou vide, l'application doit s'arrêter. On voit donc apparaître le concept de **Suite à donner** à un contrôle (fichier refusé, fichier accepté, ligne retirée, ligne acceptée, champ mis à null, valeur de champ modifiée, …).

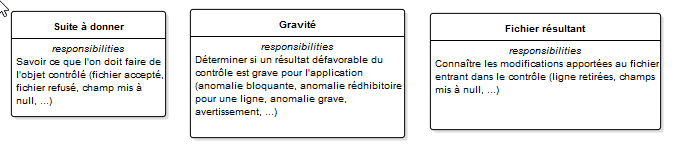
Par ailleurs, si dans une ligne d'un fichier HIT (fichier contenant des sections de comptage des trafics routiers fournis par les **D**irections **I**nterdépartementales des **R**outes (DIR)) un champ comme le "Département" n'est pas renseigné, la section de trafic correspondante à cette ligne est inutilisable puisque l'on ne pourra pas la localiser. Le champ Département est un champ "**rédhibitoire**". La MOA a stipulé une *règle de gestion* précisant qu'une telle ligne devait être *éliminée*. Le fichier en sortie du contrôle peut donc être différent du fichier en entrée puisque des lignes peuvent être éliminées.

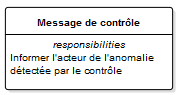
On voit donc apparaître le concept de **Gravité** d'un contrôle (anomalie bloquante, anomalie rédhibitoire pour une ligne, anomalie grave, avertissement, …). Il existe une relation entre la gravité, l'objet contrôlé et la suite à donner puisque l'on peut imaginer qu'un contrôle défavorable d'une forte gravité sur un champ rédhibitoire peut entraîner la suppression d'une ligne, la mise à null d'un champ, la modification de la valeur d'un champ, …

On voit également apparaître le concept de **Fichier résultant** d'un contrôle. Un fichier peut par exemple comporter moins de lignes en sortie d'un contrôle sur le champ Département qu'en entrée puisque l'on peut avoir retiré les lignes avec le Département non renseigné.

En outre, on constate que tout contrôle (y compris les traitements) doit générer un **Message** de contrôle à l'intention des acteurs. La MOA a donc un *exigence fonctionnelle implicite que tout contrôle ou traitement génère un message de contrôle*.

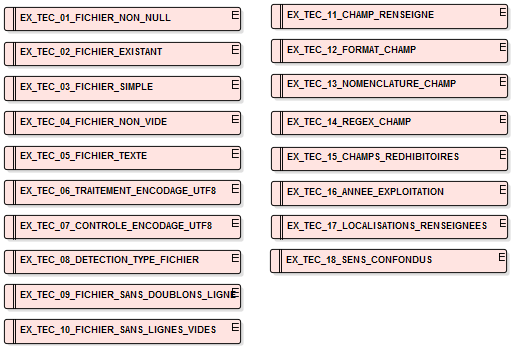
Concepts capturés :





Exigences capturées :





Règles de Gestion capturées :



Résumé tabulaire des Règles de gestion capturées :



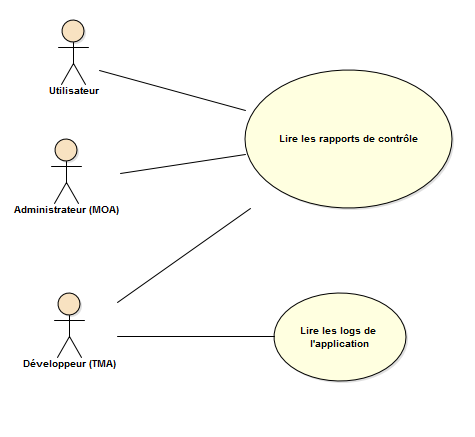
### Identifier les acteurs et imaginer comment l'application devra rendre compte des contrôles (rapports de contrôle, logs)

Objectif : capturer les besoins, CU, acteurs, concepts, exigences et règles de gestion provenant de l'étude des besoins des différents acteurs.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus par l'**examen des besoins des acteurs**.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

* L'**Utilisateur** de l'application informatique aura *besoin d'un rapport de contrôle* lui indiquant pourquoi une ressource (fichier soumis en entrée) est défectueuse.
* De même l'**Administrateur** de l'application informatique (**M**aîtrise d'**O**uvr**A**ge MOA) aura besoin d'accéder à ces rapports de contrôle pour aider les utilisateurs à mieux utiliser l'application.
* Enfin, un **Développeur** tiers (**T**ierce **M**aintenance **A**pplicative TMA) aura besoin de savoir comment les rapports de contrôle ont été conçus et sont générés. Il aura également *besoin des logs* de l'application pour en améliorer l'utilisation.

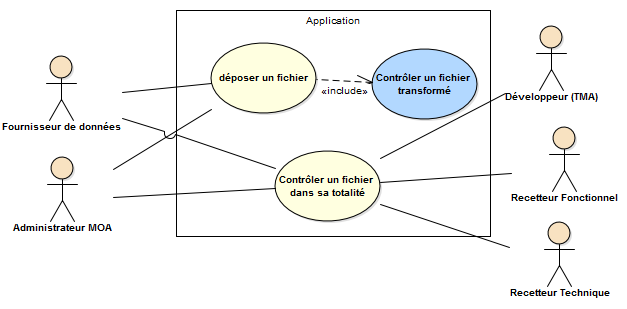


Par ailleurs, le fait qu'un fichier puisse être transformé lors de son contrôle, - et notamment le fait que des lignes puissent être retirées -, pose un réel problème : Il est impossible d'exécuter des contrôles dans une ligne retirée.

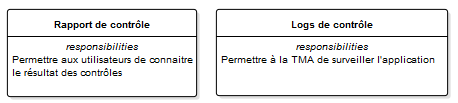
Cette élimination des lignes rédhibitoires au fur et à mesure des contrôles est intéressante lors du processus de dépôt d'un fichier pour des raisons de performance.

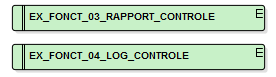
Mais si on veut pouvoir appliquer la totalité des contrôles sur un fichier, il faut donc concevoir un mode de fonctionnement des contrôles sans retrait de lignes. C'est le cas pour tout acteur qui veut connaître la qualité d'un fichier de trafic.

Même si ce point dépasse le strict niveau des contrôles et est plus du niveau applicatif, il faut le prendre en compte car il apporte une nouvelle exigence technique : EX\_TEC\_28\_CONTROLE\_INTEGRAL (L'application doit proposer un mode d'exécution des contrôles sur la totalité d'un fichier (sans retrait de ligne, …)).



#### Concepts et exigences mis en évidence



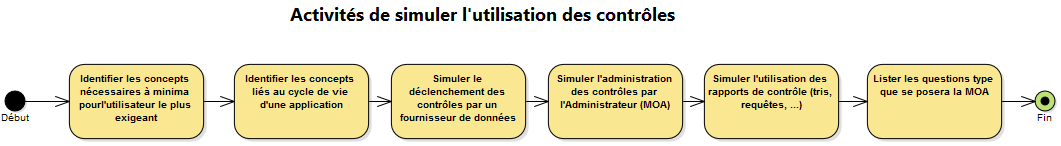
 

## Simuler l'utilisation des contrôles

Objectif : L'objectif de cette phase "simuler l'utilisation des contrôles" est de comprendre ce que "doit savoir" un contrôle pour pouvoir être utilisé dans de bonnes conditions par les différents acteurs.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus en **simulant conceptuellement l'utilisation des contrôles**.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.



### Identifier les concepts nécessaires à minima pour l'acteur le plus exigeant (Quand-Qui-Quoi-Qu'est ce - Que-Quel-Où)

Objectif : identifier grossièrement les besoins, CU, acteurs, concepts, exigences et règles de gestion permettant d'**identifier** un concept Contrôle.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus en **simulant conceptuellement l'utilisation des contrôles par la MOA et les acteurs très exigeants**.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

* Un **Utilisateur** (fournisseur de données) qui vient de déposer un fichier sait quel fichier il vient d'utiliser dans l'application. Il n'aura donc aucun problème pour identifier à quel objet contrôlé (fichier) se rapporte le rapport de contrôle que l'application vient de générer.
* En revanche, l'**Administrateur** ou le **Développeur** auront probablement besoin d'examiner les rapports de contrôles à postériori. Ils examineront probablement des consolidations de rapports de contrôle. Ils auront donc besoin de renseignements leur permettant d’identifier quel objet contrôlé (fichier, ligne du fichier, champ…) a déclenché tel contrôle…

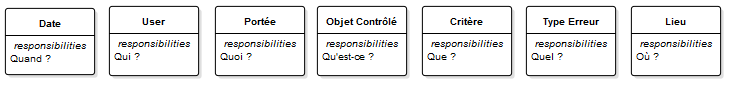
Il y a donc une *exigence fonctionnelle de pouvoir parfaitement identifier un contrôle* à postériori pour l'**Administrateur** et le **Développeur**. La suite de questions **Quand-Qui-Quoi-Qu'est ce-Que-Quel-Où** est une bonne piste pour commencer à trouver l'identifiant.

1. **Quand** le contrôle a-t-il eu lieu ? : concept **Date.** Date d'exécution du contrôle.
2. **Qui** a déclenché le contrôle ? : concept **User.** Utilisateur qui a déclenché le contrôle.
3. **Quoi** – le contrôle contrôlait quoi ? Un fichier ?, un champ d'une ligne du fichier ? : concept **Portée.** Type d'objet(s) sur lequel porte le contrôle.
4. **Qu'est-ce** – sur quel objet le contrôle s'applique-t-il ? le fichier DIRA\_2014 ? Le champ département de la 3ème ligne du fichier DIRE\_2013 ? : concept **Objet**. L'objet sur lequel porte le contrôle.
5. **Que** détecte le contrôle ? : concept **Critère.** Le critère que doit vérifier l'objet contrôlé. Par exemple, "le fichier ne comporte aucun caractère indésirable" ou "une ligne dans un HIT comporte exactement 520 caractères".
6. **Quel** type d'erreur le contrôle recherche-t-il ? : concept **Type**. Le type de l'erreur contrôlée comme par exemple "taille", "règle de gestion", "renseigné", …
7. **Où** le contrôle a-t-il eu lieu ? (contrôle de surface, contrôle métier …) : concept **Lieu**

Par exemple :



#### Concepts et exigences mis en évidence





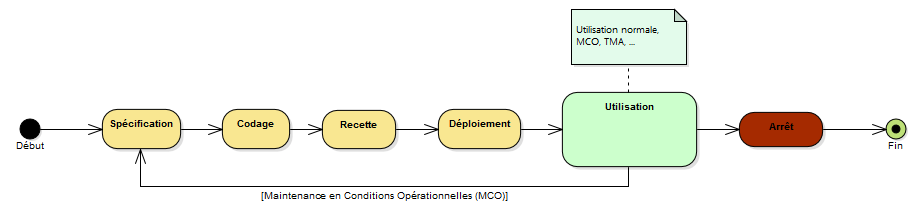
### Identifier les concepts liés au cycle de vie d’une application

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de l'étude du cycle de vie d'une application informatique.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus en **simulant conceptuellement le cycle de vie de l'application**.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

On peut modéliser le cycle de vie d’une application informatique comme suit :



Les phases de **codage**, de **recette** (tests), de **déploiement** et d’**utilisation** de l’application engendrent des besoins et exigences en sus de la phase de *spécification*.

#### Imaginer le codage de l’application en équipe

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de l'étude du codage d'une application informatique en équipe.

Le travail en équipe nécessite de se mettre d'accord au préalable sur le comportement attendu de l'application.

Il faut s'accorder sur ce que l'on entend par "un contrôle a répondu favorablement". Par exemple, que conclure si le Contrôle du contenu du fichier (fichier pas vide) a retourné true ? Le fichier était-il vide ou pas ?

Il faut donc une règle de gestion appliquées par tous les développeurs de l'équipe :



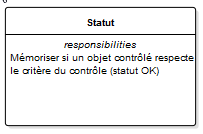
Définition : Un contrôle est dit "favorable" si l'objet contrôlé *remplissait* le critère de contrôle.

RG-100\_TRUE\_SI\_CONTROLE\_FAVORABLE : Un contrôle favorable (qui a détecté que l'objet remplissait le critère) doit répondre true dans l'application.

* Par exemple le résultat du contrôle "Pas de doublons" à true signifie que le fichier ne comporte pas de lignes en doublon.
* Autre exemple : le résultat du contrôle "Contenu du fichier pas vide" à true signifie que le fichier n'est pas vide.

Par ailleurs, la phrase "le contrôle a répondu favorablement (true)" fait apparaître un nouveau concept : le **Statut** du contrôle après son exécution.

* Lorsqu'un contrôle sur un objet répond favorablement (true), cela signifie que l'objet respectait le critère du contrôle (critère = champ département renseigné par exemple). Le statut du contrôle est donc favorable (OK)
* Lorsqu'un contrôle sur un objet répond défavorablement (false), cela signifie que l'objet ne respectait pas le critère du contrôle (critère = champ département renseigné par exemple). Le statut du contrôle est donc défavorable (KO).



#### Imaginer la recette de l'application

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de la simulation des futures recettes fonctionnelles et techniques de l'application informatique.

La recette d'une application informatique suppose qu'un acteur "**Recetteur**" puisse fabriquer des jeux d'essais et les jouer dans l'application afin de tester les contrôles.

Il est donc indispensable que le recetteur ait un *accès spécifique aux contrôles* de l'application afin de pouvoir tester ses jeux d'essais.

Le **Recetteur fonctionnel** doit pouvoir préparer des fichiers particuliers, les déposer dans l'application et vérifier que les contrôles voulus se sont bien déclenchés et que l'application délivre le bon rapport de contrôle.

Le **Recetteur technique** doit pouvoir tester chaque classe de contrôle individuellement (tests unitaires JUnit) et vérifier qu'il n'y a pas de régression entre les versions successives de l'application.



#### Imaginer le déploiement de l’application

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de l'étude du déploiement de l'application informatique.

Si l'application informatique est une application web, elle est susceptible d'être hébergée par un centre serveur.

Dans le cas où les contrôles nécessitent des ressources (fichiers externalisés properties, fichiers de nomenclatures pour certains champs, fichiers de description des champs pour les HIT, DARWIN, …), il est recommandé de permettre au centre serveur de choisir où il stockera ces ressources.

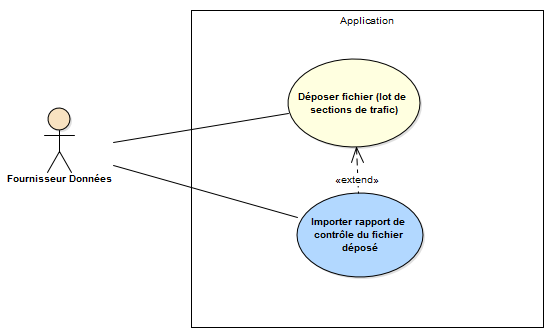
L'application doit donc prévoir le paramétrage des chemins des ressources nécessaires aux contrôles (Exigence fonctionnelle de l'application).

#### Imaginer la « vie réelle de l’application »

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de la simulation de l'utilisation réelle de l'application informatique.

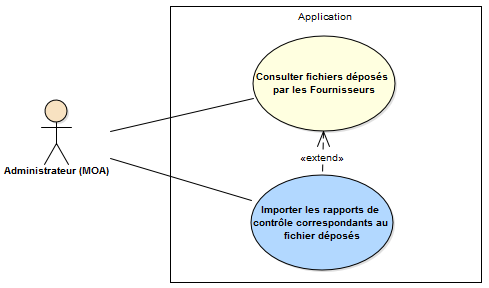
La question est de savoir comment les acteurs accèderont concrètement aux rapports de contrôle et comment ils bénéficieront de ce rapport.

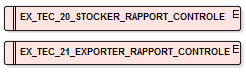
Un Acteur "**Fournisseur de données**" (**D**irection **I**nterdépartementale des **R**outes (DIR), **S**ociété **C**oncessionnaire d'**A**utoroute (SCA alimentant la base DARWIN), …) aura sans doute besoin d'importer sur son propre disque dur le rapport de contrôle du fichier qu'il vient de déposer dans l'application afin de pouvoir l'étudier. L'application doit donc disposer de ce rapport de contrôle sur le serveur applicatif et permettre de l'*exporter vers le Fournisseur de données*.



Un acteur "**Administrateur (MOA)**" aura besoin de surveiller la qualité des fichiers déjà déposés par des fournisseurs de données. Il aura donc besoin de consulter à postériori les rapports de contrôle correspondants aux fichiers déposés.

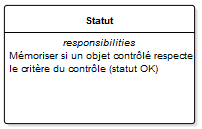
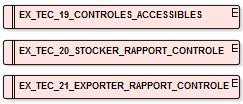
Il souhaitera sans doute importer ces rapports de contrôle sur son propre disque dur afin de les agréger, trier, comparer, …





Par ailleurs, le Fournisseur de Données qui a exporté sur son disque dur le rapport de contrôle dispose du fichier original de données trafic qu'il a soumis et du rapport de contrôle. Or, le fichier de données trafic peut être modifié (lignes retirées, champs mis à null, valeurs de champ modifiées, …) lors du processus de traitement des trafics. L'application doit donc *mémoriser les numéros de ligne dans les fichiers originaux* et les utiliser dans les rapports de contrôle afin de simplifier le travail du fournisseur.

#### Concepts, exigences et RG mis en évidence

### Simuler le déclenchement des contrôles par un fournisseur de données

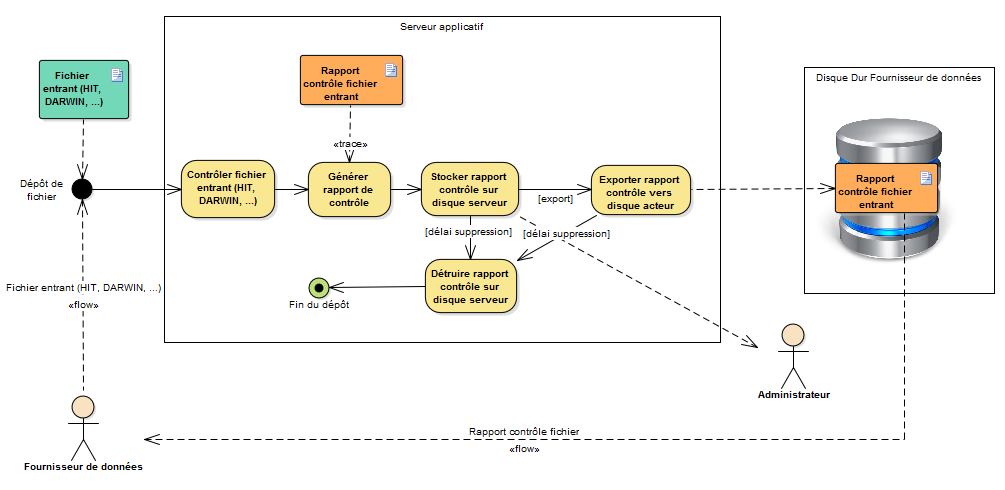
Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de la simulation du dépôt d'un fichier dans l'application informatique.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus en **simulant conceptuellement le déclenchement des contrôles lors d'un dépôt de fichier**.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

Le Fournisseur de données (DIR, SCA, …) dépose un fichier (HIT, DARWIN, HISTONAT, ...) dans l'application TraficWeb. TraficWeb réalise alors le processus suivant :

Import du fichier déposé > Contrôle > Génération du rapport de contrôle > Stockage du rapport sur le serveur



Les acteurs peuvent ensuite importer les rapports de contrôle sur leur propre disque dur comme vu précédemment. L'Administrateur MOA ira chercher directement le rapport de contrôle sur le serveur.

Le problème est que le stockage "sans limite" des rapports de contrôle sur le serveur peut conduire à un problème de taille de stockage.

Il est nécessaire qu'un acteur du type "**Administrateur Applicatif**" puisse s'assurer que le stockage des rapports de contrôle n'occupe pas un volume excessif sur le serveur. Le cas échéant, l'Administrateur Applicatif doit pouvoir *retirer manuellement des rapports du stockage*.

On peut également imaginer que l'application "sache" *automatiquement écraser les rapports de contrôle* hors d'une période déterminée par la MOA (par exemple, l'année d'exploitation).

#### Concepts, exigences et RG mis en évidence





### Simuler la gestion des contrôles par l'Administrateur

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de la simulation de la gestion des contrôles "lors de la vie réelle" par l'Administrateur MOA dans l'application informatique.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus en **simulant conceptuellement la future gestion des contrôles** par l'Administrateur lors de la vie réelle de l'application.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

L'**Administrateur (MOA)** de l'application peut être amené lors de la phase d'utilisation normale de l'application à activer/désactiver certains contrôles (par exemple si des contrôles de faible gravité polluent les rapports de contrôle). Les contrôles doivent donc être *individuellement* *paramétrables* pour s'appliquer ou non. On voit apparaître le concept d'**Activité** d'un contrôle.

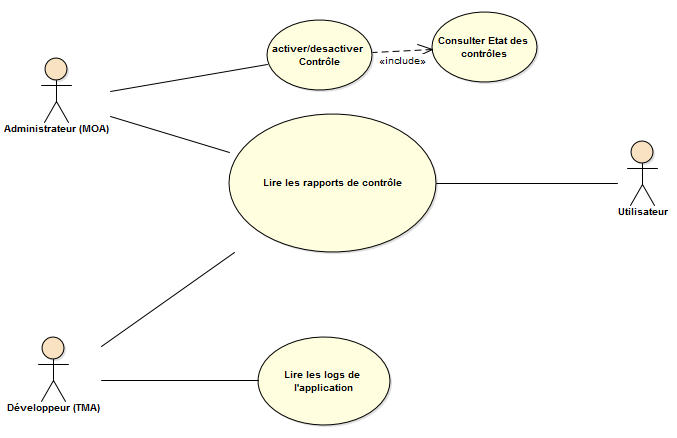
La MOA a donc une *exigence fonctionnelle de contrôles paramétrables*.

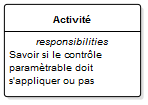
La MOA a stipulé une *règle de gestion* précisant qu'un contrôle désactivé ne devait pas s'appliquer et émettre de rapport.

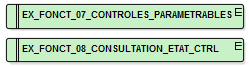
En pratique, l'Administrateur MOA aura besoin d'une aide logicielle pour connaître la liste des contrôles et savoir quels sont les contrôles activés avant de pouvoir activer/désactiver un contrôle.

La MOA a donc également une *exigence fonctionnelle de consultation de l'état des contrôles*.

#### Concepts, exigences et RG mis en évidence





### Simuler l'utilisation des rapports de contrôle (tris, requêtes, …)

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de la simulation de l'utilisation des rapports de contrôle par l'Administrateur MOA dans l'application informatique.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus en **simulant conceptuellement l'utilisation future des rapports de contrôle**.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

L'**Administrateur (MOA)** aura sans doute besoin de vérifier les rapports de contrôles des derniers lots de sections de trafic déposés par les **Fournisseurs de Données** avant de décider de consolider les données et de générer un fichier national annuel HISTONAT\_F07.

Il doit pouvoir agréger les données trafic provenant de chaque Fournisseur de Données (exigence fonctionnelle de l'application) pour procéder à une consolidation nationale.

Il doit pouvoir également *agréger les rapports de contrôle* afin d'avoir une vue globale de la qualité des données.

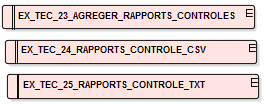
Comment peut-il utiliser ce rapport de contrôle agrégé ?

Une bonne solution semble être de présenter ce rapport agrégé sous forme de tableau *lisible par n'importe quel tableur*. Le format csv semble donc tout indiqué puisque – sous réserve que l'on respecte un certain ordre des colonnes – il permet l'agrégation des données comme leur vision sous forme de tableau (Excel, Calc, …). Il y a donc une *exigence fonctionnelle de consolidation des rapports* et une *exigence technique de présentation des lignes de rapport au format csv*.

L'Administrateur (MOA) pourra ensuite faire ses tris, ses analyses croisées, compter combien il y a de messages de contrôle défavorable pour telle DIR, rechercher une ligne de rapport particulière dans le tableur…

Si l'application est de type "Desktop" (avec des fenêtres et tournant directement sur la machine de l'acteur) et pas une application Web, les zones de texte ne se prêtent pas à l'affichage de lignes csv. Il faut donc prévoir de présenter également les messages de contrôle sous forme textuelle (pas csv) pour l'affichage des messages de contrôle dans des zones de texte. Il y a donc également une *exigence technique de présentation des lignes de rapport au format textuel*.

#### Concepts, exigences et RG mis en évidence

### Lister les questions types que se posera la MOA

Objectif : identifier les besoins, CU, acteurs, concepts, exigences et règles de gestion liés aux contrôles provenant de la simulation des questions posées par l'Administrateur MOA à l'application informatique.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus en **simulant conceptuellement les interrogations futures** de la MOA.

Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

La MOA aura besoin d'étudier les rapports de contrôle pour :

* Evaluer la qualité des données trafic.
* Déterminer les objets (fichiers, lignes, champs, …) qui posent problème.
* Superviser le travail des Fournisseurs de données (DIR, SCA, …)
* Prioriser ses interventions pour améliorer la qualité des données trafic.

On peut dégager quelques questions type :

#### Quelle est la qualité d'une année d'exploitation ?

La MOA doit donc pouvoir consolider les rapports de contrôle d'une année d'exploitation et bénéficier d'un *indicateur de qualité*. Par exemple, **combien de lignes de rapport de contrôle comportent au moins une anomalie rédhibitoire par rapport au total des lignes**.

Le concept de Statut du contrôle (OK/KO) déjà évoqué est un bon moyen pour créer cet indicateur.

