



# Projet de plateforme « sol virtuel »

# Conception informatique détaillée

# Version 2011

Auteurs : Nicolas Moitrier[1] et Nathalie Moitrier[2]

Relecteur : Cédric Nouguier[3]

Dernière modification : 24 juillet 2011

1 Chef de projet informatique de la plateforme « sol virtuel » <nicolas.moitrier@avignon.inra.fr>
2 Ingénieur en développement informatique de la plateforme « sol virtuel » <nathalie.moitrier@avignon.inra.fr>
3 Ingénieur en développement informatique de la plateforme « sol virtuel » < cedric.nouguier@avignon.inra.fr>

# Table des matières

# Index des illustrations

# Index des tables

# 1 Introduction

Ce document est un complément du document de conception générale informatique. L'ordre du « comment » est décrit de manière plus concrète et plus précise. Les spécifications de ce document doivent être considérées comme stables et être respectées. A l'image de la méthode agile dans laquelle ce projet s'inscrit, ce document doit nécessairement être revu et mis à jour régulièrement. Notamment à chaque fois que des problèmes techniques -rencontrés lors de la production informatique- l'imposent. Ce document peut donc devenir obsolète -au bout d'une ou deux années- puisqu'il s'agit de procéder par touches successives, de s'autoriser des corrections et des re-cadrages sur les réalisations.

Ce document exprime les aspects techniques détaillés. Le public visé est composé d'informaticiens en développement. Plus particulièrement, les « informaticiens de plateforme » et dans une moindre mesure les « informaticiens de modules ».

Outre l'introduction, ce document se décompose en six grandes parties. La première détaille les fonctions informatiques exigées dans les modules et le coupleur. La deuxième présente les règles de nommage des différents fichiers informatiques des modèles et des modules. Les troisième, quatrième et cinquième chapitres donnent des bonnes pratiques, respectivement pour la programmation du code source informatique, pour tester la « solidité » du code et pour les interfaces graphiques. La sixième partie décrit les fichiers et chemins d'accès des logiciels du projet. La dernière partie expose la manière dont sont diffusés les logiciels du projet.

# 2 Les fonctions informatiques des modèles et des modules

Un modèle « sol virtuel » est l'assemblage ordonné de modules qui représentent chacun un processus. Chaque module possède des codes numériques organisés en fonctions informatiques (voir document de conception générale chapitre 2.1.2, « Autonomie souhaitée »). Ces différentes fonctions informatiques sont gérées par un système de couplage dont la charge est d'ordonnancer les appels aux différentes fonctions des modules.

Pour ce faire, il organise :

- les flux de données en entrées/sorties des modules ;
- l'écoulement du temps ;
- un maillage du sol compatible avec tous les modules.

En dehors de l'ordonnancement des codes de calculs des modules, le coupleur procède à l'initialisation de tous les modules -donc avant les codes calculs- et à leur terminaison -donc à la fin de la simulation. Durant la simulation, le coupleur réalise à chaque pas de temps la sauvegarde des sorties.

Concernant la gestion de l'espace et donc du maillage du sol, le choix initial est d'en laisser -pour l'instant- la gestion aux modules. Le coupleur fait toutefois la distinction entre les processus/modules de surface et ceux distribués dans le maillage.

## *2.1 Les fonctions du coupleur*

Le système couplage consiste donc en un programme principal (appelé « main » en C/C++) qui procède à trois grandes étapes : initialisation, calculs et terminaison. Ces étapes concernent chacun des modules. C'est à dire que pour chaque étape, tous les modules du modèle doivent être sollicités et dans le bon ordre. Voir section 4.5 « Les algorithmes de modules et de modèles » dans le document de conception générale.

### 2.1.1 Initialisation

L'étape d'initialisation concerne à la fois le niveau modèle, avec notamment la définition des couches et du maillage du sol, et le niveau module, avec les valeurs initiales et les paramètres de chacun d'eux. Cette première étape se termine avec la sauvegarde des valeurs de sorties des modules au temps initial. Elle est suivie par l'étape des calculs.

### 2.1.2 Calculs

L'étape des calculs consiste à découper le temps de simulation en tranches de temps. Ces delta de temps sont variables et non prévisibles. Le coupleur doit décider d'une tranche de temps et solliciter successivement tous les codes de calculs des modules, en respectant l'ordre d'ordonnancement du modèle. L'appel au code de calcul d'un module consiste à :

1. récupérer les entrées nécessaires au code de calcul du module ;
2. solliciter le code de calcul du module ;
3. en retour, récupérer le statut accepté/refusé du code de calcul du module.

Si **tous** les modules acceptent la tranche de temps donnée, le coupleur passe alors à la validation (voir section suivante) et à la sauvegarde des sorties. Sinon, il doit prendre une tranche de temps plus courte et re-solliciter tous les modules à partir du premier de l'ordonnancement. L'opération et répétée jusqu'au succès de tous les modèles.

### 2.1.3 Validation

La validation signifie que tous les modules ont accepté une tranche de temps donnée. Le coupleur peut alors procéder à la sauvegarde des sorties (voir ci-dessous), faire « avancer » le temps courant -de la durée de la tranche- et lancer les calculs pour une tranche de temps suivante (voir ci-dessus).

### 2.1.4 Sauvegarde

La sauvegarde consiste à écrire les valeurs de sorties validées des modules, au temps courant. Une interpolation des valeurs peut-être nécessaire.

### 2.1.5 Terminaison

La terminaison du coupleur consiste à fermer tous les fichiers ouverts, libérer la mémoire allouée et à demander à chaque module de faire de même. Le coupleur se termine en avertissant l'utilisateur de la bonne exécution globale -ou non- du modèle.

## *2.2 Les fonctions des modules*

Pour interagir avec le coupleur, les modules doivent remplir des engagements fonctionnels. Cet ensemble de fonctions dans les modules sera sollicité par le coupleur lors des différentes étapes de l'exécution d'un modèle (voir ci-dessus). Les fonctions des modules font écho à celles du coupleur. Elles sont détaillées dans les sections suivantes.

