Capsis 4 – Documentation de Référence

F. de Coligny

v 1.1 − Capsis 4.1.0 − 26 sep 2001

Versions précédentes :

Version documentation	*Version Capsis*	*Date de publication*
v1.0	v4.0	30 avr 2001

Table des matières

1 Introduction

Capsis4 est une plate−forme de simulation pour l'étude de la production et de la dynamique des peuplements forestiers. Elle héberge des modèles dendrométriques de dynamique forestière et propose des outils pour procéder à des interventions sur les peuplements.

Les modèles sont de plusieurs types (modèles de peuplement, modèles arbre indépendants des distances – MAID, modèles arbre dépendants des distances – MADD, modèles mixtes...) et s 'appliquent sur des structures de données plus ou moins détaillées (l'arbre peut être individualisé et même spatialisé).

Ainsi, certains modèles considèrent un peuplement équienne mono−spécifique dans son ensemble pour calculer des grandeurs générales relatives à son évolution (densité, facteur d'espacement, caractéristiques de l'arbre dominant ou moyen), quand d'autres modèles s'intéressent à chaque arbre d'un peuplement hétérogène et calculent ses propriétés en tenant compte de la gène que peut lui occasionner son voisinage pour l'accès aux ressources (eaux, lumière...).

Ces modèles ont en commun de calculer l'évolution d'un peuplement forestier (au minimum la croissance des arbres existants, mais aussi parfois la mortalité, la régénération, l'élagage, les déformations...) et d'être utilisés dans une démarche d'établissement de scénarios sylvicoles alternant des phases d'évolution et des interventions (éclaircie, fertilisation...).

Pour répondre à ces besoins, Capsis4 adopte la structure modulaire des versions précédentes (Capsis2 [Dreyfus96] et Capsis3) : un module par modèle.

Capsis4 gère une structure de données "légère" axée sur l'organisation des données (session, projet, étape). Le logiciel a peu d'exigences sur la structure du peuplement sur lequel on travaille. Il considère un "peuplement générique" (*GStand*) dont il ignore par exemple s'il comporte une liste d'arbres ou s'il est associé à un objet de type terrain.

Pour ce qui est des structures de données fonctionnelles, des propositions sont faites au modélisateur en terme de base de description des objets de son modèle (arbre générique spatialisé ou non, terrain générique découpé en cellules...). Capsis4 propose des outils travaillant sur ces structures de données générales. Ces outils sont utilisables par les modèles qui s'appuient sur les structures en question.

Les objets génériques proposés ont une description minimale pour être peu contraignante (ex: `GTree` – identifiant, age, diamètre, hauteur). Cette description est complétée dans chaque module par le jeu de l'héritage de la Programmation Orientée Objets (POO) (ex: arbre du modèle A : `GTree` + hauteur du houppier et rayon de sa base).

Le modélisateur peut choisir de s’appuyer sur les objets génériques proposés, ou bien de reconstruire tout ce dont il a besoin. La seule exigence de Capsis4 envers ses modules est de pouvoir faire évoluer un objet peuplement dérivé de `GStand` (le "peuplement générique") pour qu’il soit possible d’associer ses états successifs à des étapes de scénario sylvicole.

2 Architecture

2.1 Introduction

L'architecture de Capsis4 est de type *noyau/modules* avec *pilote* séparé. Elle est également *extensible*.

L'un des principes fondateurs de l'architecture de Capsis4 est la possibilité de spécialisation dans des modules de structures de données génériques décrites dans un noyau applicatif. Ce mécanisme requiert l'emploi d'un langage de Programmation Orientée Objets (POO) pour disposer de la notion essentielle d'héritage. Capsis4 est développé en langage Java[1] [Gosling96].

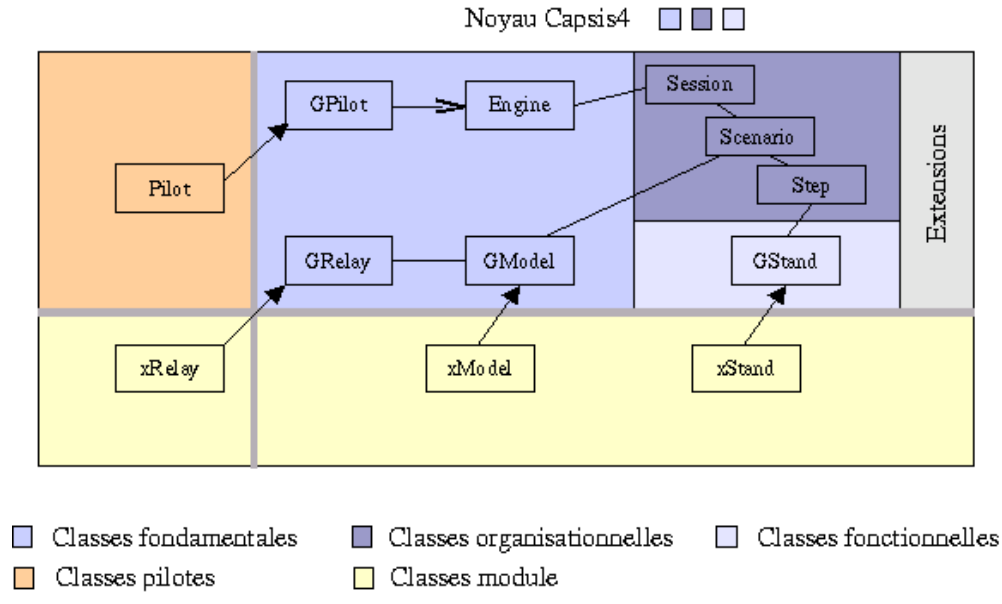


Fig. 1 − Architecture Capsis4

Le *noyau applicatif* est le coeur de Capsis4. Il fournit les opérations de base du logiciel, tels que la gestion de la structure de données et le chargement des modules. Les fonctionnalités du noyau sont accessibles par tous les composants de Capsis4, modules et extensions compris.

Capsis4 est utilisé au moyen d'un *pilote*. C'est un ensemble de classes permettant de conduire des simulations en tirant parti des possibilités offertes par le noyau. Deux pilotes au moins sont prévus : un pilote interactif graphique permettant le dialogue avec l'utilisateur au travers d'une interface graphique, et un pilote en mode console, fonctionnant automatiquement à partir de fichiers de paramètres et de scripts de scénarios préalablement construits.

1 Les notions de Programmation Orientée Objets et le langage Java sont supposés connues du lecteur.

Dans ce document, il n'est fait référence qu'au pilote interactif.

Un *module* Capsis4 est l'implémentation concrète d'un modèle de dynamique forestière au moyen de structures de données, de méthodes et d'algorithmes dans des classes dites fonctionnelles. Ces classes contiennent en quelque sorte la connaissance du modélisateur.

Les modules sont accompagnés d'un ensemble de classes *relais* pour compléter ou remplacer certaines classes du pilote générique. Le *relais* interactif du modèle comporte notamment des boites de dialogues pour l'acquisition de paramètres nécessaires au fonctionnement propre du modèle.

Le *mécanisme d'extensions* permet de rajouter des fonctionnalités à Capsis4. L'externalisation de certains traitements sous forme d'extensions permet la réutilisation d'outils par plusieurs modèles, le partage de méthodes et l'évolution ultérieure de la plate−forme par construction de nouvelles extensions.

Il existe plusieurs types d'extensions pour extraire ou représenter les données calculées, filtrer des collections d'objets (arbres, placeaux...) ou encore permettre l'importation et l'exportation des données vers d'autres logiciels.

Capsis4 se charge de gérer la compatibilité des extensions entre elles ou avec les modules. La typologie des extensions est ouverte et susceptible d'évolutions.

2.2 Le noyau − capsis.kernel

Les principales fonctionnalités proposées par le noyau Capsis4 sont décrites ici. Elles sont prévues pour être accessibles par tous les pilotes Capsis4, qu'ils soient interactifs ou non.

Les classes évoquées dans ce chapitre appartiennent au package `capsis.kernel` sauf mention contraire. Elles peuvent être regroupées en plusieurs catégories.

2.2.1 Structure de données générique

La structure de données de Capsis4 est décrite à deux niveaux : *organisationnel* et *fonctionnel*.

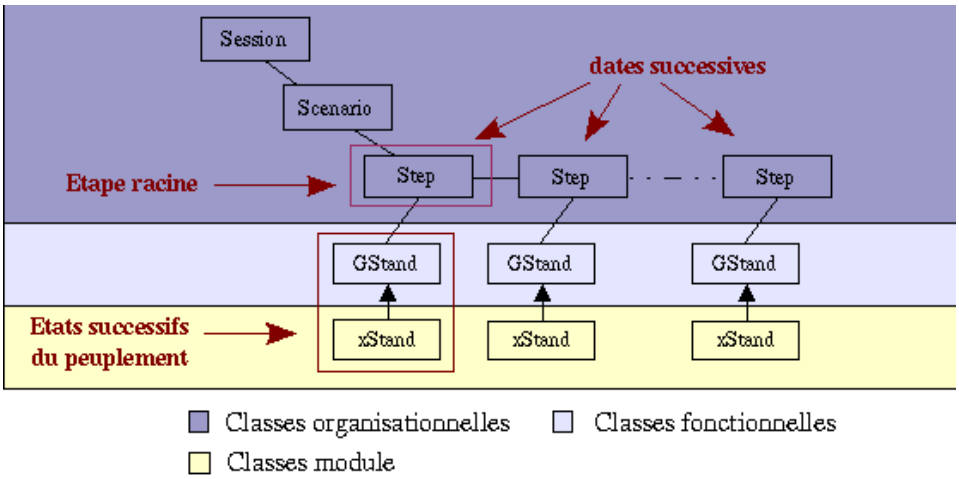


Fig. 2 − Structure de données Capsis4

2.2.1.1 Niveau organisationnel

Au *niveau organisationnel*, Capsis4 gère des scénarios sylvicoles (ou Projets) regroupés dans une session de travail et composés d'étapes successives. A chaque projet est associé un modèle avec un paramétrage déterminé lors d'une phase initiale (lors de la création du projet).

Le projet comporte une étape racine, supportant un peuplement initial fourni par le modèle (chargement d'un inventaire, génération virtuelle...).

Il est ensuite possible de créer d'autres étapes en demandant au modèle de faire évoluer le peuplement initial ou en procédant à une intervention (ex: une éclaircie).

Un projet peut contenir plusieurs scénarios sylvicoles ayant en commun une même étape racine, mais reflétant des sylvicultures différentes.

a. La Session − capsis.kernel.Session

La classe `Session` comporte une collection de projets. Il est possible d'ajouter des projets ou d'en supprimer avec les méthodes prévues à cet effet. La session est caractérisée par un nom et peut donner des indication sur son statut de sauvegarde (sauvée, déjà sauvée une fois...).

b. Le Projet − capsis.kernel.Scenario

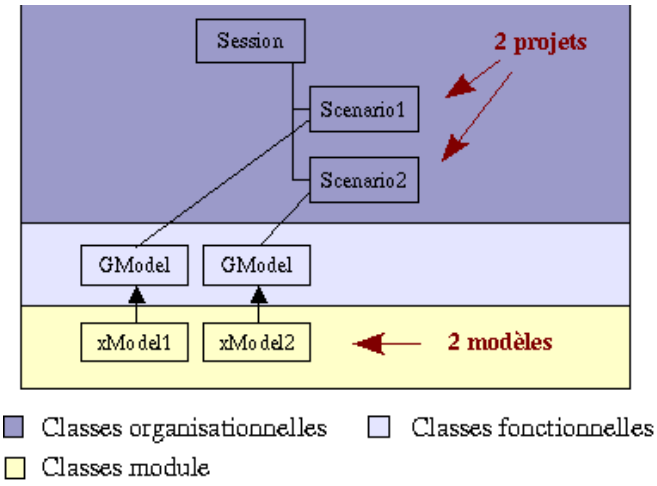


Fig. 3 − Session, projets et modèles associés

Le projet est décrit dans le noyau par la classe `Scenario`. Le projet peut en effet être considéré comme un scénario à options ayant une particularité forte : toutes les branches de sous−scénario ont la même étape racine. Le projet (`Scenario`) est donc un arbre général (chaque noeud a plusieurs fils) dont le noeud est une étape (voir ci−après `Step`).

La superclasse de `Scenario` est `capsis.util.NTree` (arbre général). Cette classe est caractérisée principalement par la gestion d'une étape racine (*"root"*). *NTree* peut renvoyer un iterateur (`Iterator` Java) qui contient la liste des noeuds de l'arbre parcouru en préordre. Cette liste peut se limiter aux noeuds *"visibles"* (voir `Step`).

c. L'étape − capsis.kernel.Step

La classe `Step` décrit l'*étape* Capsis4, dont la fonction est de représenter un peuplement calculé par un modèle donné à une date donnée. Cette entité est un noeud d'arbre général (*NTree*) par héritage de la classe `capsis.util.Node`.

`Node` est la base d'une implémentation d'arbre général de type "left−son, right−brother" [Cormen90] qui s'appuie sur la connaissance par chaque noeud de son "père", son "fils gauche" et son "frère droit". Cette implémentation est économique en place parce qu'elle évite une liste de fils pour chaque noeud.

De plus, chaque noeud porte une propriété qui le définit comme *"visible"* ou *invisible"* dans les projets Capsis4. Cette notion de visibilité sert à marquer les noeuds charnière lors de l'établissement d'un projet. Elle est modifiable à loisir. Il en résulte des représentations simplifiées des projets, ne faisant état que des noeuds visibles.

Note : La *racine*, les *noeuds porteurs* (qui sont racine de plusieurs branches de scénario) ainsi que les *noeuds feuilles* (terminant une branche de scénario) sont toujours visibles.

`Node` possède des méthodes pour insérer un nœud après un noeud donné, supprimer un nœud, supprimer tous les nœuds père d'un nœud donné jusqu'au précédent noeud visible non compris, rendre le père visible d'un nœud donné, déterminer si un nœud est racine ou feuille de l'arbre.

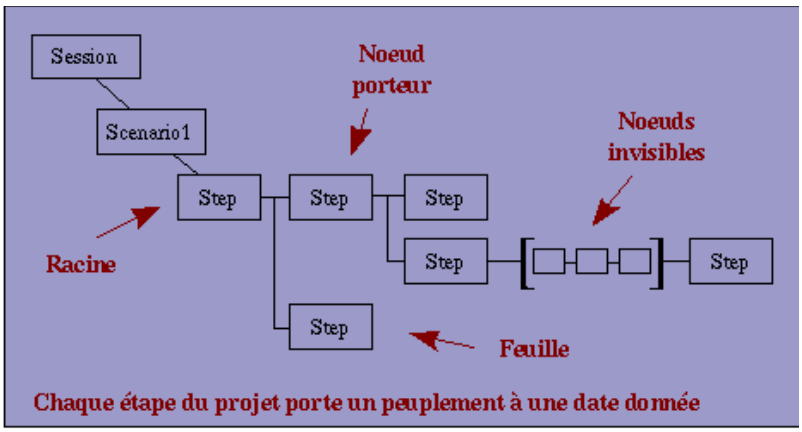


Fig. 4 − Projet et étapes

`Node` implémente en outre le *Design Pattern "Visitor"* [Gamma95] qui permet à un objet quelconque de parcourir l'arbre pour effectuer une tâche sur chaque noeud sans que le noeud ne soit conçu pour cette tâche a priori.

La classe `Step` hérite donc des fonctionnalités de `Node`, auxquelles elle ajoute des méthodes pour connaître le scénario auquel elle est rattachée et le peuplement qu'elle porte.

2.2.1.2 Niveau fonctionnel

a. Peuplement générique − capsis.kernel.GStand, GTCStand

Au *niveau fonctionnel*, Capsis4 décrit un peuplement générique (`GStand`). Cette description est volontairement "légère" pour ne pas contraindre le modélisateur, mais elle prévoit néanmoins le minimum requis pour un peuplement Capsis4 :

- Récupération de l'étape portant le peuplement et d'un élément de datation dans le projet (qui peut être par exemple une année ou un nombre d'années (de mois...) depuis l'étape racine).

