Projet Informatique – Sections Electricité et Microtechnique

Printemps 2021 : *Tchanz* © R. Boulic & collaborators

Rendu1 (3 avril 23h59)

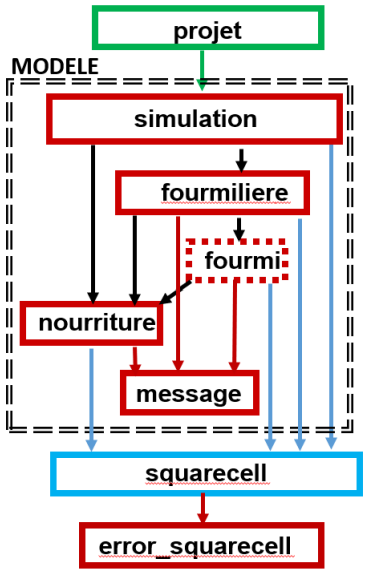
Objectif de ce document : *Ce document utilise l'approche introduite avec la série théorique du Topic1 sur les méthodes de développement de projet qu'il est important d'avoir faite avant d'aller plus loin.*

En plus de préciser ce qui doit être fait, ce document identifie des ACTIONS à considérer pour réaliser le rendu de manière rigoureuse. Ces ACTIONS sont équivalentes à celles indiquées pour le projet d'automne ; elles ne sont pas notées, elles servent à vous organiser. Vous pouvez adopter une approche différente du moment que vous respectez l'architecture minimale du projet (donnée Fig 9a).

1. Buts du rendu1 : mise au point de squarecell et lecture de fichier

Le *premier* objectif est de disposer d'un module **squarecell** dont chaque fonction est validée par des test extensifs (avec du *scaffolding et un test unitaire de ce module*). Le but est de disposer d'un outil stable pour les rendus suivants. Cela implique de définir en priorité les structures de données, assez simples, de **squarecell** puis les fonctions que ce module va offrir dans son interface. Nous demandons de mettre à disposition des types **concrets**[1] à l'aide de structures pour faciliter leur manipulation au niveaux supérieurs.

C'est seulement après la validation du module **squarecell** que vous pourrez aborder le second objectif d'ébauche du **Modèle** avec l'ensemble des modules et des dépendances visibles dans la figure ci-contre.



Donnée Fig 9b
N'utilise PAS GTKmm

Nous imposons que les modules du **Modèle** mettent en œuvre des **classes** en respectant le *principe d'encapsulation*, ce qui veut dire que les *attributs* seront **private** et qu'il faudra utiliser des *méthodes* pour y accéder. A part cette contrainte stricte, nous acceptons pour ce premier rendu que votre choix de structure de donnée soit une ébauche qui pourra être remise en question pour les rendus suivants.

Pour le rendu1, le module de plus haut niveau **projet** contient seulement la fonction **main** : sa tâche est de récupérer le *nom de fichier de test* transmis sur la ligne de commande du lancement de l'exécutable. Ce nom de fichier doit être immédiatement transmis à une fonction ou méthode du module **simulation** qui agit comme point d'entrée du **Modèle**.

Les responsabilités des modules du Modèle (donnée section 7.2) sont complétées ici :

- **simulation** : c'est le SEUL point d'entrée du Modèle vis-à-vis de l'extérieur (module **projet** pour le rendu1)
  - il gère au plus haut niveau d'abstraction les tâches de (*une*) mise à jour de la simulation, dessin, lecture et écriture de fichier. Pour le **rendu1**, on demande seulement de mettre au point l'action de **lecture** d'un fichier pour initialiser une simulation en détectant des erreurs prédéfinies.
  - en vertu du principe d'abstraction, ce module délègue l'exécution des sous-problèmes de ces tâches auprès des modules qui gèrent les entités de **nourriture** et de **fourmiliere** (cf Fig9b).