### 2.2.1 Initialisation

L'initialisation du module contient tout ce dont le module a besoin pour que sa fonction de calcul puisse s'exécuter correctement. C'est à dire les paramètres et valeurs d'états -qui sont généralement les sorties- du module. Ces valeurs peuvent venir :

- directement, si elles sont en entrée de la fonction
- indirectement
  - par la voie d'un nom de fichier qui contient les valeurs utiles
  - êtres calculées à partir des autres données.

Lorsque cette fonction se termine, l'état du module doit être cohérent. C'est à dire que ses sorties sont valides et qu'il sera capable d'exécuter sa fonction de calcul sans erreurs, notamment

celles dues à une initialisation incomplète.

### 2.2.2 Calculs

La fonction de calcul est le « cœur scientifique » du module. Elle sera invoquée à chaque tranche de temps décidée par le coupleur. Pour des raisons qui peuvent échapper au coupleur, la fonction est libre de décider si elle accepte la tranche de temps demandée ou pas.

En plus de la tranche de temps demandée par le coupleur, la fonction d'exécution reçoit en entrée les valeurs d'entrées -sorties des modules amont- qui lui sont nécessaires. Il est souhaitable qu'elle procède à leur vérification (domaine de validité) avant de faire ses calculs puis de mettre à jour les sorties calculées. Si la fonction de calcul utilise ses propres sorties pour calculer les nouvelles, alors elle devra **toujours** utiliser celles des sorties validées (voir section suivante).

### 2.2.3 Validation

La fonction de validation est appelée par le coupleur si tous les modules ont accepté la tranche de temps courante. Elle consiste simplement à considérer les sorties calculées comme valides. En conséquence, la fonction de calcul utilisera lors de son prochain appel les dernières sorties calculées à la place des précédentes.

### 2.2.4 Sauvegarde

La sauvegarde consiste à écrire les valeurs de sorties validées du module.

### 2.2.5 Terminaison

La terminaison consiste à libérer les éventuelles allocations de mémoire et les fichiers ouverts à l'initialisation.

### 2.2.6 Sorties

Pour que le coupleur puisse récupérer les sorties des modules qu'il désire à tout moment, il doit exister un accès -en lecture seule- pour chacune des sorties. Avec les langages C et C++, cela est possible avec des fonctions qui renvoient les valeurs des variables de sorties. Elles sont communément appelées les « getter ». Les fonctions de sorties donnent toujours les dernières valeurs calculées -par la fonction de calcul- même si elles n'ont pas encore étés validées -par la fonction de validation. En Fortran, depuis la norme 2003, il est possible de déclarer des variables de module en lecture seule, avec le mot-clef « protected ».

# 3 Les fichiers des modèles et des modules

## 3.1 Les fichiers des modèles

En plus des fichiers liés à ses modules, un modèle « sol virtuel » utilise des ensembles de paramètres, stockés dans des fichiers. Ces ensembles sont :

1. la définition temporelle et spatiale ;
2. les paramètres globaux.

Ces ensembles de paramètres sont partagés entre tous les modules du modèle.

### 3.1.1 La définition temporelle et spatiale

L'ensemble des données temporelles est défini à travers les champs suivants :

- la date de référence ;
- les dates de début et de fin de simulation ;
- les pas de temps minimum et maximum ;
- la fréquence de sauvegarde des sorties des modules.

Plusieurs ensembles de données temporelles peuvent être définis. Ces jeux de valeurs sont stockés dans un fichier « timestamps.xml » (voir en annexes) situé dans le répertoire racine des données utilisateur « sol virtuel ».

L'ensemble des données spatiales est défini à travers les champs suivants, pour chaque couche homogène de sol :

- la profondeur ;
- le nombre et la répartition des points de grille ;
- les espaces minimum et maximum entre deux points de grille.

Plusieurs ensembles de données spatiales peuvent être définis. Ces jeux de valeurs sont stockés dans un fichier « layers.xml » (voir en annexes) situé dans le répertoire racine des données utilisateur « sol virtuel ».

### 3.1.2 Les paramètres globaux

L'ensemble des paramètres globaux n'est actuellement pas défini. Lorsqu'il le sera, les ensembles de paramètres globaux seront stockés dans un fichier « parameters.xml » (voir en annexes) situé dans le répertoire racine des données utilisateur « sol virtuel ».

## 3.2 Les fichiers des modules

Chaque module « sol virtuel » possède plusieurs ensembles de données, stockés sous forme d'un ou plusieurs fichiers. Ces ensembles sont :

3. les codes sources informatiques ;
4. les fichiers de données de l'utilisateur ;
5. des jeux de valeurs de paramètres.

## 3.2.1 Les codes sources

L'ensemble de données « codes sources » des modules sont des fichiers situés dans le répertoire « sources » du répertoire du module. La nomenclatures des fichiers dépend du langage informatique utilisé. En option des fichiers obligatoires, décrits dans les sous-chapitres suivants, l'utilisateur peut ajouter des fichiers de code source contenant des fonctions complémentaires et propre au module.

### 3.2.1.1 Les fichiers codes sources obligatoires en C++

Le code source d'un module écrit en C++ (norme C++98 minimum) doit être réparti -comme c'est l'usage- en deux fichiers :

- un fichier contenant uniquement des déclarations
- un fichier de code d'exécution proprement dit.

En C++, le module est programmé en langage objet. Le fichier des déclarations contient donc la déclaration d'un classe. Le nom de cette classe est celui du module. L'usage veut que la première lettre du nom soit en majuscule, les autres en minuscules. Les espaces sont remplacés par le caractère « souligné » ('_'). Cette classe contient les déclarations suivantes :

- Les fonctions publiques du module comme citées dans la section 2.2 Les fonctions des modules, page 6. Ces fonctions sont librement accessibles par un code source externe à la classe, donc par le coupleur.
- Les fonctions privées du module. Accessibles uniquement par les autres fonctions du module, elles permettent de sectionner le code source -notamment de calcul- afin de le rendre plus lisible. Conseil : le découpage en sous-fonctions doit suivre une logique -à priori- scientifique.
- Les paramètres (variables constantes à l'échelle d'une simulation), les variables d'états et les variables de sorties (calculées et validées). Toutes ces données sont en accès privé.