- Possibilité d'associer un objet terrain (`GPlot`). Ce terrain est facultatif. Il peut comporter un découpage en cellules de terrain individualisées connaissant les arbres qu'elles contiennent. Ce mécanisme peut permettre d'implémenter des traitements au niveau cellule (ex: régénération).

- Mécanisme pour l'évolution : on peut demander au peuplement qu'il renvoie une base pour l'évolution. Cette base doit être une copie de lui même (mais sans arbres pour les modèles à arbre individualisé), que le processus d'évolution va compléter pour donner le peuplement évolué. Pour les implémentations lourdes de `GStand` (ex : beaucoup d'arbres détaillés et association d'un terrain avec de nombreuses cellules), ce mécanisme peut choisir de renvoyer une copie "légère" de lui même (ex: copie sans arbres avec une référence vers son propre terrain non cloné). Cette précaution peut permettre d'accélérer les traitements mais implique des conséquences qu'il faut prendre en compte.

- Mécanisme pour l'intervention. Renvoie une copie du peuplement complet (avec arbres dans le cas d'un modèle spatialisé) sur lequel le dispositif d'intervention va travailler, par exemple pour supprimer ou marquer des arbres.

- Des éléments d'identification (légende, source initiale des données...)

Une implémentation par défaut de `GStand` est fournie : `GTCStand` (pour "Generic Tree Collection Stand"). `GTCStand` est un peuplement générique qui possède une collection d'arbres de type `GTree` (ou une de ses sous−classes).

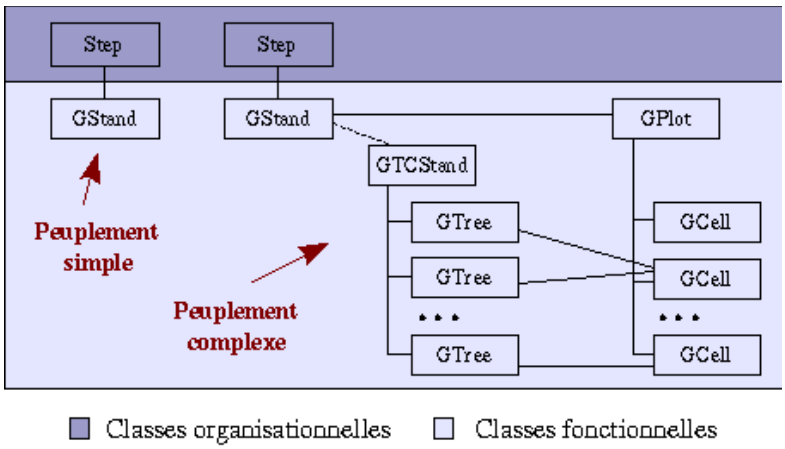


Fig. 5 − Etape et peuplement. Ex : un modèle Peuplement et un modèle arbre individuel spatialisé

L'implémentation de la collection d'arbres est laissée à la discrétion du modélisateur qui décide d'utiliser (sous−classer) `GTCStand` dans son module. Une implémentation par défaut est fournie (sur la base d'une `java.util.Hashtable`). L'implémentation doit respecter le contrat fixé par l'interface `capsis.util.TreeCollection`, implémentée par `GTCStand` par l'intermédiaire de `TreeCollectionHandler`. Cette interface prévoit en particulier l'ajout et la suppression d'un arbres, mais aussi la récupération d'un arbre à partir de son identifiant.

Par le jeu de la redéfinition de la méthode créant la `TreeCollection`, le modélisateur peut choisir de créer une collection propre implémentant l'interface `TreeCollection`, mais basée sur une structure de données qui favorise les opérations qu'il prévoit sur ses arbres (ex: recherche de voisins).

b. Arbre − capsis.kernel.GTree, GMaddTree, GMaidTree

`GTree` est une description d'arbre générique. Son utilisation par les modèles avec arbre individuel n'est pas requise, mais elle permet l'emploi d'outils qui reconnaissent les objets de ce type. Cette remarque est généralisable à beaucoup de classes génériques dont l'usage est recommandé mais pas obligatoire.

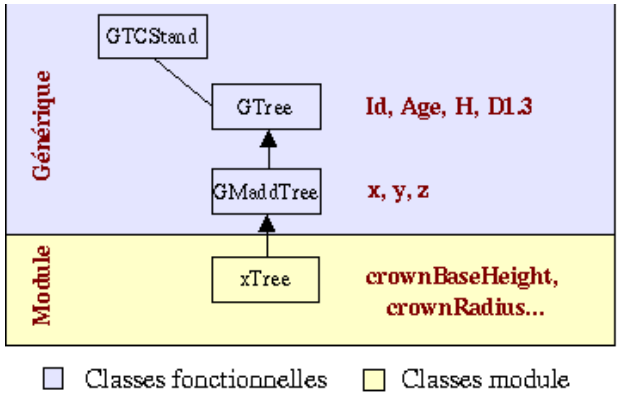


Fig. 6 − Peuplement et arbre (pour un modèle avec arbre individualisé)

`GTree` définit un identifiant pour l'arbre unique dans le peuplement, un âge, une hauteur (en mètres), un diamètre à 1.3 m. (en cm.), une propriété "marqué" utilisable pour marquer les arbres morts ou éclaircis sans les retirer du peuplement. L'arbre générique `GTree` connaît sa cellule de terrain (cette référence peut être nulle si le terrain n'est pas utilisé) et le peuplement qui le contient. `GTree` dispose de deux méthodes pour l'enregistrer ou le retirer d'un terrain (`GPlot`).

`GMaddTree` est un arbre générique spatialisé. Il reprend les propriétés de `GTree` dont il hérite. En plus, il implante l'interface `capsis.util.Spatialized` qui définit des coordonnées 3D : `x`, `y` et `z`.

`GMaidTree` est un arbre générique à effectif. Il hérite également de `GTree`, mais représente une classe d'arbres dont il porte l'effectif. Il implante l'interface `capsis.util.Numberable` qui spécifie l'emploi d'un effectif. `GMaidTree` possède également une propriété `numberOfDead` qu'on peut employer pour mémoriser le nombre de morts (ou d'éclaircis) dans la classe représentée par l'arbre depuis la dernière étape.

Dans les modules utilisant un arbre individualisé, il est d'usage d'hériter de `GMaddTree` pour les modèles spatialisés et de `GMaidTree` pour les modèles indépendants des distances. Ceci est une possibilité mais pas une règle. De même, il est possible d'utiliser une sous classe de `GMaddTree` implémentant également l'interface `Numberable` pour construire un arbre particulier : spatialisé et représentant une classe d'arbres.

c. Terrain − capsis.kernel.GPlot, GCell

Des propositions sont faites pour l'utilisation d'un objet terrain `GPlot` divisable en cellules de terrain `GCell`.

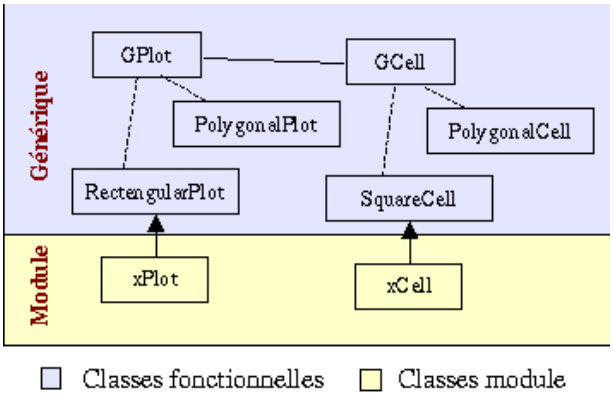


Fig. 7 − Terrain et cellules (option). Ex : un module utilisant un terrain rectangulaire découpé en cellules carrées

Le `GPlot` a pour propriétés une origine, la liste des cellules qu'il contient, une forme pour le dessiner, des méthodes pour gérer ses cellules, récupérer la référence au peuplement qu'il accompagne et ajouter un arbre dans la cellule qui doit le contenir.

La `GCell` spécifie une origine, une liste d'arbres qu'elle contient, une forme permettant de la dessiner, des méthodes pour enregistrer des arbres et déterminer son terrain.

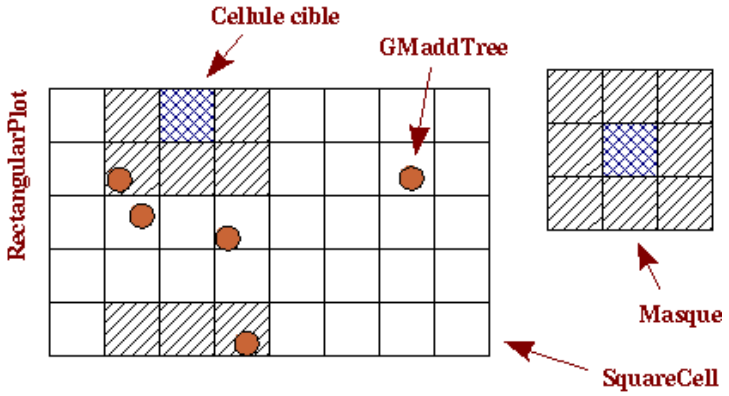


Fig. 8 − Application d'un masque de sélection sur une cellule cible d'un terrain rectangulaire

Des implémentations standard sont proposées :

`RectangularPlot` est un terrain rectangulaire composé de cellules carrées de même taille `SquareCell`. La subdivision en cellules permet d'associer des propriétés à la cellule de terrain (ex: régénération au niveau cellule).

La `SquareCell` supporte un mécanisme permettant d'appliquer un masque sur un terrain en visant une cellule de référence pour récupérer les cellules qui avoisinent la cellule visée et qui sont dans le masque. Ce mécanisme est assorti d'une option permettant de considérer le terrain comme étant torique pour éviter des effets de lisière indésirables en bordure du terrain.

`PolygonalPlot` et `PolygonalCell` sont des implémentations de `GPlot` et `GCell` qui décrivent des portions de terrain polygonales. Elles n'ont pas toutes les fonctionnalités de leurs homologues rectangulaires.

d. Modèle générique − capsis.kernel.GModel

`GModel` est la superclasse de toutes les classes "modèle" des modules Capsis4. Cette classe définit des propriétés communes à toutes les classes modèles :

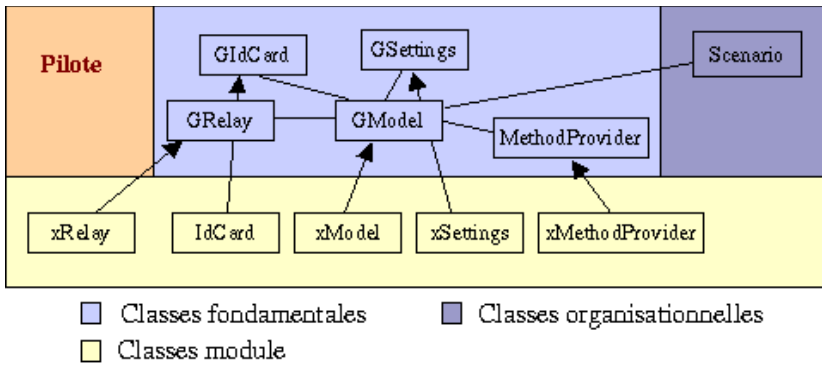


Fig. 9 − Un module Capsis4

- Possession d'une carte d'identité (`GIdCard`) redéfinie dans chaque module. Cette carte d'identité est utilisée dans le processus de chargement des modules. Elle contient notamment des renseignements sur le modèle et son auteur.

- Association d'un jeu de paramètre (`GSettings`) redéfini dans chaque module. Ces paramètres sont sérialisables pour pouvoir être sauvegardés avec un projet et restitués lors d'un chargement ultérieur. Généralement, la plupart de ces paramètres sont fixés pendant la phase d'initialisation du modèle.

- Référence du Projet associé (`Scenario`). Un modèle avec son jeu de paramètres est associé à un projet. Il définit les modalités d'évolutions pour toute la durée du projet.

- Référence au relais de pilotage (sous−classe de `GRelay`) du module dans le contexte de pilotage courant (ex : pilote *gui* − mode interactif).

- Référence au fournisseur de méthodes (`MethodProvider`) du module.

- Capacité à être sérialisée avec ses paramètres lors de l'enregistrement du projet associé.

- Appareillage pour implémenter les stratégies de gestion mémoire des projets par le moteur Capsis4 (`ProjectConfig`, géré par `capsis.kernel.Engine`).

e. Ensoleillement − capsis.kernel.GBeamSet, GBeam

`GBeam` et `GBeamSet` décrivent respectivement un rayon lumineux et un ciel utilisables pour l'ensoleillement d'une scène par lancer de rayons.

2.2.2 Paramétrage − capsis.kernel.CapsisSettings

La classe `CapsisSettings` établit les paramètres pour la session Capsis4 à partir des paramètres éventuels de la ligne de commande, de valeurs par défaut et de paramètres enregistrés sur disque lors de la dernière session de travail Capsis4.

Elle détermine la langue courante, récupère les paramètres passés par la machine virtuelle Java (JVM − *Java Virtual Machine*), repère l'emplacement du répertoire "racine" de Capsis4, ainsi que des répertoires nécessaire à son fonctionnement : `bin/`, `etc/`, `var/`, `tmp/`, `doc/`...

Cette classe assure également le chargement de fichiers de paramètres : `capsis.properties` (fichier paramètres de Capsis4) et `capsis.options`, paramètres déterminés lors de la dernière session de travail et venant modifier ceux de `capsis.properties`. Enfin, elle crée un fichier `capsis.log` qui recueillera toutes les informations écrites par Capsis4 ou ses modules pendant la session Capsis4.

2.2.3 Moteur Capsis4 − capsis.kernel.Engine

La classe `Engine` est la classe principale du noyau. Elle contient les fonctions de base liées à la gestion de la structure de donnée (niveau organisationnel) : création, ouverture, sauvegarde et fermeture des sessions et projets, création d'une nouvelle étape à partir d'un peuplement initial ou calculé, suppression d'étapes existantes. Elle assure également la détection et le chargement des modules.

Elle porte le numéro de version de Capsis4, récupérable par la méthode d'accès `getVersion ()`.

2.2.3.1 Design Pattern "Singleton"

La classe `Engine` a été conçue conformément au *Design Pattern "Singleton"* [Gamma95]. Cette disposition permet à toute classe de Capsis4, d'une extension ou d'un module de récupérer la référence à l'unique objet de type `Engine` instancié pendant une session de travail Capsis4 par la méthode de classe `Engine.getInstance ()`.

Cette méthode instancie `Engine` au premier appel (par la classe de démarrage) par invocation d'un constructeur privé. Les appels suivants se contentent de renvoyer la référence au même objet. Ainsi, on est assuré qu'une seule instance d'`Engine` est créée. Une conséquence est l'inutilité de la transmission de la référence de l'objet `Engine` à bon nombre d'objets créés et donc une simplification du code.

Le *pattern Singleton* est utilisé par d'autres classes, par exemple : `ExtensionManager` et `GroupManager` dans le package `capsis.kernel`, `Pilot` et `MainFrame` dans le package `capsis.gui` (pilote graphique).

2.2.3.2 Gestion des sessions

La méthode `processNewSession()` permet de créer une nouvelle session. Une session doit être créée avant la création ou l'ouverture d'un projet (une session *"untitled"* est éventuellement créée automatiquement). La création consiste en l'instanciation d'un objet de type `Session` et en l'attribution d'un nom.