- **fourmiliere** : gère à son niveau d'abstraction les tâches qui lui sont déléguées par le module **simulation** et délègue la mise en œuvre des sous-tâches aux modules **fourmi**(s).

[1] Une structure est un type **concret** car on peut directement accéder à ses champs

- **fourmi** : pour le rendu final il est demandé de mettre en place *une hiérarchie de classes* (deux niveaux suffisent) pour gérer les différents types de fourmis. Cependant, pour le **rendu1**, une approche simplifiée sera accepté : avec un attribut de *type* dans une classe unique **fourmi**. Ce module gère à son niveau d'abstraction les tâches qui lui sont déléguées par le module **fourmiliere**

- Le module **message** est *fourni* pour l'affichage de messages standardisés pour la tâche de lecture du Modèle. Il ne faut pas le modifier.

La section suivante précise ce qui est demandé pour le module **squarecell** et comment le tester.

2. A quoi sert le module squarecell ?

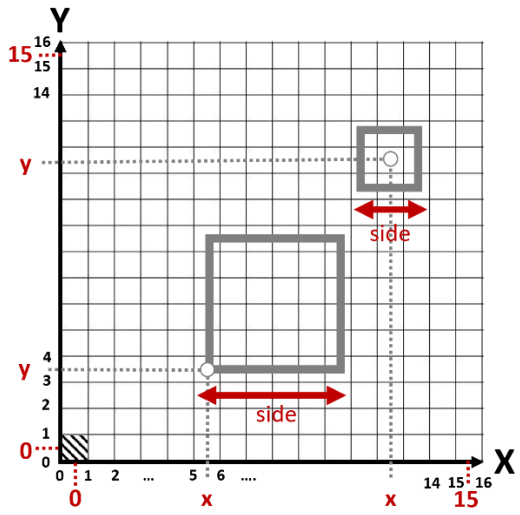
Ce module sert à gérer des entités carrées dans une grille à 2 indices de taille constante **g_max x g_max** .

Les coordonnées (x, y) de l'origine des carrés doivent respecter la convention d'axe décrite dans la Fig2(donnée) et ci-contre. La relation entre cette convention (x,y) et les indices (i,j) de ligne et colonne d'une grille à deux indices en C++ est la suivante :

- ***i* = g_max-1 – *y***
- ***j* = *x***

Un carré peut être de 2 types (cf ci-contre):

- **Centré** : son coté doit avoir une valeur impaire
- **Non-centré** : son origine est le coin inférieur gauche ; la valeur de son coté doit être strictement positive.



Rendu1 Fig1 : conventions d'axes de coordonnées x et y de la grille dans laquelle vont exister les carrés. Les valeurs [min,max] des indices possibles sont indiquées en rouge pour cet exemple avec g _max valant 16. Les carrés doivent appartenir totalement à la grille.

Cela étant posé, ce module doit offrir les fonctions suivantes :

- Test de validation d'un carré : un carré est valide si sa taille est cohérente avec son type, si ses coordonnées (x,y) sont dans l'intervalle [0, g_max-1] et s'il est entièrement à l'intérieur de l'espace [0, g_max-1]. La détection d'une incohérence dans ces tests doit conduire à effectuer l'appel d'une fonction que nous mettons à disposition dans le petit module **error_squarecell**. Voici les fonctions :

```
// violation du domaine autorisé d'un indice [0, g_max-1]
std::string print_index(unsigned index, unsigned max);

// violation du domaine couvert [0, g_max-1]
// par la combinaison de l'origine et du coté d'un carré
std::string print_outside(unsigned index, unsigned side, unsigned max);
```

Ces fonctions sont prévues pour afficher un message vers **cout** (et pas ailleurs) comme ceci :