Le fichier de code d'exécution contient le corps (c'est à dire les instructions proprement dites) des fonctions déclarées dans le fichier des déclarations.

### 3.2.1.2 Les fichiers codes sources obligatoires en Fortran

Le code source d'un module écrit en Fortran (norme F2003 minimum) doit être réparti en deux fichiers. Le Fortran n'est pas un langage à objets. Il permet toutefois de déclarer des « modules » -au sens Fortran du terme- ce qui permet d'encapsuler -au même sens que le paradigme d'encapsulation des langages à objets- des sous-routines et fonctions Fortran dans un « espace de nom » avec un contrôle de l'accessibilité de type « privé/public ». En conséquence, il conviendra d'appliquer les mêmes règles qu'avec le C++. A la différence qu'il n'y a pas de fichier contenant

uniquement des déclarations en Fortran. Il est toutefois indispensable d'avoir un fichier des déclarations (publiques) au format C. Ceci afin d'assurer l'interopérabilité avec le langage C++, utilisé par le coupleur.

### 3.2.2 Les fichiers de données de l'utilisateur

L'utilisateur peut ajouter aux fichiers des codes sources, un ensemble fichiers de données propre au module. Ces fichiers sont situés dans le répertoire « user_datafiles » du répertoire du module. L'utilisateur a une complète liberté sur le contenu de ce répertoire.

### 3.2.3 Les jeux de valeurs de paramètres

Un module défini un ensemble de paramètres nécessaires à son fonctionnement. L'utilisateur peut en définir plusieurs pour son module. Ces jeux de valeurs des paramètres sont stockés dans un fichier « parameters.xml » situé dans le répertoire du module.

# 4 Bonnes pratiques de programmation

Ce chapitre tente de guider le programmeur -informaticien de plateforme ou de module- vers des pratiques pour obtenir un code informatique de meilleure « qualité ». C'est à dire plus efficient -notamment en terme de vitesse d'exécution- mais surtout emprunt de moins de bogues et de problèmes possibles à l'exécution. Ce codage plus « solide » passe principalement par de bonnes pratiques qui aident le compilateur à détecter le code potentiellement « faible ». Le compilateur alerte alors l'utilisateur sur les problèmes potentiels. L'utilisateur doit essayer de comprendre les alertes pour juger de la suite à leur donner dans le code informatique.

## *4.1 Programmation défensive et par contrat*

La programmation défensive part du principe que le programmeur peut insérer -malencontreusement- des fautes dans son code ou il utilise des données d'entrée non valides. La programmation défensive est une technique qui consiste à écrire son code de façon à imaginer le pire. C'est donc un état d'esprit où il faut penser à toutes les sources d'erreurs possibles et prévoir un traitement pour chacune d'elles. La programmation défensive demande de détecter les erreurs « au plus tôt ». Cela permet d'agir sur les effets et non sur les causes. Ces erreurs peuvent être de différents types :

- Les **valeurs** en dehors d'un **domaine de validité** donné. Par exemple, une valeur négative pour une masse ou un indice de tableau négatif.
- Les **valeurs non initialisées**. Pour les réels, cela correspond à « NaN » (Not a Number).
- Les **valeurs divergentes**. C'est à dire une valeur qui tend vers +/- l'infini.
- Les **pointeurs invalides**. Généralement à la valeur « nil », qui correspond à la constante NULL en C/C++.
- Plus rarement, on retrouve aussi les erreurs d'entrée/sortie (fichier absent ou inaccessible), les boucles infinies, les divisions par zéro...

Il existe quelques solutions qui permettent de se prémunir en grande partie contre ces erreurs -souvent fatales- à la bonne exécution d'un programme informatique. La démarche consiste à méthodiquement utiliser des assertions et certains automatismes lorsque les cas possibles d'erreurs ci-dessus se présentent. Les bonnes habitudes d'écriture à prendre sont donc :

1. En préambule de chaque fonction avec des variables en entrée, écrire des assertions[4] qui vérifient toutes les conditions possibles d'erreurs citées ci-dessus.
2. De même, à la fin de chaque fonction avec des variables en sortie, écrire des assertions qui vérifient toutes les conditions possibles d'erreurs citées ci-dessus.
3. A chaque ouverture de fichier, prendre soin de créer le code correspondant à la fermeture du même fichier.
4. Lorsqu'il s'agit d'allocations mémoires ou de tout autre allocation de ressource, l'automatisme est identique aux fichiers.

4 En programmation informatique, une assertion est un prédicat (une expression booléenne) qui doit toujours être vrai pour que l'exécution du programme puisse continuer.

La programmation par contrat[5] est en grande partie basée sur la programmation défensive. Elle formalise l'ensemble des règles d'assertion de la programmation défensive par le terme de contrat. La programmation par contrat précise les responsabilités entre le code appelant et le code appelé, qui est en général une fonction. Un contrat est décomposé en pré et post-conditions. La programmation par contrat veut que ces conditions soient incluses dès la phase de conception, avant le codage donc.

Les pré-conditions sont les conditions nécessaires et suffisantes à la bonne exécution du corps de la fonction. Ce sont typiquement les assertions du point 1, ci-dessus. Elles portent donc sur les variables d'entrées des fonctions. Par exemple, la fonction « calculer_surface_rectangle( x , y ) » a comme pré-condition (évidente) x et y > 0. Le mot-clef « require » est employé pour décrire cette situation.