`processOpenSession()` charge une session précédemment enregistrée sur disque. On lui passe un nom de fichier contenant une session. Le fichier session est un fichier descriptif contenant des informations permettant de recréer une session (avec `processNewSession()`), notamment une chaîne de caractère (`java.lang.String`) encodée contenant des informations enregistrées lors de la sauvegarde et les chemins d'accès aux projets enregistrés avec la session. Cette méthode appelle `processOpenScenario()` pour chacun des noms de fichier projet, puis ajoute ce projet à la session nouvellement créée.

`processCloseSession()` permet de supprimer la session en cours (il y a au plus une session en cours à un moment donné).

Enfin, `processSaveAsSession()` sauvegarde la session en cours dans un fichier portant le nom précisé. Ce fichier est composé d'informations sur la session, puis des noms des fichiers contenant les projets sauvegardés. Cette méthode considère que les projets viennent d'être sauvegardés sur disque (ce qui est de la responsabilité du pilote).

2.2.3.3 Gestion des projets

Rappel : Les projets sont des instances de la classe `Scenario` (cf. §2.2.1.1).

Pour créer un nouveau projet, on utilise la méthode `processNewScenario()` en lui passant un modèle instancié et paramétré, ainsi qu'un nom pour le projet. La méthode invoque le constructeur de la classe `Scenario`, puis ajoute le projet ainsi créé à la session courante.

`processOpenScenario()` permet de charger un projet enregistré sur disque. Comme pour la session, une chaîne de caractères encodée précède les données du projet, qui sont sous forme sérialisée (cf. §2.2.6). La méthode crée un *relais de pilotage* pour le modèle associé dans le mode courant (ex: pilote interactif *gui*). Le projet n'est pas associé à la session courante (responsabilité du pilote) mais juste retourné par la méthode.

La suppression d'un projet s'opère en invoquant la méthode `processCloseScenario`() qui supprime le projet donné de la session courante, puis invoque sa méthode `dispose()` pour libérer la mémoire explicitement.

La méthode `processSaveAsScenario()` permet de sauver un projet donné dans un fichier au nom précisé. Une chaîne de caractères encodée descriptive précède le projet sérialisé dans le fichier.

2.2.4 Gestion des étapes

La classe `Engine` permet d'ajouter et de supprimer des étapes dans un projet. Pour l'ajout, la méthode `processNewStep()` nécessite la référence au projet concerné, l'étape parente, le peuplement à associer à la nouvelle étape, un booléen concernant l'aspect visible de l'étape et une chaîne de caractères donnant la raison de l'ajout.

`processNewStep()` procède à l'instanciation du `Step`, puis attache la nouvelle étape à son étape parente en respectant les options de mémorisations courantes du projet. Ainsi, l'étape pourra être mémorisée durablement ou bien seulement provisoirement en tant qu'étape intermédiaire.

La méthode `processDeleteStep()` supprime l'étape passée en paramètre ainsi que toutes ses étapes parentes jusqu'à l'étape visible précédente ou la racine du projet.

2.2.4.1 Détection et chargement des modules

Le chargement d'un module s'opère en deux temps : détermination de son nom de classe principale, puis chargement proprement dit. Ces deux opérations sont effectuées grâce à des méthodes de la classe `Engine`.

Les modules sont des groupes de classes dans des packages. Ces derniers sont disposés dans le répertoire `bin/` (cf §2.5). La stratégie de gestion des modules sur disque prévoit par convention qu'un module terminé est compacté dans une archive java (`.jar`) qui porte le nom du module. Ce nom de module est également le nom de package de premier niveau du module.

Par exemple, le module `arthur` est constitué de deux package disposés dans `bin/` : `arthur.model` et `arthur.gui` (contenant les *classes relais* pour son usage par le pilote gui) et il est destiné à être compacté dans une archive dans `bin/` nommée `arthur.jar`.

Ces archives java servent de support au procédé de détection des modules. Ainsi, pour détecter les modules disponibles, Capsis4 recherche les fichier archives (par leur extension `.jar`) dans le répertoire `bin/` : méthode `getJars()` qui renvoie une collection des noms de modules.

`Engine` récupère ensuite les cartes d'identité de chaque modèle dont le nom est fixé par convention : `<nomModule>.model.IdCard` (ex : `arthur.model.IdCard`).

La méthode `createHshIdCard()` crée une *Map* ayant pour clé le nom de module spécifié dans la carte d'identité et pour valeur la carte d'identité elle−même. Cette *Map* permet par exemple au pilote graphique de créer et proposer à l'utilisateur une liste des noms de modèles disponibles (méthode `getModelNames()`).

Une fois le choix fait, la carte d'identité correspondant au nom dans la Map permet de charger le modèle dynamiquement à partir de la classe principale du modèle qu'elle contient.

Remarque : pendant le développement d'un module, les classes qui le composent ne sont pas encore compactées dans une archive java. Il est nécessaire dans ce cas de créer un fichier (vide) portant le nom permettant la détection du modèle par la classe `Engine` et de le disposer dans `bin/` (ex: `arthur.jar`). Les classes seront lues dans les packages en cours de mise au point (disposés également dans `bin/`).

2.2.5 Démarrage − capsis.util.Starter

L'application Capsis4 est lancée a l'aide d'un script de démarrage. Ce script porte le nom de l'application (*capsis.bat* sous Win32, *capsis.sh* sous Linux) et il accepte des paramètres qu'il transmet à Capsis4 au démarrage.

Pour permettre à Capsis4 de détecter les modules et les extensions compactés dans des archives java (fichiers `.jar`), le script utilise le paramètre *−cp* (*classpath*) de la machine virtuelle java et le fait suivre de la liste des fichiers *jar* présents dans `bin/`.

Quand une archive contenant un module ou une extension est ajoutée ou retirée de `bin/`, il convient de regénérer le script de démarrage de capsis en utilisant un autre script : *makescript* (*makescript.bat* / *makescript.sh*). Ce script lance l'application `MakeScript.class` qui se charge de la détection des archives dans le répertoire `bin/` et de la génération du script capsis (*.sh* ou *.bat* suivant le système d'exploitation).

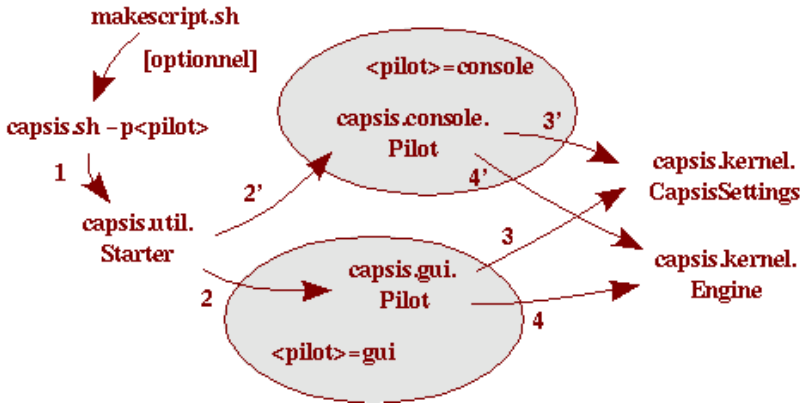


Fig. 10 − Démarrage de Capsis4. Starter détecte et lance le pilote demandé

Pour connaître la liste des options de démarrage de capsis, utilisez l'option *−h* (pour *help*). La liste des paramètres est présentée avec les valeurs par défaut et des exemples d'utilisation.

```
./capsis.sh -pgui -len
```

L'exemple ci−dessus lance Capsis4 en mode "pilote graphique" (fenêtres et menus) et en anglais.

`capsis.util.Starter` est la classe de démarrage de l'application. Elle contient la méthode `main()` est c'est elle qui est invoquée par le script *capsis*.

Son rôle est de déterminer quel pilote elle doit lancer. Elle interprète donc la ligne de commande à la recherche du paramètre *−p* (pour *pilote*). En cas d'absence du paramètre recherché, `Starter` tente de lancer le pilote par défaut : *gui*.

La classe principale d'un pilote a un nom qui suit les conventions suivantes : *capsis.<pilotName>.Pilot* (ex : *capsis.gui.Pilot*). Cette convention permet a `Starter` d'instancier la classe en ne connaissant que le nom du pilote (ex : *gui*).

Dans son constructeur, la classe principale du pilote a deux responsabilités. Elle commence par créer une instance de `CapsisSettings` (cf. §2.2.2) en lui passant les paramètres de la ligne de commande (dont le nom de pilote) , puis elle crée l'instance d'`Engine`, le moteur Capsis4 (cf. §2.2.3) qui servira durant toute la session Capsis4. Elle peut ensuite créer tous les objets nécessaires au pilotage de Capsis4 pour la session de travail qui vient de débuter (ex : une fenêtre principale avec des menus pour *gui*).

2.2.6 Sauvegarde par sérialisation

2.2.7 Les groupes − capsis.kernel.AbstractGroup, DynamicGroup, StaticGroup, GroupManager

Les groupes sont utilisés dans le cadre d'une simulation pour travailler sur un sous−ensemble d'arbres − pour les modèles avec arbres individualisés − ou de cellules − pour les modèles utilisant le terrain générique `Gplot` − (visualisation, extraction de données, interventions).

`AbstractGroup` décrit un groupe Capsis4. Elle permet d'appliquer un groupe constitué d'une combinaison de "filtres" à un objet "Filtrable". Classiquement, on travaille sur des groupes d'arbres ou de cellules de terrain et les "Filtrable" sont respectivement `Gplot` et `GStand`.

`DynamicGroup` enregistre les filtres paramétrés pour les rejouer sur le "Filtrable" considéré.

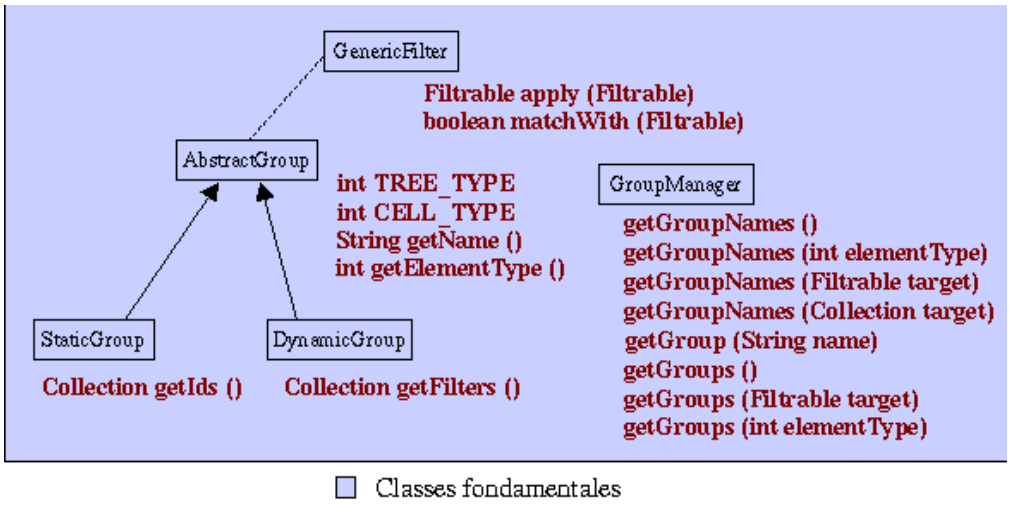


Fig. 11 − Les groupes Capsis4

`StaticGroup` enregistre seulement les identifiants des éléments sélectionnés pour les retrouver sur un autre "filtrable" (ex: le même peuplement l'année d'après).

`GroupManager` (implémentant le *design pattern "Singleton"* cf. §2.2.3.1) peut sauvegarder les groupes créés pendant une session de travail Capsis4 et les recharger au début de la session suivante.

Une fois un groupe créé (cf. § pour la construction), il est déclaré auprès du gestionnaire de groupes.

Tout composant Capsis4 peut demander au gestionnaire de groupes la liste des noms de groupes (liste complète ou limitée à un type d'élément ou de "Filtrable"). Une fois le groupe identifié, on en récupère une instance auprès du `GroupManager`.

Exemple d'utilisation :

```
GroupManager gm = GroupManager.getInstance ();
AbstractGroup group = gm.getGroup (name);
try {
        fi = group.apply (fi);    // a group is also a
Filter
} catch (Exception e) {...}
```

2.2.8 Le mécanisme d'extensions − capsis.kernel.ExtensionManager − compatibilité

La classe `ExtensionManager` (implémentant le *design pattern "Singleton"* cf. §2.2.3.1) est chargée de la gestion des extensions Capsis4. C'est elle qui lit dans le fichier paramètre `etc/capsis.extensions` l'inventaire des extensions disponibles. Cet inventaire est interrogeable à tout moment par son intermédiaire.

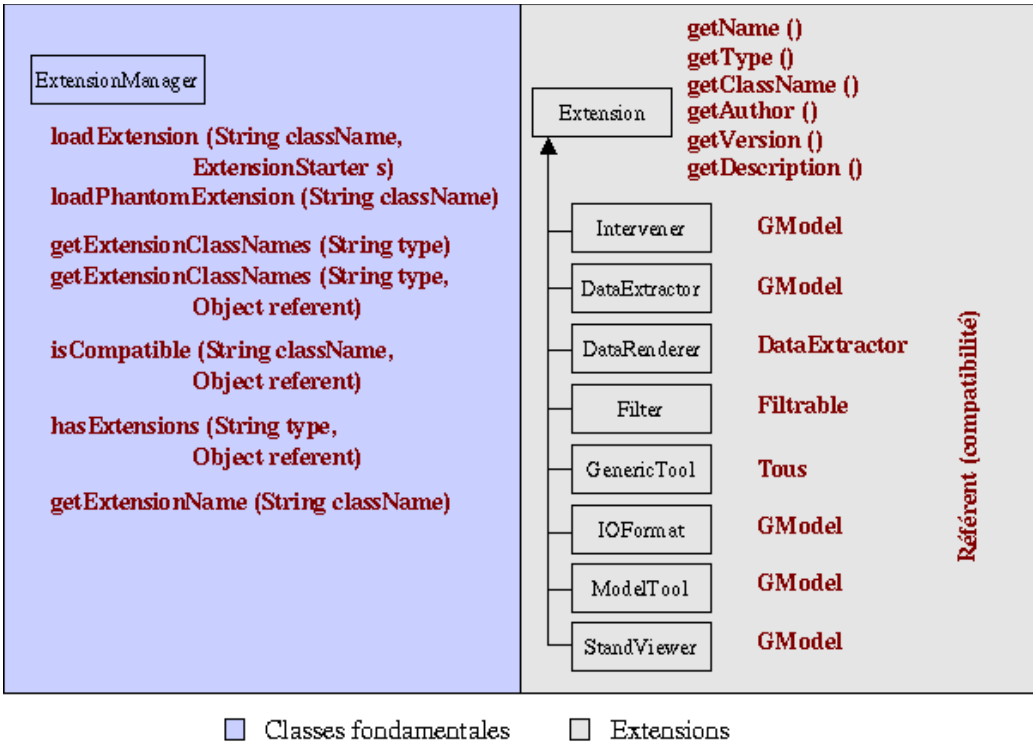


Fig. 12 − Les extensions

Le fichier d'inventaire des extensions contient un paragraphe par extension, composé des paramètres à mot−clé suivants :

- **extension** : nom de la classe principale de l'extension (package compris). Certaines extensions peuvent comporter plusieurs classes et tenir dans un package individuel. Ce nom de classe permet de charger l'extension dynamiquement.

- **type** : type de l'extension. Détermine son usage possible. Capsis4 peut proposer les extensions en fonction du contexte de travail. Les types connus sont : `GenericTool`, `StandViewer`, `DataExtractor`, `DataRenderer`, `Filter`, `Intervener`, `IOFormat`, `ModelTool`.

- **settings** : paramètre optionnel, dépendant du type d'extension. Si le paramètre apparaît, il est transmis à l'extension lors de son chargement. L'extension a la responsabilité du décodage de ce paramètre.