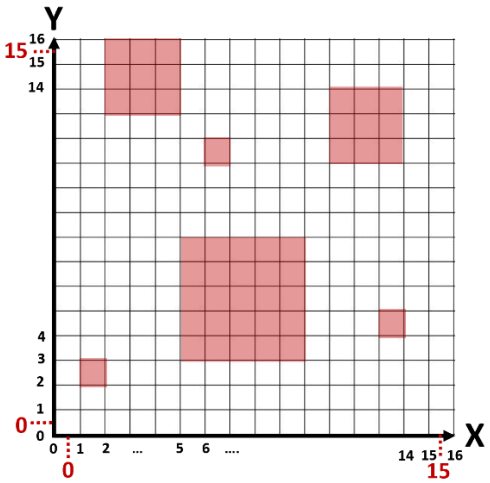
```
if(une détection d'erreur est vraie)
{
      cout << message::appel_de_la_fonction(paramètres éventuels);
      std ::exit(EXIT_FAILURE) ; // Rendu1
}
```

A partir du rendu2, il ne faudra pas quitter le programme mais renvoyer un booléen d'échec.

- Test de superposition de deux carrés : plusieurs variantes peuvent exister selon qu'on veut obtenir un booléen ou quantifier le nombre de cellules qui se superposent

- Test de superposition d'un carré avec une mémorisation de l'occupation d'un ensemble de carrés : Cette approche permet de déterminer à faible coût si un carré intersecte un ensemble d'autres carrés dont l'occupation a été mémorisée au préalable dans une grille de booléens. Pour cela nous nous donnons les moyens suivants :

- Il est autorisé de cacher dans le module une grille de **booléens** de taille **g_max x gmax**
- Cette grille a le droit d'être globalement accessible mais pas partout : elle doit être *cachée* dans ce module[2]
- Cette grille est initialement à *false* pour tous les booléens (les cellules sont libres).
- Il faut mettre à disposition :
  - une fonction pour déclarer l'occupation des cellules correspondant à un carré passé en paramètre, ce qui fait passer les cellules de ce carré à *true* dans la grille globale du module
  - une autre fonction qui libère les cellules de la grille correspondant à un carré passé en paramètre en les faisant passer à *false*.
  - une ou plusieurs fonctions effectuant un test de superposition d'un carré passé en paramètre avec l'état courant de la grille de booléens et renvoyer un booléen ou un nombre de cellules en superposition.



Rendu1 Fig2 : grille de booléens (rouge= occupé, blanc = vide) globale au module et indiquant l'occupation de la grille **g_max x g_max** par un ensemble de carrés

Voici la déclaration d'une telle grille qui doit être cachée dans l'implémentation de **squarecell** :

```
vector<vector<bool>> grid(g_max,vector<bool>(g_max));
```

L'ordre de test des cellules du carré passé en paramètre vis-à-vis de la mémoire d'occupation doit être *ligne par ligne*, de *haut en bas* et de *gauche à droite*, afin de détecter la même première cellule que l'autograder.

3. lecture de fichiers de configuration

3.1 Module projet

Le module **projet** contient la fonction **main()** en charge d'analyser si la ligne de commande est bien conforme à la syntaxe suivante : `./projet test1.txt`

Où `test1.txt` est un fichier de configuration (Donnée **Section 4**).

Nous ne testerons pas l'absence de l'argument. Pour le rendu1, votre programme doit se terminer en cas d'absence d'argument. Si l'argument est présent nous supposons qu'il correspond à un fichier de test présent dans le répertoire courant. Ce nom de fichier doit être transmis à une fonction ou méthode de **lecture** du module **simulation.**

Deux stratégies sont autorisées concernant *l'instance* de la classe **Simulation** qui pilote le Modèle :

- Soit elle existe dans le module **projet** et une méthode **lecture** est appelée sur cette instance.

- Soit elle est cachée dans le module **simulation** (car elle est unique) et une fonction **lecture** du module simulation est appelée sans avoir besoin de préciser en paramètre sur quelle instance elle travaille.

