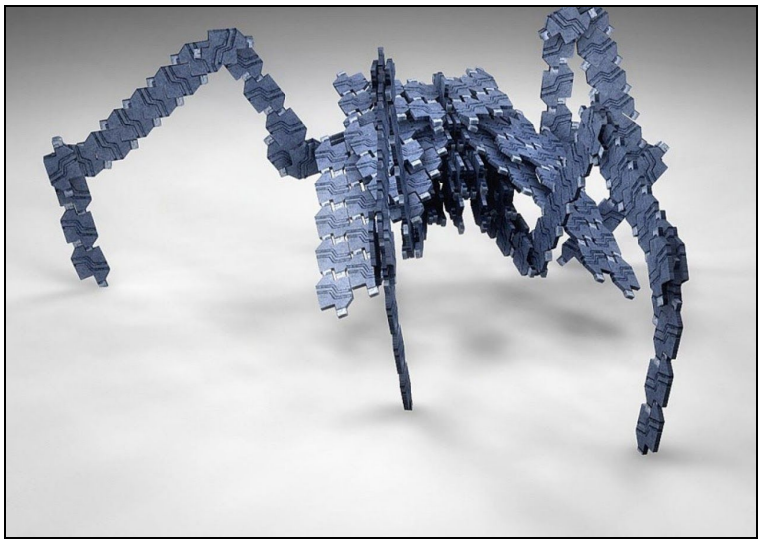
TP C++ : Replicators



Les Réplicateurs sont des petits robots ayant la capacité de s'assembler pour en former des plus gros. Avides de conquête, ces petites bêtes se reproduisent rapidement et interagissent avec leur environnement pour en tirer de l'énergie.

Le cycle de vie classique de ces réplicateurs est :

- Je cherche des objets,
- J'absorbe leur énergie,
- Je communique avec les autres réplicateurs pour augmenter de catégorie,
- Je crée une Reine quand je le peux,
- La Reine crée de nouveaux réplicateurs ,
- etc...

Nous allons suivre dans ce TP la vie de deux petits réplicateurs : Minus et Cortex.

<u>Commençons par créer nos réplicateurs.</u>

Chaque réplicateur est représenté par :

- Une catégorie (un entier)
- Un identifiant (chaine)
- Un niveau d'énergie (un entier)

Chaque réplicateur peut :

- Se charger (augmenter son énergie)
- Agir (diminuer son énergie)

1. Créez la classe Replicator.

Replicator
categorie id energie
charger() agir()

★ Attention à faire les bonnes inclusions, les bonnes portées, et à bien séparer vos fichiers.

2. Créez une méthode ping()

❑ Cette méthode permettra à chaque réplicateur d'afficher son identifiant, sa catégorie, et son énergie.

Replicator
categorie id energie
charger() agir() ping()

3. Surchargez le constructeur

❑ Ceci permettra de pouvoir choisir la valeur des attributs lors de la création d'un réplicateur.

> ★ Par défaut, les réplicateurs sont de catégorie 1, ont 100 d'énergie, et un identifiant "Reese".

Bon, assez codé, construisons nos deux vedettes.
Instanciez deux réplicateurs : Minus, et Cortex.

> ★ Attention, passez les bons paramètres à votre constructeur pour créer vos deux réplicateurs.

> ❖ Vérifiez par des pings que chacun est bien qui il doit être.

4. Codez les méthodes charger et agir.

> ❑ Celles-ci agissent sur l'énergie du réplicateur (respectivement en l'augmentant et en la diminuant).

C'est bien, mais si il suffisait qu'ils se chargent tous seuls dans leur coin, Minus et Cortex auraient la vie facile. En pratique, ils sont obligés de consommer des choses pour en tirer de l'énergie.

5. Créons donc une classe Chose

> ❑ Celle-ci ne sera représentée que par un niveau d'énergie (un entier).

Chose
energie

Ok, l'idée maintenant est de faire en sorte que l'énergie de nos réplicateurs n'augmente qu'en consommant des Choses.

6. Modifions la méthode charger.

- ❏ Renommons la en absorber(), et faisons en sorte qu'elle prenne en paramètre une chose. C'est l'énergie de cette chose qui sera absorbée.

Replicator
categorie id energie
absorber() agir() ping()

Problème : l'énergie d'une chose est privée. Or, la seule raison d'être d'une Chose est d'avoir une énergie à disposition. Rajoutons donc une méthode getEnergie() publique, qui nous renverra l'énergie de chaque Chose.