Les post-conditions sont les conditions nécessaires et suffisantes à la bonne terminaison de l'exécution de la fonction. Ce sont typiquement les assertions du point 2, ci-dessus. Elles portent donc sur les variables de sorties des fonctions. Par exemple, la fonction « calculer_surface_rectangle( x , y ) » a comme post-condition évidente une valeur de retour > 0. Le mot-clef « ensure » est employé pour décrire cette situation.

Le respect de ces conditions est de la responsabilité de l'appelé. C'est à dire que c'est à la fonction qui décrit ses pré et post-conditions de les faire respecter. La stratégie à employer lors de la détection d'une assertion fausse dépend des capacités du langage d'implantation. Lorsqu'elles sont disponibles, ce qui est généralement le cas avec les langages à objets, les exceptions sont un excellent mécanisme à employer. Sinon, il est toujours possible de renvoyer des codes d'erreurs pour signifier le problème à la fonction appelante. C'est donc à la fonction appelante de décider quoi faire lors de la remontée d'une assertion fausse. Souvent, le choix se limite à « erreur fatale » et un arrêt du programme. Il convient alors d'afficher les informations utiles à la compréhension de cet état désagréable pour l'utilisateur. C'est à dire le lieu précis et les conditions exactes de l'erreur détectée.

Attention toutefois à ne pas faire peser sur le contrat la vérification des données saisies par l'utilisateur durant l'exécution du programme. La validation de données saisies est une étape indispensable que la programmation par contrat ne supprime pas. Cependant, elle peut permettre de détecter facilement un manquement dans cette étape.

## *4.2 Bonnes pratiques pour le 64 bits*

Si le besoin en puissance de calculs est toujours grandissant, le besoin de représenter des objets toujours plus volumineux en mémoire est lui aussi toujours plus important. C'est pourquoi augmenter la taille de l'adressage mémoire est intéressant pour représenter des objets de très grande taille en mémoire. Par exemple, la limite théorique est de 4 Go sur des architectures 32 bits. Dans le cas de « sol virtuel », cela peut se révéler utile de passer cette barrière, notamment pour des modèles qui doivent travailler avec des représentations du sol en 3D.

Ce chapitre est plus particulièrement dédié au problèmes de portage du code source d'un programme conçu sur une architecture système 32 bits vers du 64 bits. De manière générale, il est aussi valable pour tout changement dans la taille de l'adressage d'une architecture système vers une autre. De plus, s'il porte sur le langage C++ et les librairies utilisées par « sol virtuel », il n'est pas

5 Les anglophones utilisent le terme « design by contract ».

spécifique à ces outils.

La principale règle permettant de s'affranchir des problème de taille des adresses mémoire consiste à utiliser des types architecture-dépendant pour les variables qui représentent :

1. des adresses mémoire ;
2. des tailles d'objets en mémoire ;
3. des tailles de structures de données.

### 4.2.1 Cas des adresses mémoire

Ce cas ne pose pas de problèmes particuliers, à partir du moment où les types « pointeur » (caractère '*' en C/C++) sont utilisés pour stocker l'adresse mémoire des objets. Ces types sont naturellement architecture-dépendant, ils permettent donc d'adresser 4 Go[6] de mémoire en 32 bits et très largement plus en 64 bits[7].

### 4.2.2 Cas des tailles d'objets en mémoire

Ce cas se rencontre occasionnellement. Connaitre la taille d'un objet en mémoire s'effectue avec la fonction « sizeof() » en C/C++. Elle renvoi une valeur de type « site_t ». Il est toutefois fréquent de rencontrer dans le code source des variables de type entier (« int » en C/C++) pour stocker le résultat de cette fonction. En environnement 32 bits, cela ne pose pas de problème, puisque la taille d'un entier correspond aux nombres d'octets adressable sur 32 bits, soit 4 Go. Mais en environnement 64 bits, le type « int » reste codé sur 32 bits. Donc, si la taille de l'objet dépasse 4 Go, alors la variable contiendra une valeur invalide.

Ces 4 Go peuvent paraitre largement suffisant. Prenons le cas d'une matrice 3D de réels (« double » en C/C++) servant à représenter un sol avec des volumes isométriques. La limite dans ce cas est proche de 800 points pour chaque axe[8]. La bonne pratique consiste donc à utiliser le type architecture-dépendant « size_t ».

### 4.2.3 Cas des tailles de structures de données

Par le terme « structures de données », nous entendons les objets de type : tableau, matrice, liste, etc. Le terme « taille » représente aussi les « index » qui servent à désigner directement un élément dans une structure de données.

Le cas des tailles (ou index) des structures de données se rencontre souvent. Connaitre la taille d'une structure de données s'effectue avec la fonction « sizeof() » en C ou avec les fonctions « size() » de la libraire standard STL du C++. Elles renvoient toutes une valeur de type « site_t ». Il est toutefois fréquent de rencontrer dans le code source des variables de type entier (« int » en C/C++) pour représenter un index de tableau, pour parcourir ses éléments. En environnement 32 bits, cela ne pose toujours pas de problème. Mais en environnement 64 bits, le type « int » reste toujours codé sur 32 bits. Il est donc possible que le nombre d'éléments du tableau dépasse la valeur maximale possible avec un entier sur 32 bits, soit 4 milliards. Cela conduit inévitablement à une

6 4 Go = $2^{32}$ octets
7 18 Eo soit 1000 x 1000 x 18 To ! On est tranquille pour un moment.:-)
8 Valeur proche de la racine cubique de ($2^{32}$)/8.

violations d'accès mémoires ou à une boucle infinie, selon les cas.

Là aussi, ces 4 milliards d'éléments possibles peuvent paraitre démesurés. Reprenons le cas de la matrice 3D de réels. La limite dans ce cas est proche de 1600 éléments, pour chaque axe[9]. La bonne pratique consiste donc là aussi à utiliser le type architecture-dépendant « size_t ».