Exemple :

```
extension = capsis.extension.dataextractor.DETimeG
type = DataExtractor
settings = "defaultDataRenderer =
capsis.extension.datarenderer.drcurves.DRCurves"
```

Pour chaque extension connue, `ExtensionManager` charge également sa dernière configuration connue depuis le fichier `etc/extensions.settings`.

Plusieurs méthodes permettent de chercher des extensions dans la liste des extensions disponibles :

- `getExtensionClassNames(String type)` renvoie la liste des noms de classes des extensions du type considéré.

- `getExtensionClassNames(String type, Object referent)` renvoie la liste des noms de classes des extensions du type considéré et compatibles avec l'objet référent spécifié. Cette méthode s'appuie sur la méthode `isCompatible(String className, Object referent)`.

- `isCompatible(String className, Object referent)` : la compatibilité prise en compte est de niveau objet. Elle s'appuie sur l'invocation d'une méthode de classe `matchWith(Object)` décrite par la classe de l'extension. Il y a vérification que `matchWith ()` renvoit `true` si on l'invoque avec comme paramètre l'objet `referent`. Toutes les extensions sont concernées par ce mode de vérification de la compatibilité.

  Le type du référent diffère suivant le type d'extension concerné. Ainsi, *DataExtractor*, *Intervener*, *IOFormat*, *ModelTool* et *StandViewer* attendent un référent de type *GModel*. *DataRenderer* attend un référent *DataExtractor*, *Filter* attend un référent *Filtrable* et *GenericTool* est compatible avec tout le monde.

L'`ExtensionManager` permet ensuite de charger une extension dont on connaît le nom de classe principale par la méthode `loadExtension()`. Celle méthode prend le nom de classe de l'extension et un paramètre commun à toutes les extensions pour leur construction : une instance d'`ExtensionStarter`. Cet objet est préalablement préparé pour contenir les informations nécessaires à la construction de l'extension.

Une autre méthode, `loadPhantomExtension(String className)`, permet d'obtenir une référence utilisable seulement pour interroger les méthodes de récupération du nom de l'extension, son auteur, sa version, etc.

D'autres classes outils d'`ExtensionManager` permettent d'obtenir le nom d'une extension dans la langue courante à partir de son nom de classe ou encore les informations lues dans `capsis.extensions` pour une extension.

2.3 Le pilote générique

2.3.1 Introduction

Les pilotes de Capsis (sous classes de `capsis.util.GPilot`) permettent d'utiliser les fonctionnalités du noyau complétées par les méthodes fonctionnelles d'un module pour créer des simulation. Les pilotes sont définis pour un contexte d'utilisation.

Ainsi la première version de Capsis4 est−elle accompagnée d'un pilote interactif (appelé *gui* pour *Graphical User Interface*) permettant d'utiliser Capsis4 dans un environnement à base de fenêtres, de menus et de dialogues constitués de composants graphiques.

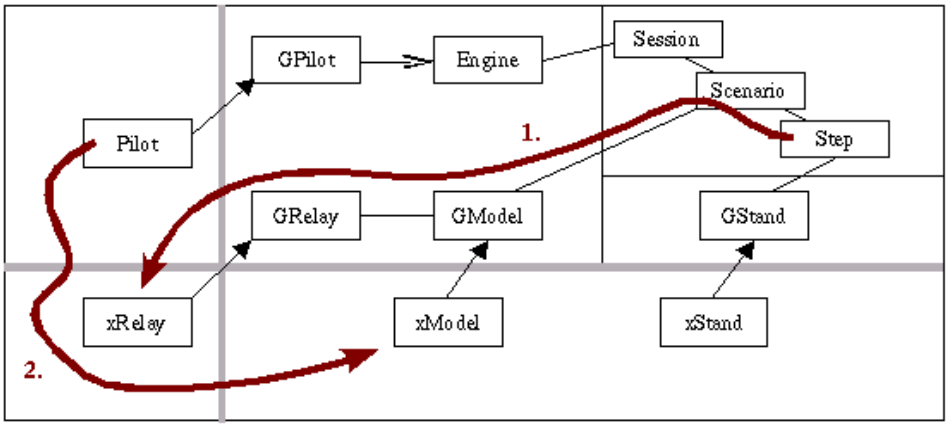
De même, il est prévu un pilote en mode console, c'est à dire non interactif, prenant ses instructions dans des fichiers de paramètres préparés à l'avance pour jouer des simulations longues ou répétitives.

Ces pilotes génériques sont complétés par du code accompagnant chaque module, permettant d'utiliser le module dans tel ou tel contexte et dénommés *relais de pilotage du module*. Ainsi, pour utiliser un module en mode console, il faut fournir un *relais* de pilote console pour le module.

Tous les pilotes génériques ont en commun la contractualisation de certaines action avec les *relais* des modules. Il partagent également une stratégie de construction et de reconstruction du *relais* de pilotage des modules dans le contexte d'utilisation courant lors de la réouverture d'un projet sauvegardé par sérialisation.

2.3.2 Les contrats du relais générique

Le relais générique (`capsis.util.GRelay`) impose des contrats au *relais* du module, c'est à dire l'implémentation de certaines méthodes au prototype imposé. C'est par ce moyen que le pilote courant peut déclencher des actions dans le module.



GRelay définit les commandes (contrats) exigibles par le pilote générique
A partir d'une étape de référence :
1. Détection du relais pour le modèle associé et le pilote courant
2. Le pilote transmet une commande au modèle par l'intermédiaire du relais

Fig. 13 − Pilote et relais

Le relais générique est décrit par une superclasse abstraite `GRelay` qui décrit elle même des méthodes particulières : abstraites ou bien proposant des implémentations par défaut, redéfinissables dans les sous−classes relais `xRelay` des modules.

Cette superclasse doit être impérativement sous classée par le *relais* du module pour fournir des implémentations à ces méthodes.

Plusieurs type d'actions sont concernés : des requêtes et des commandes. Les requêtes sont du type demande de paramètres dans la perspective d'effectuer une action. Les commandes sont le déclenchement d'actions dévolues au module, concenant principalement l'initialisation et l'évolution.

Généralement, le *relais* du module répond lui même aux *requêtes* du pilote et transmet les *commandes* à des méthodes des classes modèles du module. Une règle primordiale est que le *relais* du module n'implémente jamais de méthodes fonctionnelles, c'est à dire faisant partie du modèle (algorithmes, méthodes de calcul...). Toutes ces méthodes fonctionnelles sont implémentées dans les classes fonctionnelles du module (la première d'entres elles étant `xModel`) pour pouvoir être utilisées dans plusieurs contextes d'utilisation, c'est à dire par plusieurs pilotes différents.

Par convention, les méthodes de calcul des classes fonctionnelles correspondant aux méthodes du *relais* du module portent le même nom (ex: `module.gui.xRelay.processEvolution (protype imposé)` appelle `module.model.xModel.processEvotution (prototype_libre)`). Pour désigner les classes de même nom du *relais* du module et de sa classe fonctionnelle, on parle de méthodes *homologues*.

Les méthodes du *relais* de module ont donc un prototype imposé par la superclasse `GRelay`, mais leurs méthodes homologues de la classe fonctionnelle du module ont un prototype libre, connu du *relais* de module, mais inconnu de Capsis4 (noyau et pilote).

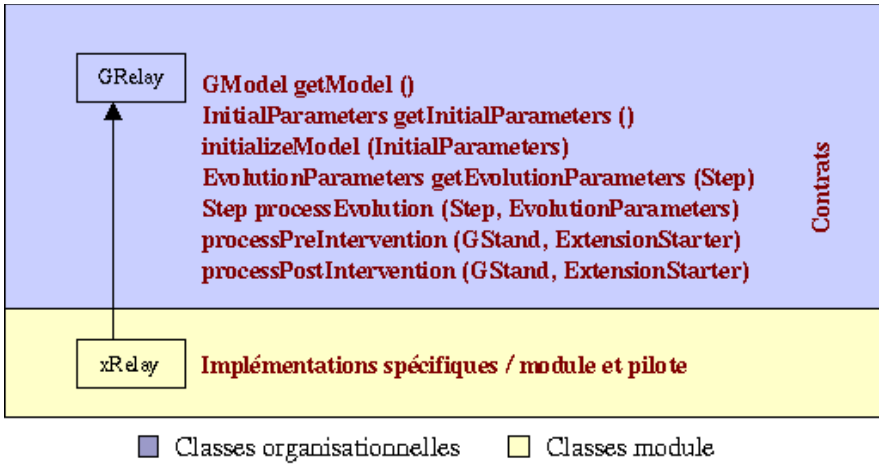


Fig. 14 − Les contrats d'un relais de pilotage

Le *relais* du module a toute latitude pour effectuer des actions de son ressort avant et après la redirection des commandes vers les classes fonctionnelles du module.

Les contrats sont les suivants :

- `getInitialParameters ()` : obtention des paramètres initiaux du modèle, renvoie un `InitialParameters`, c'est à dire un objet sur lequel on peut invoquer `getInitStand ()` pour obtenir un peuplement initial pour l'associer à l'étape racine de projet en cours de construction.

- `initializeModel (InitialParameters)` : Initialisation du modèle, permet au modèle d'effectuer des prétraitements qui ne doivent être effectués qu'une fois.

- `getEvolutionParameters (Step)` : récupération des paramètres conditionnant l'évolution (`EvolutionParameters`), qui sont spécifiques au modèle, ils seront repassés en paramètre à la méthode d'évolution.

- `processEvolution (EvolutionParameters)` : processus d'évolution, reçoit les paramètres récupérés par `getEvolutionParameters ()`, retourne la dernière étape (`capsis.kernel.Step`) créée par le processus d'évolution.

- `processPreIntervention (GStand, ExtensionStarter)` : appelée par Capsis4 avant une intervention pour permettre une éventuelle configuration par le module du mécanisme d'intervention choisi par l'utilisateur.

- `processPostIntervention (GStand, ExtensionStarter)` : appelée après une intervention. Le module peut choisir d'effectuer un post traitement après l'intervention, par exemple un ensoleillement de la scène après une éclaircie.

Les implémentations concrètes des pilotes génériques existants sont traitées au chapitre 3.

2.3.3 Construction du Pilote générique

Le pilote générique est construit par `capsis.util.Starter`, la classe qui décrit le démarrage de Capsis4 a partir de paramètres par défaut et des paramètres de la ligne de commande (cf. §2.2.5).

2.3.4 Construction des relais et reconstruction lors de l'ouverture d'un projet

Lors de la sauvegarde d'un projet par sérialisation, le *relais* du module n'est pas sauvegardé. En effet, le projet peut être réouvert dans un autre contexte d'utilisation, nécessitant un *relais* de module différent. Après l'ouverture d'un projet sérialisé, il faut donc reconstruire le *relais* de module correspondant au contexte d'utilisation courant (ex: mode *gui*).

C'est la classe `Engine` qui décrit la méthode qui se charge de cette construction : `createRelay (GModel model)`. Cette méthode est directement appelée lors du chargement d'un module par `Engine.loadModel (String className)` et lors de l'ouverture d'un projet sauvé sur disque par `Engine.processOpenScenario (String fileName)`. La méthode `createRelay ()` est privée (*private*). Elle n'est utilisable que dans ces deux cas.

Le nom de la classe principale du *relais* (qui peut en compter plusieurs, ex : des boîtes de dialogue) est construit suivant les conventions suivantes : `<nomModule>.<nomPilote>.xRelay` (avec `x` = préfixe du modèle). Ces conventions permettent de reconstruire le nom et d'instancier le relais dynamiquement.

2.4 Structure d'un module

2.4.1 Organisation

Capsis4 héberge des modèles de croissance, de production ou de dynamique forestière. Ils peuvent s'appliquer à des plantations ou à des forêts naturelles. Dans la suite, ces modèles sont dénommés *"modèles de croissance"*.

On considère des modèles de plusieurs types, notamment modèles de type *"Peuplement"* −− travaillant sur les propriétés globales d'un peuplement (densité, facteurs d'espacement...), *modèles "Arbre, Indépendant des Distances"* (*MAID*) −− prenant en compte des arbres "à effectif" représentant plusieurs individus ayant les mêmes propriétés, ou encore *modèles "Abre, Dépendant des Distances"* (*MADD*) −− où l'on individualise chaque arbre dont on connaît les coordonnées sur le terrain. Si Capsis4 est concu pour héberger tout type de modèles (a priori dendrométrique, mais ce n'est pas une limitation), ceux−ci sont jugés représentatifs et ont donné lieu à une attention particulière.

Chaque modèle est implémenté dans un module Capsis4.

Le module contient des classes *"fonctionnelles"* ou *"classes modèle"* contenant la connaissance du modélisateur et des "*classes relais"* proposant les outils nécessaires à l'exploitation du module dans un contexte d'utilisation donné (contexte de pilotage, ex: interactif − *gui*).

Par exemple, en mode de pilotage interactif, des boîtes de dialogue spécifiques sont nécessaires à l'initialisation et au fonctionnement du modèle (évolution du peuplement...). Elles sont décrites parmis les classes *relais*.

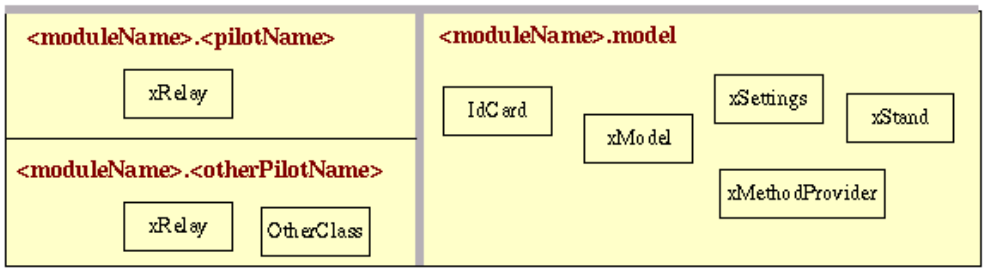


Fig. 15 − Les packages d'un module

Les classes des modules sont regroupées dans des "packages java". Par convention, le premier membre du nom de package est le nom du module (ex: `eucalypt`) ci dessous dénommé `<moduleName>`.

Les classes fonctionneles sont dans un package `<moduleName>.model` (ex: `eucalypt.model`) et les classes du *relais* graphique sont dans `<moduleName>.gui` (ex: `eucalypt.gui`). Dans ces packages, toutes les classes à l'exception de `<moduleName>.model.IdCard` voient leur nom débuter par un préfixe propre au modèle (ci après désigné par `<Prefix>`) : `<moduleName>.model.<Prefix>Stand` (par exemple: `eucalypt.model.EptusStand` − nom du module : `eucalypt`, préfixe : `Eptus`). Cette convention permet de reconnaître facilement les classes de chaque modèle.

La classe `IdCard` est la "*carte d'identité*" du modèle. Elle contient des informations relatives au modèle et à son auteur, ainsi que des informations nécessaires au chargement du module par le moteur de Capsis4 (nom de classe principale...). Le nom de cette classe est le seul qui n'est pas préfixé, pour répondre aux conventions du mécanisme de détection et de chargement de modules (cf §2.2.4.1).

2.4.2 Classes modèle − <moduleName>.model

Ces classes regroupent les descriptions des structures de données utilisées, les algorithmes et les méthodes du modèle, ainsi que des données de paramétrage.

Le modélisateur doit utiliser une implémentation de l'interface `capsis.kernel.GStand` (Generic Stand) pour représenter son peuplement. C'est le contrat minimal qu'un module doit honorer pour ce qui concerne sa structure de données. Il est possible d'utiliser directement une implémentation proposée par le noyau. Par exemple, les modèles gérant une liste d'arbres peuvent s'appuyer sur `capsis.kernel.GTCStand` (*Generic Tree Collection Stand*).

Généralement, le modélisateur décide de sous classer une classe de base (par exemple `GTCStand`) pour ajouter des propriétés particulières à son objet peuplement, ou bien redéfinir certaines méthodes pour lesquelles il souhaite un comportement particulier. Cette dernière possibilité permet par exemple de créer une liste d'arbres ou un terrain différent des implémentations par défaut en redéfinissant `createTreeCollection ()` ou `createPlot ()`.