[2] Le cours présente deux possibilités pour cela: le *namespace non-nommé* et le mot clef **static**

3.2 Liste des erreurs à détecter au niveau du Modèle (Donnée section 4.2):

Le programme cherche à initialiser l'état du **Modèle** avec les données lues dans le fichier de configuration. Pour le Rendu1 il **s'arrête** dès la ***première*** erreur trouvée dans le fichier en appelant la fonction mise à disposition pour l'affichage du message d'erreur puis on quitte le programme en appelant **exit(EXIT_FAILURE)**.

Le programme **s'arrête** aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce cas, il y a affichage d'un message indiquant le succès de la lecture (section 3.3) puis on quitte en appelant **exit(0**).

Remarque sur l'ordre des données après la lecture : la donnée insiste à plusieurs endroits sur le fait que les ***ensembles d'instances doivent être organisés dans le même ordre que dans le fichier*** (et pas dans l'ordre inverse par exemple). Cela est important pour les rendus suivants pour comprendre l'exécution séquentielle de la simulation et pour le codage de la couleur de l'affichage.

Tout d'abord, on délègue au niveau du module utilitaire **squarecell** la vérification des entités carrées et l'éventuel affichage d'erreurs détectée à son niveau.

Ensuite, voici la liste des points à vérifier au niveau du Modèle

a) Absence de superposition entre fourmilieres
b) Absence de superposition entre un élément de nourriture et l'ensemble des autres éléments de nourriture (mémorisés dans la grille des booléens de **squarecell**)
c) Absence de superposition entre une fourmi et l'ensemble des autres fourmis et celui des éléments de nourriture (mémorisés dans la grille des booléens de **squarecell**)
d) Les fourmis Generator et Defensor doivent être entièrement dans leur fourmiliere

IMPORTANT : l'ordre des tests b) et c) doit impérativement respecter celui de la lecture et être effectué au moment de la lecture de chaque entité. La raison de cette contrainte est la suivante : nous vous demandons d'utiliser la mémoire d'occupation d'un ensemble de carrés offerte par **squarecell**. Cette mémoire doit seulement contenir les carrés des éléments de **nourriture** et des **fourmis** (*pas* les carrés des fourmilières). La mémoire va être remplie au fur et à mesure de la lecture (Donnée section 4.1), c'est-à-dire avec d'abord les éléments de **nourriture** dans l'ordre d'apparition dans le fichier de test, puis la fourmi Generator, puis les fourmis Collector, Defensor et Predator (aussi dans l'ordre d'apparition dans le fichier).

Nous ne testerons pas vos projets sur la longueur des lignes de fichier ni sur d'autres éventuelles incohérences.

3.3 Utilisation du module message :
Le module **message** est fourni dans un fichier archive sur moodle ; il ne doit pas être modifié. Son interface **message.h** détaille l'ensemble des fonctions à appeler. Comme pour la détection d'erreur au niveau de **squarecell**, il faudra utiliser les messages de cette façon :

```
if(une détection d'erreur est vraie)
{
      cout << message::appel_de_la_fonction(paramètres éventuels);
     std ::exit(EXIT_FAILURE) ; // Rendu1
}
```

A partir du rendu2, il ne faudra pas quitter le programme mais renvoyer un booléen d'échec pour les cas d'échec et un booléen de succès quand la lecture de fichier et toutes les validations ont été effectuées avec succès. Dans ce cas, c'est la fonction **success** qu'il faut appeler.

Le message affiché par ces fonctions se termine par un passage à la ligne. Il ne faut pas en ajouter un.

3.4 Méthode de travail

Plusieurs approches sont possibles ; utilisez le document *méthodes de développement de projet* comme guide.

Au niveau de l'exécution, nous fournirons un nombre limité de fichiers de tests sur lesquels votre programme sera évalué. Le succès de ces tests ne peut pas garantir l'absence de bugs pour d'autres fichiers de tests. Donc, commencez à *organiser votre propre batterie de tests* indépendamment de ce que nous mettons à disposition.