Un contre exemple concerne la librairie Qt. En effet, les méthodes qui donnent la taille d'une structure de données Qt (Qvector, Qlist, Qhash, etc) renvoient une valeur de type « int ». L'usage de la librairie Qt est donc à proscrire pour tout ce qui concerne la représentation d'objets mémoire d'une taille potentiellement très grande, même sur des machine 64 bits.

### 4.2.4 Cas particulier de « time_t »

Outre les problèmes liés au changement de taille des adresses mémoire, il existe un cas particulier pour les fonctions utilisant un type architecture-dépendant -non lié aux adresses mémoire- : « time_t ». Ces fonctions sont définies dans l'include « time.h » du C.

Le type « time_t » représente un nombre de secondes écoulées. Il est fréquent de rencontrer dans le code source des variables de type entier (« int » en C/C++) pour représenter le résultat de la fonction « time() ». Cette fonction renvoie le nombre de secondes écoulées depuis une date appelée Epoch[10]. En environnement 32 bits, la valeur maximale possible arrivera à partir du 19 janvier 2038. Date lointaine, certes, mais pas tant que ça. Surtout si l'on souhaite faire des simulations dans un avenir simulé. La bonne pratique consiste donc aussi à utiliser le type architecture-dépendant « time_t ». En environnement 64 bits, cela autorisera à utiliser les dates bien au delà de 2038.

## *4.3 Bonnes pratiques « dans le code »*

Les pratiques suggérées dans ce chapitre sont des suggestions issues d'expériences vécues. Elles s'appliquent le plus souvent au langage C++ ou aux langages à objets. Les langages procéduraux sont aussi concernés, donc le C et le Fortran. Les conseils sont classés avec un certain ordre d'importance, mais il n'est pas strict. De plus, des entêtes sur chaque point -entre crochets- permettent de savoir à quel langage informatique il s'applique :

- [Fortran][C][C++] Placer les déclarations de variables sur des lignes séparées et les commenter. Les noms des variables doivent être explicites, sans dépasser trois mots.
- [C][C++] Concernant l'inclusion des fichiers d'en-tête (ceux en suffixes « .h » et « .hpp ») dans les codes sources, les directives d'inclusion (« #include ») doivent aller « **du plus spécifique au plus général** ». La principale raison est d'augmenter la probabilité -pour le compilateur- de découvrir au plus tôt la raison d'un éventuel problème. En effet un fichier d'en-tête doit pouvoir s'analyser de par lui-même. Il ne doit dépendre que de son propre contexte, pas de celui de ses inclusions. En conséquence, les inclusions des fichiers d'en-tête « système » (comme « cmath » ou « cstdlib ») doivent toujours être parmi les dernières.
- [C][C++] Le C++ hérite de la commande « #define » pour déclarer des valeurs constantes. Il faut lui préférer les déclarations « **const** » et « **enum** ». Ces dernières permettent le typage des constantes et donc la vérification du bon contexte de leur usage par le compilateur. Depuis la nome C89, le langage C permet aussi l'utilisation de « const ».

9 Valeur proche de la racine cubique de $2^{32}$.

10 Sous Unix, c'est le 1er janvier 1970 à 0 heure (UTC).

- [Fortran][C][C++] Toutes les variables en entrée des fonctions doivent être des **constantes**.
- [Fortran][C][C++] Toutes les variables en entrée ou sortie des fonctions qui sont de type non-simple (cad objets ou tableaux) doivent être passées par **référence** ou éventuellement par pointeur.
- [C++] Concernant les fonctions qui retournent des objets par valeur, il est important que les **valeurs retournées** soient « **const** ». Ceci afin de prévenir l'utilisation de l'objet retourné comme une « lvalue ».
- [Fortran][C][C++] S'assurer de la **bonne initialisation** des variables avant de les utiliser.
- [Fortran][C][C++] Respecter les **standards** de chaque langage, les **normes** préconisées et les **usages**. Cela inclu, d'éviter les constructions déclarées comme obsolètes et d'utiliser les formes de codages les plus fréquemment rencontrées, afin de rendre le code plus lisible par des tiers.
- [C++] Dans le constructeur, préférer **l'initialisation** des objets plutôt que leur assignation. Ex : « lvalue = rvalue » doit être remplacé par « lvalue( rvalue ) ».
- [C++] Préférer les « **iostream** » du C++ aux « scanf » et « printf » du C. C'est à dire utiliser : « **std::cin** » pour l'accès au flux d'entrée standard du programme ; « **std::cout** » pour le flux de sortie standard ; « **std::cerr** » pour le flux de messages d'erreurs et « **std::clog** » pour le flux de messages d'informations.
- [C++] Utiliser la « **Standard Template Library** » (STL) du C++ pour manipuler les structures de données. Notamment le type « **std::vector** » pour manipuler des tableaux unidimensionnels au lieu de pointeurs. Note : pour la manipulation de tableaux de valeurs numériques, on lui préfèrera le type spécialement optimisé « **std::valarray** ».
- [C++] Utiliser la même forme de « delete » que de « new ». Notamment à un « **new[]** » doit toujours correspondre un « **delete[]** ».
- [C++] **Empêcher** le compilateur de **générer** et d'appeler **silencieusement** des fonctions, comme les constructeurs (par défaut et de copie) et les opérateurs (de copie), si ce n'est pas explicitement souhaité. Ceci n'est qu'occasionnellement le cas et doit être murement réfléchi. Par exemple, pour éviter que le compilateur génère le code du constructeur par défaut d'une classe, il faut déclarer le constructeur par défaut et ajouter « =delete ».
- [C++] Une fonction ne doit **pas retourner de référence** ou de pointeur sur des **objets locaux**, dont l'existence ne dépasse pas celle de la fonction.
- [C++] Les **membres** (variables) d'une classe doivent être **tous privés**, sauf s'il s'agit de constantes. Cela protège les données de la classe des accès extérieurs et garantit donc leur cohérence.
- [C][C++] **Déclarer** et initialiser les variables **le plus tard possible**. C'est à dire juste avant le moment de leur utilisation. Cela clarifie le code et rend l'exécution plus efficiente.
- [C++] Préférer les « cast » du C++ à celui du C. Ex : « **static_cast<int>**(value) » doit remplacer « **(int)**value ».
- [C][C++] Ne mettre en « **inline** » que les fonctions -très courtes- qui renvoient -ou calculent de manière simple- des valeurs. Notamment en C++ pour les objets, les fonctions qui permettent l'accès aux données membres sans modifier leur valeur sont « **inline** ».
- [C++] Utiliser le mécanisme des **exceptions** pour provoquer un arrêt du programme suite à

la détection d'une erreur. En effet, la fonction « exit » du C ne permet pas la libération « propre » des ressources systèmes allouées.