Le modélisateur peut choisir de sous classer les classes génériques décrites dans le noyau Capsis4 (`GTCStand`, `GPlot`, `GTree`, `GCell`...) pour les enrichir, ce qui lui permet par la suite de bénéficier d'outils travaillant sur ces structures de données (visualisateurs, filtres, extracteurs de données...). D'un autre côté, il a la liberté de créer toutes les structures de données dont il a besoin. Dans ce cas, il ne dispose que des outils spécifiquement développés pour son modèle.

Le modélisateur doit implémenter une classe "modèle" nommée par convention `<moduleName>.model.<Prefix>Model` (ex: `eucalypt.model.EptusModel`) et héritant de la classe `capsis.kernel.GModel` (*Generic Model*). Cette superclasse définit certaines fonctions disponibles sur les classes modèles des modules. Capsis4 peut ainsi notamment obtenir :

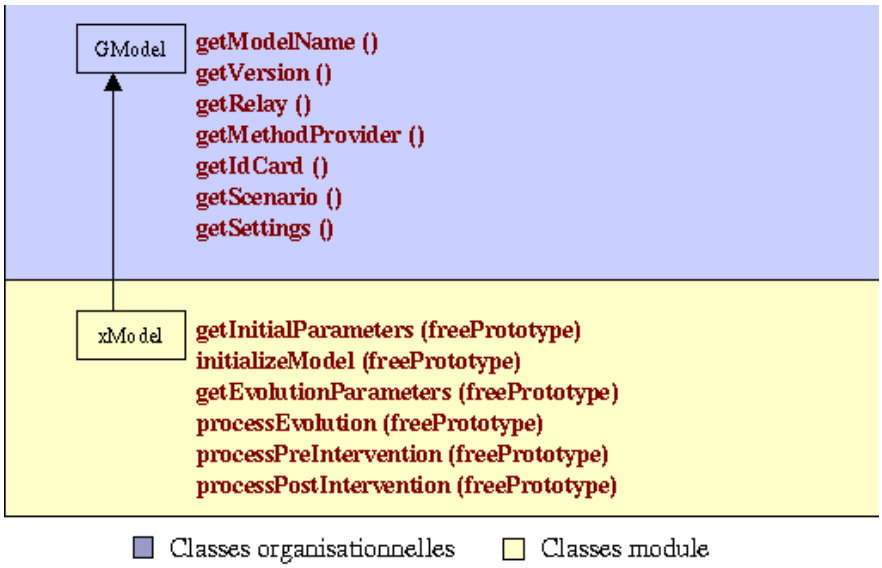


Fig. 16 – La classe modèle d'un module

- le nom du modèle (nom de package en minuscules) – `getModelName ()`,
- la version du modèle – `getVersion ()` (la version est mémorisée dans la carte d'identité du modèle. Cette méthode est un raccourcis),
- le relais de pilotage pour le contexte courant – `getRelay ()`,
- un fournisseur de méthodes compatible – `getMethodProvider ()`,
- la carte d'identité du modèle – `getIdCard ()`,
- le projet (notion interne : `Scenario`) auquel est associé le modèle – `getScenario ()`,
- les paramètres actuels du modèle – `getSettings ()`,
- diverses informations sur le mode de gestion mémoire du projet associé.

Il n'est pas fixé de contrats à ce niveau sur les propotypes des méthodes pour le chargement d'un peuplement initial, l'initialisation du modèle ou encore l'évolution. Cela permet de laisser libre la signature des méthodes qui implémentent ces processus dans la classe modèle. De tels contrats sont toutefois fixés dans les classes relais (cf. §2.4.3) pour permettre au pilote générique d'invoquer des méthodes de calcul du module.

Il est à noter que certaines conventions existent sur les noms des méthodes principales de la classe modèle. Le principe est que les méthodes doivent porter le même nom que leurs *homologues* de la classe *relais* correspondante :

- `getInitialParameters (...)` : en général résolu par le pilote,
- `initializeModel (...)`,
- `getEvolutionParameters (...)` : en général résolu par le pilote,
- `processEvolution (...)`,
- `processPreIntervention (...)`,
- `processPostIntervention (...)`.

Seuls les noms sont concernés par la convention, les paramètres de ces méthodes peuvent différer d'un module à l'autre dans les classes modèle (cf §2.3.2).

Ainsi, quand le pilote Capsis4 demande l'évolution au relais du module depuis une étape de référence d'un projet donné, ce relais peut décider de récupérer ou de calculer des paramètres pour l'évolution et il les transmet à sa classe modèle en utilisant une méthode de prototype connu et spécifique au module (ex: `processEvolution (int nbYears)` ou `processEvolution (int nbYears, double gMax)`).

Classiquement, la classe modèle propose une méthode de chargement depuis un inventaire sur disque ou de génération d'un peuplement initial − `loadInitStand ()`, une méthode d'initialisation du modèle − `initializeModel ()` − invoquée après la récupération des paramètres initiaux par le pilote et d'une méthode d'évolution − `processEvolution ()` − procédant au calcul des états successifs des peuplements et à leur ajout au projet courant les uns à la suite des autres grâce à la méthode `Engine.processNewStep ()`.

2.4.3 Classes relais − ex: <moduleName>.gui

Capsis4 est utilisé au travers d'un pilote (cf §2.3). Le pilote est un groupe de classes permettant de construire des simulations dans un contexte donné. Dans cette section, on évoque le pilote interactif en mode graphique nommé *gui*.

Le noyau Capsis4 est accompagné d'un pilote graphique permettant l'utilisation de l'application interactivement. Les fonctionnalités de base sont proposées dans les menus d'une fenêtre principale.

Chaque module est accompagné de *relais* (compléments de pilote) pour un ou plusieurs contextes d'utilisation. Ainsi, tel module dispose d'un relais graphique, tel autre possède un relais pour le mode *console* (non interactif, utilisant des fichiers paramètres pré−établis). Certains module auront plusieurs relais permettant de les utiliser dans plusieurs contextes.

Dans le contexte interactif, chaque module doit sous classer `capsis.util.GRelay` pour constituer sa "classe relais". Un contrat est défini par `GRelay` avec cette classe relais (prototypes de méthodes imposés) qui permettra à Capsis4 d'invoquer des méthodes du module par l'intermédiaire de son pilote.

Les méthodes de `GRelay` dont l'implémentation est attendue dans la classe relais sont présentées dans la section traitant le relais générique de Capsis4 (cf §2.3.2).

Ces méthodes sont discutées dans les sections qui suivent.

Note: certaines des méthodes ci−dessus peuvent être appelées dans des threads (gestion multi−tâche) par certains pilotes. C'est le cas du pilote graphique pour `GRelay.initializeModel ()`, `processEvolution ()` et `processPostIntervention ()`. Ces trois méthodes peuvent opérer des traitements longs et le pilote interactif peut ainsi libérer le thread de rafraichissement de l'interface graphique pour éviter à l'utilisateur un blocage apparent de cette dernière.

2.4.4 Paramétrage − <moduleName>.model.<Prefix>Settings

Les paramètres du modèle sont par convention regroupés dans une classe héritant de `capsis.kernel.GSettings`. La classe `GModel`, superclasse de la classe modèle de chaque module, propose une méthode `getSettings ()` qui renvoie ces paramètres.

Note : La méthode `GModel.getSettings ()` renvoie un objet de type `GSettings`. L'utilisation dans un module nécessite le transtypage (`cast`) du résultat dans le type de la sous classe de settings du module à chaque utilisation (`((ModSettings) getSettings ())`). Il est possible de définir dans `<Prefix>Model` une méthode renvoyant directement les settings dans le type attendu :

```
protected <Prefix>Settings get<Prefix>Settings () {
    return (<Prefix>Settings) getSettings ();
}
```

La classe *settings* de chaque module comporte une série de variables d'instances publique, accessible directement (sans accesseurs). Cet objet est en effet considéré comme une simple structure de données et on évite ainsi un bon nombre d'accesseurs inutiles.

Il est possible de prévoir une série de valeurs par défaut pour les variables d'instances, sous la forme d'autant de constantes portant le même nom mais avec la convention de nommage des constantes en java.

Exemple :

```
public static final String PLOT_NAME = "Plot 1";
...
public String plotName = PLOT_NAME;
```

Généralement, les paramètres du modèle sont déterminés en début de simulation, au moment de la construction du projet. On crée alor un "`GSettings`" (une sous classe) que l'on attribue au module (`GModel.setSettings (laClasse)`).

Ce mécanisme de centralisation des paramètres est important. En effet, lors des sauvegardes de projets, le modèle associé est sauvegardé par *sérialisation* et ces paramètres sont sauvegardés également. Ainsi, lors de la réouverture ultérieure du projet, les paramètres sont restitués dans le même état.

2.4.5 Peuplement initial

Pendant la création d'un projet, immédiatement après le chargement du module à lui associer, le pilote générique invoque la méthode `getInitialParameters ()` du relais du module. Cette méthode a pour obligation de renvoyer un objet qui implémente l'interface `InitialParameters` qui impose la méthode `GStand getInitStand ()`.

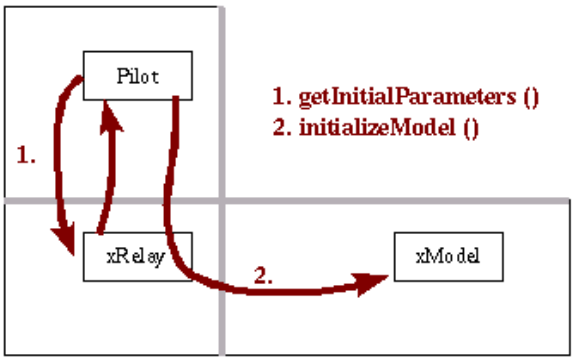


Fig. 17 − Acquisition d'un paramétrage et initialisation du modèle

De cette manière, le pilote générique s'assure que le processus d'initialisation du modèle, qui comporte par ailleurs une acquisition des paramètres du modèles (boîtes de dialogue en mode interactif), renverra un peuplement initial qu'il pourra accrocher sous la première étape − ou étape racine − du projet.

Le module peut choisir de charger un fichier inventaire au format libre (cf. §4.9) ou d'en générer un par une méthode qui lui est particulière (peuplement virtuel).

2.4.6 Initialisation du modèle

Après l'acquisition des paramètres du modèle, le pilote générique invoque la méthode `initializeModel ()` du relais du module.

Cette méthode peut soit ne rien faire si aucun pré−traitement n'est à faire pour ce module, soit déléguer l'initialisation à une *méthode homologue* au prototype libre de la classe `<moduleName>.model.<Prefix>Model`.

Note : En aucun cas l'initialisation du module ne doit être décrite dans une méthode *relais*. Il en est de même pour tous les processus fonctionnels (calculs, méthodes, algorithmes). En effet, suivant le contexte d'utilisation (pilote graphique, console...), ce ne sont pas les mêmes *relais* qui sont chargés pour jouer l'intermédiaire dans le pilotage du module.

La méthode du *relais* peut executer des actions qui lui sont attribuables avant et après l'appel de la méthode de la classe modèle. Il s'agit par exemple de toute action graphique dans le contexte graphique.

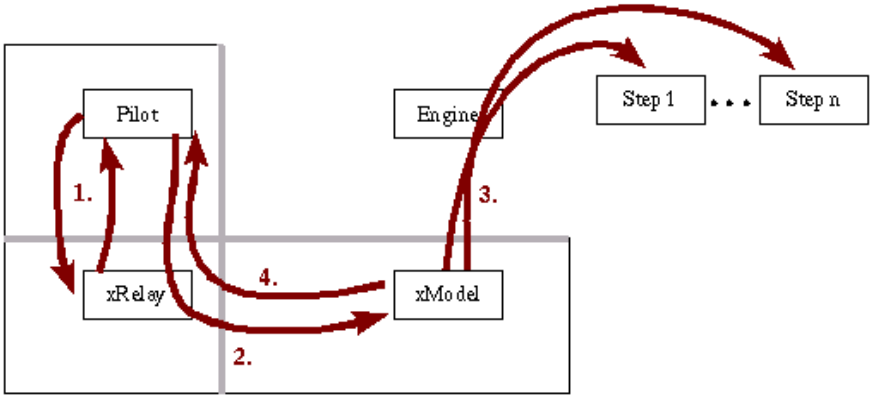
La classe modèle fournit l'implémentation concrète des pré−traitements envisagés. Par exemple, le module *Mountain* procède à la construction d'un ciel (ensemble de rayons) pour ensoleiller les scènes successivement calculées au cours du temps. Ce ciel est construit une seule fois au cours de ce pré−traitement.

2.4.7 Procédure d'évolution

Dans le contexte graphique par exemple, l'évolution est demandée par l'utilisateur sur une étape de référence à l'aide de la souris ou du clavier. Cette requête donne lieu à l'invocation de la méthode `getEvolutionParameters ()` du pilote du module.

Ce pilote peut alors ouvrir un dialogue générique ou spécifique pour acquérir les paramètres déterminant l'évolution, typiquement une date cible (le pas de temps est déterminé par le modèle).

Une fois ces paramètres retournés, le pilote générique invoque la méthode `processEvolution ()` du pilote du module. Cette méthode délègue également le travail à une *méthode homologue* de la classe modèle du module au prototype non imposé. C'est dans cette méthode `<moduleName>.model.<Prefix>Model.processEvolution ()` que la croissance et les processus éventuellement affiliés (mortalité, régénération...) sont décrits.



1. getEvolutionParameters ()
2. processEvolution ()
3. Construction des nouvelles étapes et chaînage derrière l'étape de référence
4. Retour de la dernière étape

Fig. 18 − Délégation de l'évolution au module à partir d'une étape de référence

La méthode d'évolution du modèle peut classiquement opérer une itération sur le nombre de pas de temps demandés et produire des peuplements successifs résultant à chaque fois de la croissance (au sens large) de celui de la précédente étape. Chaque peuplement calculé est raccroché au projet courant après l 'étape de référence par l'utilisation d'une méthode de la classe `capsis.kernel.Engine` : `processNewStep ()`. Cette dernière connaît et gère les options mémoires choisies pour le projet.

La dernière étape calculée est renvoyée jusqu'au pilote générique qui a initié l'action.

2.4.8 Actions avant et après intervention

Les interventions (éclaircie, élagage, fertilisation...) sont déléguées à des extensions de type "*Intervener*" (cf. §4.3). Les modélisateurs profitent des *Interveners* compatibles avec leur module et développent certains autres spécifiques le cas échéant.

La méthode `processPreIntervention ()` du *relais* de module est appelée juste avant l'invocation de l'*Intervener*. On lui passe le peuplement qui va supporter l'intervention, résultat d'un `getInterventionBase ()` sur le peuplement de l'étape de référence, et l'objet destiné à paramétrer l'extension (`ExtensionStarter`). Le module a donc la possibilité d'agir avant l'intervention, par exemple pour configurer un "*éclaircisseur*" en mode "*marquer les arbres*" (au lieu de les supprimer de la liste).