3.4.1 ACTION : test unitaire de squarecell

Pour effectuer le test unitaire d'un module vous devez écrire un programme de test (*scaffolding*) qui effectue des appels de toutes les fonctions offertes par le module et ce programme compare le résultat renvoyé par chaque appel au résultat attendu (que vous avez calculé indépendamment par un autre moyen).

A partir des indications de ce rendu1 décidez le nom des fonctions et leur prototype et mettez les fonctions exportées dans **squarecell.h**. Une fois cela fait on peut inclure **squarecell.h** dans un programme de test pour tester chacune des fonctions de **squarecell.cc**. C'est votre responsabilité de trouver un nombre suffisant d'exemples pertinents pour s'assurer qu'on n'oublie pas de cas particuliers.

Il est essentiel d'effectuer ce test unitaire sur **squarecell** avant d'utiliser ses fonctions dans le Modèle.

3.4.2 ACTION : test du Makefile de l'architecture du rendu1

A partir du dessin de l'architecture du rendu1 (page 1) en déduire les dépendances et écrire le fichier Makefile. Testez-le avec des modules contenant le minimum pour être compilable et exécutable, c'est-à-dire les includes et, au plus haut niveau, une fonction main vide ou simplement avec affichage d'un message.

3.4.3 ACTION : tests du module simulation

Au stade du rendu1, seul le constructeur et la fonction/méthode de lecture de fichier sont nécessaires. Dans ce module l'opération de lecture met en place *l'automate de lecture* (cours Topic3) qui filtre les lignes inutiles du fichier et délègue l'analyse fine de lecture d'une ligne aux autres modules plus spécialisés (nourriture, fourmiliere, fourmi) qui ont la responsabilité de faire les vérifications nécessaires. L'automate peut être testé avec des *stubs* de ces fonctions/méthodes plus spécialisées de lecture qui renvoient toujours true pour indiquer que le décodage de la ligne de fichier s'est bien passé.

A ce stade l'intégration avec le module projet est immédiate puisqu'il suffit de remplacer le stub de la fonction/méthode de lecture du fichier par un appel de celle mise au point pour le module simulation.

3.4.4 ACTION : tests du module nourriture

Au stade du rendu1, seul le constructeur et la fonction/méthode de décodage d'une ligne lue dans un fichier sont nécessaires. Chaque élément de nourriture lu dans le fichier est ajouté à la mémoire de **squarecell** pour vérification de ses coordonnées et pour la détection d'une éventuelle superposition.

Une fois effectué cette validation il est possible d'intégrer ce module avec celui de Simulation ; c'est à vous de décider où l'ensemble des éléments de nourriture doit exister selon l'usage qu'il est prévu d'en faire. : est-ce un attribut de Simulation ou est-ce un ensemble caché dans l'implémentation du module nourriture ?

3.4.5 ACTION : tests du module fourmilière

Au stade du rendu1, seul le constructeur et la fonction/méthode de décodage d'une ligne lue dans un fichier sont nécessaires. On mettra en place des stubs pour simuler le décodage d'une ligne de fichier contenant une fourmi. Sachant que chaque fourmiliere gère ses fourmis il semble logique de prévoir la mémorisation des ensembles de ces entités[3] comme des attributs de la classe Fourmiliere.

De plus sachant qu'il faut mémoriser un ensemble d'instances de Fourmiliere, il est possible d'écrire un peu de code de scaffolding qui ajoute une nouvelle Fourmiliere à un tel ensemble seulement si elle n'intersecte pas une Fourmiliere déjà présente dans cet ensemble. On ne peut pas utiliser la mémoire d'occupation de **squarecell** pour les Fourmiliere car elle ont le droit de contenir les carrés de plusieurs types de fourmis. A la place la classe Fourmiliere peut effectuer un appel au module **squarecell** pour demander la vérification de superposition des carrées de 2 Fournilieres.