- [C++] Préférer les « **strongs enums** » de la norme « C++0x » plutôt que les « enums »venant du C. Les premiers permettent un typage fort et donc un meilleur contrôle de la part du compilateur.

## *4.4 Bonnes pratiques de « compilation »*

Les bonnes pratiques de compilations correspondent à l'usage d'analyseurs de codes statiques. Cela couvre une variété de méthodes utilisées pour obtenir des informations sur le comportement d'un programme sans l'exécuter. Cet usage le distingue de l'analyse dynamique (comme le « débogage » ou le « profiling ») qui concerne le suivi de l'exécution du programme.

L'analyse statique est donc utilisée pour repérer des erreurs de programmation ou de conception, mais aussi pour déterminer la facilité ou la difficulté à maintenir le code. Elle peut aussi permettre l'optimisation du code au niveau de l'exécution, de la mémoire, de sa sécurité.

### 4.4.1 L'analyse de code statique avec GCC

Comme la plupart des compilateurs modernes, le compilateur « GCC » ne se contente pas de transformer des lignes de code -au format texte- en langage machine -au format binaire, arrêtant le processus de compilation en cas d'erreurs. Il possède de nombreuses capacités de vérification du code. Ainsi le programmeur peut être alerté sur des pratiques douteuses ou éventuellement dangereuses. Ce peut être :

- l'usage d'instructions devenues obsolètes ;
- erreurs potentielles de programmation ;
- des conversions de type avec perte de précision ;
- des lourdeurs inutiles, comme des déclarations ou des conversions implicites ;
- et bien d'autres sujets encore...

Il est obligatoire que chaque code de la plateforme « sol virtuel », modules comme codes d'infrastructures et d'ateliers logiciels, soit soumis à un maximum de vérifications. Il est toutefois inconcevable d'activer toutes les options de vérification du code. Ceci pour deux raisons :

- Le compilateur n'alerte que sur des erreurs potentielles. Activer toutes les options afficherait tellement d'alertes que les dangers les plus vraisemblables seraient « noyés » dans le nombre d'alertes affichées. Trop d'informations tue l'information.
- La plupart (si ce n'est pas tous) des codes des librairies existantes ne peuvent être exempts de pratiques détectées comme des alertes par le compilateur. Là aussi le programmeur serait « noyé » par le flot d'erreurs potentielles. Ce serait d'autant plus inutile qu'il n'y pourrait rien puisque ce ne serait même pas sur son propre code[11].

Concrètement, chaque code devra être compilé avec les options suivantes, par ordre de priorité :

11 Cette pratique est toutefois intéressante pour connaitre un des aspects de la qualité du code d'une librairie. Il peut être utile de se donner une idée du nombre d'alertes en compilant avec une attitude très « paranoïaque ».

1. « -Wall », les vérifications les plus usuelles et les plus utiles ;
2. « -Wextra », des vérifications plus poussées, généralement utiles ;
3. « -Weffc++ », des vérifications très poussées, issues de l'excellente suite de livres de Scott Meyers « Effective C++ », mais qui posent souvent des problèmes aux codes sources des librairies.

### 4.4.2 L'analyse de code statique avec Sparse

Basé sur le compilateur « GCC », l'outil « Sparse » permet de détecter des erreurs supplémentaires. Son usage le plus simple consiste à l'invoquer à la place de GCC. C'est à dire, utiliser « cgcc », plutôt que « gcc ». Un exemple pour l'utiliser avec un « Makefile » tout en gardant le possibilité d'utiliser le compilateur « colorgcc » :

```
make CC=cgcc REAL_CC=colorgcc
```

Concrètement, chaque code devra être compilé avec les options suivantes :

```
-Wbitwise -Wcast-to-as -Wdefault-bitfield-sign -Wdo-while -Wparen-
string -Wptr-subtraction-blows -Wreturn-void -Wtypesign -Wundef
```

### 4.4.3 L'analyse de code statique avec cppcheck

Cppcheck est un outil complémentaire aux précédents. Dédié aux langages C et C++, il détecte entre autres :

- des défauts de style, comme l'inutilité de tester la nullité d'un pointeur avant de l'effacer ;
- les variables membres d'une classe C++ non initialisées dans le constructeur :
- les fonctions inutilisées.

Son utilisation est très simple. Par exemple, pour lancer la vérification de tous les codes sources du répertoire courant, il faut lancer la commande suivante :

```
cppcheck --enable=style,exceptNew,exceptRealloc --quiet .
```

## *4.5 L'analyse de code dynamique*

La mise en œuvre de l'analyse de code statique est la plus facile des bonnes pratiques de programmation. A ce titre, il ne faut pas s'en priver. Mais elle ne suffit pas à assurer un code informatique d'une qualité suffisante pour une diffusion large d'un logiciel. Pour obtenir un exécutable avec le moins de défauts de conception possible, il faut constituer un ensemble de méthodes capables de détecter les erreurs et mauvaises pratiques à l'exécution. Cette analyse dynamique est principalement constituée du « débogage » et du « profiling ».

Un bon usage de l'analyse dynamique peut repérer des erreurs de programmation ou de conception, comme des fuites mémoires, de manière très efficace. C'est aussi un passage obligé pour l'optimisation du code au niveau de l'exécution, de la mémoire, et de sa sécurité.