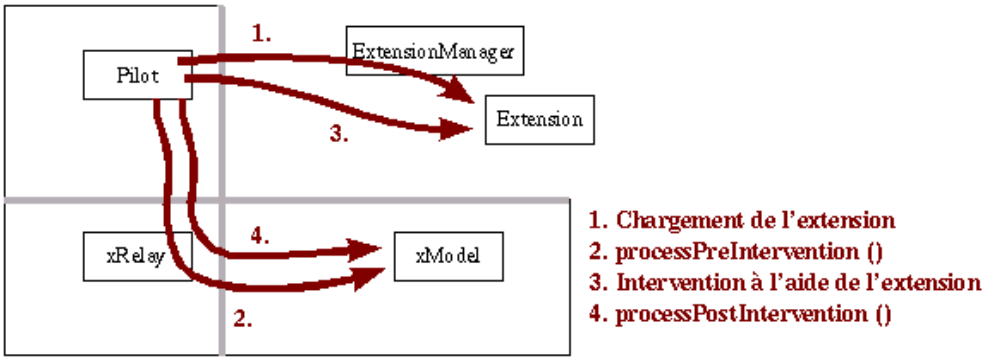


Fig. 19 − Intervention avec délégation au module de pré et post−traitements

De même, après l'intervention, une autre méthode du *relais* du module − `processPostIntervention ()` − est appelée pour donner la possibilité au module d'opérer un post−traitement. Il est important de noter que le traitement doit être implémenté dans une *méthode homologue* le la classe modèle du module. Il s'agit par exemple pour le module *Mountain* d'un ensoleillement du peuplement une fois éclairci.

2.4.9 Fournisseurs de méthodes − MethodProvider

2.4.9.1 Les fournisseurs de méthodes

Un mécanisme a été mis en place pour gérer les méthodes de calcul de grandeurs dendrométriques (ou autres) par les différents modules. Ces modules ayant des structures de données potentiellement très différentes, des méthodes calculant une même grandeur peuvent nécessiter des implémentations très différentes également.

Pour des raisons de stabilité par rapport à la *sérialisation* (cf. §2.2.6), ces méthodes ne doivent pas être implémentées dans la classe peuplement du module comme cela pourrait sembler logique. On préfère les regrouper dans des classes outils appelées par la suite *fournisseurs de méthodes*, sous classes de `MethodProvider` (package `capsis.util.methodprovider`).

La classe `GModel` possède une déclaration de méthode abstraite `createMethodProvider ()`. Cette méthode doit être implémentée dans la classe modèle de chaque module. Elle instancie une sous−classe de `MethodProvider` qui implémente des méthodes de calcul compatibles avec la structure de données du module.

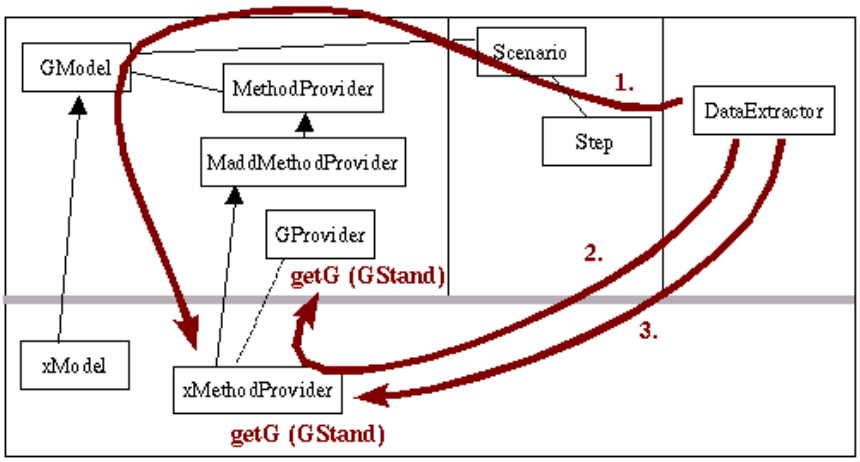
Il est possible d'instancier un fournisseur de méthodes déjà existant, comme `MaddMethodProvider` ou `MaidMethodProvider` qui sont proposés dans le package `capsis.util.methodprovider`. On peut également sous classer l'un de ces fournisseurs de méthodes pour bénéficier d'un modèle, duquel on peut redéfinir certaines méthodes et auquel on peut en ajouter de nouvelles.

Il est également possible d'hériter directement de `MethodProvider` et de définir de nouvelles méthodes de calcul sans s'appuyer sur un fournisseur de méthodes existant.

Par la suite, la méthode `getMethodProvider ()` de la classe modèle du module renvoie la référence du fournisseur de méthodes préparé par le modélisateur. Ce fournisseur de méthode est utilisable par tout composant souhaitant faire un calcul sur une structure de données généré par le modèle (ex : un peuplement, un arbre...). C'est notamment le cas des extracteurs de données.

2.4.9.2 Les interfaces de méthodes de calculs

Pour permettre de déterminer quelles méthodes de calcul implémente un fournisseur de méthodes, on s'appuie sur une convention reposant sur des interfaces java. Chacune de ces interfaces définit un accesseur et indique que la classe qui l'implémente possède cet accesseur.



1. L'extension récupère la référence du fournisseur de méthodes du module associé au projet de l'étape de référence
2. L'extension vérifie que la méthode getG (GStand) est implémentée par le fournisseur de méthodes : implémente-t-il GProvider ?
3. Si oui, l'extension invoque la méthode et obtient le résultat du calcul

Fig. 20 − Les fournisseurs de méthodes

Les noms de l'interface et de l'accesseur suivent la norme suivante :

Pour le calcul de `<Grandeur>` :

- interface : `<Grandeur>Provider.java`
- accesseur : `get<Grandeur> (paramètres au choix de l'auteur)`

Ainsi, par exemple, l'interface `GProvider` définit l'accesseur `getG (GStand s)`.

```
public interface GProvider {
    // G : Basal area
    public double getG (GStand s);
}
```

Ces interfaces sont toutes regroupées dans le package `capsis.util.methodprovider` pour être accessible en tout point du code capsis, des modules et des extensions.

Note : la gestion des interfaces dans `capsis.util.methodprovider` doit être centralisée. Toute nouvelle interface créée par un modélisateur doit être intégrée à la version de référence de la plate−forme Capsis4.

2.4.9.3 Application aux fournisseurs de méthodes

Les fournisseurs de méthodes des modules implémentent les interfaces pour les méthodes qu'elles définissent. Ce système permet aux outils de Capsis4 (par exemple les extracteurs de données) de juger de leur compatibilité avec tel ou tel module par une question du type : "le fournisseur de méthodes du module implémente−t−il l'interface `StuffProvider` me garantissant une méthode `getStuff ()` ?", ce qui ce traduit par exemple pour `G` (surface terrière) comme suit :

```
if (model.getMethodProvider () instanceof GProvider) {
    // I can invoke mp.getG ()
    GProvider mp = (GProvider) model.getMethodProvider ();
    double G = mp.getG (stand);
}
```

Les implications de cette conception sont les suivantes :

- Pas de liste de méthodes obligatoires à implémenter par les modules.
- Les modules peuvent bénéficier de méthodes par défaut qu'ils peuvent redéfinir.
- Ils peuvent créer de nouvelles méthodes de calcul qui n'existaient pas jusqu'alors. Il convient alors de disposer la nouvelle interface de méthode de calcul dans le package `capsis.util.methodprovider`.
- `GStand` est stable (peu de maintenance), ce qui facilite la relecture des projets sauvegardés par sérialisation.
- Le fournisseur de méthodes du module n'est pas sérialisé (inutile).

2.4.9.4 Autres utilisation des interfaces de méthodes de calcul

Les interfaces de méthodes de calcul sont utilisables en dehors des fournisseurs de méthodes. Il est possible de définir une interface spécifiant un accesseur pour une grandeur quelconque et de la faire implémenter par un objet de la structure de données du module.

Cette démarche permet potentiellement à un objet extérieur (par exemple une extension) de détecter que l'objet a cette capacité de calcul et de l'exploiter.

Par exemple, l'extension `Viewer3D` représente les arbres d'un peuplement dans une scène en trois dimensions.

```
public interface Viewer3DOptionalData {
    static public final int CONIC = 1;
    static public final int SPHERIC = 2;

    public int getCrownShape ();           // CONIC, SPHERIC
    public double getCrownBaseHeight ();   // in m.
    public double getCrownMaxDiameter ();  // in m.
    public Color getCrownColor ();
}
```

Avant de dessiner chaque arbre, il est vérifié s'il implémente l'interface `Viewer3DOptionalData` qui spécifie des méthodes concernant la couronne de l'arbre. Si oui, les méthodes sont utilisées sur l'arbre pour rechercher les valeurs retournées, sinon, des valeurs par défaut sont utilisées.

2.5 Architecture sur disque − Fichiers spéciaux

Capsis4 est installé dans un répertoire sur disque, ci−après nommé `install/`. Après installation, ce répertoire contient lui même une série de répertoires :

- `bin/` : c'est le répertoire qui contient les sources et les classes de Capsis4 et des modules installés.

  Ce répertoire est la base des packages de Capsis4. Il contient les scripts permettant de lancer Capsis4 à partir d'un terminal système (ou shell pour les systèmes Unix). Ces scripts lancent Capsis4 en spécifiant que la variable `CLASSPATH` contient `bin/`, ce qui est indispensable au démarrage de l'application.

  Les développeurs et modélisateurs qui veulent compiler des classes de Capsis4 ou de modules doivent positionner la variable `CLASSPATH` de leur système d'exploitation sur `install/bin/` pour permettre au compilateur `javac` de localiser les packages qu'il recherche. `bin/` contient également les fichiers archive `.jar` destinés à contenir les modules une fois stabilisés. Ces archives servent à la détection des modules (cf. §2.2.4.1) et doivent exister dans `bin/` (provisoirement vide) pour permettre leur utilisation dans Capsis4.

- `data/` : il est destiné à contenir des données pour l'utilisation des modules intégrés. Il n'est évidement pas obligatoire de disposer les données des modélisateurs dans ce répertoire. Cependant, le modélisateur peut créer un répertoire dans `data/` pour y disposer un jeu de données de test (pas trop volumineux) qui sera utilisable par un utilisateur installant la plate−forme Capsis4. Cette démarche est encouragée.

- `doc/` : ce répertoire accueille la documentation `javadoc` générable automatiquement par la lecture des sources java. Pour générer la documentation correspondant à la version installée de Capsis4, utiliser le script `jmk` ("Java MakeFile") dans `bin/`. Le fichier `bin/Makefile.jmk` joint comporte une commande `doc` qui lance `javadoc` avec pour cible le répertoire doc. Il est possible de compléter le fichier `Makefile.jmk` pour générer la documentation de modules non prévus.

- `etc/` : le répertoire de paramétrage contient plusieurs fichiers de paramètres nécessaires au bon fonctionnement de Capsis4.

    - `capsis.properties` contient les paramètres de l'application.

    - `capsis.options` modifie `capsis.properties` avec les paramètres choisis par l'utilisateur lors de la dernière session Capsis4.

    - `capsis.extensions` est le fichier de description externe des extensions. Il est lu par le gestionnaire d'extensions Capsis4.

    - `extensions.settings` est le fichier ou l'`ExtensionManager` enregistre la dernière configuration connue de chaque extension.

    - `capsis.groups` est une sauvegarde des groupes (paramétrés) connus lors de la dernière session Capsis4. Il est rechargé au démarrage par le gestionnaire de groupes.

- `ext/` : il contient les extensions utilisées par Capsis4, éventuellement sous forme de fichier archives `.jar`.

  Les scripts de lancement de Capsis4 précisent la liste des extensions employées et les recherchent dans ce répertoire. Ce mécanisme se substitue au mécanisme d'extension de la plate−forme Java qui prévoie que les classes d'extensions compactées en "*jarFile*" sont placées dans `installjdk/jre/lib/ext` où `installjdk` est le répertoire d'installation du java development kit (ex: `/usr/java/jdk1.3_01`).

  Cette mesure facilite l'installation de la plate−forme Capsis4 qui contient tout ce dont elle a besoin pour fonctionner dans l'arborescence définie à partir de son répertoire d'installation.

- `project/` : proposé pour sauvegarder les projets individuellement.
- `session/` : proposé pour sauvegarder les sessions.
- `tmp/` : utilisable par Capsis4 pour héberger des fichiers temporaires.
- `var/` : contient des fichiers de longueur variable (dont la taille augmente). `capsis.log`, le fichier dans lequel Capsis4 écrit des traces des événements qu'il rencontre, est dans ce répertoire.

3 Les pilotes

3.1 Le pilote graphique − capsis.gui

3.1.1 La classe principale − capsis.gui.Pilot

3.1.2 La fenêtre principale − capsis.gui.MainFrame

3.1.3 Les commandes du pilote gui − capsis.gui.command

3.1.4 Le gestionnaire de scénarios − ScenarioManager

3.1.5 Le gestionnaire de visualisateurs − ViewerMediator

3.1.6 Le gestionnaire de sorties graphiques − OutputMediator

3.1.7 Le sélecteur d'interventions − DIntervention

3.1.8 Le constructeur de groupes − DGroupDefiner

3.2 Le pilote console

4 Les extensions

4.1 Introduction

Capsis4 est une plate−forme évolutive. Elle est destinée à évoluer au fur et à mesure de l'intégration de modèles nouveaux par des modalisateurs ayant des problématiques sensiblement différentes.

Par ailleurs, il faut que le logiciel soit suffisament stable, pour que l'intégration d'un nouveau modèle ne nécessite pas des modifications telles dans le noyau, qu'une mise à niveau de tous les modules existants soit rendue nécessaire à chaque fois.

Dans ce contexte, il a été développé une architecture d'extensions (comparables aux *plugins* de certaines autres applications). Les extensions Capsis4 sont des outils qui peuvent être écrits ou modifiés parallèlement au développement des modules sans déstabiliser l'architecture du noyau, en se conformant à des spécifications de développement.

Ces outils prennent en charge plusieurs fonctionnalités de Capsis4, dont la principale est l'*Intervention*. En effet, si l'Evolution est à la charge de chaque module, l'Intervention est déportée dans des extensions dont certaines implémentations sont utilisables par plusieurs modules.

C'est le cas des mécanismes d'éclaircie qui opèrent à un niveau suffisament abstrait, par exemple sur un peuplement (`GStand`) comportant une liste d'arbres (`TreeCollection`) contenant une sous classe d'arbre générique (`GTree`) éventuellement spatialisée (`Spatialized`). Autant de propriétés détectables dynamiquement par l'extension quand on lui demande si elle peut être appliquée sur un objet donné.

4.2 Spécifications communes

4.2.1 Présentation

Les extensions sont classées par *type*, suivant l'usage auquel elles sont destinées par conception. Le gestionnaire d'extensions `ExtensionManager` permet de chercher une extension correspondant à des critères (type, compatibilité avec un objet référent sur lequel on détient une référence) et de la charger.

Les extensions sont construites avec un constructeur prenant un paramètre normalisé : une instance d'`ExtensionStarter`. Cet objet est préalablement renseigné avec les informations nécessaires au fonctionnement de l'extension.

Chaque extension doit être définie dans le fichier `etc/capsis.extensions` pour être connus du gestionnaire d'extension. Parmis les entrées, on spécifie le nom complet (package compris) de la classe principale de l'extension. Certaines extensions volumineuses (plusieurs classes) peuvent tenir dans des *packages* particuliers (ex: `capsis.extension.datarenderer.drcurves` contient quatre *data renderers* : `DRCurves`, `DRHistogram`, `DRScatterPlot` et `DRTable`).

4.2.2 Spécifications de l'interface Extension

Toutes les extensions Capsis4 implémentent l'interface `capsis.extension.Extension` qui exige les méthodes suivantes :

```
// This definition is inherited from interface Namable
public String getName ();

// Return extension type : STAND_VIEWER, DATA_EXTRACTOR...
public String getType ();

// Return getClass ().getName () : complete class name
public String getClassName ();

// Optional initialization processing. Called after
constructor.
public void activate ();

// Return version.
public String getVersion ();

// Return author name.
public String getAuthor ();

// Return short description.
public String getDescription ();
```

L'interface `Extension` définit également les types d'extension connus :

```
public static final String STAND_VIEWER = "StandViewer";
public static final String DATA_EXTRACTOR = "DataExtractor";
public static final String DATA_RENDERER = "DataRenderer";
public static final String GENERIC_TOOL = "GenericTool";
public static final String MODEL_TOOL = "ModelTool";
public static final String FILTER = "Filter";
public static final String INTERVENER = "Intervener";
public static final String IO_FORMAT = "IOFormat";
```

Les extensions Capsis4 héritent d'une superclasse implémentant l'interface `Extension` ou bien implémentent une interface qui en dérive.