[3] Il est recommandé de mémoriser un ensemble de pointeurs sur des instances de Fourmi au lieu d'un ensemble de Fourmi afin de bénéficier du polymorphisme du C++ pour les rendus suivants.

Une fois effectué cette validation il est possible d'intégrer ce module avec celui de Simulation ; c'est à vous de décider où l'ensemble des Fourmilieres doit exister : est-ce un attribut de Simulation ou est-ce un ensemble caché dans l'implémentation du module Fourmiliere ?

3.4.6 ACTION : tests du module fourmi puis méthode de lecture puis intégration avec le module Fourmiliere
Les fourmis peuvent être représentées par une seule classe à l'aide d'un attribut de type mais cela n'est accepté que pour le rendu1 ; il faut proposer une hiérarchie de classes (2 niveaux suffisent) pour les rendus 2 et 3. Indépendamment de votre choix pour le rendu1, utilisez le type offert par **squarecell** pour décrire le carré occupé par une fourmi et utilisez la mémoire d'occupation de **squarecell** pour tester l'ajout d'une nouvelle fourmi. Ecrivez du code de scaffolding pour initialiser des instances des différentes types de fourmis et vérifier la valeur des attributs avec une méthode d'affichage dans le terminal. La fourmi Generator est créée en même temps que la Fourmilière ; elle doit exister en tant que type de Fourmi.

Ensuite seulement écrivez la méthode qui décode la ligne du fichier pour chaque type de Fourmi. Testez cette méthode en transmettant une ligne avec la syntaxe du fichier et vérifiez avec la méthode d'affichage que le décodage s'est bien passé. Ensuite intégrez de proche en proche avec les niveaux supérieurs.

4. Forme du rendu1

Documentation : l'entête de vos fichiers source doit indiquer les noms des membres du groupe, etc.

Rendu : pour chaque rendu **UN SEUL membre d'un groupe** (noté **SCIPER1** ci-dessous) doit téléverser un fichier **zip**[4] sur moodle (pas d'email). Le non-respect de cette consigne sera pénalisé de plusieurs points. Le nom de ce fichier **zip** a la forme :

SCIPER1_ SCIPER2.zip

Compléter le fichier fourni **mysciper.txt** en remplaçant 111111 par le numéro SCIPER de la personne qui télécharge le fichier archive et 222222 par le numéro SCIPER du second membre du groupe.

Le fichier archive du rendu1 doit contenir (**aucun répertoire**) :

- Fichier texte édité **mysciper.txt**
- Votre fichier **Makefile** produisant un executable **projet**
- Tout le code source (.cc et .h) nécessaire pour produire l'exécutable.

*On doit obtenir l'exécutable **projet** en lançant la commande* **make** *après décompression du fichier **zip**.*

Auto-vérification : Après avoir téléversé le fichier **zip** de votre rendu sur moodle (upload), récupérez-le (download), décompressez-le et assurez-vous que la commande **make** produit bien l'exécutable et que celui-ci fonctionne correctement.

Exécution sur la VM: votre projet sera évalué sur la VM à distance.

Backup : Il y a un backup automatique sur votre compte myNAS.

Gestion du code au sein d'un groupe :

- vous pouvez envisager d'utiliser **gdrive.epfl.ch** pour définir un répertoire partagé par les 2 membres du groupe et pas plus. Cependant il n'y a pas d'éditeur de code en mode partagé.
- Une approche qui demande un apprentissage supplémentaire serait d'utiliser **github** : cf ce tutorial sur moodle. Attention : il FAUT restreindre l'accès du code aux seuls 2 membres du groupes.

Rappel sur l'outil GDB de recherche de bug :

- Guide utilisateur de GDB avec son code de test ; GDB tutorial in english

[4] Nous exigeons le format zip pour le fichier archive