### 4.5.1 L'analyse de code dynamique avec Valgrind

Valgrind permet de trouver les fuites mémoires et un mauvais usage des allocations mémoires. C'est un « profiler » de code et un outil de déboguage de mémoire. Dans son usage le plus simple, il surveille les usages les plus commun de la mémoire : allocations, initialisations, écritures en dehors des tableaux, libérations des pointeurs. C'est son composant « Memcheck » qui est chargé de cette tâche.

Concrètement, chaque exécutable devra être testé en le démarrant avec Valgrind. En ligne de commande, cela signifie de précéder le nom de l'exécutable par celui de Valgrind.

Exemple :

```
valgrind --read-var-info=yes --track-origins=yes -leak-check=yes \
vsoil-processes
```

Note : la plupart des librairies -notamment graphiques- contiennent un certain nombre de mauvaises pratiques et même de fuites mémoires qu'il convient bien entendu d'ignorer.

# 5 Bonnes pratiques de tests

En complément du document de conception générale et à l'image des bonnes pratiques de programmation (cf. chapitre 4, page 11), cette section éclaire le programmeur sur les tâches de tests qui lui incombent, avant de livrer son code source et l'éventuel logiciel exécutable qui en découle. De part leur coté relativement générique, ces bonnes pratiques pourraient se révéler intéressantes pour d'autres acteurs, hors de la plateforme « sol virtuel ».

## *5.1 Mettre en œuvre la testabilité*

Rendre une application testable se joue dès les débuts de la réalisation d'une application et à toutes les étapes du développement (spécifications, conception...). Le testeur et le développeur doivent donc travailler ensemble avec comme objectif de rendre l'application testable. Les méthodes agiles facilitent la gestion de cette problématique au travers de certaines recommandations : écriture des tests avant le codage, participation des utilisateurs finaux... Toutefois, les tests sont fastidieux à spécifier et exécuter. Il convient donc de cibler ceux les plus pertinents et de les automatiser autant que possible.

Pour mettre en œuvre les tests dans les meilleures conditions, les pré-requis suivants sont nécessaires :

- Sans **spécifications** précises, pas de références pour les tests.
- Un logiciel testable va de pair avec une bonne conception, notamment basée sur les principes de **modularité**.
- Respecter les différentes phases de test (unitaires, intégration et système).
- Des journaux (logs) d'exécution et une gestion d'erreurs facilitent le débogue.
- Planifier les livraisons de logiciels en prenant en compte les phase de tests.

Évaluer la testabilité et mettre en amont les moyens pour l'améliorer sont donc nécessaires à la bonne exécution de l'activité de test. Cela a un cout, nécessite de la rigueur et du travail en amont, mais accélère les phases de test. Les experts du test logiciel estiment que les couts impliqués par les activités de test peuvent représenter autour de 25-30% du cout total du développement d'un logiciel.

## *5.2 Les phases de tests*

Les tests s'inscrivent dans quatre phases ordonnées. La réussite d'une phase de tests conditionne donc le passage à la suivante.

### 5.2.1 Les tests unitaires

Les tests les plus élémentaires sont les tests unitaires. Ils visent à vérifier la conformité des composants logiciels par rapport aux exigences de la tâche à laquelle ils doivent répondre.

Les tests unitaires doivent être effectués à chaque fois qu'une production de code source est réalisée. C'est-à-dire à l'issue d'une tâche -plutôt élémentaire- donnée. C'est en général au

développeur informatique de son propre code de les effectuer. Dans l'idéal, c'est un personne tierce -spécialisée dans le domaine- qui écrit les tests liés à la tâche réalisée.

Les tests unitaires vont faciliter le déroulement des tests système (décrits ci-dessous). Les anomalies liées à des erreurs de codage sont plus facilement détectées et corrigées à ce niveau. Ils vont permettre de localiser les erreurs de façon fine et permettre de corriger des anomalies ou d'améliorer le code sans crainte de violer la règle de non-régression. C'est donc une phase essentielle, qui doit retenir une attention particulière de la part de tous les intervenants informatiques autours du projet « sol virtuel ».

### 5.2.2 Les tests d'intégration

Les tests d'intégration permettent de valider les interfaces entre composants logiciels. Ils visent à s'assurer que les modules (fonctions ou objets informatiques) de l'application communiquent et interagissent de manière correcte, stable et cohérente.

Dans un contexte de développement agile, ils sont provoqués par chaque changement dans le code source, donc à la suite de chaque test unitaire ou éventuellement batterie cohérente de tests unitaires. Le développeur concerné par ce code peut prendre en charge les tests d'intégration. Dans l'idéal, c'est à une personne tierce -spécialisée dans le domaine- de les prendre en charge.

Dans certains cas, il s'agit de tester un changement non lié au code source. Par exemple, lors de changements dans les versions des librairies informatiques utilisées. Toute l'équipe de développement est alors potentiellement concernée par les tests.

Lorsque des changements dans le code source concernent des grandes parties en parallèles, il convient de planifier ces tests et de prévoir l'implication de toute l'équipe de développeurs.

### 5.2.3 Les tests système

Les tests système permettent de valider un système intégré par rapport à des exigences -logicielles ou matérielles- spécifiques. Dans le cas actuel de « sol virtuel », c'est en général un logiciel exécutable. Ces tests couvrent les aspects fonctionnels et non fonctionnels. Cela peut recouvrir des aspects de performance, d'installation, de robustesse, de sauvegarde...

Là aussi, le développeur concerné par ce code peut prendre en charge les tests d'intégration. Mais il est souhaitable de faire appel à une personne ou même une équipe de validation- dédiée.

### 5.2.4 Le test d'acceptation

Le test d'acceptation -autrement appelé recette ou réception- n'est pas a proprement parler un type de test. C'est plutôt un processus qui permet au « client » de valider que la livraison des fonctionnalités -développées par le « fournisseur »- correspond au cahier des charges. Dans le cas d'une démarche de développement de type agile, ce processus sur le logiciel est quasi-continu, puisque des utilisateurs sont intégrés à l'équipe de développement.