4.2.3 Compatibilité avec un référent

Toutes les extensions doivent implémenter une méthode capable de dire si l'extension est compatible avec un objet référent donné :

```
static public boolean matchWith (Object referent);
```

La méthode est *statique* pour pouvoir être appelée sans instancier l'extension mais en chargeant seulement sa classe (méthode de classe), ce qui permet de faire des économies de temps dans un contexte prospectif (ex : obtenir une liste d'extensions compatibles avec un référent pour en choisir une). L'invocation de la méthode se fait dynamiquement (cf. méthode `isCompatible (className, referent)` dans `ExtensionManager`).

Le modifieur *static* interdit d'inscrire la méthode parmis les contrats de l'interface `Extension`. En effet, une interface java ne peut faire référence qu'à des méthodes d'instances.

En revanche, les superclasses des extensions (une par type d'extension) implémentent une méthode `matchWith (Object)` par défaut qui envoie un message d'erreur dans le *fichier Log* de Capsis4 en cas d'invocation. Cette erreur révèle un oubli de redéfinition de la méthode dans l'extension sur laquelle elle a été invoquée (c'est la méthode héritée de la superclasse qui a été utilisée).

Le corps de la méthode `matchWith (Object)` procède a une série de tests sur le référent pour vérifier qu'elle pourra fonctionner avec lui si elle est instanciée. En fonction des résultats de ces tests, la méthode renvoie true ou false.

4.3 Les interventions

4.3.1 Présentation

Les interventions sont des actions qui sont appliquées sur un peuplement à un moment donné et qui modifient ce peuplement. C'est le complément naturel de l'évolution, qui est la charge du module, pour la construction de scénarios sylvicoles dans Capsis4.

L'intervention type est l 'éclaircie. Elle consiste en la simulation de la coupe d'arbres suivant une stratégie d'éclaircie donnée. Différents mécanismes d'éclaircie existent dans Capsis4 (ex: un *éclaircisseur* par filtrage d'arbres en utilisant des extensions de type *Filtre* et un éclaircisseur de type Capsis2 développé pour le module *PNN*).

La technologie des extensions permet au modélisateur de choisir parmi les *éclaircisseurs* existants ou bien d'en développer un nouveau pour son module (ou pour une famille de modules compatibles).

Des modèles de fertilisation, d'élagage, d'attaque par des insectes, de simulation de feux de forêts... sont autant de mécanismes d'intervention potentiels dans Capsis4.

En mode interactif (pilote `capsis.gui`), les interventions sont choisies par la commande `capsis.gui.command.Intervention` et la boîte de dialogue `DIntervention`, qui produit la liste des noms des extensions de type Intervener qui sont compatibles avec le module associé au projet contenant l'étape de référence (celle qui porte le peuplement qui est la base de l'intervention).

Le peuplement qui supporte l'intervention n'est pas directement le peuplement sous l'étape de référence : on demande à ce dernier de fournir une base pour l'intervention en utilisant `GStand.getInterventionBase ()`. Il s'agit d'une forme de clone du peuplement origine. Dans le cas du `GTCStand`, c'est un peuplement de même type, comportant des clones des arbres de l'original, mais pointant sur l'objet terrain (s'il y a lieu, en fonction du module) de l'original. Cette concession permet d'accélérer la copie du peuplement à éclaircir.

Le modélisateur peut adapter cette stratégie si besoin est en redéfinissant `getInterventionBase ()` dans sa classe peuplement.

4.3.2 Spécifications

La spécification de l'interface des interveners est donnée par la classe abstraite `capsis.extension.Intervener` qui implémente `Extension`.

Cette classe rajoute les contrats suivants à `Extension` :

```
/**
 * Tells if construction was ok.
 * It could be wrong because of wrong parameters in console
 * mode or cancel in gui mode.
 */
abstract public boolean isReadyToApply ();

/**
 * Makes the actual intervention.
 */
abstract public Object apply () throws Exception;
```

Elle fournit également les implémentations par défaut de certaines méthodes d'Extension :

```
/**
 * From Extension interface.
 */
public String getType () {
    return Extension.INTERVENER;
}

/**
 * From Extension interface.
 */
public String getClassName () {
    return this.getClass ().getName ();
}

/**
 * From Extension interface.
 * May be redefined by subclasses. Called after constructor
 * at extension creation (ex : for view2D zoomAll ()).
 */
public void activate () {}
```

4.3.3 ExtensionStarter

- *model* : modèle associé au projet.
- *stand* : résultat de `getInterventionBase ()` sur le peuplement de l'étape de référence.
- *step* : étape de référence.

Exemple : Le paramètre `ExtensionStarter` pour les *interveners* est constitué comme suit (`wStep` est l'étape de référence à partir de laquelle on souhaite éclaircir) :

```
GStand fromStand = wStep.getStand ();

// This object is a partial copy of fromStand.
// Its step instance variable is null !
GStand newStand = (GStand) fromStand.getInterventionBase ();

// Load an extension of type : the one chosen by user
// we're gonna cut trees in newStand
final ExtensionStarter starter = new ExtensionStarter ();
starter.setModel (wStep.getScenario ().getModel ());
starter.setStand (newStand);
starter.setStep (wStep);      // passed for convenience
```

4.3.4 Situation

Packages `capsis.extension.intervener` et inférieurs.

4.3.5 Exemples

Dans les packages `capsis.extension.intervener` et inférieurs, voir `FilterThinner`, `IndividualThinner` et `C2Thinner`.

4.4 Les visualisateurs de peuplement

4.4.1 Présentation

Les visualisateurs de peuplements sont des outils utilisables seulement dans un contexte interactif en mode graphique. Il s'agit de représentation graphiques (cartes2D...) d'un peuplement dans un état donné.

Ces outils sont disponibles en utilisant le pilote graphique de capsis (`capsis.gui`) qui permet de synchroniser (`ViewerMediator`) des "*outils d'étape*" avec les boutons représentant les étapes dans le gestionnaire de scénarios (`ScenarioManager`).

4.4.2 Spécifications

Les spécifications d'interface des visualisateurs de peuplement sont données par la classe abstraite `capsis.extension.StandViewer`. Elle définit en plus d'`Extension` les méthodes suivantes :

```
/**
 * A stand viewer may be updated to synchronize itself with
a
 * given step button.
 */
public void update (StepButton sb) {
    super.update (sb);
}

/**
 * Dispose the extension.
 */
public void dispose () {
    super.dispose ();
}

/**
 * Print method. Called by a printJob. See
 * capsis.gui.command.PrintPreview
 * and capsis.gui.command.Print.
 */
public int print (Graphics g, PageFormat pf, int pi)
        throws PrinterException {...}
```

Parmis les *stand viewers*, certains sont destinés à être sous−classés pour servir de base à l'élaboration de visualisateurs spécifiques−modules. C'est le cas de `SVSimple`.

Ce visualisateur propose une représentation sous forme de carte vue de dessus des peuplements spatialisés. Il dessine les arbres à leur place par un cercle proportionnel au diamètre trouvé dans la définition de `GTree`. Si le peuplement est relié à un terrain (`GPlot`), `SVSimple` tente de dessiner les cellules de terrain.

Des options de représentation sont proposées dans une boîte de dialogue de paramétrage et on peut appliquer des seuils sur le diamètre des arbres à représenter. La zone utile du visualisateur supporte des fonctionnalités de zoom et de déplacement commandés par la souris, ainsi qu'une fonctionnalité de sélection dont le résultat est un panel d'introspection par arbre contenu dans le rectangle de sélection. Ces panels sont disposés par `SVSimple` dans une boîte de dialogue.

Le modélisateur qui souhaite spécialiser `SVSimple` procède par dérivation. Les méthodes de dessin de l'arbre et de la cellule de terrain (le cas échéant) sont réécrites pour rendre plus précisément les propriétés de l'arbre du modèle. Par exemple, si l'arbre possède un rayon de base du houppier, on peut dessiner des disques de rayon proportionnels et de coulleur fonction de la hauteur de l'arbre (cf. `capsis.extension.standviewer.SVMountain`). Il est possible de spécifier des options particulières à la nouvelle version de visualisateur, elles apparaissent dans un deuxième onglet de la boîte de paramétrage.

4.4.3 ExtensionStarter

- *model* : modèle associé au projet.
- *stepButton* : bouton représentant l'étape de référence dans le `ScenarioManager`.
- *mediator* : Objet gérant la synchronisation des visualisateurs de peuplement avec le gestionnaire de scénarios.

4.4.4 Situation

Packages `capsis.extension.standviewer` et inférieurs.

4.4.5 Exemples

Dans les packages `capsis.extension.standviewer` et inférieurs, voir `View2D`, `SVText`, `SVSimple`, `SVMountain`, `SVVentoux`.

4.5 Les extracteurs/metteurs en forme de données

Cette section traite de deux type d'extensions qui fonctionnent généralement ensemble. Les *extracteurs de données* constituent des séries de données par lecture d'un étape ou d'une suite d'étapes en mémoire. Les *metteurs en forme* présentent ces données sous forme de courbes, de tables etc...

Capsis4 propose une série d'extracteurs de données génériques pour extraire des grandeurs forestières classiquement utilisées dans la profession (surface terrière, diamètre ou hauteur dominante ou de l'arbre moyen...). Ces extracteurs s'appuient parfois sur les fournisseurs de méthodes des modèles (cf. §2.4.9).

Les extracteurs de données construisent des structures de données d'un format décrit parmis les formats regroupés dans le package `capsis.extension.dataextractor.format`. Ces structures de données sont visualisables par les *rendeurs de données* compatibles avec ces formats.

Les extracteurs et rendeurs de données sont gérés dans le contexte de pilotage interactif par un `OutputMediator` qui "écoute" les actions dans le gestionnaire de scénario. En fonction de ses actions, il crée, modifie ou supprime les extracteurs de données sélectionnés à un moment donné dans la fenêtre des sorties graphiques (`OutputBox`) et rafraichit les rendeurs de données connectés.

`OutputMediator` peut également proposer les rendeurs alternatif compatibles et gérer les modifications de rendeur (ex: courbe −> table).

On peut immaginer que les extracteurs et rendeurs de données soient utilisés dans d'autres domaines, par exemple dans un visualisateur de peuplement générique.

4.5.1 Les extracteurs de données − capsis.extension.DataExtractor

4.5.1.1 Présentation

Les extracteurs de données ont été conçus pour pouvoir caluler des grandeurs dendrométriques données à partir d'étapes de références de scénarios sylvicoles associés à des modèles différents.

Le but est de comparer sur le même graphe des données de différents scénarios à des fins de comparaisons (modèles différents, paramétrages différents, sylvicultures différentes...).

Dans cette optique, plusieurs extracteurs de données chacun synchronisé sur une étape de référence sont regroupés autour d'un `DataBlock` qui est relié à un rendeur de données courant. Le rendeur dessine les données de chaque extracteur du bloc.

Les extracteurs de données implémentent une des interfaces de `capsis.extension.dataextractor.format` qui définit leur type.

4.5.1.2 Configuration

Les extracteurs reliés à un `DataBlock` sont tous du même type (et même de la même classe) et sont paramétrés de manière cohérente. Ainsi, si l'on raisonne en hectare, tous les extracteurs du bloc sont configurés pour fonctionner par hectare. Certains extracteurs utilisent par ailleurs des éléments de configuration individuels. Ainsi, deux extracteurs pourront suivre l'évolution de la hauteur pour deux groupes d'arbres donnés.

Il y a par conséquent deux niveaux de configuration pour les extracteurs de données : configuration commune à tous les extracteurs reliés au même bloc (multi−configuration) et configuration individuelle.

Les extracteurs de données implémentent deux interface pour gérer ces aspects de configuration.

La multi−configuration est gérée par l'interface `capsis.util.MultiConfigurable` :

```
public interface MultiConfigurable {
    public String getMultiConfLabel ();
    public ConfigurationPanel getMultiConfPanel (Object
param);
    public void multiConfigure (ConfigurationPanel p);
    public void postConfiguration ();
}
```

La configuration indivudiuelle est gérée par l'interface `capsis.util.Configurable` :

```
public interface Configurable {
    public String getConfigurationLabel ();
    public ConfigurationPanel getConfigurationPanel (Object
param);
    public void configure (ConfigurationPanel panel);
    public void postConfiguration ();
}
```

4.5.1.3 Spécifications

Les spécifications des extracteurs de données sont décrites dans la classe abstraite `capsis.extension.DataExtractor`. En plus d'`Extension` qu'elle implémente et des deux interfaces de configuration, elle définit :

- Des méthodes liées à l'héritage de propriétés de configuration :

  ```
  abstract public void setConfigProperties ();
  public void addConfigProperty (String property) {
  public boolean hasConfigProperty (String property) {
  ```

  Ces méthodes sont liées à la configuration des extracteurs et permettent d'hériter de champs de configuration automatiquement dans les panneaux de configurations.

- Des méthodes liées au paramétrage courant de l'extracteur :

  ```
  protected void retrieveSettings () {
  public DESettings getSettings () {
  ```

Les settings sont une sous classes de GSettings, ils sont sérialisés par le gestionnaire d'extension à chaque modification et récupérés depuis le disque à chaque nouvelle session Capsis4.

- Des méthodes liées à l 'étape de référence :

```
public void setStep (Step stp) {
public Step getStep () {return step;}
```

- Des méthodes liées à la restriction à un groupe d'éléments et à l 'extraction proprement dite :

```
public Filtrable doFilter (Filtrable s) {
abstract public boolean doExtraction ();
```

Les sous classes implémentant concrètement les extracteurs redéfinissent `doExtraction ()` pour procéder à la lecture des données et construire le résultat en fonction des options courantes.

- Une méthode pour récupérer le metteur en forme par défaut :

```
public String getDefaultDataRendererClassName () {
```

- Des méthodes relative à l'aspect :

```
public Color getColor () {
public void setHidden (boolean val) {
public boolean isHidden () {
```

4.5.1.4 ExtensionStarter

- *step* : étape de référence.

4.5.1.5 Situation

Packages `capsis.extension.dataextractor` et inférieurs.

4.5.1.6 Exemples

Packages `capsis.extension.dataextractor` et inférieurs, voir un exemple d'extracteur de données Configurable : `DETimeH`. Un extracteur MultiConfigurable : `DETimeG`.

4.5.2 Les rendeurs de données − capsis.extension.DataRenderer

4.5.2.1 Présentation

Les rendeurs de données savent représenter d'une manière donnée un format de donnée connu. Les formats sont décrits dans le package `capsis.extension.dataextractor.format`. Les extracteurs de données implémentent l'interface du format qu'ils produisent (cf. §4.5.1). Ils sont donc directement représentables par les rendeurs de données.

4.5.2.2 Spécifications

Pour vérifier la compatibilité d'un *renderer* avec un extracteur, le gestionnaire d'extensions invoque sa méthode `matchWith ()` en lui passant l'extracteur.

C'est par ce dispositif qu'un *data renderer* peut vérifier sa compatibilité avec un extracteur. Par exemple, le *data renderer* `DRCurves` représente des données de type `DFCurves` (`DataFormat`). Sa méthode `matchWith ()` est la suivante :

```
static public boolean matchWith (Object target) {
    boolean b = false;
    if (target instanceof DataExtractor
            && target instanceof DFCurves) {b = true;}
    return b;
}
```

Les rendeurs de données sont configurables. DataRenderer implémente l'interface Configurable et peut donc renvoyer un panneau de configuration à partir duquel il peut ensuite se configurer.