#### Quelle est la tendance de la qualité ?

La MOA doit donc pouvoir comparer les rapports de contrôle consolidés entre plusieurs années d'exploitation. Le fait de disposer des rapports de contrôle sous forme de tableau (csv) répond à ce besoin dès lors que le tableau contient le champ année d'exploitation et dès lors que la MOA stocke les rapports annuels.

#### Quelles sont les Fournisseurs de données (DIR, …) qui posent problème ?

La MOA doit pouvoir comparer les rapports de contrôle consolidés annuellement par DIR par exemple.

Le fait de disposer des rapports de contrôle sous forme de tableau (csv) répond à ce besoin dès lors que le tableau contient le champ Fournisseur de données.

#### Quels sont les champs qui posent problème ?

La MOA souhaitera savoir par exemple quels sont les champs très peu renseignés. Elle souhaitera également savoir quels sont les types d'erreur (non renseigné, mauvais format, hors nomenclature, hors Expression Régulière (RegEx), ...) les plus fréquentes pour un champ donné.

Le fait de disposer des rapports de contrôle sous forme de tableau (csv) répond à ce besoin dès lors que le tableau contient le champ contrôlé et le type d'erreur diagnostiqué.

#### Concepts, exigences et RG mis en évidence

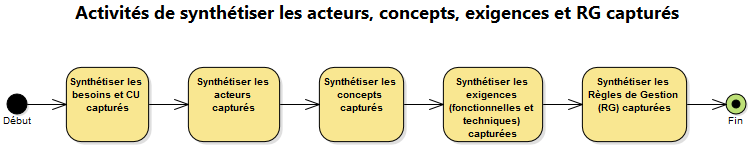
Les concepts et exigences capturés précédemment sont suffisants ici.

## Synthétiser les acteurs, concepts, exigences et RG capturés

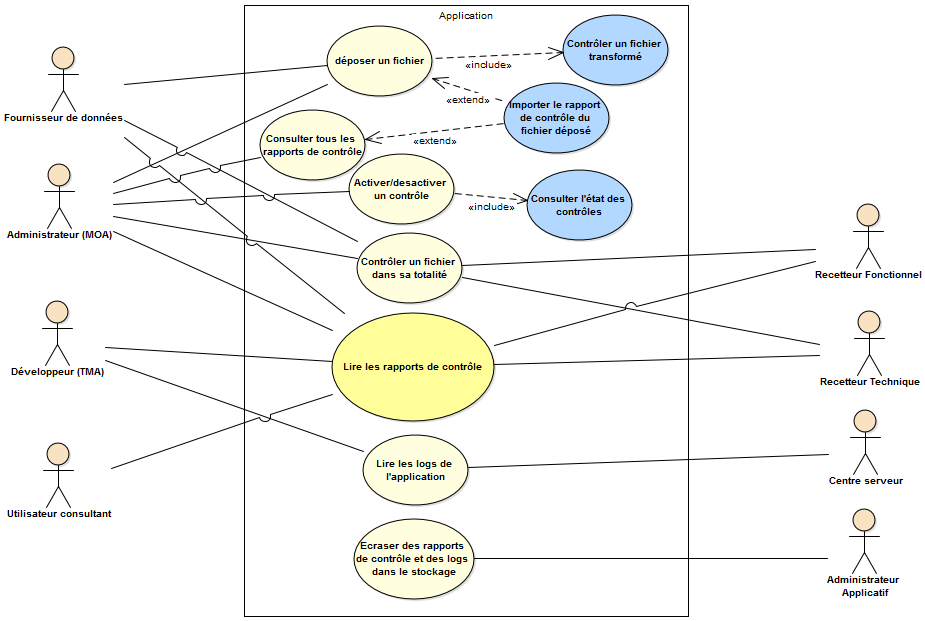
Objectif : l'objectif de cette partie est de faire à ce stade une synthèse des divers besoins, CU, acteurs, concepts, exigences et RG précédemment capturés. Cette phase réalisée dans l'EA Khefren permet de contrôler que l'on n'a pas commis d'oublis majeurs lors des phases précédentes.

Bénéfices attendus : **liste** des besoins, CU, acteurs, concepts, exigences et règles de gestion obtenus lors des paragraphes précédents.

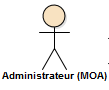
Ce travail est réalisé dès l'**E**tude **A**mont (EA) Khefren.

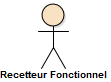


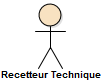
### Synthèse des besoins et CU



### Synthèse des acteurs capturés

#### Fournisseur de Données

Rôle tenu par tout acteur qui dépose un fichier (lot de sections de trafic) dans l'application.

A priori, il dépose des fichiers HIT ou DARWIN. C'est souvent un agent en **D**irection **I**nterdépartementale des **R**outes (DIR) qui dépose les trafics pour sa DIR. Mais on peut imaginer également le dépôt d'un fichier HISTONAT\_F07 (fichier consolidé national annuel des trafics) par la MOA pour effectuer des contrôles dessus par exemple.

#### Administrateur MOA

Rôle tenu par tout acteur qui supervise le travail déposé par les fournisseurs de données.

En particulier, il a besoin de consulter les rapports de contrôles correspondants aux fichiers déposés par les fournisseurs pour évaluer la qualité des données. Il peut également paramétrer les contrôles (actif/inactif). Il peut également déposer à postériori dans l'application un fichier annuel national consolidé HISTONAT\_F07 pour en vérifier la qualité.

#### Administrateur Applicatif

Rôle tenu par tout acteur chargé par la MOA d'administrer techniquement l'application (vidage des logs, vidage du stockage des rapports de contrôle, gestion des droits applicatifs, arrêt du serveur, …).

#### Utilisateur Consultant

Rôle tenu par tout acteur qui se borne à lire les rapports de contrôle.

#### Développeur (TMA)

Rôle tenu par tout acteur chargé de développer et maintenir l'application informatique. Il a besoin de consulter les rapports de contrôle ainsi que les logs générés par les contrôles. Il assure la **T**ierce **M**aintenance **A**pplicative (TMA) et le **M**aintien en **C**onditions **O**pérationnelles (MCO) de l'application.

#### Recetteur Fonctionnel

Rôle tenu par tout acteur chargé de la recette fonctionnelle de l'application. Il doit pouvoir "jouer" des jeux d'essais prédéterminés pour solliciter les contrôles de l'application.

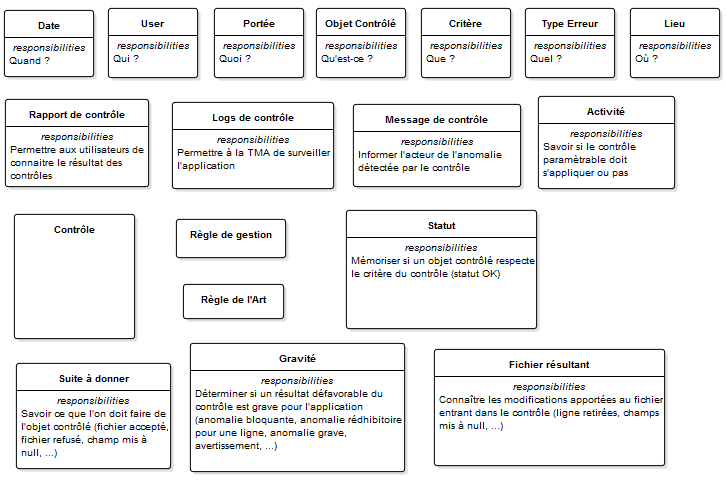
#### Recetteur Technique

Rôle tenu par tout acteur chargé de la recette technique de l'application. Il doit pouvoir "jouer" des tests unitaires (JUnit) et tester la non-régression de l'application entre différentes versions.

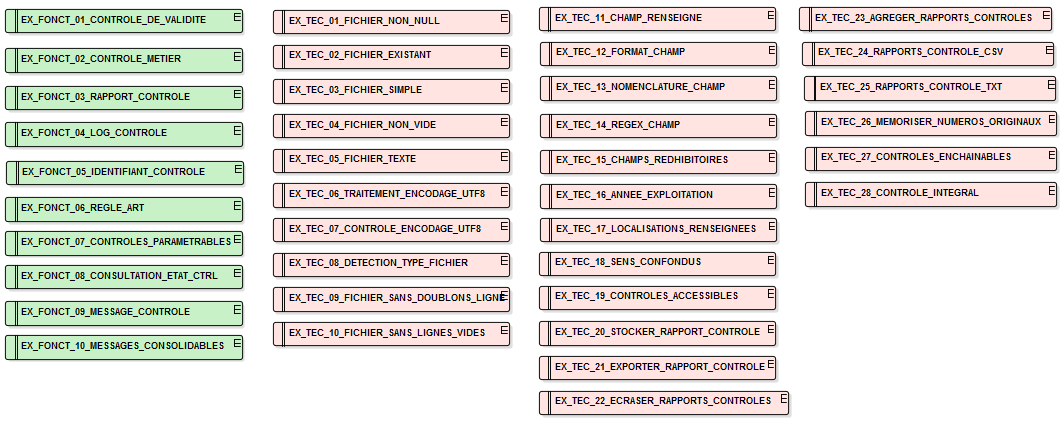
#### Centre Serveur

Rôle tenu par tout acteur chargé d'héberger l'application informatique, et par conséquent ses contrôles. Il a besoin de savoir où placer les ressources (properties, fichiers de nomenclature, fichiers de description, ...) utilisées par les contrôles.

### Synthèse des concepts capturés

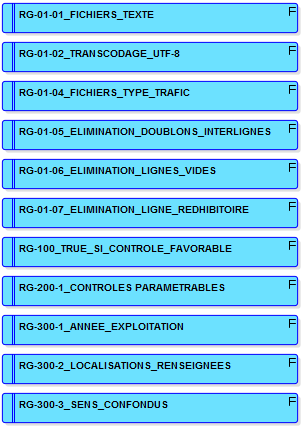


### Synthèse des exigences capturées



| **DEMANDEUR** | **IDENTIFIANT DE L'EXIGENCE** | **INTITULE** | **Description** | **Catégorie de l'exigence** |
| --- | --- | --- | --- | --- |
| MOA | EX\_FONCT\_01\_CONTROLE\_DE\_VALIDITE | Contrôle de la validité des fichiers en entrée | L'application cible doit contrôler la validité des fichiers acceptés en entrée. | Fonctionnelle |
| MOA | EX\_FONCT\_02\_CONTROLE\_METIER | Contrôle des données métier | L'application cible doit contrôler que les données métier satisfont à des règles de gestion | Fonctionnelle |
| MOA | EX\_FONCT\_03\_RAPPORT\_CONTROLE | Tout contrôle doit générer un rapport de contrôle | L'application cible doit fournir des rapports de contrôle expliquant le résultat de tous ses contrôles et traitements | Fonctionnelle |
| MOA | EX\_FONCT\_04\_LOG\_CONTROLE | Tout contrôle doit envoyer un message dans un Log applicatif | L'application cible doit fournir des logs applicatifs pour les contrôles | Fonctionnelle |
| MOA | EX\_FONCT\_05\_IDENTIFIANT\_CONTROLE | Tout contrôle doit être identifié | L'application cible doit identifier les contrôles afin de pouvoir les retrouver, les trier, … | Fonctionelle |
| MOA | EX\_FONCT\_06\_REGLE\_ART | Les contrôles doivent respecter les règles de l'art | L'application cible doit respecter les règles de l'art | Fonctionnelle |
| MOA | EX\_FONCT\_07\_CONTROLES\_PARAMETRABLES | Tout contrôle doit pouvoir être activé/désactivé | L'application cible doit mettre à disposition de l'Administrateur un dispositif de paramétrage individuel des contôles | Fonctionnelle |
| MOA | EX\_FONCT\_08\_CONSULTATION\_ETAT\_CTRL | Tout contrôle doit savoir si il est actif ou pas | L'application cible doit mettre à disposition de l'Administrateur un dispositif de consultation du paramétrage individuel des contôles | Fonctionnelle |
| MOA | EX\_FONCT\_09\_MESSAGE\_CONTROLE | Tout contrôle doit générer un message de contrôle à l'attention des acteurs | L'application cible doit générer un message de contrôle pour tout contrôle appliqué | Fonctionnelle |
| MOA | EX\_FONCT\_10\_MESSAGES\_CONSOLIDABLES | Les messages de contrôles doivent être consolidables (géographiquement au niveau national et temporellement sur une année d'exploitation) | L'application cible doit générer des messages de contrôle consolidables (géographiquement au niveau national et temporellement sur une année d'exploitation) | Fonctionnelle |
| MOE | EX\_TEC\_01\_FICHIER\_NON\_NULL | Un fichier en entrée ne doit pas être null | L'application cible doit contrôler que le fichier en entrée n'est pas null | Technique |
| MOE | EX\_TEC\_02\_FICHIER\_EXISTANT | Un fichier en entrée doit être existant | L'application cible doit contrôler que le fichier en entrée est existant | Technique |
| MOE | EX\_TEC\_03\_FICHIER\_SIMPLE | Un fichier en entrée ne doit pas être un répertoire | L'application cible doit contrôler que le fichier en entrée n'est pas un répertoire | Technique |
| MOE | EX\_TEC\_04\_FICHIER\_NON\_VIDE | Un fichier en entrée ne doit pas être vide | L'application cible doit contrôler que le fichier en entrée n'est pas vide | Technique |
| MOE | EX\_TEC\_05\_FICHIER\_TEXTE | Un fichier en entrée doit être un fichier texte | L'application cible doit contrôler que le fichier en entrée est un fichier texte (pas un mp3, un wav, …) | Technique |
| MOE | EX\_TEC\_06\_TRAITEMENT\_ENCODAGE\_UTF8 | Tous les fichiers gérés par l'application doivent être transcodés en interne en UTF-8. | Tous les fichiers gérés par l'application doivent être transcodés en interne en UTF-8. | Technique |
| MOE | EX\_TEC\_07\_CONTROLE\_ENCODAGE\_UTF8 | Contrôle de l'encodage UTF-8 des fichiers | L'application cible doit contrôler que les fichiers internes sont encodés en UTF-8 | Technique |
| MOE | EX\_TEC\_08\_DETECTION\_TYPE\_FICHIER | Détection du type de fichier (HIT, HISTO\_F07, DARWIN, …) | L'application cible doit détecter le type du fichier entrant (HIT, HISTO\_F07, DARWIN, …) et n'accepter en entrée que ces types de fichiers trafic | Technique |
| MOE | EX\_TEC\_09\_FICHIER\_SANS\_DOUBLONS\_LIGNE | Un fichier en entrée ne doit pas comporter de lignes en doublons | L'application cible doit détecter les lignes en doublon et éliminer les surplus | Technique |
| MOE | EX\_TEC\_10\_FICHIER\_SANS\_LIGNES\_VIDES | Un fichier en entrée ne doit pas comporter de lignes vides | L'application cible doit détecter les lignes vides et les éliminer | Technique |
| MOE | EX\_TEC\_11\_CHAMP\_RENSEIGNE | Un champ doit être renseigné dans une ligne | L'application doit détecter les champs non renseignés et leur appliquer une règle de gestion | Technique |
| MOE | EX\_TEC\_12\_FORMAT\_CHAMP | Un champ doit être au bon format dans une ligne | L'application doit détecter les champs au mauvais format et leur appliquer une règle de gestion | Technique |
| MOE | EX\_TEC\_13\_NOMENCLATURE\_CHAMP | Certains champs doivent respecter une nomenclature | L'application doit détecter les champs ne respectant pas une nomenclature et leur appliquer une règle de gestion | Technique |
| MOE | EX\_TEC\_14\_REGEX\_CHAMP | Certains champs doivent respecter une expression régulière (RegEx) | L'application doit détecter les champs ne respectant pas une expression régulière et leur appliquer une règle de gestion | Technique |
| MOE | EX\_TEC\_15\_CHAMPS\_REDHIBITOIRES | Un contrôle défavorable sur certains champs dits "rédhibitoires" doit aboutir au retrait de toute la ligne | L'application doit retirer toute la ligne du fichier si un de ses champs rédhibitoires est mal formé | Technique |
| MOE | EX\_TEC\_16\_ANNEE\_EXPLOITATION | L'année de toutes les sections d'un lot doit être l'année d'exploitation | L'application doit retirer toute la ligne du fichier si l'année de la ligne n'est pas l'année d'exploitation | Technique |
| MOE | EX\_TEC\_17\_LOCALISATIONS\_RENSEIGNEES | Les localisations de début et de fin de toutes les sections d'un lot doivent être renseignées | L'application doit retirer toute la ligne du fichier si une des localisations de la ligne n'est pas renseignée | Technique |
| MOE | EX\_TEC\_18\_SENS\_CONFONDUS | Les trafics sens confondus doivent être égaux à la somme des trafics sens croissants et décroissants pour une même section | L'application doit signaler les erreurs dans les cumuls de sens | Technique |
| MOE | EX\_TEC\_19\_CONTROLES\_ACCESSIBLES | Les contrôles doivent être accessibles pour les recetteurs | L'application doit mettre à disposition des recettes techniques et fonctionnelles un moyen d'accès aux contrôles | Technique |
| MOE | EX\_TEC\_20\_STOCKER\_RAPPORT\_CONTROLE | Les rapports de contrôles doivent être stockés sur le serveur | L'application doit permettre le stockage des rapports de contrôle sur le serveur pendant une période fixée par la MOA | Technique |
| MOE | EX\_TEC\_21\_EXPORTER\_RAPPORT\_CONTROLE | Les rapports de contrôle stockés sur le serveur doivent pouvoir être exportés vers les disques durs des acteurs | L'application doit permettre aux acteurs d'importer sur leur disque dur les rapports de contrôle stockés sur le serveur | Technique |
| MOE | EX\_TEC\_22\_ECRASER\_RAPPORTS\_CONTROLES | Les rapports de contrôle stockés sur le serveur doivent pouvoir être écrasés automatiquement après un délai fixé par la MOA ou manuellement | L'application doit pouvoir écraser automatiquement les rapports de contrôle sur le serveur après un délai fixé par la MOA. Elle doit en outre permettre à un Administrateur applicatif d'écraser manuellement les rapports de contrôle | Technique |
| MOE | EX\_TEC\_23\_AGREGER\_RAPPORTS\_CONTROLES | Les rapports de contrôle doivent pouvoir être agrégés afin que la MOA dispose d'un rapport unique pour une période déterminée | L'application doit pouvoir servir des rapports de contrôle concaténables afin d'obtenir un rapport unique pour une période déterminée | Technique |
| MOE | EX\_TEC\_24\_RAPPORTS\_CONTROLE\_CSV | Les rapports de contrôle doivent être enregistrés au format csv | L'application doit pouvoir servir des rapports de contrôle au format csv | Technique |
| MOE | EX\_TEC\_25\_RAPPORTS\_CONTROLE\_TXT | Les rapports de contrôle doivent être enregistrés au format textuel | L'application doit pouvoir servir des rapports de contrôle au format textuel | Technique |
| MOE | EX\_TEC\_26\_MEMORISER\_NUMEROS\_ORIGINAUX | Les rapports de contrôle doivent utiliser les numéros de ligne originaux (numéros de ligne dans le fichier soumis) | L'application doit servir des rapports de contrôle utilisant les numéros de ligne originaux (numéros de ligne dans le fichier soumis) | Technique |
| MOE | EX\_TEC\_28\_CONTROLE\_INTEGRAL | Les contrôles doivent pouvoir être appliqués sur la totalité d'un fichier sans application de la suite à donner (retrait de ligne, …) | L'application doit proposer un mode d'exécution des contrôles sur la totalité d'un fichier (sans retrait de ligne, …) | Technique |

### Synthèse des RG capturées

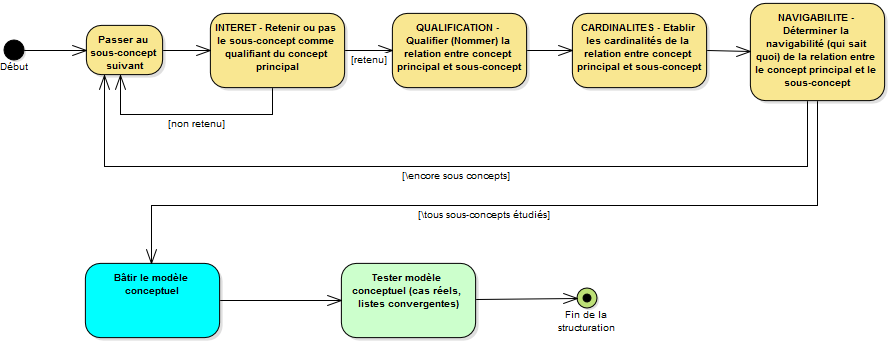


| **Dem.** | **IDENTIFIANT** | **DESCRIPTION** | **PORTEE** | **EXIGENCE ASSOCIEE** |
| --- | --- | --- | --- | --- |
| MOA | RG-01\_FICHIER\_CONTROLE\_VALIDITE | L'application cible doit contrôler la validité des fichiers acceptés en entrée | Tous Fichiers | EX\_FONCT\_01\_CONTROLE\_DE\_VALIDITE |
| MOE | RG-100\_TRUE\_SI\_CONTROLE\_FAVORABLE | Un contrôle favorable (qui a détecté que l'objet remplissait le critère) doit répondre true dans l'application.  Par exemple le résultat du contrôle "Pas de doublons" à true signifie que le fichier ne comporte pas de doublons. | Tous Fichiers |  |
| MOA | RG-101\_TOUT\_CONTROLE\_RAPPORTE | L'application cible doit fournir des rapports de contrôle expliquant le résultat de tous ses contrôles et traitements | Tous Fichiers | EX\_FONCT\_03\_RAPPORT\_CONTROLE |
| MOA | RG-102\_REGLES\_ART | L'application cible doit exécuter tous les contrôles conformes aux règles de l'art | Tous Fichiers | EX\_FONCT\_06\_REGLE\_ART |
| MOA | RG-02\_FICHIER\_CONTROLE\_METIER | L'application cible doit contrôler que les données métier satisfont à des règles de gestion | Tous Fichiers | EX\_FONCT\_02\_CONTROLE\_METIER |
| MOA | RG-01-01\_FICHIERS\_TEXTE | L'application cible ne doit admettre en entrée que des fichiers texte (HIT, HISTO\_F07, DARWIN, ,,,) | Tous Fichiers | EX\_TEC\_05\_FICHIER\_TEXTE |
| MOE | RG-01-02\_TRANSCODAGE\_UTF-8 | Tous les fichiers gérés par l'application doivent être transcodés en interne en UTF-8. | Tous Fichiers | EX\_TEC\_06\_TRAITEMENT\_ENCODAGE\_UTF8 |
| MOE | RG-01-03\_FICHIERS\_ENCODAGE\_UTF-8 | L'application cible doit contrôler que les fichiers internes sont encodés en UTF-8 | Tous Fichiers | EX\_TEC\_07\_CONTROLE\_ENCODAGE\_UTF8 |
| MOA | RG-01-04\_FICHIERS\_TYPE\_TRAFIC | L'application cible doit détecter le type du fichier entrant (HIT, HISTO\_F07, DARWIN, …) et n'accepter en entrée que ces types de fichiers trafic | Tous Fichiers | EX\_TEC\_08\_DETECTION\_TYPE\_FICHIER |
| MOE | RG-01-05\_ELIMINATION\_DOUBLONS\_INTERLIGNES | Aucun fichier ne doit comporter de lignes en doublons. Si des doublons (triplettes, …) sont trouvés, seul le premier exemplaire doit être conservé. Les autres sont éliminés et le programme continue. | Tous Fichiers | EX\_TEC\_09\_FICHIER\_SANS\_DOUBLONS\_LIGNE |
| MOE | RG-01-06\_ELIMINATION\_LIGNES\_VIDES | Aucun fichier ne doit comporter de lignes vides. Si des lignes vides sont trouvées, elles sont éliminées eet le programme continue. | Tous Fichiers | EX\_TEC\_10\_FICHIER\_SANS\_LIGNES\_VIDES |
| MOA | RG-01-07\_ELIMINATION\_LIGNE\_REDHIBITOIRE | Aucun fichier ne doit comporter de lignes avec des champs "redhibitoires" mal formés. Si une telle ligne est trouvée, elle est éliminée et le programme continue | Tous Fichiers | EX\_TEC\_15\_CHAMPS\_REDHIBITOIRES |
| MOA | RG\_DETECTION\_HIT\_01\_LIGNES | Dans un HIT, toutes les lignes doivent comporter exactement 520 caractères | HIT | EX\_TEC\_08\_DETECTION\_TYPE\_FICHIER |
| MOA | RG-200-1\_CONTROLES PARAMETRABLES | Un contrôle ne doit s'appliquer et rapporter que si il est activé | Tous Fichiers | EX\_FONCT\_07\_CONTROLES\_PARAMETRABLES |
| MOA | RG-300-1\_ANNEE\_EXPLOITATION | L'application doit retirer toute la ligne du fichier si l'année de la ligne n'est pas l'année d'exploitation. Si une telle ligne est trouvée, elle est éliminée et le programme continue | Tous Fichiers | EX\_TEC\_16\_ANNEE\_EXPLOITATION |
| MOA | RG-300-2\_LOCALISATIONS\_RENSEIGNEES | L'application doit retirer toute la ligne du fichier si une des localisations de la ligne n'est pas renseignée, Si une telle ligne est trouvée, elle est éliminée et le programme continue. | Tous Fichiers | EX\_TEC\_17\_LOCALISATIONS\_RENSEIGNEES |
| MOA | RG-300-3\_SENS\_CONFONDUS | Les sens confondus doivent être égaux à la somme des sens croissants et décroissants pour une même section. Si une telle ligne est trouvée, l'application rédige un message d'alerte, la ligne est conservée et le programme continue | Tous Fichiers | EX\_TEC\_18\_SENS\_CONFONDUS |

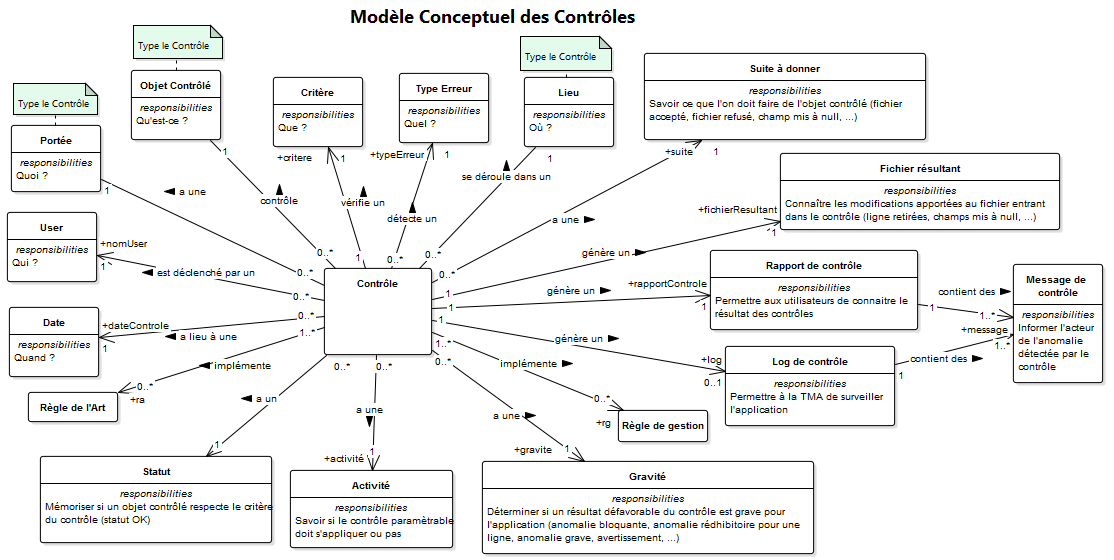
## Affiner et Structurer les concepts

Objectif :

* Décider de l'**intérêt** des sous-concepts capturés pour qualifier le concept Contrôle (sous-concepts retenus ou pas dans le modèle conceptuel)
* **Qualifier** les relations entre concepts
* Révéler les **cardinalités** des relations entre concepts
* Révéler la **navigabilité** des relations entre concepts

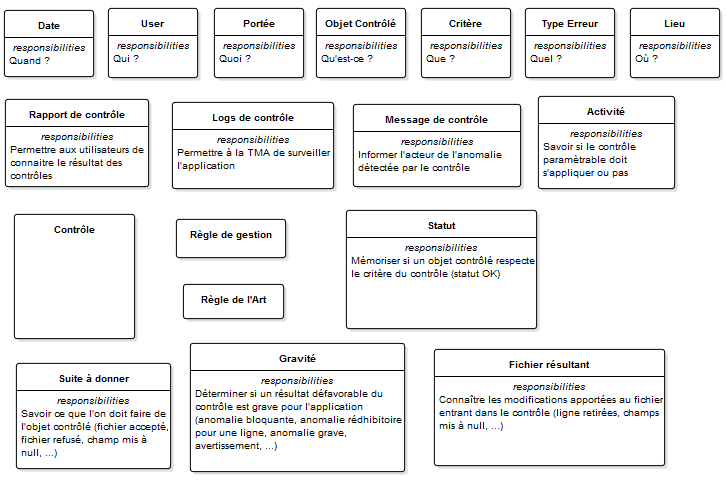


Bénéfices attendus : **liste** **des relations entre concepts** avec leurs qualifications, cardinalités et navigabilités afin d'établir le **modèle conceptuel** (aussi appelé modèle du domaine).