# 6 Règles graphiques des interfaces utilisateurs

En complément du document de conception générale, cette section décrit la présentation visuelle spécifique des logiciels du projet « sol virtuel », avec des règles de représentation en fonctions des types de données affichées.

En effet, quelque soit l'interface graphique de « sol virtuel » dans laquelle l'utilisateur se trouve, il est nécessaire de bien faire la distinction entre les deux grandes parties de la plateformes que sont :

- la partie processus et squelettes ;
- la partie modules et modèles.

Pour cela, **toutes** les interfaces graphiques de « sol virtuel » doivent adopter une couleur de fond commune à tous les objets graphiques en fonction de la partie dans laquelle ils se situent. C'est à dire que :

- Les objets graphiques qui représentent la partie « processus et squelettes » ont une couleur de fond spécifique qui est un « saumon pâle ». Nom X11 = « LightSalmon1 », codage RVB décimal = « 255 160 122 » / hexadécimal = « ff a0 7a ».
- Les objets graphiques qui représentent la partie « modules et modèles » ont une couleur de fond spécifique qui est en un « vert pâle ». Nom X11 = « PaleGreen1 », codage RVB décimal = « 154 255 154 » / hexadécimal = « 9a ff 9a ».

Afin de faire facilement la différence entre les entrées et les sorties des processus/modules, **toutes** les interfaces graphiques de « sol virtuel » doivent adopter :

- Pour les entrées des processus/modules, une icône commune de couleur de fond bleu. Codage RVB hexadécimal = 4e 7e c6.
- Pour les sorties des processus/modules, une icône commune de couleur de fond rouge. Codage RVB hexadécimal = ed 23 27.

# 7 Fichiers et chemins d'accès des logiciels

Les logiciels de la plateforme « sol virtuel » sont nommés avec la nomenclature suivante :

- « vsoil-processes », pour l'atelier des processus et squelettes ;
- « vsoil-modules », pour l'atelier de construction des modules ;
- « vsoil-models », pour l'atelier d'assemblage des modèles ;
- « vsoil-*compute* », pour l'atelier d'utilisation des modèles.

Chacun de ces logiciels est composé d'un fichier binaire propre[12] :

- pour l'atelier des processus et squelettes, « vsoil_processes » ;
- pour l'atelier de construction des modules, « vsoil_modules » ;
- pour l'atelier d'assemblage des modèles, « vsoil_models » ;
- pour l'atelier d'utilisation des modèles, « vsoil_*compute* ».

De plus, chacun des logiciels est accompagné des fichiers propres suivants :

- La documentation utilisateur[13], « userManual.pdf ».
- Le fichier qui décrit la licence, « LICENCE.txt ».
- Le fichier qui décrit les bogues actuellement connus, « BUGS.txt ».

Les différents logiciels doivent partager les données suivantes :

- Les données « officielles ». Ceci inclut les données des piscines, la documentation propre à chaque processus, ainsi que les sources des modules « officiels ». Ce sont des données en lecture seule.
- Les données propres à l'utilisateur. Ceci inclut les données des piscines et les sources des modules propres à l'utilisateur. Ces données sont spécifiques et à l'usage exclusif de l'utilisateur.

Il y a trois familles de chemins d'accès pour stocker les différents fichiers :

- Le chemin d'accès des fichiers binaires, « BINDIR » ;
- le chemin d'accès des données « officielles », « APPDATA » ;
- le chemin d'accès des données propres à l'utilisateur, « USERDATA ».

L'emplacement de ces données dépend du système d'exploitation employé :

12 Sous Windows, les binaires ont suffixe « .exe ».
13 Le format « Portable Document File » semble le plus adéquat dans un contexte multiplateforme.

| Donnée | Nom duchemin d'accès | Valeur pour Windows | Valeur pour Linux |
|---|---|---|---|
| vsoil-XXXX | BINDIR | C:/Program files | /usr/bin |
| userManual.pdf<br>LICENCE.txt<br>BUGS.txt | APPDATA | BINDIR/vsoil/vsoil-XXXX | /usr/share/doc/vsoil-XXXX |
| Données officielles | APPDATA | BINDIR/vsoil | /usr/share/vsoil-XXXX-doc |
| Données utilisateur | USERDATA | %USERPATH%[14]/vsoil[15] | $HOME[16]/.vsoil |

*Tableau 1: chemins d'installation des fichiers*

La plupart des librairies multiplateforme permettent de masquer -en partie- les différences entre les systèmes d'exploitation. Avec Qt, la méthode statique « QDir::home () » renvoie le chemin d'accès complet au répertoire de l'utilisateur. Elle renvoie donc « %USERPATH% » ou « $HOME », selon le système hôte.

14 La variable d'environnement Windows « USERPATH »
15 Le répertoire doit porter l'attribut « caché »
16 La variable d'environnement Unix « HOME »

# 8 Diffusion des logiciels « sol virtuel »

Comme il est indiqué dans la licence logicielle du projet (voir le document de conception générale, chapitre 5), la diffusion des logiciels est possible uniquement par le site Web « sol virtuel ». Avant de pouvoir télécharger et installer les logiciels, l'utilisateur doit s'enregistrer un première fois, en donnant des informations d'ordre général : nom, email... Il lui sera alors posé quelques questions permettant au site Web de lui proposer le(s) logiciel(s) adapté(s) à sa situation. Ainsi, en fonction de son profil (voir les acteurs, dans le document d'analyse informatique, section 3.2) et de son système d'exploitation, l'utilisateur pourra sélectionner les liens proposés par le site Web :

- Sur les systèmes Windows, cela sera des liens vers des programmes exécutables dédiés à l'installation.
- Sur les système Linux de la famille Debian/Ubuntu, cela sera un lien de type « apturl » qui installera automatiquement tous les logiciels et leurs dépendances éventuelles.