Les rendeurs de données possèdent d'autre méthodes, dont des méthodes de mise à jour :

```
public void update () {
public void update (int w, int h) {
```

Des méthodes d'ordre général :

```
public boolean isFrozen () {
public DataBlock getDataBlock () {
public int getUserWidth () {
public int getUserHeight () {
```

Et une classe interne définissant un menu contextuel qui apparaît à la demande de l'utilisateur. Des méthode de détection des clics souris décrivent la gestion de ce menu :

```
class DataRendererPopup extends JPopupMenu implements
ActionListener {
    public void mousePressed (MouseEvent evt) {
    public void mouseReleased (MouseEvent evt) {
    ...
}
```

Ce menu a une partie fixe (configuration...) et une partie créée dynamiquement et proposant une option par *rendeur de données* compatible. Quand l'utilisateur choisit l'un de ces *rendeurs*, il remplace le *rendeur* courant. Ces comportements sont hérités par tous les rendeurs de données de leur superclasse commune `DataRenderer`.

Il est bien sûr possible pour le modélisateur de créer d'autres types de formats de données, avec les extracteurs et rendeurs de données correspondant. C'est par ce système que l'on prévoit par exemple de reproduire les dessins d'arbres et les tableaux de productions de Capsis2.

4.5.2.3 ExtensionStarter

- *dataBlock* : le block de données (groupe d'extracteurs de même types synchronisés sur des étapes de projets quelconques) à représenter.

4.5.2.4 Situation

Packages `capsis.extension.datarenderer` et inférieurs.

4.5.2.5 Exemples

Dans les packages `capsis.extension.datarenderer` et inférieurs, voir `DRCurves`, `DRHistogram`, `DRScatterPlot` et `DRTable`.

4.6 Les outils génériques

4.6.1 Présentation

Ce sont des extensions destinées à remplir des fonctions d'ordre général, qui ne dépendent d'aucun modèle particulier. Elles sont détectées au démarrage par le gestionnaire d'extensions comme les autres extensions. En contexte de pilotage *gui* (interactif), la fenêtre principale (`MainFrame`) crée autant d'entrées dans un menu "*Outils*".

On peut écrire plusieurs types d'outils, comme des éditeurs de texte, des moniteurs de surveillance des ressources utilisées par Capsis4 ou la JVM, ou des mécanismes facilitant le déverminage du code en montrant la structure de données par introspection par exemple.

Ces outils sont généralement développés pour fonctionner en contexte interactif et ont accès à la totalité de la structure de données de Capsis4.

Il est envisageable de créer ici des mécanismes de connexion régulant les interactions de Capsis4 avec d'autres logiciels du même domaine à travers le réseau pour travailler à une tâche commune

4.6.2 Spécifications

Les *generic tools* sont des `GenericTool` (`JInternalFrame`) ou des `GenericDialogTool` (`GDialog`) qui implémentent `Extension` et `Repositionable`.

Cette dernière extension permet à un `Positionner` courant de placer la fenêtre contenant l'outil en respectant une stratégie de placement courante.

```
/**
 * From Repositionable interface.
 */
public void reposition () {
```

4.6.3 ExtensionStarter

Les outils génériques ne nécessitent aucune donnée particulière dans leur `ExtensionStarter`.

4.6.4 Situation

Packages `capsis.extension.generictool` et inférieurs.

4.6.5 Exemples

Dans les packages `capsis.extension.generictool` et inférieurs, voir `MemoryView` et `TextBrowser`.

4.7 Les outils de modèles

4.7.1 Présentation

Les outils de modèles sont des outils spécifiques pouvant être employés pour procéder à une action particulière sur une étape d'un scénario associé à un modèle donné.

Il ne fonctionnent qu'en mode interactif (pilote *gui*).

Par exemple, `AmapSimConnector` établit une connexion réseau vers le logiciel *AMAPsim* pour lui demander de calculer des maquettes 3D pour les arbres d'un peuplement calculé dans Capsis (avec un modèle "Arbre Indépendant des Distances"). En retour, il est attendu des descriptions des houppiers de ces arbres calculés par *AMAPsim.*

4.7.2 Spécifications

Les outils de modèles héritent de la classe abstraite `capsis.extension.ModelTool`. En contexte de pilotage interactif, une boîte de dialogue est utilisée pour acquérir des paramètres et/ou afficher les résultats.

L'interface graphique est mise en place dans le constructeur de la boîte de dialogue. Les opérations suivantes sont déclenchées par l'utilisateur par action sur elle.

4.7.3 ExtensionStarter

- *model* : le modèle associé au projet de l'étape de référence.
- *step* : l'étape de référence.

4.7.4 Situation

Packages `capsis.extension.modeltool` et inférieurs.

4.7.5 Exemples

Packages `capsis.extension.modeltool` et inférieur, voir `AmapSimConnector` et `Viewer3D`.

4.8 Les filtres

4.8.1 Présentation

Les filtres sont des extensions s'appliquant sur des objets complexes que l'on peut "*réduire*" par filtrage. C'est par exemple le cas d'objets composés d'éléments sur lesquels on peut donc opérer une sélection. Les objets sur lesquels on peut opérer des filtrages doivent implémenter l'interface `Filtrable`.

Les filtres sont destinés à être utilisés dans un contexte interactif ou non.

`GStand` et `GPlot` héritent de l'interface `Filtrable`. C'est classiquement sur des peuplements et sur des terrains que l'on opère des filtrages dans Capsis4.

Le résultat d'un filtrage sur un `Filtrable` est un autre `Filtrable`, ce qui permet de combiner des filtres, par exemple pour réduire une sélection petit à petit.

Les filtres servent de support à la création de groupes (cf §2.2.7), mais ils peuvent avoir d'autres utilisations. Par exemple, l'intervener `FilterThinner` permet d'utiliser des filtres pour opérer des éclaircies dans un peuplement.

L'interface `Filtrable` spécifie notamment une méthode `getFiltrationBase ()` dont l'implémentation doit renvoyer une base de filtration, image de l'objet auquel on la demande et prête à subir l'opération de filtrage. Si l'original contient des éléments, la base de filtration doit contenir des clones de ces éléments.

Généralement, le `Filtrable` renvoie un clone plus ou moins complet de lui même (en tout cas de même type). Ainsi, par exemple, `GTCStand` renvoie une copie de lui même contenant des copies de ses arbres, mais avec une référence vers son propre terrain (si la référence de terrain, qui est optionnelle, est renseignée). Cette disposition permet d'économiser le clonage de l'objet terrain (qui peut contenir beaucoup de cellules) qui n'est pas déterminant pour l'opération de filtrage d'arbres.

Comme toutes les extensions, les filtres proposent le dispositif de compatibilité `matchWith (Object)` employé par le gestionnaire d'extension (cf. §4.5.2) pour déterminer si une extension est compatible avec un objet référent donné. Le filtre vérifie qu'il peut filtrer l'objet qu'on lui passe (un `Filtrable`).

Ainsi par exemple, FThreshold vérifie les qualités suivantes pour la cible qu'on lui propose : `Filtrable`, `GStand`, `TreeCollection`, non vide, contenant des `GTree`.

Il est à noter que rien n'empêche d'écrire des filtres pour des GStand qui ne sont pas des TreeCollection (modèles peuplement).

Les filtres peuvent servir à montrer une partie seulement d'un `Filtrable`, à construire un groupe à partir d'une sélection d'éléments d'un `Filtrable`, ou encore dans un contexte d'éclaircie, à supprimer une sélection d'éléménts d'un `Filtrable`.

4.8.2 Spécifications

Les Filtres sont des sous classes de `capsis.extension.Filter` qui implémente les interfaces `Extension` et `GenericFilter`.

```
public interface GenericFilter {
    public Filtrable apply (Filtrable f) throws Exception;
    public boolean matchWith (Filtrable filtrable);
}
```

Il est à noter que la méthode `matchWith (Filtrable)` décrite par `GenericFilter` n'est pas statique et qu'elle prend pour paramètre un `Filtrable`. Elle permet de vérifier auprès d'un filtre générique (`Filter` ou `AbstractGroup`) qu'il peut être appliqué sur un `Filtrable`.

Filter décrit comme prévu la méthode statique `matchWith (Object)` utilisée par le gestionnaire d'extension.

```
static public boolean matchWith (Object referent) {
```

Note : La méthode d'instance `matchWith (Filtrable)` s'appuie sur la méthode de classe `matchWith (Object)` (appel dynamique de l'implémentation de la méthode statique d'une sous classe) pour répondre.

La méthode `apply ()` de `GenericFilter` reçoit dans les sous classes le traitement de filtrage proprement dit, différent pour chaque filtre.

Certains filtres peuvent être configurables. Il implémentent la méthode `Configurable` vue plus haut.

Finalement, les filtres sont des objets délicats à mettre en oeuvre et il convient d'être très attentif à leur performance (méthode `apply ()`) car ils peuvent être appelés souvent (par exemple, un extracteur en évolution sur *n* étapes travaillant sur un groupe constitué de trois filtres appliquera *3n* filtres à chaque raffraichissement). La rapidité de réponse de `getFiltrationBase ()` sur les *filtrables* est également déterminante.

4.8.3 ExtensionStarter

- *filtrable* : l'objet à filtrer (peuplement ou terrain).
- *stand* : le peuplement correspondant.
- *plot* : l'objet terrain correspondant.

4.8.4 Situation

Dans les packages `capsis.extensions.filter` et inférieurs.

4.8.5 Exemples

Dans les packages `capsis.extensions.filter` et inférieurs, voir `FThreshold`, `FTreeGridSelector`, `FTreeMouseSelector`, `FCellGridSelector`, `FIndividualSelector` et `FQualitativeProperty`.

4.9 Les formats d'import/export

4.9.1 Présentation

Les formats d'import/export sont un moyen de créer un peuplement (objet don la classe implémente `GStand`) à partir d'un fichier inventaire et/ou de créer un fichier inventaire à partir d'un peuplement.

Lors de la phase de récupération des paramètres initiaux déléguée au pilote du module, qui doit renvoyer un `InitialParameters` (cf. §2.3.2), il est possible dans une méthode fonctionnelle du module de faire appel à un format d'import pour charger un fichier au format connu.

De même, pour une étape de référence, il est possible d'exporter des données vers un fichier en utilisant un format d'exportation (ex: par un menu contextuel en mode interactif − pilote *gui*).

4.9.2 Spécifications

Ces formats sont des extensions héritant de `capsis.util.RecordSet` et implémentant l'interface `capsis.extension.IOFormat`.

```
public interface IOFormat extends Extension {
    public GStand load (GModel model) throws Exception;
    public void save (String fileName) throws Exception;
    public boolean isImport ();
    public boolean isExport ();
}
```

`RecordSet` est une sous classe de `Vector` qui propose un appareillage pour automatiser les opération `RecordSet -> fichier` et `fichier -> RecordSet`. Il reste au développeur à écrire les méthodes `load ()` (`RecordSet -> GStand`) et `createRecordSet (<Prefix>Stand stand)` (`GStand -> RecordSet`).

`RecordSet` est un ensemble d'enregistrements qui héritent tous de `capsis.util.Record`. Cette classe fournit un constructeur qui prend une ligne (chaine de caractères) et tente de construire avec une instance d'elle même par *introspection* en appliquant un `StringTokenizer` sur la ligne pour en extraire les éléments.

`RecordSet` décrit certains formats d'enregistrements qui peuvent donc être utilisés dans tous les fichier import/export (hérités) :

- `KeyRecord` : de type `clé = valeur`. Evaluation dans la méthode `load ()` par le modélisateur et chargement des valeurs dans les variables d'instances souhaitées. Des accesseurs permettent de récupérer la valeur dans le type souhaité (ex: `double getDoubleValue ()`) ;

- `EmptyRecord` : une ligne blanche ;

- `CommentRecord` : une ligne de commentaire commençant par le caractère `'#'` ;

- `FreeRecord` : une ligne au format libre. Cette ligne n'ayant pas de schéma, elle ne peut pas être reconnue lors d'une importation. `FreeRecord` sert donc seulement à l'exportation.

Dans son format (classe dérivant de `RecordSet` par le biais de `StandRecordSet` ou de `ParameterRecordSet`), le modélisateur décrit ses formats d'enregistrements de la même manière : classes internes `public static` étendant `Record`, avec un constructeur par défaut faisant appel à celui de la superclasse et un constructeur prenant une chaîne de caractères (une ligne de fichier) et invoquant le constructeur de la superclasse de meme signature. Les variables d'instance de la classe interne sont publiques et dans l'ordre attendu dans la ligne du fichier.

4.9.2.1 Chargement en mémoire

C'est la méthode (implémentation par introspection) `createRecordSet (String fileName)` qui lit le fichier ligne à ligne et tente pour chacune d'entre elle de créer un `Record` dont il connaît la description. Les descriptions d'enregistrement sont des classes internes de `RecordSet` et de la sous classe considérée qui implémente le format. Le résultat du chargement de `RecordSet` est que le `Vector` contient un `Record` par ligne "*significative*" du fichier d'entrée. Au cas où une ligne du fichier d'entrée ne permet de construire aucun `Record` connu, une exception est lancée pour signifier une erreur de format de fichier.

Une fois le `RecordSet` créé, la méthode `load ()` (implémentée par le développeur) est utilisée pour construire un `GStand`. Pour cela, il convient d'itérer sur les enregistrements du *RecordSet* et d'en reconnaître le type avec l'opérateur `instanceof`. Pour chaque ligne, créer un arbre, une cellule de terrain ou toute autre structure de données prévue par le modélisateur.

Exemple : la méthode fonctionnelle de chargement de fichier d'inventaire de `mountain.model.MountModel` (utilisation directe, sans passer par le gestionnaire d'extensions) :

```
/**
 * Loads the inventory file given in parameter.
 * File format is described in MountInventory.
 */
public GStand loadInitStand (String fileName) throws
Exception {
    // this == model
    GStand initStand = new MountInventory (fileName).load
(this);
    return initStand;
}
```

4.9.2.2 Ecriture sur disque

L'écriture sur disque consiste en la création du `RecordSet` à partir du peuplement (par la méthode `createRecordSet (GStand)` à implémenter par le modélisateur), puis à l'écriture proprement dite par la méthode `save (String fileName)` fournie par `RecordSet` et utilisant la méthode `toString ()` de chaque `Record`.

Exemple :

```
(1) Dans DExport : GStand -> RecordSet

ExtensionManager em = ExtensionManager.getInstance ();
ExtensionStarter starter = new ExtensionStarter ();
starter.setStand (Current.getStepButton ().getStep
().getStand ());  // export mode
try {
    String cn = (String) extensionKey_classname.get
(formatClassName);
    ioFormat = (IOFormat) em.loadExtension (cn, starter);
    ...

(2) Dans ExportStep : RecordSet -> File

// Export format class name is retrieved from DExport dialog
IOFormat ioFormat = dlg.getIOFormat ();
String fileName = dlg.getFileName ();
// 4. Export now
try {
    ioFormat.save (fileName);
    ...
```

4.9.3 ExtensionStarter

- *stand* : le peuplement concerné par l'opération.

4.9.4 Situation

Dans les packages `capsis.extension.ioformat` et inférieurs.

4.9.5 Exemples

Dans les packages `capsis.extension.ioformat` et inférieurs, voir `MountInventory` et `EptusInventory`.

Bibliographie

Dreyfus96: Ph Dreyfus, F.−R. Bonnet − INRA Unité de Recherches Forestières Méditerranéennes, av. Vivaldi, 83000 Avignon, France, *CAPSIS − An interactive simulation and comparison tool for tree and stand growth, silvicultural treatments and timber assortment*, 1996, IUFRO WP S5.01−04 Biological Improvement of Wood Properties

Gosling96: J. Gosling, B. Joy, G. Steele, The Java Language Specification (Java Series), 1996

Cormen90: Th. Cormen, Ch Leiserson, R. Rivest , Introduction to algorithms, 1990

Gamma95: E. Gamma, R. Helm, R. Johnson, R. Vlissides, *Design Patterns : Elements of rRusable Object−Oriented Software*, 1995, Addison−Wesley Professional Computing