Ce travail est réalisé dans le **D**ossier d'**A**nalyse **F**onctionnel – Partie Analyse (DAF Analyse) d'un projet Khefren.

Concepts déjà mis en évidence :



### Concept Date

Le concept Date permet de savoir quand a eu lieu le contrôle.

#### Intérêt du concept Date

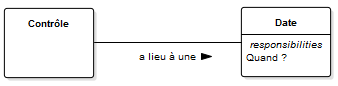
Ce concept est indispensable pour identifier un contrôle. La date du contrôle doit en outre apparaître dans les rapports de contrôle (EX\_FONCT\_05\_IDENTIFIANT\_CONTROLE).

Par ailleurs, la date d'exécution d'un contrôle *varie* en fonction de la date système à laquelle le contrôle s'est exécuté. Le concept Date participe donc à l'**état** (tout ce que sait le Contrôle) du concept Contrôle et doit pouvoir changer.

Le concept de date du contrôle est donc **retenu** pour qualifier un contrôle.

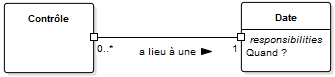
#### Qualification de la relation entre Contrôle et Date

Un Contrôle **a lieuà une** Date de contrôle.



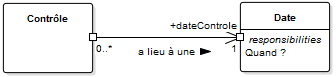
#### Cardinalités entre Contrôle et Date

* Un contrôle a lieu à **une et une seule** Date de contrôle donnée.
* A une Date donnée peuvent s'exécuter **0 ou** **plusieurs contrôles**.



#### Navigabilité entre Contrôle et Date

* Un Contrôle **sait** toujours à quelle Date il a eu lieu
* Une Date ne **sait pas** quels contrôles ont été exécutés à cette Date dans l'application TraficWeb. On obtiendra cette information en triant sur la Date le rapport de contrôle sous forme tabulaire (csv).



### Concept User

Le User désigne traditionnellement l'acteur qui a déclenché une action en informatique. Dans une application web, c'est l'internaute loggé en session web qui est en train de réaliser un cas d'utilisation. Il a le plus souvent un nom, un prénom, un Login, un mot de passe, un profil applicatif (profil Cerbere au Ministère), …

#### Intérêt du concept User

Un contrôle n'a pas besoin de toutes ces informations. En revanche, il est nécessaire de connaître le **nom** du User qui a déclenché un contrôle à des fins de tri (ou plus exactement le nom de l'organisation qui emploie le User comme DIRA, DIRE, DARWIN, …). Le User qui a déclenché le contrôle doit en outre apparaître dans les rapports de contrôle (EX\_FONCT\_05\_IDENTIFIANT\_CONTROLE).

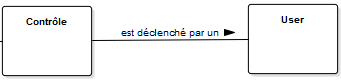
Par exemple, il est nécessaire de savoir que c'est la DIR Atlantique (DIRA) qui vient de déposer le fichier HIT\_DIRA\_2014, et que ce dépôt a déclenché un contrôle prouvant qu'un département était mal formé.

Par ailleurs, le User qui a déclenché un contrôle *varie* potentiellement à chaque exécution du contrôle. Le concept User participe donc à l'**état** (tout ce que sait le Contrôle) du concept Contrôle et doit pouvoir changer.

User est donc un concept **retenu** même si c'est *le nom du User qui sera utilisé*.

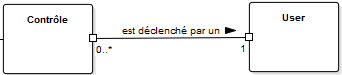
#### Qualification de la relation entre Contrôle et User

Un Contrôle **est déclenché par un** User



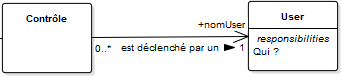
#### Cardinalités entre Contrôle et User

* Un Contrôle est déclenché par **un et un seul** User.
* Un User peut déclencher **0 ou plusieurs contrôles**.



#### Navigabilité entre Contrôle et User

* Un Contrôle **sait** toujours quel User l'a déclenché
* Un User ne **sait pas** quels Contrôles il a déclenché dans l'application TraficWeb. On obtiendra cette information en triant sur le User le rapport de contrôle sous forme tabulaire (csv).



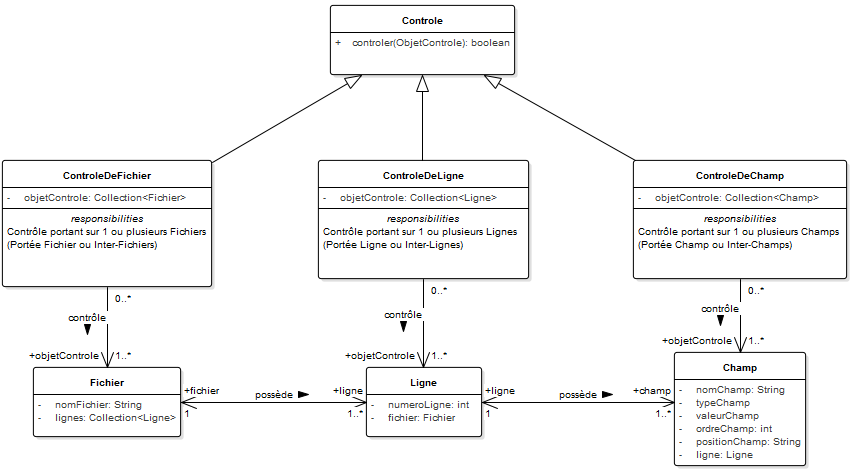
### Concept Portée

Le concept Portée détermine sur quoi (un fichier, inter-fichiers, une ligne, inter-lignes, un champ, inter-champs) porte un contrôle.

Ce concept est intéressant pour typer les messages de contrôle qui diffèrent en fonction de l'objet contrôlé.

* Par exemple, un contrôle inter-fichiers qui contrôlera que "le fichier A est plus grand que le fichier B" nécessitera un message du type : "true, le fichier A est plus grand que B".
* Un contrôle de portée fichier sur le contenu du fichier HIT\_DIRA\_2014 nécessitera un message du type : "le fichier HIT\_DIRA\_2014 n'est pas vide. OK".
* Un contrôle sur le renseignement du champ Département d'une ligne n dans un fichier HIT\_DIRA\_2014 nécessitera un message du type : "le champ Département (1er champ) situé entre les positions 0 et 3 inclus à la ligne n du fichier HIT\_DIRA\_2014 n'est pas renseigné".

Le concept Portée est donc intéressant pour qualifier l'objet contrôlé et le message de contrôle à générer. Ce concept peut aider le Développeur à packager son application en rangeant les contrôles de portée inter-fichiers dans un package, les contrôles de portée fichier dans un autre, puis les contrôles de portée inter-lignes à part, etc…



En revanche, il n'y a pas d'intérêt pour la MOA à trier les rapports de contrôles sur la portée. Il n'y a donc pas lieu de qualifier les contrôles (en ajoutant un attribut) avec ce concept Portée. Le code s'en chargera.

La portée d'un certain type de contrôle ne varie jamais ni à l'instanciation, ni lors de la durée de vie du contrôle. Si par exemple on veut contrôler qu'un fichier n'est pas vide (contrôle de contenu non vide de fichier), on sait dès l'instanciation que ce contrôle a une portée fichier. Et cette portée n'a aucune chance de varier durant toute la durée de vie de ce contrôle.

Le concept Portée n'est donc **pas retenu** pour qualifier le concept Contrôle.

Il est contenu dans le **type** du contrôle. On type les contrôles de fichier, de ligne, de champ, … en définissant des **sous-classes** d'une classe de haut niveau Contrôle.

Cette **hiérarchie de classes** permet de dire qu'un contrôle de fichier est bien un contrôle, mais un peu particulier puisqu'il ne s'applique qu'à des fichiers.

### Concept Objet Contrôlé

Tout contrôle s'applique à un "objet contrôlé" (2 ou plusieurs fichiers, un seul fichier, deux ou plusieurs lignes, une seule ligne, deux ou plusieurs champs, un seul champ).

L'objet contrôlé est par exemple : "les fichiers HIT\_DIRA\_2013 et HIT\_DIRA\_2014", ou "le fichier HIT\_DIRA\_2014", ou encore "le champ Département de la ligne n du fichier HIT\_DIRA\_2014", …

Par exemple :

* Un contrôle déterminera si le fichier HIT\_DIRA\_2014 n'est pas vide
* Un autre contrôle déterminera si la 3ème ligne du fichier HIT\_DIRA\_2014 n'est pas vide.
* Un contrôle déterminera si le champ Département d'une ligne n d'un fichier HIT\_DIRA\_2014 est bien renseigné.
* Un autre contrôle déterminera si le champ Département d'une ligne n d'un fichier HIT\_DIRA\_2014 est bien numérique.
* Un autre contrôle déterminera si l'écart entre le Trafic Moyen Journalier Annuel (TMJA) de l'année n et la TMJA de l'année n-1 respecte une certaine fourchette dans une ligne n du fichier HIT\_DIRA\_2014.

La MOA aura besoin de comptabiliser le nombre de champs Département mal renseignés dans le fichier HIT\_DIRA\_2014 par exemple.

Le concept Objet Contrôlé est donc à priori retenu.

Mais l'exigence fonctionnelle EX\_FONCT\_07\_CONTROLES\_PARAMETRABLES précise que *chaque contrôle doit pouvoir être activé/désactivé individuellement*.

Le développeur devra donc coder (pour le champ Département par exemple) :

* Un contrôle *ControleChampDepartementRenseigne* qui contrôlera que le Département est bien renseigné à chaque ligne d'un fichier.
* Un contrôle *ControleChampDepartementNumerique* qui contrôlera que le Département est bien homogène à un numérique à chaque ligne d'un fichier.
* Un contrôle *ControleChampDepartementDansNomenclature* qui contrôlera que le Département respecte une nomenclature à chaque ligne du fichier
* Un contrôle *ControleChampDepartementRegex* qui contrôlera que le Département respecte une expression régulière (RegEx) à chaque ligne du fichier.

On constate donc que chaque contrôle **porte** entre autres dans son **type** l'objet contrôlé (champ Département ci-dessus). Comme pour le concept Portée, le typage des classes dans le code se charge de qualifier le contrôle concernant l'objet contrôlé.

De plus, le type de l'Objet Contrôlé est variable en fonction du type du Contrôle.

* L'Objet Contrôlé est de type Fichier (File en Java) pour un contrôle de contenu non vide de fichier.
* L'Objet Contrôlé est de type Ligne (String en Java) pour un contrôle de ligne non vide.
* L'Objet Contrôlé est de type Champ (String en Java) pour un contrôle de format du Département.

Il est donc impossible de dégager un seul type d'Objet Contrôlé pour tous les types de Contrôles.

Le concept Objet Contrôlé n'est donc **pas retenu** pour qualifier le concept Contrôle. Il est contenu dans le **type** du contrôle.

### Concept Critère

Le concept de Critère détermine *ce que le contrôle a contrôlé*.

Par exemple :

|  |  |
| --- | --- |
| Contrôle | Critère |
| *ControleChampDepartementRenseigne* | Le champ Département est renseigné |
| *ControleChampDepartementNumerique* | Le champ Département est homogène à un numérique |
| *ControleChampDepartementDansNomenclature* | Le champ Département respecte la nomenclature du Département dans un HIT |
| *ControleChampDepartementRegex* | Le champ Département respecte l'expression régulière (RegEx) imposée dans les HIT |

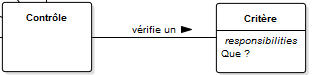
#### Intérêt du concept Critère

Ce concept Critère est intéressant pour pouvoir trier les rapports de contrôles en fonction des critères. Il qualifie le contrôle dans la mesure où il précise ce qui a été contrôlé.

Le concept Critère est **retenu**.

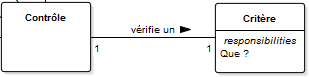
#### Qualification de la relation entre Contrôle et Critère

Un Contrôle **vérifie un** Critère



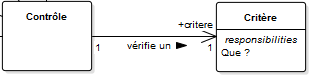
#### Cardinalités entre Contrôle et Critère

* Un Contrôle vérifie **un et un seul** Critère
* Un Critère est vérifié par **un et un seul** Contrôle.



#### Navigabilité entre Contrôle et Critère

* Un contrôle **sait** toujours quel Critère il vérifie.
* Un Critère ne **sait pas** quel Contrôle le vérifie. On obtiendra cette information en triant sur le Critère le rapport de contrôle sous forme tabulaire (csv).



### Concept Type d'Erreur

Le concept Type d'Erreur (TypeErreur) détermine quel *type d'anomalie* le contrôle a détecté.

Par exemple :

| **Type d'Erreur** | **Contrôle** | **Message de contrôle** |
| --- | --- | --- |
| Non renseigné | Controle  ChampDepartementRenseigne | Le Département à la ligne n du fichier HIT\_DIRA\_2014 n'est pas renseigné |
| Non renseigné | Controle  ChampTMJARenseigne | La TMJA de l'année N à la ligne n du fichier HIT\_DIRA\_2014 n'est pas renseignée |
| Mauvais format | Controle  ChampDepartementNumerique | Le Département à la ligne n du fichier HIT\_DIRA\_2014 n'est pas homogène à un entier |

#### Intérêt du concept TypeErreur

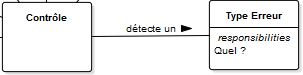
La MOA aura besoin de trier les rapports de contrôle pour comptabiliser les anomalies en fonction du TypeErreur.

Par exemple, comptabiliser le nombre de champs "non renseigné" dans un certain fichier.

Le concept TypeErreur est **retenu**.

#### Qualification de la relation entre Contrôle et TypeErreur

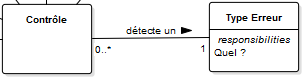
Un contrôle **détecte un** TypeErreur.



#### Cardinalités entre Contrôle et TypeErreur

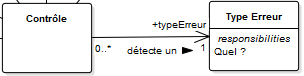
Du fait de l'exigence fonctionnelle EX\_FONCT\_07\_CONTROLES\_PARAMETRABLES qui impose une classe concrète de contrôle pour chaque type d'erreur afin de pouvoir activer/desactiver individuellement chaque contrôle :

* Un Contrôle détecte **un et un seul** TypeErreur
* Un TypeErreur peut être détecté par **plusieurs contrôles (ou zéro)**. C'est le cas par exemple du type d'erreur "non renseigné" qui peut être détecté pour tous les champs d'un fichier (TMJA, Département, …) et donc par divers contrôles individualisés *ControleChampDepartementRenseigne*, *ControleChampTMJARenseigne*, …).



#### Navigabilité entre Contrôle et TypeErreur

* Un Contrôle **sait** toujours quel TypeErreur il détecte
* Un TypeErreur ne **sait pas** quel Contrôle le détecte. On obtiendra cette information en triant sur le TypeErreur le rapport de contrôle sous forme tabulaire (csv).



### Concept Lieu

Le concept de Lieu détermine où le contrôle a eu lieu dans l'application (contrôle de surface qui a lieu dès la lecture d'un fichier déposé, contrôle métier qui a lieu dans la couche métier après que le fichier déposé ait été décomposé, …).

Ce concept peut aider le Développeur à packager son application en rangeant les contrôles de surface dans un package, les contrôles métier dans un autre, etc…

Mais la MOA n'aura aucun intérêt à trier les rapports de contrôle en fonction des contrôles de surface, des contrôle métier, …

Le concept de Lieu n'est donc **pas retenu**.

### Concept Activité

Le concept Activité détermine si un contrôle doit être exécuté ou pas. Il est lié à la notion de contrôle paramétrable.

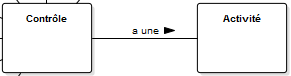
#### Intérêt du concept Activité

La MOA aura besoin de paramétrer les contrôles afin de les activer/désactiver. Elle aura également besoin d'obtenir de l'application la liste des contrôles implémentés et de savoir lesquels sont actifs.

Le concept Activité est donc **retenu**.

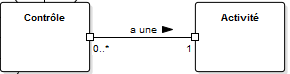
#### Qualification de la relation entre Contrôle et Activité

Un Contrôle a une Activité (il est actif ou inactif)



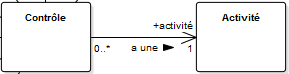
#### Cardinalités entre Contrôle et Activité

* Un Contrôle a une et une seule Activité (il est actif ou inactif).
* Une Activité peut concerner 0 ou plusieurs Contrôles.



#### Navigabilité entre Contrôle et Activité

* Un Contrôle **sait** toujours s'il est actif ou inactif.
* Une Activité **ne sait pas** quels contrôles sont actifs (ou inactifs).



### Concept Statut

Le concept Statut sert à mémoriser si un contrôle a matché favorablement (l'objet contrôlé respecte le critère de contrôle et le statut est OK) ou défavorablement (l'objet contrôlé ne respecte pas le critère de contrôle et le statut est KO).

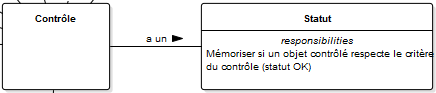
#### Intérêt du concept Statut

La MOA aura besoin de trier les rapports de contrôle pour savoir quels sont tous les contrôles qui ont matché défavorablement.

Le concept Statut est donc **retenu**.

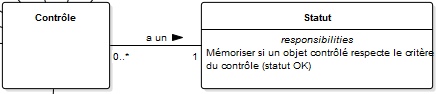
#### Qualification de la relation entre Contrôle et Statut

Un Contrôle a un Statut après son exécution.



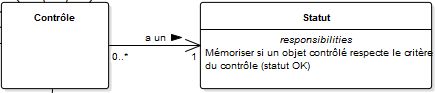
#### Cardinalités entre Contrôle et Statut

* Un Contrôle a un et un seul Statut.
* Un Statut (OK/KO) peut concerner 0 ou plusieurs Contrôles.



#### Navigabilité entre Contrôle et Statut

* Un Contrôle sait toujours quel est son Statut (OK/KO) après son exécution.
* Un Statut ne sait pas quels contrôles sont concernés (par exemple, le Statut KO ne sait pas quels contrôles sont KO). Cette information sera accessible en triant les rapports de contrôle sur le statut.



### Concept Gravité

Le concept Gravité permet de mémoriser si un Contrôle a un impact important pour l'application (anomalie bloquante pour un fichier, anomalie rédhibitoire pour une ligne, avertissement, …). Par exemple, un Département non renseigné ou mal formé dans une ligne d'un fichier HIT est une anomalie rédhibitoire. En effet, cette ligne ne pourra pas être convenablement localisée et perd tout intérêt.

#### Intérêt du concept Gravité

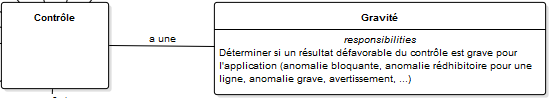
La MOA aura besoin de trier les rapports de contrôle pour connaître toutes les anomalies rédhibitoires pour des lignes de fichier par exemple.

Le concept Gravité est donc **retenu**.

#### Qualification de la relation entre Contrôle et Gravité

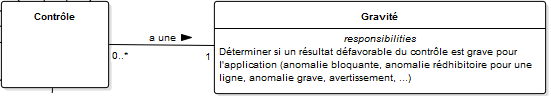
Un Contrôle **a une** Gravité.

Par exemple, un contrôle sur le format numérique d'un Département dans une ligne d'un fichier HIT peut déclencher une anomalie rédhibitoire pour la ligne.



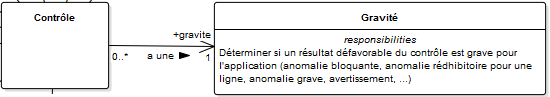
#### Cardinalités entre Contrôle et Gravité

* Un Contrôle a **une et une seule** Gravité.
* Une Gravité peut concerner **0 ou plusieurs** Contrôles.



#### Navigabilité entre Contrôle et Gravité

* Un Contrôle **sait** toujours quelle est sa Gravité.
* Une Gravité ne **sait pas** quels Contrôles elle concerne.



### Concept Suite A Donner

Le concept Suite A Donner permet de savoir ce que le contrôle doit faire avec l'objet contrôlé *en cas de contrôle défavorable*.

Par exemple, un contrôle de surface défavorable de format numérique d'un champ Département dans une ligne d'un fichier HIT devra supprimer la ligne.

Un contrôle métier défavorable de respect d'une fourchette sur le **T**rafic **M**oyen **J**ournalier **A**nnuel (TMJA) dans une ligne de fichier HIT devra annuler cette TMJA (s'il existe une telle **R**ègle de **G**estion RG) mais conserver la ligne.

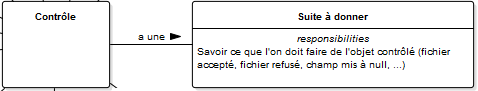
#### Intérêt du Concept Suite A Donner

La MOA aura besoin de connaître les lignes supprimées, les champs annulés, les champs conservés mais ayant déclenché un avertissement, …

Le concept Suite A Donner est **retenu**.

#### Qualification de la relation entre Contrôle et Suite A Donner

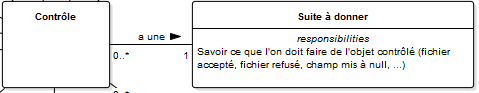
Un Contrôle **a une** Suite A Donner.



#### Cardinalités entre Contrôle et Suite A Donner

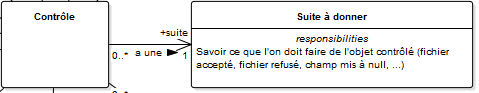
Un Contrôle a **une et une seule** Suite A Donner à un objet contrôlé.

Une Suite A Donner concerne **0 ou plusieurs** Contrôles.



#### Navigabilité entre Contrôle et Suite A Donner

* Un Contrôle **sait** toujours quelle Suite il doit donner à l'objet contrôlé.
* Une Suite A Donner ne **sait pas** quels Contrôles sont concernés.



### Concept Fichier Résultant

Le concept Fichier Résultant permet de mémoriser les changements apportés à un fichier par un contrôle. Certains contrôles peuvent en effet transformer le fichier entrant en lui retirant des lignes, en annulant des champs, …

Ce concept est primordial pour pouvoir enchaîner les contrôles

#### Intérêt du Concept Fichier Résultant

La MOA aura besoin de connaître le fichier résultant de la **chaîne des contrôles** de l'application.

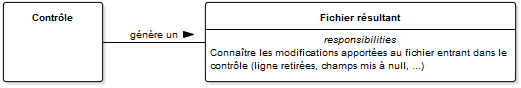
En outre, les développeurs ont impérativement besoin d'accéder au Fichier Résultant d'un contrôle puisque le contrôle suivant ne doit s'appliquer qu'au Fichier Résultant.

Le concept Fichier Résultant est **retenu**.

#### Qualification de la relation entre Contrôle et Fichier Résultant

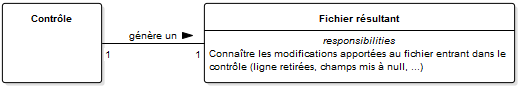
Un Contrôle **génère un** Fichier Résultant.

Ce Fichier Résultant peut le cas échéant être le même que le fichier en entrée (cas des contrôles qui n'exécutent aucun traitement).



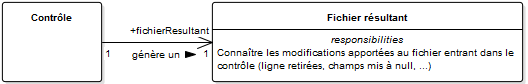
#### Cardinalités entre Contrôle et Fichier Résultant

* Un Contrôle génère **un et un seul** Fichier Résultant.
* Un Fichier Résultant et généré par **un et un seul** Contrôle.



#### Navigabilité entre Contrôle et Fichier Résultant

* Un Contrôle **sait** toujours quel Fichier Résultant il a généré.
* Un Fichier Résultant ne **sait pas** quel Contrôle l'a généré.



### Concept Règle de Gestion

Un contrôle met en application 0, 1 ou plusieurs Règles de Gestion.

#### Intérêt du Concept Règle de Gestion

La MOA n'a pas besoin de trier les rapports de contrôle en fonction des règles de gestion.

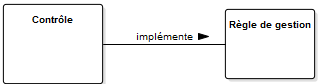
En revanche, le **chef de projet informatique** et le **développeur (TMA)** peuvent souhaiter vérifier que toutes les règles de gestion ont bien été implémentées lors de l'administration des contrôles.

Il est intéressant que le code "sache dire" où et comment les règles de gestion ont été implémentées dans le cadre de l'élaboration du projet informatique (traçabilité des concepts dans un projet Entreprise Architect par exemple).

Le concept de Règle de Gestion est donc **retenu**.

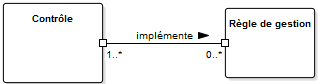
#### Qualification de la relation entre Contrôle et Règle de Gestion

Un Contrôle **implémente** des Règles de Gestion



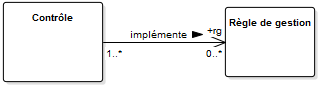
#### Cardinalités entre Contrôle et Règle de Gestion

* Un Contrôle implémente **0, 1 ou plusieurs** Règles de Gestion.
* Un Règle de Gestion est implémentée par **1 ou plusieurs** Contrôles.



#### Navigabilité entre Contrôle et Règle de Gestion

* Un Contrôle **sait** quelles Règles de Gestion il implémente.
* Une Règle de Gestion ne **sait pas** Quels Contrôles l'implémentent. Nous aurons accès à cette information en triant le rapport d'administration des contrôles – qui liste pour chaque contrôle les règles de gestion qu'il implémente - sur les règles de gestion.



### Concept Règle de l'Art

Un contrôle met en application 0, 1 ou plusieurs Règles de l'Art.

#### Intérêt du Concept Règle de l'Art

La MOA n'a pas besoin de trier les rapports de contrôle en fonction des règles de l'art.

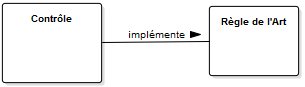
En revanche, le **chef de projet informatique** et le **développeur (TMA)** peuvent souhaiter vérifier que toutes les règles de l'art ont bien été implémentées lors de l'administration des contrôles.

Il est intéressant que le code "sache dire " où et comment les règles de l'art ont été implémentées dans le cadre de l'élaboration du projet informatique (traçabilité des concepts dans un projet Entreprise Architect par exemple).

Le concept de Règle de l'art est donc **retenu**.

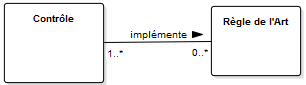
#### Qualification de la relation entre Contrôle et Règle de l'Art

Un Contrôle **implémente** des Règles de l'Art



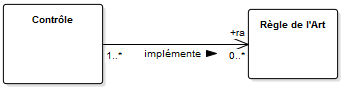
#### Cardinalités entre Contrôle et Règle de l'Art

* Un Contrôle implémente **0, 1 ou plusieurs** Règles de l'Art.
* Un Règle de l'Art est implémentée par **1 ou plusieurs** Contrôles.



#### Navigabilité entre Contrôle et Règle de l'Art

* Un Contrôle **sait** quelles Règles de l'Art il implémente.
* Une Règle de l'Art ne **sait pas** Quels Contrôles l'implémentent. Nous aurons accès à cette information en triant le rapport d'administration des contrôles – qui liste pour chaque contrôle les règles de l'art qu'il implémente - sur les règles de l'art.



### Concept Rapport de Contrôle

Le concept Rapport de Contrôle détermine ce dont l'acteur va bénéficier pour prendre connaissance de la qualité d'un fichier de trafic soumis à l'application.

Le Rapport de contrôle doit être au format csv pour être exportable vers des tableurs et au format textuel pour des **I**nterfaces **H**omme-**M**achine (IHM) autonomes à fenêtres (Desktop).

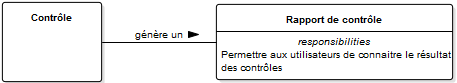
#### Intérêt du Concept Rapport de Contrôle

Tous les acteurs auront besoin de consulter les rapports de contrôle.

Le concept Rapport de Contrôle est donc **retenu**.

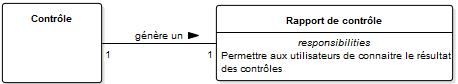
#### Qualification de la relation entre Contrôle et Rapport de Contrôle

Un Contrôle **génère un** Rapport de Contrôle.



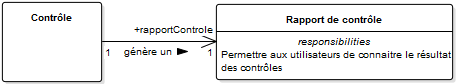
#### Cardinalités entre Contrôle et Rapport de Contrôle

* Un Contrôle génère **un et un seul** Rapport de Contrôle.
* Un Rapport de Contrôle est généré par **un et un seul** Contrôle.



#### Navigabilité entre Contrôle et Rapport de Contrôle

* Un Contrôle **sait** quel Rapport de Contrôle il a généré.
* Un Rapport de Contrôle ne **sait pas** quel Contrôle l'a généré.



### Concept Log de Contrôle

Le concept Log de contrôle détermine ce que les acteurs applicatifs (Administrateur Applicatif, Développeur (TMA), Recetteur Technique, Centre Serveur) vont pouvoir observer à postériori afin d'assurer le suivi de l'application informatique.

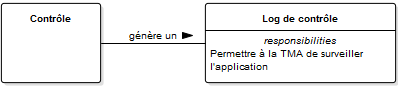
#### Intérêt du Concept Log de Contrôle

L'**Administrateur Applicatif**, le **Développeur (TMA)**, le **Recetteur Technique** et le **Centre Serveur** peuvent avoir besoin de consulter les Logs de Contrôle.

Le concept Log de Contrôle est **retenu**.

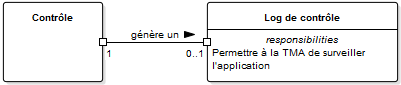
#### Qualification de la relation entre Contrôle et Log de Contrôle

Un Contrôle **génère un** Log de Contrôle.



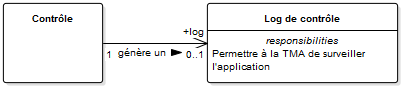
#### Cardinalités entre Contrôle et Log de Contrôle

* Un Contrôle génère **zéro ou un seul** Log de Contrôle.
* Un Log de Contrôle est généré par **un et un seul** Contrôle.



#### Navigabilité entre Contrôle et Log de Contrôle

* Un Contrôle **sait** quel Log de Contrôle il a généré.
* Un Log de Contrôle ne **sait pas** quel Contrôle l'a généré.



### Concept Message de Contrôle

Le concept Message de Contrôle détermine ce que l'application "dit" lorsqu' un contrôle a matché favorablement ou défavorablement.

Par exemple, un contrôle de format numérique sur le champ Département dans une ligne d'un fichier HIT pourra émettre le message : "le Département de la ligne 3 du fichier HIT\_DIRA\_2014 n'est pas homogène à un entier : '20A'. Cette ligne 3 est donc rejetée".

Le Message de Contrôle est encapsulé dans un Rapport de Contrôle ou un Log. Ce n'est donc pas directement un concept de Contrôle. C'est un concept lié à Rapport de Contrôle et Log de Contrôle.

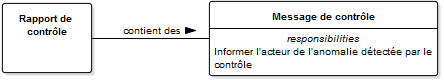
#### Intérêt du Concept Message de Contrôle

Tous les acteurs auront besoin de lire les messages de contrôle via le rapport de contrôle ou le Log.

Le concept Message de Contrôle est donc **retenu**.

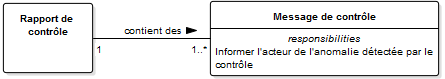
#### Qualification de la relation entre Rapport de Contrôle et Message de Contrôle

Un Rapport de Contrôle (ou Log) **contient** **des** Messages de Contrôle.



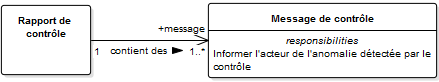
#### Cardinalités entre Rapport de Contrôle et Message de Contrôle

* Un Rapport de Contrôle contient **un ou plusieurs** Message de Contrôle.
* Un Message de Contrôle est contenu dans **un et un seul** Rapport de Contrôle.



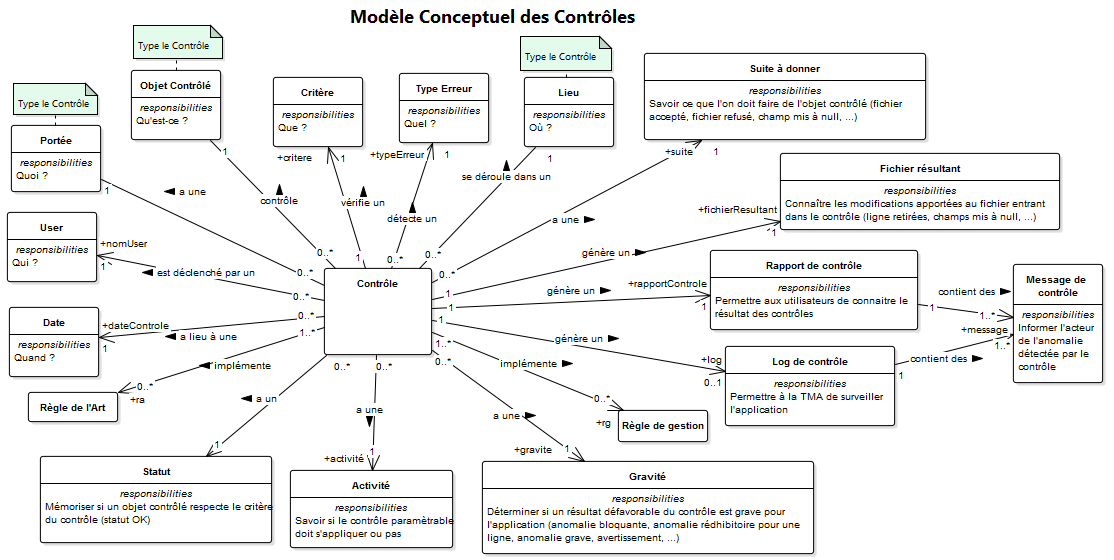
#### Navigabilité entre Rapport de Contrôle et Message de Contrôle

* Un Rapport de Contrôle **sait** quels Messages de Contrôle il contient.
* Un Message de Contrôle **ne sait pas** quel Rapport de Contrôle le contient.



## Bâtir le modèle conceptuel des Contrôles

Objectif : **synthétiser les concepts et relations** précédentes. A ce stade, nous avons bâti le modèle conceptuel suivant :

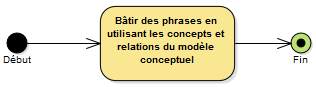


## Tester le modèle conceptuel

Objectif : **tester** le modèle conceptuel précédemment établi afin de s'assurer qu'il permet effectivement de décrire un contrôle.

Bénéfices attendus : **modèle conceptuel** opérationnel.

Ce travail est réalisé dans le **D**ossier d'**A**nalyse **F**onctionnel – Partie Analyse (DAF Analyse) d'un projet Khefren.



### Bâtir des phrases utilisant le modèle conceptuel

Objectif : vérifier que les concepts et leurs relations avec leurs cardinalités et leur navigabilité permettent d'exprimer un contrôle.

Test d'un contrôle de nullité de ligne d'un fichier :

Le *contrôle de nullité* de la **3ème ligne du fichier HIT\_DIRA\_2014** déclenché par la DIRA (User) à la date du 15/06/2016 à 17h35 (Date) a matché favorablement (statut OK).

Il a vérifié que la ligne n'était pas vide (critère) et n'a pas déclenché d'anomalie de type "ligne vide dans fichier" (typeErreur).

La nullité de la ligne aurait constitué une anomalie bloquante (gravité) et aurait entrainé la suppression de la ligne (suiteDonnee). Le contrôle étant favorable (statut OK), la ligne a été conservée (suiteDonnee) et le fichier résultant (fichierResultant) en sortie du contrôle est identique au fichier en entrée.

Le contrôle a généré un message favorable (messageControle) stocké dans le rapport de contrôle (rapportControle).

Test du bon format d'un département d'une ligne d'un fichier HIT :

Le *contrôle de bon format numérique* du **champ Département à la 3ème ligne du fichier HIT\_DIRA\_2014** déclenché par la DIRA (User) à la date du 15/06/2016 à 17h35 (Date) a matché défavorablement (statut KO).

Il a vérifié que le champ Département était homogène à un entier (critère) et ce n'était pas le cas. Il a déclenché une anomalie de type "mauvais format de champ" (typeErreur).

Le mauvais format du champ constitue une anomalie rédhibitoire (gravité). Le contrôle étant défavorable (statut KO), le champ a été annulé (suiteDonnee) et le fichier résultant (fichierResultant) en sortie du contrôle est différent du fichier en entrée.

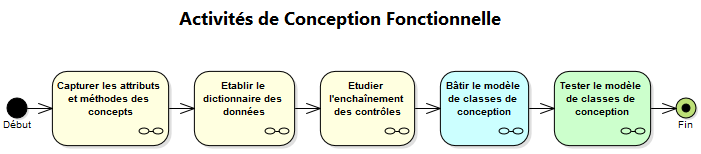
Le contrôle a généré un message défavorable (messageControle) stocké dans le rapport de contrôle (rapportControle).

# CONCEPTION FONCTIONNELLE

Objectif : L'objectif du chapitre "Conception fonctionnelle" est d'**affiner** le modèle conceptuel en **capturant** les **attributs** (*ce que doit "savoir" l'objet*) et les **méthodes** (*ce que doit "savoir-faire" l'objet*) de chaque concept afin d'obtenir un **modèle de classes de conception**.

Bénéfices attendus : la conception fonctionnelle doit fournir un **modèle de classes de conception** *testé*.

Ce travail est réalisé dans le **D**ossier d'**A**nalyse **F**onctionnel – Partie Conception (DAF Conception) d'un projet Khefren.



## Capture des attributs et méthodes des concepts

Objectif : l'objectif de cette phase "capture des attributs et méthodes des concepts" est d'obtenir un **modèle de classes de conception** *à partir du* **modèle conceptuel** précédent. Les concepts sont progressivement transformés en classes (au sens Objet) et voient apparaître leurs attributs et leurs premières méthodes. On détermine dans cette phase le *type des attributs* et les *attributs dépendants*. On commence également à définir une *hiérarchie de classes*.

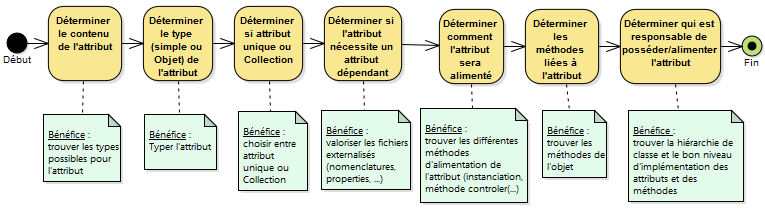
On établit le **dictionnaire des données** de la classe Contrôle, c'est-à-dire la liste de ses attributs (= ce que la classe "sait").

Bénéfices attendus : **liste des attributs et méthodes** des concepts.

Ce travail est réalisé dans le **D**ossier d'**A**nalyse **F**onctionnel – Partie Analyse (DAF Analyse) d'un projet Khefren.

### Méthode de capture des attributs et méthodes

L'enchaînement de questions suivant est une méthode pour capturer les attributs :



#### Déterminer le contenu de l'attribut (examiner la relation dans le modèle conceptuel) :

Objectif : deviner le type à donner à l'attribut en se basant sur le concept.

L'examen de la relation entre concepts dans le modèle conceptuel donne une bonne idée de ce que doit contenir l'attribut. Par exemple :

 Permet de deviner qu'une Personne aura un attribut 'nom' qui sera probablement un String (chaîne de caractères). Le contenu de l'attribut est "homogène" à un nom.

#### Déterminer le type (simple ou Objet) de l'attribut (examiner la navigabilité de la relation dans le modèle conceptuel)

Objectif : déterminer si l'attribut peut avoir un type primitif Java ou un type Objet Java prédéterminé. Sinon, l'attribut doit avoir pour type un objet du domaine.

* Les types que j'appelle "**simples**" sont les :
* **Types primitifs en Java** (comme les entiers **int**, les entiers longs **long**, les caractères **char**, les booléens **boolean**…). Ces primitifs ne sont pas des Objets en Java. Ils ne savent rien "faire" car ils ne possèdent aucune méthode. Ils ont été intégrés au langage Java pour des motifs de performance.
* **Les types Java Objets usuels prédéterminés** (chaîne de caractères **String**, les dates **Date**, les wrappers comme **Integer** qui encapsule un int, …). Ce sont des Objets en Java (donc possédant des méthodes) très couramment utilisés dans les développements informatiques. Une String sait par exemple dire quelle est sa longueur. On les utilise tels quels dans notre code mais ces objets ne savent rien de notre problème particulier puisqu'ils ont été conçus par les inventeurs du langage Java qui ignoraient tout de notre application.
* Les types que j'appelle "**Objet**" sont des types utilisant des Objets du domaine (Bateau, CompteBancaire, Contrôle, …) élaborés par un développeur.

L'examen de la navigabilité de la relation dans le modèle conceptuel permet de savoir si on peut se contenter d'un type simple ou s'il faut fabriquer un Objet.

*Par exemple :*

Permet de dire qu'une Personne connaît son Nom mais qu'un Nom ne sait rien des Personnes qui portent ce nom.

* Le Nom n'ayant rien à savoir des Personnes, on peut mettre un simple attribut 'nom' de type Java prédéfini **String** dans la classe Personne comme suit :

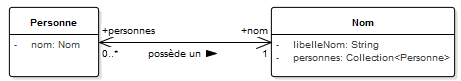


*En revanche, dans le modèle conceptuel suivant (avec navigabilité dans les 2 sens) :*

Le concept Nom connaît les Personnes qui portent ce Nom.

Nom ne peut plus être un type simple (String) puisque Nom doit *connaître* la Collection de Personnes qui portent ce nom.

La traduction en classes de conception devient alors :



* Nom devient un *type Objet* à part entière (et plus simplement un **String**). Le développeur doit fabriquer un type Nom (une classe Nom). Cette classe Nom possède un attribut 'libelleNom' qui contient le nom (Dupont, Durand, …) et un attribut 'personnes' qui est une Collection de Personnes (la collection de Personnes qui portent ce nom).
* L'attribut 'nom' dans la classe Personne a maintenant ce *type* Nom et plus le type simple Java prédéfini String.

#### Déterminer les collections (examiner les cardinalités de la relation dans le modèle conceptuel) :

Objectif : déterminer si l'attribut doit être unique (0 ou 1 valeur) ou multiple (Collection de 0 ou plusieurs valeurs).

L'examen des cardinalités des relations dans le modèle conceptuel permet de déterminer si la classe a besoin d'un attribut unique ou d'un attribut "multiple" que l'on appelle une **Collection**.

 Signifie qu'une Personne possède 0 ou 1 Voiture.

Un attribut unique 'maVoiture' de type Voiture est donc suffisant. Il sera rempli ou pas (nullable).

Attention : Dans ce cas, une Personne ne pourra plus jamais posséder 2 Voitures ou plus puisqu'il n'y a qu'*un seul* attribut 'maVoiture'.

En revanche :

 Signifie qu'une Personne possède 0 ou *plusieurs* Voitures.

La classe Personne a donc ici besoin d'un attribut 'voitures' qui est une **Collection de Voitures**.

#### Déterminer si l'attribut nécessite un attribut dépendant

Objectif : déterminer s'il convient d'associer un attribut plus explicite à l'attribut prévu par le modèle conceptuel.

Il peut être utile de concevoir des attributs que le modèle conceptuel ne prévoyait pas pour faciliter des fonctionnalités de l'application (affichage, calcul, …). Par exemple, une classe Cercle comprenant un attribut déclaré dans le modèle conceptuel 'rayonCercle' peut être enrichie avec un attribut 'circonférenceCercle' *dépendant* de 'rayonCercle' pour afficher directement la circonférence.

Il y a plusieurs types d'attributs *dépendants* :

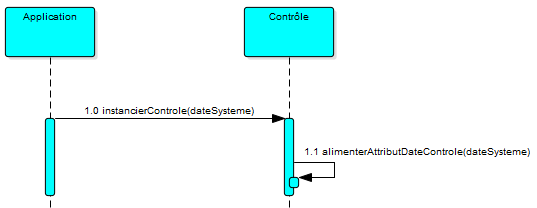
* Les attributs **calculés** à partir d'un attribut maître. L'exemple type est l'*âge* d'une personne que l'on calcule à partir de sa *date de naissance*.
* Les **labels** servant à clarifier l'affichage d'un attribut maître. L'usage est de ne stocker en base que l'attribut maître (par exemple le type d'erreur d'un contrôle sous sa forme entière comme 1, 2, …) mais d'afficher le label correspondant à ce type d'erreur ("NON RENSEIGNE" pour 1, "MAUVAIS FORMAT pour 2, …). Les champs à label sont donc notamment les champs concernés par des **fichiers externalisés** comme des **nomenclatures** au format csv, des fichiers **properties**, …

#### Déterminer comment l'attribut sera alimenté

Objectif : vérifier qu'un attribut est pertinent en vérifiant qu'il est possible de l'alimenter.

Ce point est intéressant car il permet de se pencher dès la conception sur le "comment ça va marcher ? ".

Par exemple, l'application alimente l'attribut 'dateControle' du Contrôle avec la date système dès son instanciation :



Mais il faudra ensuite pouvoir rafraîchir 'dateControle' à chaque fois que le contrôle sera exécuté, c'est-à-dire à chaque fois que l'on invoquera sa méthode contrôler(…).

On conçoit que l'attribut 'dateControle' est pertinent et doit être accessible à chaque exécution du contrôle.

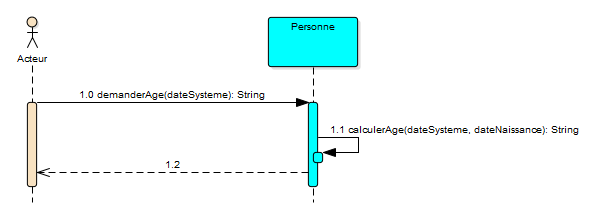
#### Déterminer les méthodes liées à l'attribut

Objectif : trouver les premières méthodes du concept.

Il s'agit de concevoir les méthodes du concept nécessaires au bon fonctionnement de l'attribut.

Par exemple, si on veut demander l'âge d'une personne au système informatique, il faut une classe Personne possédant :

* Un **attribut** 'dateNaissance' de type Date
* Une **méthode** calculerAge(Date pDateSysteme, Date pDateNaissance) qui calcule l'âge



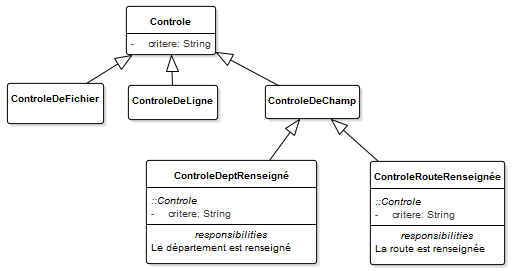
#### Déterminer qui est responsable de l'attribut

Objectif : vérifier que l'attribut figure bien dans le bon concept (la bonne classe en Objet) et commencer à concevoir la **hiérarchie de classes**.

Par exemple, **tout Contrôle vérifie toujours un Critère** (comme la classe ControleDeptRenseigné qui vérifie que le champ Département est renseigné…).

L'attribut 'critere' doit donc figurer dans la classe de haut niveau Contrôle. En effet, tous les descendants de la classe Contrôle (classes ControleDeChamp, ControleRouteRenseignee, …) bénéficient de cet attribut 'critere' par héritage en langage Objet.

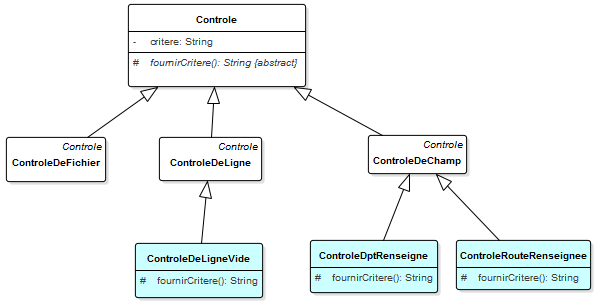
Mais on constate que c'est **chaque classe concrète** de Contrôle (ControleDptRenseigne, ControleRouteRenseignee, …) qui a la responsabilité d'expliciter le critère vérifié :



Plus précisément, tout contrôle vérifie forcément un critère. L'attribut 'critère' est donc **situé au plus haut** niveau dans la hiérarchie des classes de contrôle (aussi appelée arborescence des classes) de manière à **garantir que tous les contrôles en héritant auront un critère**.

Mais ce n'est que lorsque l'on exécutera précisément le contrôle de vérification de la route renseignée (ControleRouteRenseignée) que l'on saura que le critère vérifié est : "la route est renseignée".

C'est donc le contrôle de vérification de la route renseignée (ControleRouteRenseignée) qui détient la *responsabilité* de connaître son critère et de venir alimenter l'attribut 'critere' défini au plus haut niveau. Les méthodes permettant d'alimenter le critère seront donc situées dans un bas niveau dans la hiérarchie des classes (dans chaque classe concrète).



La classe de haut niveau Contrôle définit **abstraitemen**t que tout contrôle devra fournir son critère

Les classes concrètes de bas niveau ont la responsabilité d'**implémenter** la méthode fournirCritere()

Tout contrôle vérifie un critère

On sait dès la conception de la classe de haut-niveau Contrôle que :

* Tout contrôle connaît le critère qu'il vérifie. Toute classe de contrôle a donc un attribut 'critere'. On peut donc définir l'attribut 'critere' dans la classe de plus haut niveau Contrôle, **ce qui garantit que tous ses descendants auront cet attribut 'critere'**.
* Tout contrôle doit savoir fournir le critère qu'il vérifie. On peut donc définir une méthode fournirCritere() au plus haut niveau dans la classe Contrôle. Mais, on ne peut pas implémenter cette méthode à ce niveau puisque la classe de haut niveau Contrôle ne sait rien du critère à vérifier. On définit donc la méthode *fournirCritere()* dans la classe Contrôle comme **abstraite** (elle apparaît en italiques en UML). Elle sert de définition mais *ne fait rien*. Elle ne contient pas de code. Le rôle de cette méthode *abstraite* est d'imposer au descendants de la classe Contrôle d'implémenter cette méthode *fournirCritere()*.
* Ce sont les classes concrètes en bleu sur le diagramme (ControleDeLigneVide, ControleDptRenseigne, …) qui connaissent le critère qu'elles vérifient. Elles détiennent donc la responsabilité d'**implémenter** la méthode fournirCritere(). C'est à ce niveau que se trouve le code qui permet de fournir le critère.

### Capture des attributs et méthodes

#### Attribut simple dateControle (Date)

1. Contenu : date.
2. Type : Simple. Objet Java prédéterminé **java.util.Date**.

Un Contrôle sait toujours à quelle Date il a eu lieu. La Date ne sait rien du Contrôle. Il est donc inutile de créer un Objet particulier (pure fabrication) pour la date du contrôle puisque notre concept ne doit rien savoir et/ou savoir-faire de particulier.

On peut mettre un attribut simple 'dateControle' dans la classe Contrôle et utiliser directement le type **java.util.Date** (Objet) fourni par Java. Une **java.util.Date** est un Objet (pas un primitif) en Java puisque l'on peut lui demander de faire des opérations comme par exemple se décaler d'un mois.

1. Multiplicité : attribut unique.

Un contrôle est déclenché à une date unique. Donc, pas besoin d'une Collection.

1. Nécessité d'un attribut dépendant : Oui.

Il faudra afficher cette date de contrôle à notre convenance dans les rapports de contrôle. On conçoit donc un attribut dépendant 'dateControleStringFormatee' chargé de l'affichage formaté de cette date.

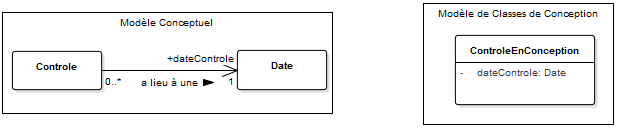
1. Alimentation de l'attribut : par l'application.

En pratique, il suffira de renseigner cet attribut 'dateControle' avec la date système lors de l'instanciation du contrôle, puis lors de chaque exécution du contrôle. L'application devra donc pouvoir injecter la date système dans 'dateControle' à chaque exécution du contrôle (appel de sa méthode controler(…)).

1. Méthodes liées à l'attribut : aucune
2. Détenteur de la Responsabilité de l'attribut : Tout contrôle.

Tout Contrôle doit connaître sa date d'exécution et c'est l'application qui sera chargée d'alimenter cet attribut lors de l'instanciation puis à chaque exécution du contrôle avec la date système. Tous les contrôles auront leur attribut 'dateControle' alimentée de la même manière. Ce sera donc un attribut de haut niveau dans la hiérarchie des classes alimenté par des méthodes de haut niveau dans l'arborescence. Aucun contrôle n'a la responsabilité de remplir sa 'dateControle' de façon particulière.

Le modèle conceptuel est donc transformé en modèle de classes de conception comme suit :



Devient

#### Attribut simple dépendant dateControleStringFormatee (String)

1. Contenu : date mais affichée à notre convenance.
2. Type : Simple. Objet Java prédéterminé **String**.

Le type **java.util.Date** en Java est donc un Objet qui "sait faire" des choses grâce à ses méthodes (se décaler d'un mois, compter un nombre de jours entre 2 Date, …). En revanche, cet Objet **java.util.Date** ne sait afficher la date qu'il encapsule que selon le format que les concepteurs de Java ont prédéterminé.

Nous savons déjà que nous aurons besoin d'émettre des messages de contrôle incorporés dans des rapports de contrôle. Il faudra donc obligatoirement pouvoir "écrire" l'attribut 'dateControle' sous forme de chaîne de caractères (**String**) pour l'affichage dans les rapports de contrôle comme nous le voulons.

Java permet de "formater" des dates comme par exemple pour le 1er Mai 2016 à 17h35 :

* Dimanche 1er Mai 2016 à 17 heures 35
* 01/05/2016-17:35
* 2016-05-01-17-35

Java utilise pour ce faire un Objet (technique) déjà fabriqué par les concepteurs de Java que l'on appelle un **DateFormat**.

Il suffit de savoir qu'avec un certain **DateFormat**, on obtiendra " Dimanche 1er Mai 2016 à 17 heures 35", avec un autre "2016-05-01-17-35", …

Il est intéressant de prévoir dès maintenant un attribut 'dateControleStringFormatee' dans la classe Contrôle. Cet attribut 'dateControleStringFormatee' est forcément *dépendant* de 'dateControle'. C'est un attribut **calculé** à partir de l'attribut 'dateControle'.

1. Multiplicité : attribut unique. Un contrôle est déclenché à une date unique.
2. Nécessité d'un attribut dépendant : non.
3. Alimentation de l'attribut : par l'application.

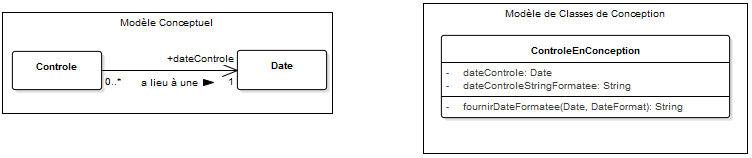
En pratique, il suffira de renseigner cet attribut 'dateControleStringFormatee' à chaque modification de l'attribut 'dateControle'.

1. Méthodes liées à l'attribut : fournirDateFormatee(…)

Puisque la classe Contrôle devra fournir la date sous forme de String formatée à notre convenance, il faut la doter d'une méthode qui formate une Date.

* La méthode fournirDateFormatee(Date pDate, DateFormat pDateFormat) prend en paramètre une Date pDate et un certain DateFormat pDateFormat et retourne une chaîne de caractères (String) qui correspond à la date formatée à notre convenance (par exemple 2016-05-01\_17-35 avec un certain DateFormat).
* L'attribut dateControleStringFormatee est de type String.

1. Détenteur de la Responsabilité de l'attribut : Tout contrôle.



#### Attribut simple userName (String)

1. Contenu : le nom de l'organisation de l'acteur qui déclenche le contrôle
2. Type : Simple. Objet Java prédéterminé **String**.

Un Contrôle sait toujours quel User l'a déclenché. Un concept User est le plus souvent un Objet qui connaît son nom, son prénom, son profil Cerbère, …

Mais le concept Contrôle n'a besoin que du nom du User, ou plus exactement du nom de l'organisation du User qui a déclenché le contrôle (DIRA, DIRE, DARWIN, …).

Il est donc inutile de mettre dans la classe Contrôle un attribut de *type* User puisque l'on ne se servira jamais de toutes ses informations lors de l'exécution des contrôles et de la génération des rapports de contrôle.

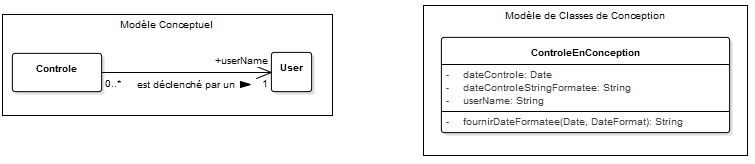
* On peut se contenter d'incorporer dans la classe Contrôle un attribut simple userName et utiliser le type **String** fourni par Java puisque l'on utilisera une simple chaîne de caractères.

1. Multiplicité : attribut unique. Un contrôle est déclenché par un unique User.
2. Nécessité d'un attribut dépendant : Non
3. Alimentation de l'attribut : par l'application

En pratique, il suffira de renseigner cet attribut 'userName' avec le nom de l'organisation du User lors de l'instanciation du contrôle. L'application doit donc pouvoir injecter ce userName dans le *constructeur* du contrôle (le constructeur est la méthode qui permet d'instancier in Objet Contrôle) en le prélevant dans le User loggé en session.

1. Méthodes liées à l'attribut : aucune
2. Détenteur de la responsabilité de l'attribut : tout contrôle

Tout Contrôle doit connaître le nom du User qui le déclenche et c'est l'application qui sera chargée d'alimenter cet attribut lors de l'instanciation (avec le nom de l'organisation du User loggé en session). Tous les contrôles auront leur attribut 'userName' alimentée de la même manière. Ce sera donc un attribut de haut niveau dans la hiérarchie des classes alimenté par des méthodes de haut niveau dans l'arborescence. Aucun contrôle n'a la responsabilité de remplir son 'userName' de façon particulière.

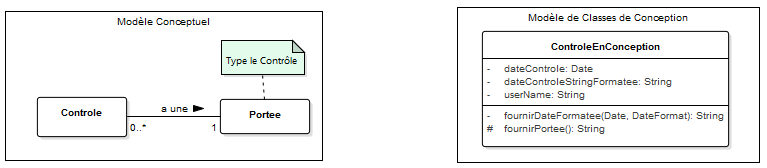


#### Concept Portée (pas un attribut)

Comme précisé lors de l'étude du concept portée dans l'analyse, il n'y a pas lieu à priori de complexifier la classe Contrôle avec un attribut portée. On peut le faire, mais ce n'est pas indispensable puisque l'on peut **typer les contrôles** en créant une **hiérarchie de classes** pour tenir compte de la portée.

Une simple méthode fournirPortee() avec un type de retour String dans la classe Contrôle est suffisante si l'on a besoin d'afficher la portée d'un contrôle. Ce sera peut-être le cas dans les rapports d'administration des contrôles par exemple.

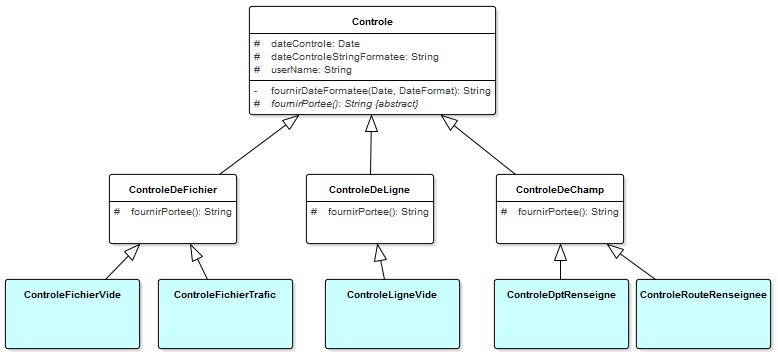
* S'il s'agit d'un contrôle portant sur un fichier, cette méthode retournera simplement "Fichier".
* S'il s'agit d'un Contrôle portant sur une ligne, cette méthode retournera simplement "Ligne".
* S'il s'agit d'un contrôle portant sur un champ, cette méthode retournera simplement "Champ".
* Etc…



En revanche, la classe de haut-niveau Contrôle ne connaît pas la portée du contrôle qui est en train de s'exécuter. Il peut s'agir d'un contrôle de fichier, d'un contrôle de ligne, d'un contrôle de champ, d'un contrôle inter-champs, …

La méthode *fournirPortee()* ne peut donc qu'être **abstraite** dans la classe de haut niveau Contrôle (ControleEnConception sur le diagramme ci-dessus).

Il est donc nécessaire de **sous-classer** la classe de haut niveau Contrôle dès la conception pour tenir compte des diverses portée :



Les sous-classes '**typent**'

Les contrôles en fonction

de la portée et **implémentent**

la méthode fournirPortee()

La classe de haut niveau Contrôle définit **abstraitement** une méthode *fournirPortee()* pour garantir qu'elle sera implémentée plus bas dans la hiérarchie

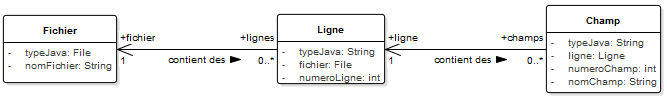
* Les classes 'ControleDptRenseigne' et 'ControleRouteRenseignee' **sont** des 'ControleDeChamp' puisqu'ils **héritent** de la classe 'ControleDeChamp'.
* La classe 'ControleDeChamp' se charge d'**implémenter** la méthode fournirPortee() et retourne "Champ". De même, la classe 'ControleDeLigne' se charge d'implémenter la méthode fournirPortee() et retourne "Ligne".
* Les classes 'ControleDptRenseigne' et 'ControleRouteRenseignee' **possèdent** la méthode fournirPortee() par **héritage**. Comme ces deux classes héritent de la classe 'ControleDeChamp', leur méthode fournirPortee() retournera "Champ" comme le stipule l'implémentation dans leur classe mère.
* Les sous-classes de la classe 'Contrôle', à savoir les classes 'ControleDeFichier', 'ControleDeLigne" et 'ControleDeChamp' *suffisent à préciser la portée* puisqu' elles **typent leurs contrôles descendants** en fonction de leur portée.

#### Concept Objet Contrôlé (pas un attribut)

L'étude du concept Objet Contrôlé dans l'analyse précédente a montré qu'il était impossible de définir un type unique pour les objets contrôlés de tous les types de contrôles du fait de leurs différentes portées. Certains contrôles s'appliquent à des fichiers (File en Java), d'autres à des lignes de fichier (String en Java), d'autres à des champs de lignes d'un fichier, …

Il ne pourrait donc en aucun cas y avoir un type unique pour un attribut 'objetControle' que l'on mettrait dans la classe Contrôle.

On ne peut pas concevoir un attribut unique 'objetControle' dans la classe de haut niveau Contrôle puisque nous serions incapables de le typer.

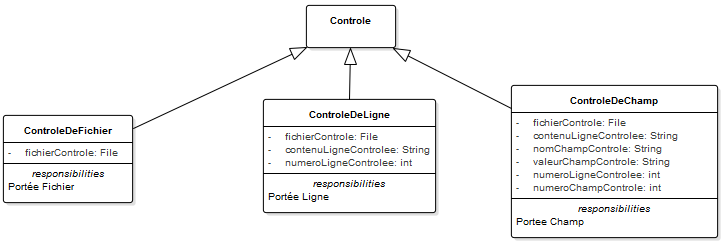


En effet :

* Un fichier contient des lignes, lesquelles contiennent des champs.
* Une ligne doit savoir à quel fichier elle appartient (ex : la 2ème ligne du fichier HIT\_DIRA\_2014).
* Un champ doit savoir à quelle ligne il appartient, et sa ligne doit savoir à quel fichier elle appartient (ex : le champ Département de la 2ème ligne du fichier HIT\_DIRA\_2014).

On constate donc que les informations nécessaires **diffèrent** pour identifier l'objet contrôlé en fonction de la portée du contrôle (fichier, ligne, champ, …).

On en déduit qu'il faudra **sous-classer** la classe Contrôle pour tenir compte de l'objet contrôlé par le contrôle comme par exemple :



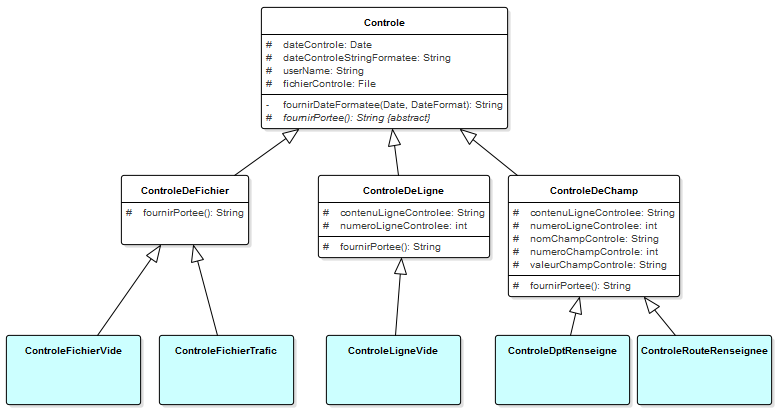
Pour un contrôle de portée "Champ" comme par exemple un contrôle de format numérique du Département dans la deuxième ligne d'un fichier HIT\_DIRA\_2014, nous aurons alors :

* fichierControle : HIT\_DIRA\_2014
* contenuLigneControlee : "13056987445 565623 LA VANNE …."
* nomChampControle : Département
* valeurChampControle : "130"
* numeroLigneControlee : 2
* numeroChampControle : 1

Il sera donc possible pour la MOA de trier le rapport de contrôle sur le nom du champ "nomChampControle" et de lister par exemple toutes les lignes dont le Département a donné lieu à un contrôle défavorable (tri sur le nom du champ contrôlé et le statut du contrôle).

Mais cette notion de nom du champ "nomChampControle" n'a de sens que pour la sous-classe ControleDeChamp. Aucun intérêt pour la classe Contrôle puisqu'il pourrait aussi bien s'agir à priori d'un fichier que d'une ligne ou d'un champ…

On peut modifier en conséquence l'ébauche de modèle des classes conceptuelles comme suit :



Dans ce modèle :

* Tous les contrôles héritant de la classe Contrôle (les descendants de Contrôle) connaissent forcément le fichier qu'elles contrôlent puisque l'attribut 'fichierControle' figure dans la classe de haut niveau Contrôle**. Tous les descendants le possèdent donc par héritage**. 'ControleDeFichier', 'ControleDeLigne' et 'ControleDeChamp' connaissent donc leur 'fichierControle' et c'est évidemment vrai pour tous leurs descendants en bleu.
* Tous les 'ControleDeLigne' connaissent en plus du fichier contrôlé le contenu de la ligne contrôlée 'contenuLigneControlee' ainsi que le numéro de la ligne contrôlée 'numeroLigneControlee'.
* Tous les 'ControleDeChamp' connaissent en plus du fichier contrôlé le contenu de la ligne contrôlée 'contenuLigneControlee' ainsi que le numéro de la ligne contrôlée 'numeroLigneControlee'.

Ils possèdent en plus des informations relatives au champ contrôlé : le nom du champ 'nomChampControle', le numéro d'ordre du champ dans une ligne du fichier 'numeroChampControle' et la valeur du champ 'valeurChampControle'.

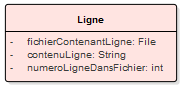
On peut améliorer ce modèle qui n'est pas très "Objet" et fait apparaître des redondances. En effet :

* Pourquoi ne dit-on pas qu'un champ contrôlé appartient à une "ligne" plutôt que de le doter de son numéro de ligne et du contenu de la ligne dans 'ControleDeChamp' ?
* Pourquoi répète-t-on l'attribut 'contenuLigneControlee' dans les classes 'ControleDeLigne' et 'ControleDeChamp' ?

Un des intérêts de la conception Objet est la conception de "**pures fabrications**". Au lieu de traiter un contenu et un numéro de ligne pour préciser la ligne contrôlée, on crée ses *propres concepts* qui **encapsulent** toutes ses notions.

Une pure fabrication n'est *pas un objet du domaine* (Contrôle, Bateau, CompteEnBanque, ...), *ni un objet simple* prédéfini par Java (**String**, **Date**, **DateFormat,** …), mais un objet *créé par le concepteur* pour simplifier le problème.

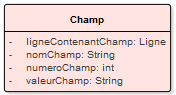
Par exemple, on peut concevoir une pure fabrication 'Ligne' qui encapsule :



La pure fabrication 'Ligne' connait donc :

* Le fichier auquel la ligne qu'elle encapsule appartient (attribut 'fichierContenantLigne').
* Le contenu (chaîne de caractères) de la ligne qu'elle encapsule (attribut 'contenuLigne').
* Le numéro de la ligne qu'elle encapsule dans le fichier auquel elle appartient (attribut 'numeroLigneDansFichier').

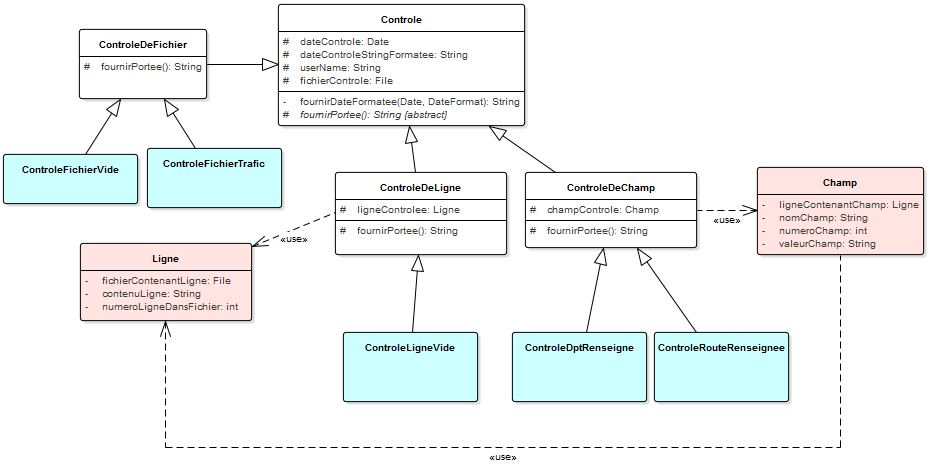
De même, on peut concevoir une pure fabrication 'Champ' comme suit :



La pure fabrication 'Champ' connaît donc :

* La ligne contenant le champ qu'elle encapsule grâce à l'attribut 'ligneContenantChamp' qui est de **type Ligne**. La ligne connaît elle-même le fichier dont elle est extraite. Par conséquent, le champ connaît aussi le fichier dont il est extrait.
* Le nom du champ qu'elle encapsule (attribut 'nomChamp').
* Le numéro d'ordre du champ dans une ligne du fichier (par exemple, le champ route est le 3ème champ dans une ligne d'un fichier HIT) par son attribut 'numeroChamp'.
* La valeur prise par le champ qu'elle encapsule (par exemple RN20) par son attribut 'valeurChamp'.

En utilisant ces pures fabrications, le modèle précédent devient :



#### Attribut simple critere (String)

1. Contenu : homogène à la description du critère vérifié par le contrôle
2. Type : Simple. Objet Java prédéterminé **String**.
3. Multiplicité : attribut unique. Un contrôle vérifie un unique critère.
4. Nécessité d'un attribut dépendant : non
5. Alimentation de l'attribut : lors de l'instanciation de chaque contrôle

Une fois que l'on a implémenté un Contrôle, il n'y a plus aucune ambiguïté sur le critère que le Contrôle vérifie. La valeur du critère peut être fixée dès l'instanciation du contrôle et ne peut être modifiée durant toute la durée de vie du contrôle (attribut immuable). Le critère ne doit pas être paramétré ou rapatrié depuis un fichier properties. Il est fixé par le développeur.

Par exemple, un Contrôle chargé de vérifier qu'un champ Département a un format numérique dans une ligne de fichier HIT sait évidemment que le critère qu'il vérifie est " le champ Département est homogène à un numérique entier". Le développeur a fixé le critère dès l'implémentation et ce point ne peut pas être modifié.

1. Méthodes liées à l'attribut : méthode fournirCritere()

Une simple méthode fournirCritere() avec un type de retour String dans chaque classe concrète de contrôle est à priori suffisante si l'on a juste besoin d'afficher le critère d'un contrôle.

* S'il s'agit d'un contrôle portant sur le contenu (non vide) d'un fichier, cette méthode retournera simplement " le fichier soumis n'est pas vide".
* S'il s'agit d'un contrôle portant sur le format numérique d'un champ Département, cette méthode retournera simplement "le champ Département est homogène à un numérique entier".

Un attribut 'critere' n'est donc pas strictement nécessaire. Il existe néanmoins deux cas où il est judicieux de créer un attribut 'critere' dans la classe Contrôle :

1. Lorsque l'on sait qu'il existe un fort risque d'utiliser 'critere' *à plusieurs endroits dans la classe*.
2. Lorsque l'on sait que l'on aura besoin d'accéder à l'attribut *depuis une classe externe*.

En effet, les attributs sont visibles dans toute la classe et peuvent donc être appelés par n'importe quelle méthode de la classe. Comme il est probable que nous aurons besoin d'afficher le critère dans le rapport de contrôle au format csv, dans les logs, … et donc depuis différentes méthodes de la classe, il est prudent de prévoir un attribut 'critere' qui sera alimenté par la méthode fournirCritere() une seule fois dès l'instanciation du contrôle.

En outre, des classes externes comme AdministrateurDesControles peuvent avoir besoin d'accéder à l'attribut 'critere' via un **accesseur public** getCritere() qui retourne cet attribut.

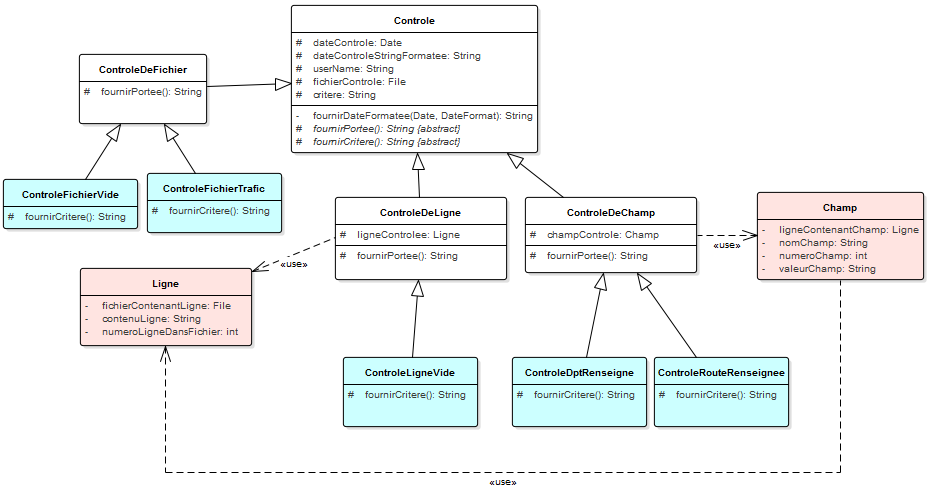
* L'attribut critere est une simple chaine de caractères (String).
* La méthode fournirCritere() retourne une String.

1. Détenteur de la responsabilité de l'attribut : chaque contrôle concret.

Chaque contrôle concret (comme le contrôle du bon format numérique du Département) connaît le critère qu'il vérifie. La classe de haut niveau Contrôle n'a aucune connaissance du critère du contrôle concret.

* On définit donc l'attribut 'critere' dans la classe de haut niveau Contrôle pour garantir que tous ses descendants possèderont un 'critere'.
* On définit une méthode *abstraite* *fournirCritere()* dans la classe de haut niveau Contrôle pour garantir que tous ses descendants implémenteront cette méthode
* On implémente la méthode fournirCritere() dans chaque descendant concret.
* On appellera fournirCritere() une seule fois lors de l'instanciation de chaque contrôle (donc dans son constructeur).

A ce stade, le modèle des classes de conception devient :



#### Attribut simple typeErreur (Integer - attribut à Label)

1. Contenu : homogène au numéro d'une description d'erreur (1 pour "NON RENSEIGNE", …)
2. Type : Simple. Objet Java prédéterminé **Integer**.

Un Contrôle sait toujours quel type d'erreur il vérifie (par exemple "nullité d'un fichier", "fichier vide", "ligne vide", "champ non renseigné", "champ de mauvais format", …). Un type d'erreur ne sait rien des contrôles qui le vérifient.

1. Multiplicité : attribut unique. Un contrôle vérifie un unique type d'erreur.
2. Nécessité d'un attribut dépendant : Oui. labelTypeErreur.

Il est toujours plus facile de trier dans un tableau ce type de champs lorsqu'ils sont homogènes à des entiers numériques.

En revanche, un label sous forme de chaîne de caractères (String) est toujours plus lisible pour l'acteur. En outre, la MOA pourra le cas échéant modifier à postériori le label à sa convenance si celui-ci est stocké dans un fichier externalisé comme une nomenclature en csv.

Le label 'labelTypeErreur' est forcément *dépendant* de l'attribut 'typeErreur'.

C'est un attribut qui doit être **rapatrié** **à partir d'un fichier externalisé** comme un properties ou une nomenclature csv en fonction de l'attribut 'typeErreur'.

Par exemple :

| **typeErreur** | **Label du typeErreur** |
| --- | --- |
| 13 | CHAMP NON RENSEIGNE |
| 14 | MAUVAIS FORMAT DE CHAMP |
| 15 | CHAMP HORS NOMENCLATURE |
| 11 | LIGNE VIDE DANS FICHIER |
| 12 | LIGNE SANS LOCALISATIONS |

1. Alimentation de l'attribut : lors de l'instanciation du contrôle.

Le développeur connaît le type d'erreur contrôlé dès l'implémentation du contrôle. La valeur du type d'erreur peut être fixée dès l'instanciation du contrôle et ne peut être modifiée durant toute la durée de vie du contrôle (attribut immuable).

1. Méthodes liées à l'attribut : fournirTypeErreur(), fournirLabelTypeErreur()

Chaque contrôle concret (comme le contrôle du bon format numérique du Département) connaît le type d'erreur qu'il vérifie. La classe de haut niveau Contrôle n'a aucune connaissance du type d'erreur vérifié par le contrôle concret.

* On définit donc l'attribut 'typeErreur' dans la classe de haut niveau Contrôle pour garantir que tous ses descendants possèderont un 'typeErreur'.
* On définit une méthode *abstraite* *fournirTypeErreur()* dans la classe de haut niveau Contrôle pour garantir que tous ses descendants implémenteront cette méthode
* On implémente la méthode fournirTypeErreur() dans chaque descendant concret.
* On appellera fournirTypeErreur() une seule fois lors de l'instanciation de chaque contrôle (donc dans son constructeur).

Puisque nous savons que labelTypeErreur doit être rapatrié depuis un fichier externalisé en fonction de typeErreur, il est intéressant de prévoir une méthode chargée de cette tâche de rapatriement.

Nous pouvons donc concevoir une méthode fournirLabelTypeErreur() qui retournera le label stocké dans la nomenclature ("champ non renseigné", "champ de mauvais format", ..) sous forme de String.

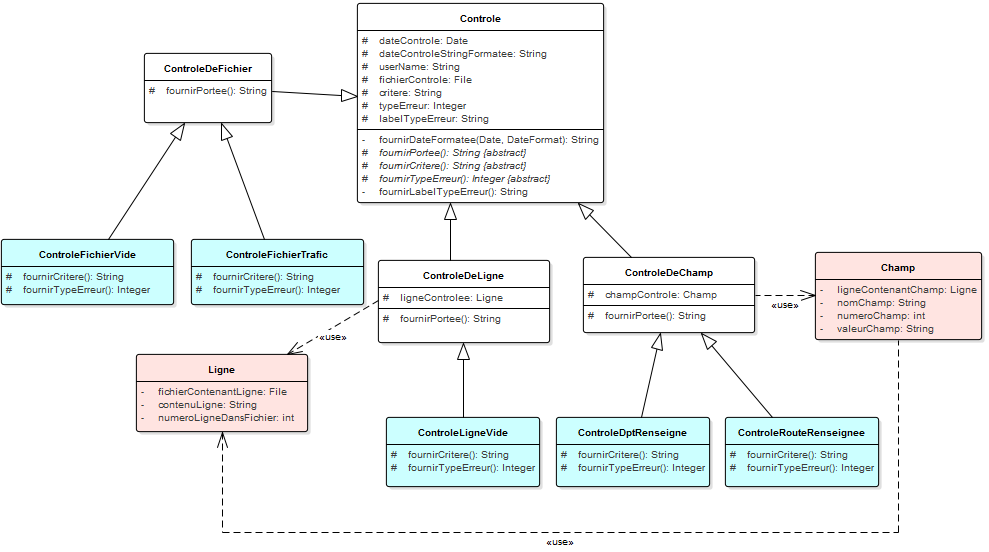
Cette méthode fournirLabelTypeErreur() sera appelée juste après fournirTypeErreur() dès la construction de l'instance du contrôle.

1. Détenteur de la responsabilité de l'attribut :

Seul chaque contrôle concret connait le type d'erreur qu'il détecte. Par conséquent, la méthode fournirTypeErreur() qui alimente l'attribut 'typeErreur' doit se situer au niveau des contrôles concrets.

En revanche, la méthode fournirLabelTypeErreur() qui va rechercher le label dans une nomenclature ne dépend pas du contrôle dans lequel elle est implémentée. Elle opère toujours de la même manière quel que soit le contrôle. Elle peut donc être factorisée au plus haut niveau.

A ce stade, le modèle des classes de conception devient :



#### Attribut simple aEffectuer (boolean)

1. Contenu : homogène à oui/non pour savoir si le contrôle doit être effectué ou pas.
2. Type : Simple. Primitif Java **boolean**.

Un contrôle sait toujours s'il est actif/inactif pour être exécuté ou pas. Le concept Activité ne sait rien du Contrôle. En pratique, j'ai appelé l'attribut actif 'aEffectuer' dans ma classe de conception.

Un type simple Java de type primitif (non Objet) boolean est donc suffisant pour cet attribut aEffectuer.

1. Multiplicité : attribut unique. Un contrôle est actif (aEffectuer) ou pas.
2. Nécessité d'un attribut dépendant : non
3. Alimentation de l'attribut : lors de l'instanciation du contrôle.

L'application va aller chercher dans un fichier externalisé (properties) dès l'instanciation du contrôle s'il doit être exécuté ou pas. Cette valeur ne pourra plus être changée pendant l'exécution du contrôle (attribut immuable).

1. Méthodes liées à l'attribut : fournirAEffectuer()

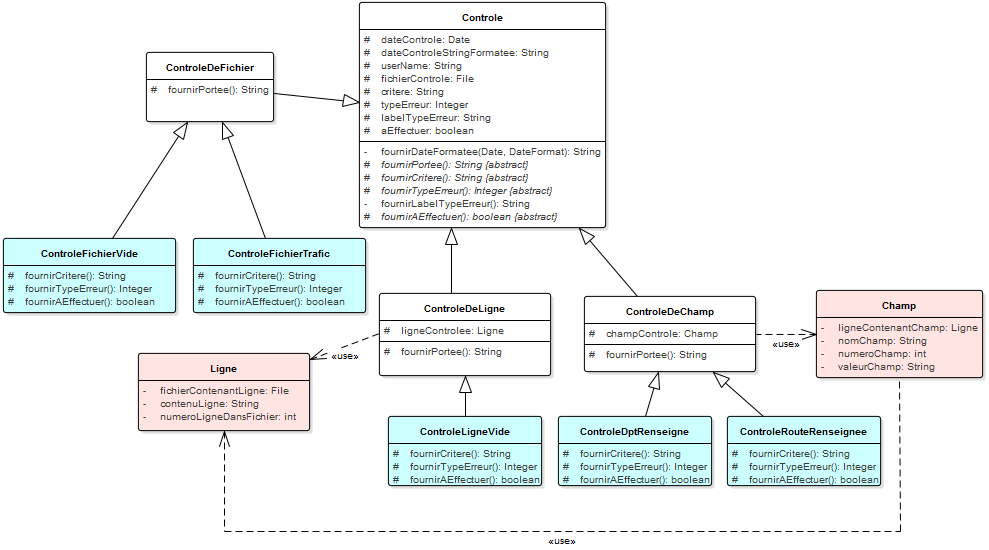
Les contrôles doivent être **individuellement paramétrables** (EX\_FONCT\_07\_CONTROLES\_PARAMETRABLES). La MOA aura donc accès à un properties (fichier externalisé) dans lequel elle pourra activer/desactiver chaque contrôle.

Il est donc intéressant de prévoir dans chaque classe de contrôle concrète une méthode fournirAEffectuer() chargée de rapatrier la valeur de ce boolean depuis le fichier properties.

1. Détenteur de la responsabilité de l'attribut : chaque contrôle concret.

Chaque contrôle concret sait s'il doit être effectué ou pas. La méthode fournirAEffectuer() doit donc être placée au niveau des contrôles concrets.

A ce stade, le modèle des classes de conception devient :



#### Attribut simple statut (boolean - attribut à Label)

1. Contenu : homogène à un OK/KO en fonction du résultat du contrôle.
2. Type : Simple. Primitif Java **boolean**.

Un contrôle sait toujours si le contrôle qu'il vient d'effectuer a matché favorablement ou défavorablement. Le concept Statut ignore tout du Contrôle qui l'a généré.

Un type simple Java de type primitif boolean est donc suffisant pour l'attribut statut.

1. Multiplicité : attribut unique. Un contrôle est OK ou KO après son exécution.
2. Nécessité d'un attribut dépendant : labelStatut

Il est toujours plus facile de trier dans un tableau ce type de champs lorsqu'ils sont homogènes à des entiers numériques (1 ou 0) ou booléens (true, false).

En revanche, un label (String) est toujours plus lisible pour l'acteur.

L'usage est de dire qu'un contrôle est OK s'il a matché favorablement et KO s'il a matché défavorablement.

| **statut** | **labelStatut** |
| --- | --- |
| true | OK |
| false | KO |

1. Alimentation de l'attribut : par la méthode controler(…) des contrôles concrets.

Il appartient à la classe concrète de contrôle de décider (via sa méthode controler(…)) si le statut est OK ou KO.

Le label du statut 'labelStatut' doit être rafraîchi dès que le statut est modifié, donc à chaque exécution de la méthode controler(…).

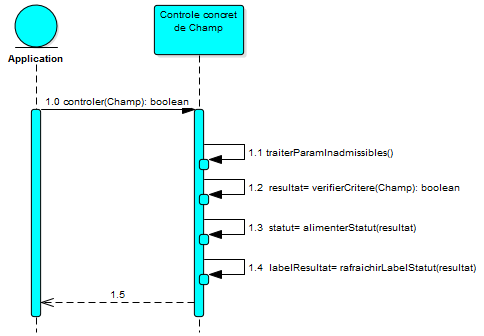
1. Méthodes liées à l'attribut : controler(), fournirLabelStatut(boolean pStatut)

C'est la méthode controler(…) du contrôle concret qui alimentera l'attribut 'statut' en fonction du résultat du contrôle.

Il est intéressant de prévoir une méthode fournirLabelStatut(boolean pStatut) qui retournera une String ("OK" si statut vaut true, "KO" si statut vaut false).

1. Détenteur de la responsabilité de l'attribut : chaque contrôle concret.

Chaque contrôle concret effectue le contrôle via sa méthode controler(…). Lors de son exécution, cette méthode doit alimenter l'attribut 'statut' avec le résultat du contrôle (true/false) et rafraîchir l'attribut dépendant 'labelStatut'.

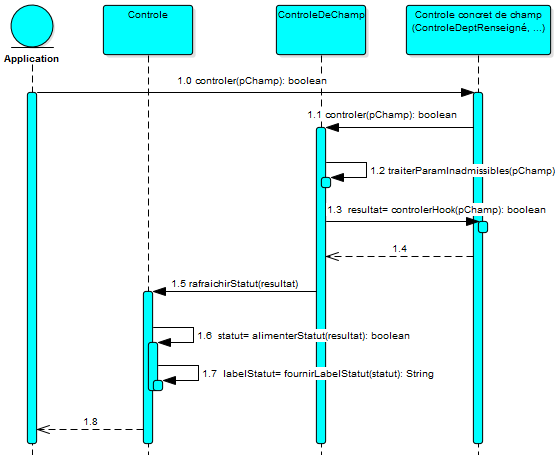


Dans cette conception, la méthode controler(…) doit donc être située au niveau de chaque classe concrète de contrôle. L'inconvénient de cette conception est que toutes les méthodes controler(…) de tous les contrôles concrets devront exécuter des tâches répétitives comme évacuer le cas de paramètres inadmissibles ...

Je préfère donc utiliser dès la conception le **design pattern Hook** :

Le principe est de :

* Placer la méthode controler(…) au niveau de la classe mère des contrôles concrets (par exemple la classe ControleDeChamp par rapport à la classe ControleDeptRenseigne). Rappelons que le contrôle concret dispose alors de la méthode controler(…) par héritage.
* Factoriser les traitements répétitifs (traitement des paramètres inadmissibles, …) dans la méthode controler(…) au niveau de la classe mère des contrôles concrets.
* De définir une méthode *abstraite* *controlerHook(…)* dans la classe mère afin de garantir que tous ses descendants implémenteront cette méthode
* De n'implémenter que la vérification du critère propre au contrôle concret dans la méthode controlerHook(…) de chaque contrôle concret.

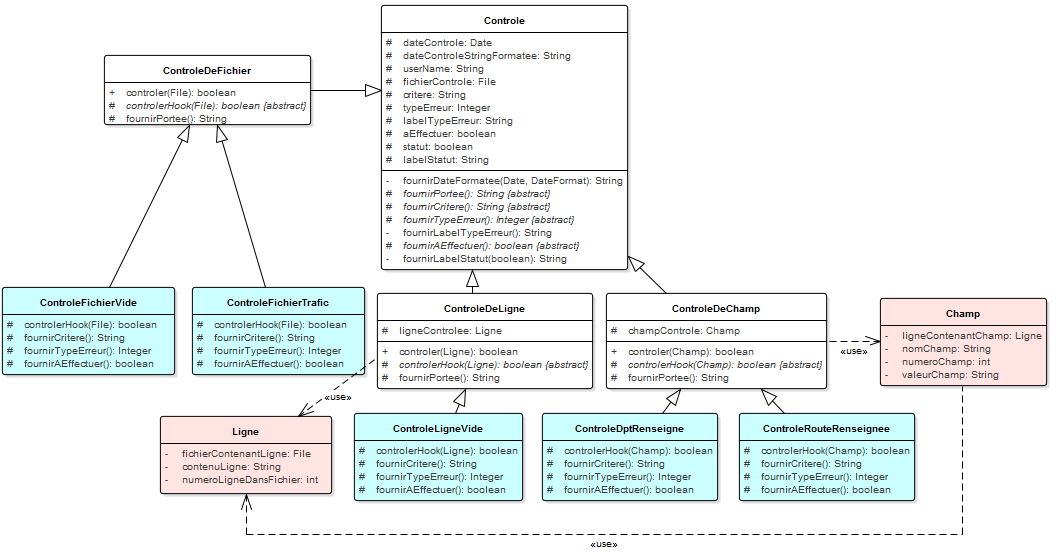


Lorsque l'on demande à un contrôle concret ControleDptRenseigne de contrôler un champ Département en écrivant en Java controleDptRenseigne.controler('le champ département voulu') :

1. La méthode controler(Champ pChamp) du contrôle concret ControleDeptRenseigne est invoquée (appelée). **Flèche 1.0** dans le diagramme de séquence ci-dessus.
2. Comme le contrôle concret ControleDeptRenseigne ne possède cette méthode que par héritage, la méthode controler(Champ pChamp) de la classe mère ControleDeChamp est automatiquement invoquée par Java grâce au mécanisme appelé **polymorphisme**. **Flèche 1.1**.
3. Les traitements répétitifs comme le traitement des paramètres inadmissibles sont effectués dans la partie traiterParamInadmissibles(pChamp) de la méthode controler(Champ pChamp) de la classe mère ControleDeChamp. **Flèche 1.2**.
4. Puis, cette méthode comporte un appel de la méthode controlerHook(Champ pChamp) qui est implémentée dans le contrôle concret ControleDeptRenseigne. Java invoque donc automatiquement la méthode controlerHook(Champ pChamp) du bon contrôle concret grâce au polymorphisme. **Flèche 1.3**.
5. La méthode controlerHook(Champ pChamp) du contrôle concret retourne le résultat de la vérification du critère. **Flèche 1.4**.
6. La méthode controler(Champ pChamp) de la classe mère ControleDeChamp comporte alors un appel de la méthode rafraichirStatut(boolean pResultat) de la classe de haut niveau Contrôle. Java invoque donc rafraichirStatut(resultat). **Flèche 1.5**.
7. rafraichirStatut(resultat) passe alors le résultat du contrôle à l'attribut 'statut'. **Flèche 1.6**.
8. rafraichirStatut(resultat) calcule ensuite le label du statut et passe le résultat du calcul à l'attribut 'labelStatut' grâce à la méthode fournirLabelStatut(boolean pStatut). **Flèche 1.7**.
9. Puis retour (boolean) de la méthode controler(Champ pChamp) du contrôle concret ControleDeptRenseigne à l'appelant. **Flèche 1.8**.

Gains : Tout le code répétitif de traitement des paramètres inadmissibles et d'alimentation des attributs 'statut' et 'labelStatut' est maintenant factorisé dans les classes de haut niveau. Les classes concrètes de contrôle n'ont plus que la responsabilité de vérifier leur critère.

A ce stade, le modèle de classes de conception devient :



#### Attribut simple gravite (String - attribut à Label)

1. Contenu : homogène au numéro d'une description de gravité du contrôle ("1" pour anomalie bloquante, "2" pour anomalie grave, …).
2. Type : Simple. Objet Java prédéterminé **String**.

Un contrôle connaît toujours sa gravité et le concept Gravité ne sait rien des concepts Contrôle qu'elle concerne.

La gravité et donc un attribut simple (pas un objet).

J'ai choisi de le typer **String** pour éviter d'avoir à parser des String en Integer lors de la lecture de cette valeur dans un properties (qui ne contient bien évidemment que des chaines de caractères String). Je souhaite également concaténer cette 'gravite' avec d'autres chaînes de caractères. Mais j'aurai pu "fonctionnellement" choisir le type Integer.

1. Multiplicité : attribut unique. Un contrôle a une et une seule gravité.
2. Nécessité d'un attribut dépendant : labelGravite

Il est toujours plus facile de trier dans un tableau ce type de champs lorsqu'ils sont homogènes à des entiers numériques.

En revanche, un label (String) est toujours plus lisible pour l'acteur. En outre, la MOA pourra le cas échéant modifier à postériori modifier le label à sa convenance dans un fichier externalisé comme un properties.

Le label 'labelGravite' est forcément *dépendant* de l'attribut 'gravite'.

C'est un attribut qui doit être **rapatrié** **à partir d'un fichier externalisé** comme un properties ou une nomenclature csv en fonction de l'attribut 'gravite'.

Par exemple :

| **gravite** | **labelGravite** |
| --- | --- |
| 0 | Indéfini |
| 1 | Anomalie Bloquante |
| 2 | Anomalie grave |
| 3 | Avertissement |
| 4 | Anomalie rédhibitoire |

1. Alimentation de l'attribut : lors de l'instanciation du contrôle.

L'application va aller chercher dans un fichier externalisé (properties) dès l'instanciation du contrôle sa gravité. Cette valeur ne pourra plus être changée pendant l'exécution du contrôle (attribut immuable).

J'ai choisi de rendre cet attribut **paramétrable** par la MOA via un fichier externalisé (properties). Ce choix peut paraître discutable pour certains contrôles bloquants. Par exemple, un contrôle de contenu (non vide) de fichier est forcément bloquant puisque l'on ne pourra rien contrôler dans un fichier vide.

En revanche, la MOA peut avoir envie de modifier la gravité d'un contrôle de **T**rafic **M**oyen **J**ournalier **M**ensuel (TMJM) renseigné de 2 (anomalie grave) à 3 (avertissement).

J'ai retenu l'option de rendre la gravité entièrement paramétrable, même si le bon sens s'impose pour les contrôles bloquants (qui doivent arrêter le programme) ou rédhibitoires (qui doivent conduire à l'annulation d'un champ ou l'écrasement d'une ligne).

1. Méthodes liées à l'attribut : fournirGravite(), fournirLabelGravite()

Il est intéressant de prévoir une méthode fournirGravite() chargée de rapatrier le niveau d'anomalie paramétrable depuis un properties, et de fournir labelGravite contenu également dans le properties.

L'attribut simple gravite a été renommé dans ma classe de conception niveauAnomalie et est de type String pour permettre des concaténations de chaînes et éviter d'avoir à parser en Integer la String contenue dans le fichier properties.

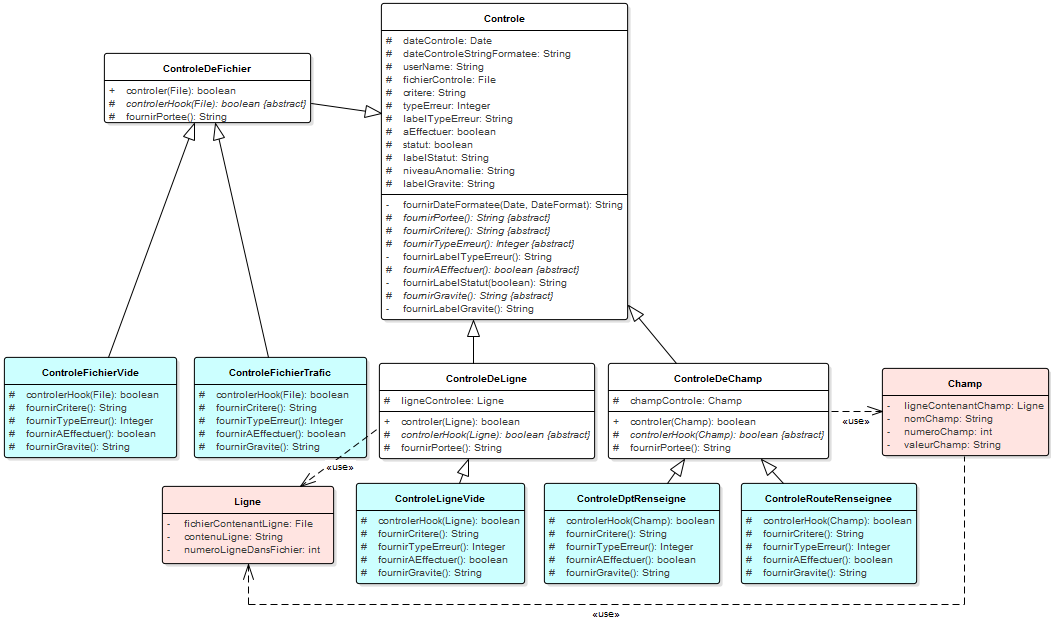
L'attribut simple labelGravite (attribut *dépendant* de gravite) est de type **String**.

1. Détenteur de la responsabilité de l'attribut : chaque contrôle concret.

Chaque contrôle concret connaît sa gravité. En fait il sait qu'il doit aller chercher sa gravité dans un properties. Mais le concept gravite est dépendant de chaque contrôle concret.

La méthode fournirGravite() doit donc être placée au niveau de chaque contrôle concret.

A ce stade, le modèle de classes de conception devient :



#### Attribut simple suiteDonnee (Integer - attribut à Label)

1. Contenu : homogène au numéro d'une description de la suite à donner à l'exécution d'un contrôle (1 pour "FICHIER ACCEPTE", 2 pour "FICHIER REFUSE", …).
2. Type : Simple. Objet Java prédéterminé **Integer**.

Tout contrôle sait la suite à donner à l'objet contrôlé (fichier refusé, ligne retirée, champ mis à null, champ conservé, …). Le concept Suite A Donner ne sait rien du Contrôle qui l'a généré.

La suite à donner est donc un attribut simple (pas un objet). J'ai choisi **Integer**.

1. Multiplicité : attribut unique.

Un contrôle détermine une unique suite à donner après son exécution.

En revanche, la suite à donner diffère en fonction du fait que le contrôle est favorable ou défavorable.

Par exemple, on peut avoir suite à donner = "champ accepté" si le contrôle de champ est favorable et "champ modifié" si le contrôle est défavorable.

1. Nécessité d'un attribut dépendant : labelSuiteDonnee

Il est toujours plus facile de trier dans un tableau ce type de champs lorsqu'ils sont homogènes à des entiers numériques. En revanche, un label (String) est toujours plus lisible pour l'acteur. En outre, la MOA pourra le cas échéant modifier à postériori modifier le label à sa convenance.

Le label 'labelSuiteDonnee' est forcément *dépendant* de l'attribut 'suiteDonnee'.

C'est un attribut qui doit être **rapatrié** **à partir d'un fichier externalisé** comme un properties ou une nomenclature csv en fonction de l'attribut 'suiteDonnee'.

Par exemple :

|  |  |
| --- | --- |
| **suiteDonnee** | **labelSuiteDonnee** |
| 1 | FICHIER ACCEPTE |
| 2 | FICHIER REFUSE |
| 3 | LIGNE CONSERVEE |
| 4 | LIGNE RETIREE |
| 5 | CHAMP ACCEPTE |
| 6 | CHAMP ANNULE |
| 7 | CHAMP MODIFIE |

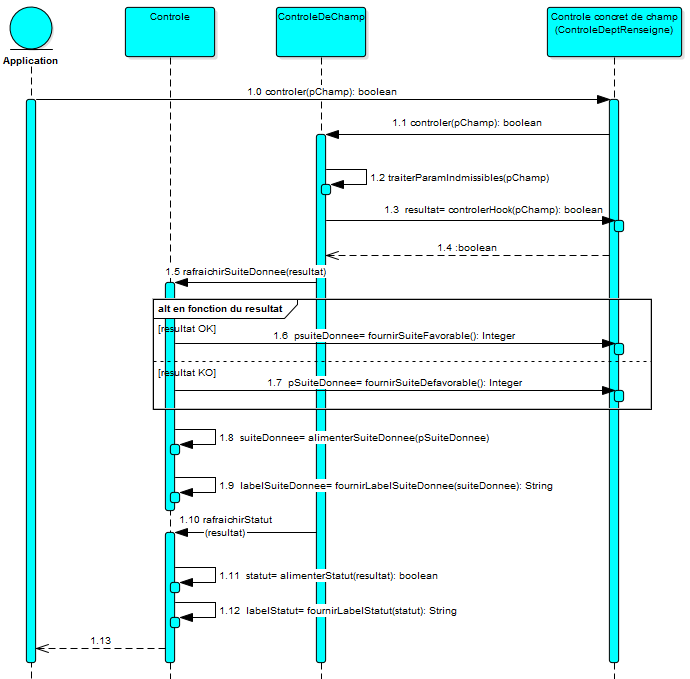
1. Alimentation de l'attribut : par la méthode controler(…) des contrôles concrets.

Il appartient à la classe concrète de contrôle de décider (via sa méthode controler(…)) si le statut est OK ou KO et de décider en conséquence quelle est la suite à donner.

Le label 'labelSuiteDonnee' doit être rafraîchi dès que l'attribut 'suiteDonnee' est modifié, donc à chaque exécution de la méthode controler(…).

1. Méthodes liées à l'attribut : controler(…), fournirSuiteFavorable(), fournirSuiteDéfavorable(), fournirLabelSuiteDonnee().

C'est la méthode controler(…) du contrôle concret qui alimentera l'attribut 'suiteDonnee' en fonction du résultat du contrôle. La suite donnée par controler(…) diffère selon que le contrôle est favorable ou pas. Le contrôle concret doit donc disposer de 2 méthodes fournirSuiteFavorable() et fournirSuiteDéfavorable() qui retournent une suite à donner différente en fonction du résultat du contrôle.

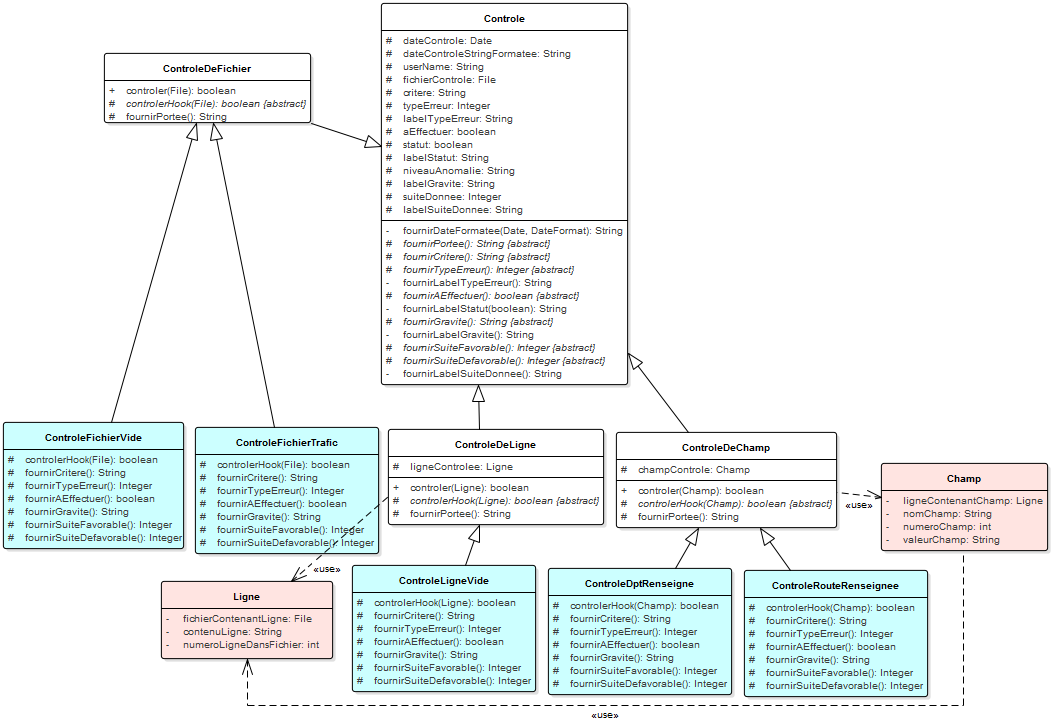


1. Détenteur de la responsabilité de l'attribut : chaque contrôle concret.

Le polymorphisme permet de factoriser dès la conception les tâches répétitives dans les classes de haut niveau.

* Tout contrôle connaît la suite à donner après son exécution. Il faut donc déclarer l'attribut 'suiteDonnee' au plus haut niveau possible dans la classe Contrôle pour garantir que tous les descendants en bénéficieront par héritage.
* Le contrôle concret fournit la suite à donner (en fonction du résultat du contrôle) car il est seul à "savoir" quelle suite il faut donner. Seul un ControleDeptRenseigné sait qu'il faut supprimer la ligne si le département n'est pas renseigné. Les contrôles concrets implémenteront donc les méthodes fournirSuiteFavorable() et fournirSuiteDéfavorable().
* Le polymorphisme permet de factoriser l'appel des méthodes fournirSuiteFavorable() et fournirSuiteDéfavorable() dans la méthode controler(…) de la super-classe ControleDeChamp. En effet, si la suite à donner varie en fonction de chaque contrôle concret, le processus de choix entre la suite à donner favorable ou défavorable est le même pour n'importe quel contrôle : il ne dépend que du résultat de la vérification du critère.
* Il faut déclarer les méthodes *fournirSuiteFavorable()* et *fournirSuiteDéfavorable()* ***abstraites*** au plus haut niveau possible pour garantir que tous les contrôles concrets implémenteront ces méthodes. La classe de haut niveau Contrôle est un candidat idéal.
* L'alimentation de l'attribut 'suiteDonnee' et de l'attribut dépendant 'labelSuiteDonnee' ne varie pas en fonction du contrôle concret. On applique toujours la même procédure consistant à alimenter l'attribut 'suiteDonnee', à aller chercher dans une nomenclature le label correspondant à l'attribut 'suiteDonnee', puis à alimenter 'labelSuiteDonnee' avec ce label. Par conséquent, on peut factoriser tout ce code répétitif au plus haut niveau dans la classe Contrôle. Une méthode fournirLabelSuiteDonnee() au niveau de la classe de haut niveau Contrôle se chargera de rapatrier le label depuis une nomenclature.

A ce stade, le modèle de classes de conception devient :



#### Attribut Objet fichierResultant (File)

1. Contenu :
2. Type :
3. Multiplicité :
4. Nécessité d'un attribut dépendant :
5. Alimentation de l'attribut :
6. Méthodes liées à l'attribut :
7. Détenteur de la responsabilité de l'attribut :

Tout Contrôle connaît le Fichier Résultant qu'il génère. Celui-ci peut être le même que le fichier entrant si le contrôle ne produit aucune modification, ou différent si le contrôle retire des lignes, annule des champs, …

Le Fichier Résultant ne sait rien du Contrôle qui l'a généré.

Il n'est donc pas nécessaire de créer un Type particulier (pure fabrication Objet) pour le concept de Fichier Résultant.

Le type File proposé par Java est suffisant.

#### Attribut Objet reglesGestion (Map<String, String>)

1. Contenu :
2. Type :
3. Multiplicité :
4. Nécessité d'un attribut dépendant :
5. Alimentation de l'attribut :
6. Méthodes liées à l'attribut :
7. Détenteur de la responsabilité de l'attribut :

Tout Contrôle connaît les Règles de Gestion (RG) qu'il implémente. Il peut implémenter plusieurs Règles de Gestion.

Les Règles de Gestion ne connaissent pas les Contrôles qui les implémentent.

Il n'y a donc pas lieu de créer une pure fabrication (Objet spécifique) pour le concept Règle de Gestion.

Un Contrôle doit posséder sa **Collection** de Règles de Gestion qu'il implémente.

Une RG possède toujours un identifiant et une description. Par exemple :

| **Identifiant** | **Description** |
| --- | --- |
| RG-01-06\_ELIMINATION\_LIGNES\_VIDES | Aucun fichier ne doit comporter de lignes vides.  Si des lignes vides sont trouvées, elles sont éliminées et le programme continue. |
| RG-01-07\_ELIMINATION\_LIGNE\_REDHIBITOIRE | Aucun fichier ne doit comporter de lignes avec des champs "rédhibitoires" mal formés.  Si une telle ligne est trouvée, elle est éliminée et le programme continue. |

reglesGestion peut donc être une Collection (objet Java) de type Map<String, String> dans laquelle les enregistrements seront des couples <identifiant, description>

#### Attribut Objet reglesArt (Map<String, String>

Les Règles de l'Art peuvent être vues comme un cas particulier de règles de gestion.

Il n'est donc pas nécessaire de créer un attribut supplémentaire.

#### Attribut Objet rapportControleTxt

1. Contenu :
2. Type :
3. Multiplicité :
4. Nécessité d'un attribut dépendant :
5. Alimentation de l'attribut :
6. Méthodes liées à l'attribut :
7. Détenteur de la responsabilité de l'attribut :

#### Attribut Objet rapportControleCsv

1. Contenu :
2. Type :
3. Multiplicité :
4. Nécessité d'un attribut dépendant :
5. Alimentation de l'attribut :
6. Méthodes liées à l'attribut :
7. Détenteur de la responsabilité de l'attribut :

## Dictionnaire des données d'un Contrôle

| **Attribut** | **Type Java** | **description** |
| --- | --- | --- |
| **dateControle** | Date | * Date d'exécution d'un contrôle. * Possède un un label pour l'affichage. * Fixée lors de l'instanciation d'un contrôle. * Rafraichie à chaque exécution du contrôle. * Obtenue en prélevant la date système. |
| *dateControleFormateeString* | String | * Date d'exécution d'un contrôle sous forme de String rédigée à notre convenance. * *Dépendant* de **dateControle** (champ calculé). * Rafraîchie à chaque modification de dateControle due à une nouvelle exécution du contrôle (invocation de la méthode controler(…)). * Calculée par une méthode de haut niveau fournirDateStringFormatee(). |
| **userName** | String | * Nom de l'organisation du User qui a déclenché le contrôle en soumettant un fichier (DIRA, DIRE, DARWIN, …). * Fixée dès l'instanciation du contrôle et immuable. * Récupérée auprès du User en session lors de l'instanciation du contrôle. |
| **critere** | String | * Critère vérifié par le contrôle comme par exemple "le fichier soumis n'est pas vide" ou "le champ Département dans la ligne du fichier HIT est renseigné". * Codé en dur dans les classes concrètes de contrôle par le développeur. * Fixé dès l'instanciation du contrôle et immuable. |
| **typeErreur** | Integer | * Numéro indiquant le type de l'erreur détectée (1 pour "champ non renseigné", 2 pour "mauvais format de champ", …). * Possède un label à récupérer dans une nomenclature pour affichage. * Fixée dès l'instanciation du contrôle et immuable. * Stockée en dur dans la classe de contrôle concrètes par le développeur. |
| *labelTypeErreur* | String | * Label associé au typeErreur ("champ non renseigné" pour 1, "mauvais format de champ" pour 2, …) contenu dans une nomenclature csv. * *Dépendant* de l'attribut '**typeErreur'**. * Rapatrié depuis une nomenclature csv par une méthode de haut niveau fournirLabelTypeErreur(). |
| **aEffectuer (actif)** | boolean | * Précise si le contrôle doit être exécuté (true). * Fixé dès l'instanciation d'un contrôle mais paramétrable par la MOA. * Paramétrable par la MOA dans un properties. * A rapatrier depuis un properties grâce à une méthode fournirAEffectuer() située dans chaque contrôle concret. |
| **statut** | boolean | * Précise si le contrôle a matché favorablement (true) ou défavorablement (false). * Possède un label à récupérer par une méthode de haut niveau fournirLabelStatut(). * Rafraichi à chaque exécution du contrôle. * Renseigné par la méthode controler(…) du contrôle. |
| *labelStatut* | String | * Label pour l'affichage du statut.   + OK si le statut est true   + KO si le statut est false * *Dépendant* de l'attribut **'statut'**. * Rafraichi à chaque exécution du contrôle. * Calculé par une méthode de haut niveau fournirLabelStatut(). |
| **niveauAnomalie (gravite)** | String | * Gravité de l'anomalie recherchée par le contrôle ("1" pour une 'anomalie bloquante', "4" pour une 'anomalie rédhibitoire' pour une ligne ou pour un champ, …). * Possède un Label à récupérer dans un properties par une méthode de haut niveau fournirGravite(). * Attribut paramétrable par la MOA (dans un properties). * Fixé dès l'instanciation du contrôle et immuable (après recherche dans le properties). * Stockée dans un properties. |
| *labelGravite* | String | * Label associé à la gravité (Anomalie bloquante pour 1, anomalie grave pour 2, …). * Dépendant de 'niveauAnomalie (gravite). * Champ rapatrié depuis un properties. |
| **suiteDonnee** | Integer | * Suite à donner à l'objet contrôlé si le contrôle est défavorable (2 pour FICHIER REFUSE, 4 pour LIGNE RETIREE, 6 pour CHAMP ANNULE, …) |
| *labelSuiteDonnee* | String | * Label de la suite à donner à l'objet contrôlé si le contrôle est défavorable (FICHIER REFUSE pour 2, LIGNE RETIREE pour 4, CHAMP ANNULE pour 6, …) |
| **fichierResultant** | File | * Fichier en sortie du contrôle comportant éventuellement moins de lignes, des champs annulés, … |
| **reglesGestion** | Map<String, String> | * Collection contenant les Règles de Gestion et Règles de l'Art implémentées par le contrôle. |
|  |  |  |

## Etudier l'enchaînement des contrôles

Puisqu'il y a plusieurs enchaînements logiques des contrôles possibles, il peut être intéressant de décider de l'ordre dans lequel on choisira d'exécuter chaque contrôle.

Il est intéressant pour les développeurs (TMA) de pouvoir consulter dans quel ordre les contrôles sont exécutés lors de la vie réelle de l'application.

### Nécessité d'enchaîner les contrôles

### Importance de l'ordre des contrôles

## Bâtir le Modèle de classes de conception

## Tester le modèle de classes de conception

# CONCEPTION TECHNIQUE

Objectif : l'objectif du chapitre "Conception Technique" est d'**élaborer les classes réelles java** que le développeur Java devra coder. On part du modèle de classes de conception et on rajoute et/ou affine les attributs et méthodes.

Bénéfices attendus : la conception technique doit fournir un **modèle de classes de codage** *testé* utilisable par un développeur Java.

Ce travail est réalisé dans le **D**ossier de **C**onception **T**echnique (DCT) d'un projet Khefren.

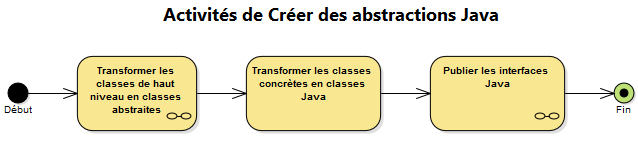
## Créer des abstractions Java

Objectif : **créer la hiérarchie de classes Java** à partir du modèle de classes de conception en utilisant des :

* **Interfaces** Java
* **Classes Abstraites** Java
* **Classes Concrètes** Java

Bénéfices attendus : hiérarchie de classes Java

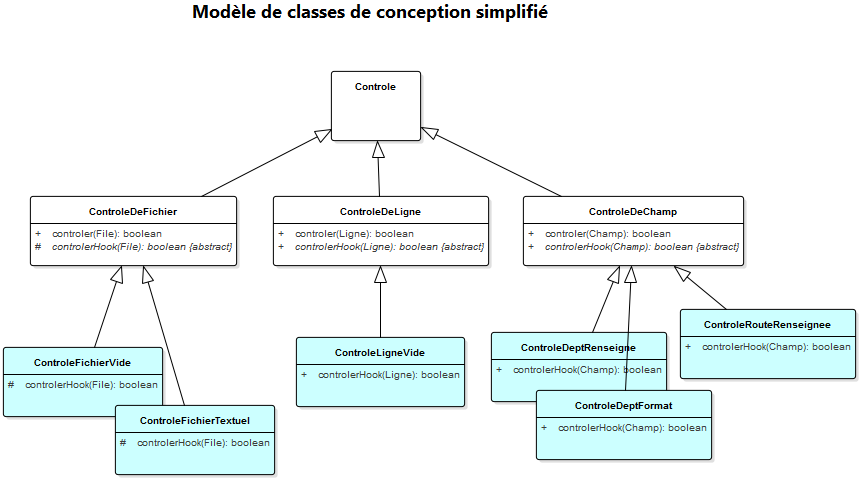
Ce travail est réalisé dans le **D**ossier de **C**onception **T**echnique (DCT) d'un projet Khefren.

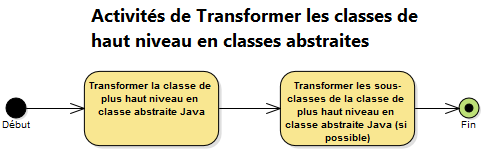


### Transformer les classes de haut niveau en classes abstraites

Objectif : **Factoriser des attributs et des méthodes** (code) dans des classes abstraites.

Le modèle de classes de conception définissait la hiérarchie de classes suivantes :





#### Transformer la classe de conception de très haut niveau Controle

* La classe de très haut niveau Contrôle définie dans le modèle de conception ne fait que :
* **Factoriser les attributs et méthodes de ses sous-classes**. Ces attributs et méthodes sont définis en un seul endroit de l'application à un très haut niveau afin de **garantir que tous les descendants bénéficieront de ces attributs et méthodes**.
* **Limiter l'écriture de code** grâce à la factorisation. En effet, le code factorisé dans la méthode fournirDateSystème(Date pDate) de la classe de haut niveau Contrôle n'est *écrit qu'une seule fois* et bénéficie à toutes les classes descendantes. Ceci contribue fortement à **faciliter la maintenance**. Si nous avions recopié cette méthode dans toutes les classes concrètes (en bleu), nous aurions non seulement dupliqué le code cette méthode, mais en plus, une modification de ce code aurait dû être répercutée dans toutes les classes concrètes.
* **Définir ce que doivent faire les descendants concrets**. Les méthodes *abstraites* dans la classe de haut niveau Contrôle comme par exemple *fournirPortee()* ne "font rien". Elles ne comportent pas de code. Néanmoins, elles **imposent aux descendants d'implémenter** ces méthodes abstraites.

La classe de haut niveau Contrôle n'a pas vocation à fournir des instances (des Objets) puisqu'une instance de Contrôle ne connaitrait pas le critère qu'elle doit contrôler. Le critère est en effet déterminé dans les classes concrètes (en bleu).

De même, la classe de haut niveau Contrôle ne rendrait pas le service attendu de contrôle puisqu'elle ne possède pas de méthode controler(…). Ce sont les sous-classes ControleDeFichier, ControleDeChamp, … qui définissent controler(…).

La classe de haut niveau Contrôle ne sera donc jamais instanciée.

L'outil Java idéal pour implémenter cette classe de haut niveau contrôle est une ***abstraction*** : la **classe abstraite Java**.

La classe abstraite Java est une abstraction qui :

* **Ne peut jamais être instanciée** (elle ne fabrique donc pas d'Objets aussi appelés instances).
* Peut **contenir des attributs et des méthodes concrètes implémentées**. Ces méthodes "font quelque chose".
* Sert pour la **factorisation des attributs et du code générique** qui ne dépend pas des classes concrètes.
* Permet de **définir des méthodes *abstraites*** ne contenant pas de code (donc ne "faisant" rien) mais garantissant que les **descendants** devront **implémenter** ces méthodes abstraites.

La classe de haut niveau Contrôle devient donc une classe abstraite Java **AbstractControle** dans le modèle de classes de codage.

#### Transformer les sous-classes de Contrôle en classes abstraites

* Les sous-classes immédiates de Contrôle définies dans le modèle de classes de conception (ControleDeFichier, ControleDeLigne, …) ne font que :
* **Typer les contrôles concrets** en fonction du fait qu'ils contrôlent un fichier, une ligne, un champ, … Ces classes typent en fonction de la portée du contrôle grâce à la hiérarchie des classes (notion d'héritage).
* **Factoriser les attributs et méthodes de ses sous-classes**.
* **Limiter l'écriture de code** grâce à la **factorisation**.
* **Définir ce que doivent faire les descendants concrets** grâce à des méthodes ***abstraites***.

Les sous-classes de Contrôle (ControleDeFichier, ControleDeLigne, …) n'ont pas vocation à fournir des instances (des Objets) puisqu'une instance de ControleDeFichier… ne connaitrait pas le critère qu'elle doit contrôler. Le critère est en effet déterminé dans les classes concrètes (en bleu).

En revanche, le service principal controler(…) est situé à ce niveau. Un ControleDeFichier connaît déjà sa portée fichier et sait que ses descendants contrôleront des Fichiers (Objet File). De même, un ControleDeLigne sait déjà que ses descendants contrôleront des Lignes (Objet Ligne) …

La méthode controler(…) peut être **définie à ce niveau** puisque le type dans la hiérarchie de classes est déjà suffisant pour savoir si on écrira controler(File), controler(Ligne), controler(Champ), … **Les sous-classes connaissent donc déjà le type d'objet contrôlé de leur méthode controler(…)**.

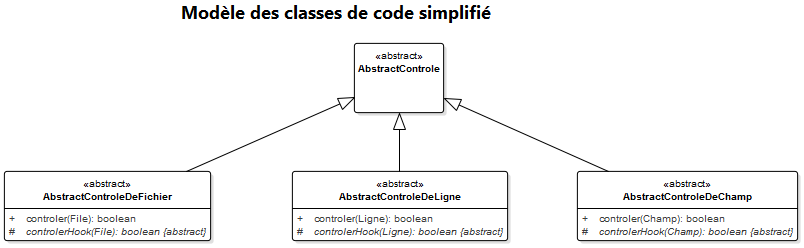
Mais on ne peut pas coder l'essentiel de la méthode controler(…) à ce niveau puisque l'on ne sait pas quel critère il faut vérifier. Il n'y a aucune raison pour que le code qui vérifie qu'un champ Département est renseigné soit le même que le code qui s'assure qu'un champ Route respecte un bon format (expression régulière). Il faut donc :

* **Factoriser tout ce qui est factorisable** dans la méthode controler(…) à ce niveau. Par exemple, le traitement des mauvais paramètres, le rafraîchissement des attributs, … Il faut essayer de *factoriser ici tous les processus qui ne dépendent pas de l'instance concrète réelle de contrôle et qui sont toujours réalisés de la même façon*.
* **Déléguer à une méthode concrète** dans chaque classe concrète que j'appelle controlerHook(…) la tâche de vérifier le critère de contrôle. C'est le Design Pattern **Hook**. On ne définit donc dans ces sous-classes que des méthodes *abstraites* *controlerHook(…)* qui garantiront que tous les descendants l'implémenteront.

Les sous-classes immédiates de Contrôle ne seront jamais instanciées puisqu'elles ne pourraient à elles-seules exécuter un contrôle (elles ne connaissent pas le critère à verifier).

L'outil classe Abstraite Java s'impose pour les sous-classes immédiates de Contrôle puisqu'elles permettent de réaliser tous ces éléments.

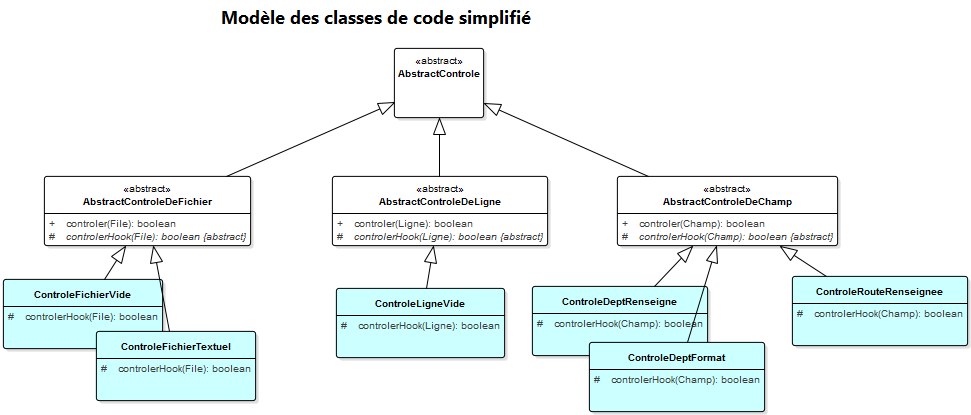
Les sous-classes de la classe Contrôle (**ControleDeFichier**, **ControleDeLigne**, **ControleDeChamp**, …) deviennent des classes abstraites Java.



### Transformer les classes concrètes de conception en classes concrètes Java

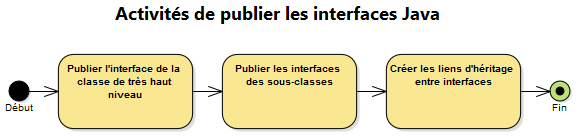
Objectif : **définir les classes concrètes Java** qui *fourniront des instances* (Objets).

Les classes concrètes (en bleu dans le modèle de conception) sont transformées en classes (concrètes) Java.



### Publier des Interfaces Java

Objectif : **définir les méthodes publiques** (services) qui pourront être accédés par toute l'application. On définit ici ce que "**doit savoir faire**" une classe ou hiérarchie de classes Java.



L'abstraction la plus "ultime" en Java est l'**interface Java**.

L'interface Java permet de "**publier**" au sens de *rendre visible partout dans l'application* tous les "**services**", c'est-à-dire toutes les méthodes publiques intéressantes d'une classe ou d'une hiérarchie de classes.

L'interface Java est donc l'outil idéal pour **définir ce que doit faire** une classe ou une hiérarchie de classes. Elle **expose les méthodes qui doivent être accessibles à toute l'application** tout en **masquant leur implémentation**. En effet, l'interface précise la définition des méthodes sans que l'on sache comment les méthodes réalisent leur travail. On sait juste ce que la méthode prend en argument et ce qu'elle doit retourner sans savoir comment elle fait.

L'interface Java ne contient absolument pas de code. Elle ne contient que la **définition des méthodes**, c'est-à-dire :

* Le **nom** des méthodes.
* La **liste des arguments typés** des méthodes (ce que la méthode accepte en entrée).
* Le **type de retour** des méthodes (ce que la méthode produit ou rien (**void**) s'il s'agit d'une méthode à effet de bord).

Toutes les méthodes définies dans une Interface Java sont donc :

* **Abstraites** (elles ne font rien puisqu'elles ne contiennent pas de code)
* **Publiques**.

L'Interface Java est une abstraction qui :

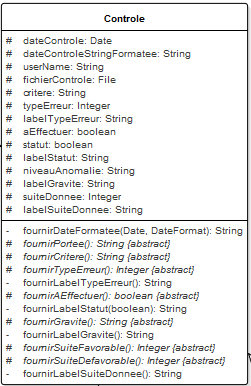
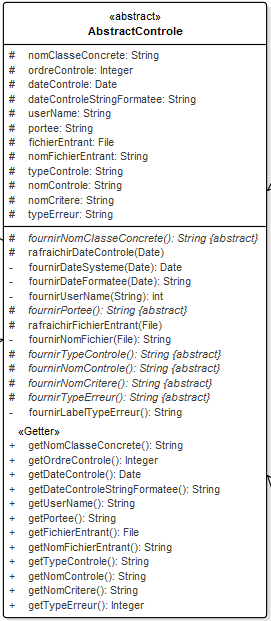
* **Ne peut jamais être instanciée** (elle ne fabrique donc pas d'Objets aussi appelés instances).
* Ne peut **contenir ni attributs, ni méthodes concrètes implémentées**. Les méthodes dans une interface "ne font rien". L'interface ne sert qu'à la *définition* des méthodes offertes par la classe ou la hiérarchie de classes.
* Permet de **définir des méthodes *abstraites*** ne contenant pas de code (donc ne "faisant" rien) mais garantissant que les **descendants** devront **implémenter** ces méthodes abstraites.

Quelques points importants pour la conception technique en Java :

* **Java ne tolère pas l'héritage multiple**. Une classe Java ne peut donc hériter que d'une seule classe Java (concrète ou abstraite).
* Une classe Java **peut implémenter de multiples interfaces** Java.
* Une **interface Java peut elle-même étendre plusieurs interfaces** Java. C'est très commode pour bien séparer divers types de services rendus par une classe (par exemple des services d'impression des rapports de contrôle que l'on pourra séparer des services de contrôle d'un fichier).

#### Publier l'interface de la classe de très haut niveau

La classe de très haut niveau Contrôle dans le modèle des classes de conception a précédemment été transformée en classe abstraite AbstractControle.

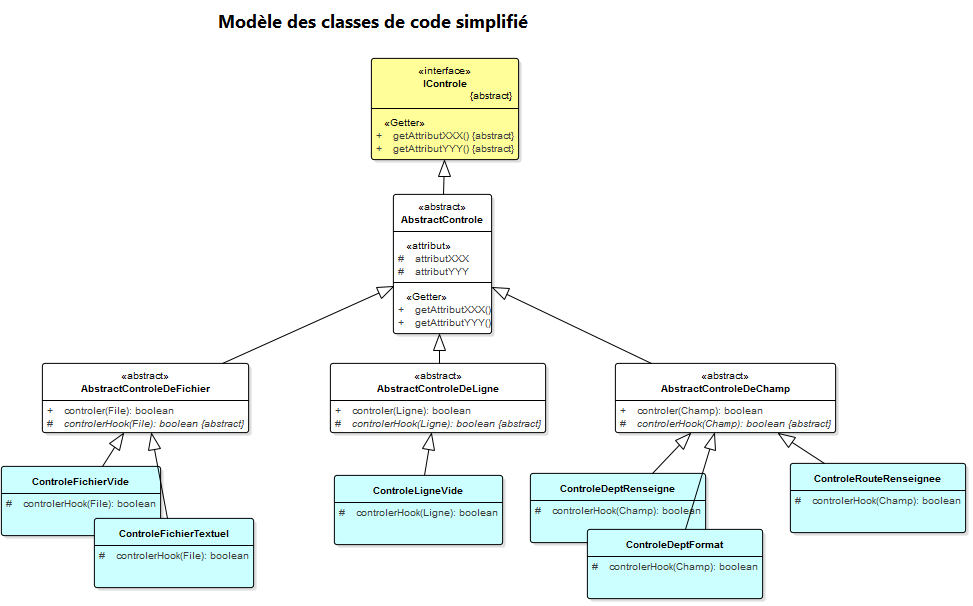
Cette classe abstraite AbstractControle :

* **Définit les attributs possédés par toutes les sous-classes de la hiérarchie**. Rappelons que toutes les sous-classes possèdent ces attributs par héritage.
* **Encapsule les attributs**. Ces attributs ne doivent pas être directement accédés par l'application sinon un développeur pourrait changer la valeur d'un attribut. Par exemple, un développeur pourrait changer le critère vérifié par un contrôle. Il faut protéger ces attributs de l'extérieur. On dit qu'on les *encapsule*, c'est-à-dire qu'on les rend invisibles depuis l'extérieur de la classe. On met devant les attributs dans AbstractControle un **modérateur** **protected** pour que l'attribut ne soit visible que par les sous-classes et pas depuis l'extérieur. Ce modérateur **protected** est symbolisé en UML par le dièse #.
* **Fournit par référence les attributs grâce aux accesseurs (getters)**. En effet, on ne veut pas qu'un développeur puisse changer la valeur d'un attribut. Seule notre classe est chargée de la modification de ses attributs (de son *état*). En revanche, il faut permettre à un développeur d'accéder à l'état de notre classe. C'est l'objet des getters (méthodes publiques getNomClasseConcrete(), getDateControle(), … modérées par **public** (symbole + en UML)).
* **Encapsule les méthodes** qui ne doivent pas être vues par un développeur tiers et qui ne sont utilisées que par notre classe. La méthode rafraîchirDateControle(Date pDate) est utilisée en interne par notre classe pour obtenir la date d'exécution d'un contrôle. Un développeur tiers n'a pas à se soucier de cette méthode ni à interférer avec le calcul de la date du contrôle. Ces méthodes sont donc modérées **private** (symbolisé par – en UML) lorsque l'on veut qu'elles ne soient visibles que dans la classe AbstractControle ou **protected** lorsque l'on veut qu'elles ne soient vues que par la hiérarchie de classes.

Les services (méthodes publiques) dotés du modérateur **public** (symbolisé par + en UML) sont donc les accesseurs (getters). Nous pouvons donc publier ces accesseurs dans une interface Java 'IControle'.

* Les getters de chaque attribut de haut niveau seront donc définis de manière *abstraite* au niveau de l'interface IControle. Ils seront forcément **public**.
* La classe abstraite AbstractControle **implémente** l'interface IControle.
* Le processus réalisé par un accesseur (getter) ne varie jamais quel que soit le niveau où il est implémenté. Tous les contrôles concrets font la même chose lorsqu'ils fournissent la valeur d'un attribut. Les getters peuvent donc être déjà implémentés dans la classe de haut niveau AbstractControle.
* Tous les descendants dans la hiérarchie de classes disposent de ces accesseurs (getters) par héritage.
* On peut maintenant dire qu'un contrôle concret comme par exemple ControleRouteRenseignee "**est un**" IControle puisqu'il en hérite.

Le modèle des classes de code est donc modifié comme suit :



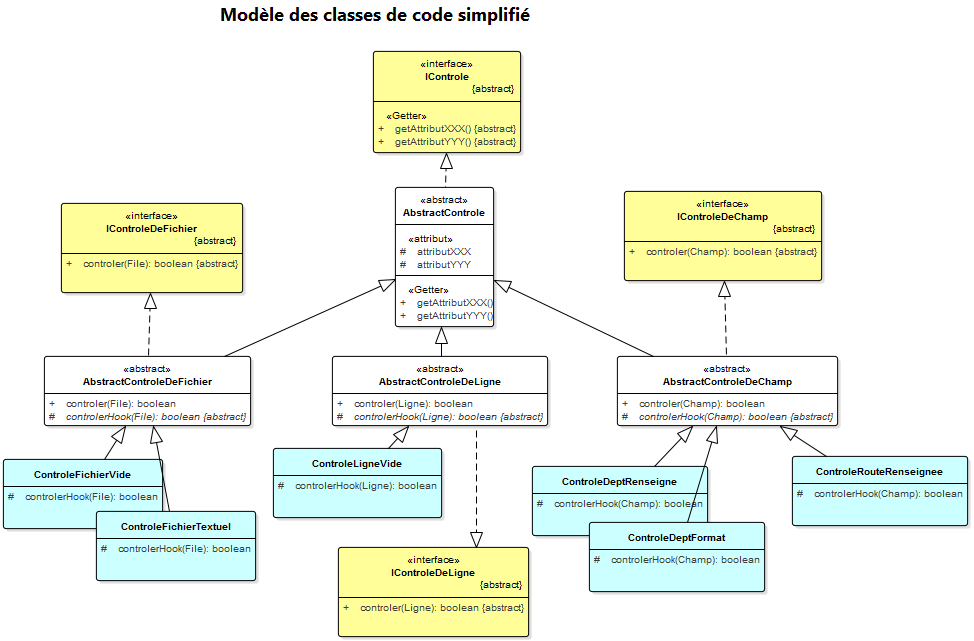
#### Publier les interfaces des sous-classes de la classe de très haut niveau

Les sous-classes de AbstractControle comme AbstractControleDeFichier, AbstractControleDeLigne typent la portée du contrôle (fichier, ligne, champ, …) et surtout permettent d'implémenter à ce niveau la méthode publique (service) controler(…).

On sait en effet à ce stade comment on doit typer la méthode controler(…) en controler(File pFile), controler(Ligne pLigne), controler(Champ pChamp), …

Il est essentiel qu'un développeur tiers ne voit pas la méthode controlerHook(…). Celle-ci fait partie de la logique interne de la hiérarchie de classes. Le développeur tiers doit pouvoir invoquer la méthode controler(…) de n'importe quel contrôle concret sans se soucier du fait que la vérification du critère est en réalité réalisée dans la méthode controlerHook(…) du contrôle concret.

On publie donc les interfaces de chaque sous-classes uniquement avec la bonne méthode controler(…).



#### Créer les liens d'héritage entre interfaces

Il est maintenant possible de dire qu'un ControleFichierTextuel "**est un**" IControleDeFichier puisque :

* ControleFichierTextuel **hérite** de AbstractControleDeFichier.
* AbstractControleDeFichier **implémente** IControleDeFichier.

Par ailleurs, ControleFichierTextuel "**est un**" IControle puisque :

* ControleFichierTextuel **hérite** de AbstractControleDeFichier.
* AbstractControleDeFichier **hérite** de AbstractControle.
* AbstractControle **implémente** IControle

Mais attention, si on "**voit**" un ControleFichierTextuel comme un IControle, on ne sait pas que le ControleFichierTextuel dispose d'une méthode controler(File pFile). En effet, la méthode controler(File pFile) n'est définie qu'au niveau de IControleDeFichier, pas dans IControle.

On pourra écrire le code suivant en Java :

// Instanciation d'un ControleFichierTextuel avec la "visibilité" de son interface IControle.

IControle controleFichierTextuel = new ControleFichierTextuel();

Mais on ne pourra pas écrire ensuite le code Java pour invoquer la méthode controler(…) :

controleFichierTextuel.**controler**('Fichier HIT\_DIRA\_2014');

Le compilateur Java signalerait une erreur de compilation. Ceci est dû au fait que l'on a instancié l'objet référencé controleFichierTextuel avec la **visibilité** de son Interface IControle. Laquelle ne dispose pas de la méthode controler(File pFile).

On pourrait par contre écrire en Java :

// Instanciation d'un ControleFichierTextuel avec la "visibilité" de son interface IControleDeFichier.

IControleDeFichier controleFichierTextuel = new ControleFichierTextuel();

controleFichierTextuel.**controler**('Fichier HIT\_DIRA\_2014');

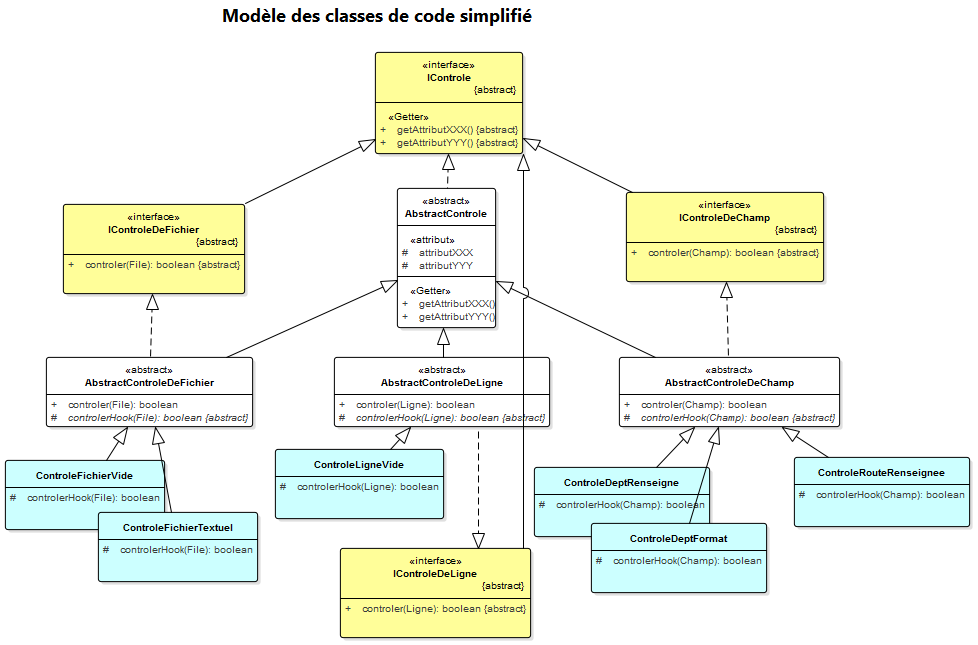
L'objet controleFichierTextuel a maintenant été instancié avec la **visibilité de son interface** IControleDeFichier qui possède elle la méthode controler(File pFile).

La question devient : "comment faire pour que l'interface IControleDeFichier dispose des getters qui ont été définis dans l'interface IControle ?".

La réponse est :

"en faisant hériter l'interface IControleDeFichier de l'interface IControle ". Java permet l'héritage entre interfaces comme entre classes Java.

Le modèle de classes de code est transformé comme suit :



## Factoriser au maximum le code au plus haut niveau (abstractions)

## Déterminer les attributs et méthodes de codage

### Attribut nomClasseConcrete

**protected** **transient** String nomClasseConcrete;

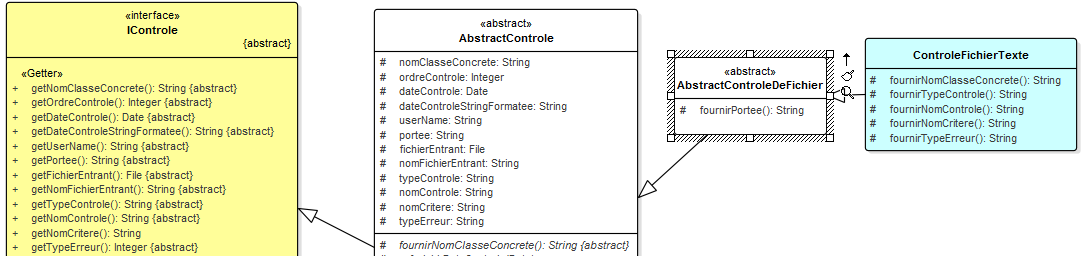
#### Type Java de l'attribut nomClasseConcrete : String

**String**.

Cet attribut *rajouté par rapport au modèle de classes de conception* ne sert qu'à afficher "'Classe ' + nom de la classe concrète" de contrôle dans les logs et les messages comme par exemple "Classe ControleFichierTexte".

#### Positionnement de l'attribut nomClasseConcrete

Tous les contrôles doivent posséder cet attribut. Il est donc déclaré dans la classe de plus haut niveau AbstractControle.



#### Modérateurs de l'attribut nomClasseConcrete

* Tous les contrôles doivent connaitre cet attribut. Il est donc modéré **protected** pour être "visible" par tous les descendants.
* Il est **transient** (volatile) puisqu'il n'a pas vocation à être sauvegardé en base.

#### Responsable de la définition de l'attribut nomClasseConcrete

Chaque contrôle concret est seul à connaître le nom de sa classe. C'est le développeur qui a fixé le nom de la classe concrète.

* On place une méthode *abstraite* dans la classe de plus haut niveau AbstractControle :

**protected** **abstract** String *fournirNomClasseConcrete()*;

* On place la méthode concrète dans chaque contrôle concret :

**protected** **final** String fournirNomClasseConcrete(){code…}

Cette méthode concrète se charge de retourner le nom de la classe concrète comme "Classe ControleRouteRenseignee".

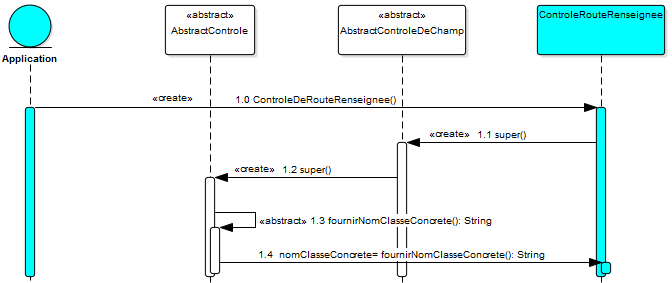
#### Méthodes liées à l'attribut nomClasseConcrete

* **protected** **abstract** String *fournirNomClasseConcrete()* dans la classe de haut niveau AbstractControle.
* **protected** **final** String fournirNomClasseConcrete() dans les classes concrètes.

Le polymorphisme réalise l'appel de de la bonne méthode fournirNomClasseConcrete() de la bonne classe concrète dès l'instanciation d'un contrôle, c'est-à-dire dès que l'on appelle son constructeur.

#### Alimentation de l'attribut nomClasseConcrete : instanciation

Cet attribut est alimenté dès l'instanciation d'un contrôle et ne peut plus être changé (immuable).



#### Accession à l'attribut nomClasseConcret

Getter **public final** String getNomClasseConcrete() dans la classe de plus haut niveau AbstractControle qui retourne nomClasseConcrete à tout objet appelant (modérateur **public**).

Ce getter est modéré **final** puisqu'il n'a jamais besoin d'être redéfini dans les sous-classes. On sait dès le plus haut niveau AbstractControle que la méthode getNomClasseConcrete() devra retourner 'nomClasseConcrete' et il n'y a aucune raison de modifier ce comportement dans les descendants.

### Attribut ordreControle

**protected** **transient** Integer ordreControle;

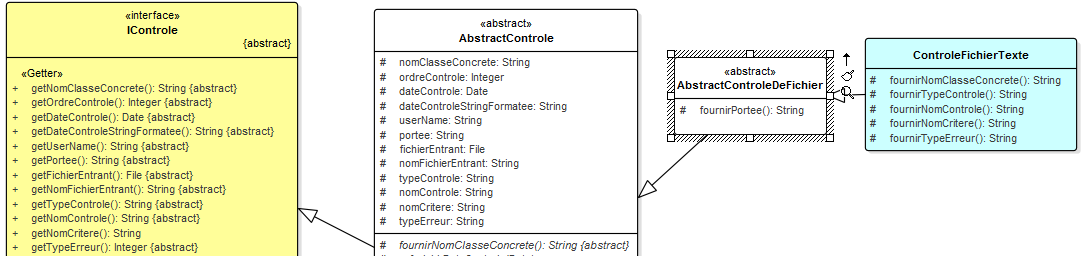
#### Type Java de l'attribut ordreControle

**Integer**.

Cet attribut rajouté par rapport au modèle de classes de conception ne sert qu'à mémoriser l'ordre du contrôle dans un enchaînement de contrôles. Il n'est pas indispensable à l'application et peut être null (nullable).

#### Positionnement de l'attribut ordreControle

Tous les contrôles doivent posséder cet attribut. Il est donc déclaré dans la classe de plus haut niveau AbstractControle.



#### Modérateurs de l'attribut ordreControle

* Tous les contrôles doivent connaitre cet attribut. Il est donc modéré **protected** pour être "visible" par tous les descendants.
* Il est **transient** (volatile) puisqu'il n'a pas vocation à être sauvegardé en base.

#### Responsable de la définition de l'attribut ordreControle

Une classe chargée d'enchaîner les contrôles décide de l'ordre des contrôles dès leur instanciation.

#### Méthodes liées à l'attribut ordreControle

#### Alimentation de l'attribut ordreControle

#### Accession à l'attribut ordreControle

Type Java de l'attribut

Positionnement de l'attribut

Modérateurs de l'attribut

Responsable de la définition de l'attribut

Méthodes liées à l'attribut

Alimentation de l'attribut

Accession à l'attribut

## Définir les constructeurs des classes

## Concevoir des classes autonomes et réutilisables

## Appliquer les règles de codage (Design Patterns)

### Utiliser le polymorphisme

### Utiliser des hooks

## Définir les objets techniques

## Créer les packages

## Bâtir le modèle de classe de codage

# TESTS ET VERIFICATIONS

## Vérifier la Traçabilité Besoins-Exigences-RG-Classes-Attributs-Méthodes

# CAPITALISATION de la méthode