- ❏ Créez une chose(énergie au choix) et vérifiez que Minus et Cortex peuvent l'absorber.

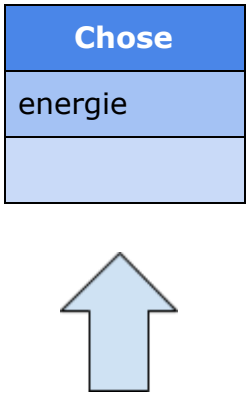
- ★ Oui mais, dans notre cas, un objet absorbé peut toujours l'être. Modifions un peu notre code pour qu'un objet absorbé ne soit plus disponible. Comment? Eh bien réduisons à 0 l'énergie d'un objet quand il est absorbé. (Changeons la méthode getEnergie en syphoner(). Syphoner renverra l'énergie de la chose, et mettra cette énergie à 0.). Attention, la chose existe toujours, mais elle n'a plus d'énergie.

Exemple :
Pile (én.150) est absorbé par Minus (én.100). Pile a maintenant 0 en énergie, et Minus 250.

❖ Testez votre code avec quelques appels à ces méthodes.

Ok, on peut maintenant se nourrir de choses.

7. Un réplicateur est une chose !



★ Attention, plus complexe : On peut considérer que les réplicateurs sont des choses. Un réplicateur doit donc pouvoir absorber un autre réplicateur.

❑ Ce qu'il faut se dire, c'est qu'un réplicateur est un type particulier de chose, et peut donc hériter de Chose.

★ Attention ! Il va falloir adapter les constructeurs, les portées, et les attributs en conséquence ! (recherchez "liste d'initialisation" pour vous aider avec les constructeurs)

Après cette étape, Minus devrait pouvoir absorber Cortex.

Mais bon, les réplicateurs ont cet avantage de réfléchir au bien commun avant le leur, ils vont donc se focaliser sur d'autres choses plutôt que sur des réplicateurs.

Nos réplicateurs peuvent donc se nourrir. Ils vont maintenant devoir apprendre les uns des autres pour pouvoir évoluer.

8. Codons la méthode telecharger

❑ Cette méthode permet à un réplicateur de monter de catégorie en téléchargeant les données d'un autre réplicateur.

Replicator
categorie id energie
absorber() agir() ping() telecharger()

★ Attention, il y a certaines règles à suivre:

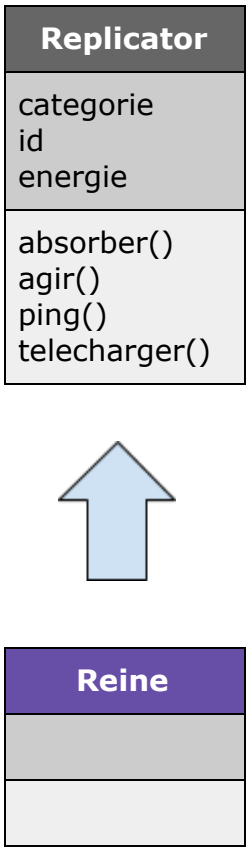
★ Télécharger coûte de l'énergie. Le réplicateur qui télécharge doit dépenser la moitié de l'énergie de sa cible * sa catégorie. Exemple : Si Cortex (cat 3 energie 200) veut absorber Minus (cat 2 energie 100), il devra dépenser 2*50=100 énergie. Cortex passera donc en catégorie 4, et aura 100 d'énergie restante).

★ Télécharger ne modifie pas le réplicateur ciblé.

❏ N'oublions pas de tester !

Nos réplicateurs ne sont toujours pas en mesure de se répliquer…. Il est temps d'y remédier.
Lorsqu'un réplicateur a accumulé assez de données, il peut construire une Reine, qui sera en charge de créer de multiples réplicateurs.

9. Construisons notre classe Reine



★ Attention ! Une Reine est bien sûr un réplicateur ! (conseil : forward dec dans Replicator.h de la classe Reine, et un include dans Replicator .cpp)

10. Permettons à un réplicateur de pouvoir créer une Reine.

Replicator
categorie id energie
absorber() agir() ping() creer_reine()

★ Attention ! Seul un réplicateur de catégorie 5 ou plus qui n'a pas de reine peut construire une Reine.

★ De plus, les Réplicateurs ne peuvent avoir qu'une seule Reine. Il va donc falloir que chaque réplicateur sache qui est sa reine. Par défaut, les réplicateurs n'ont pas de reine (Minus et Cortex n'en ont donc pas. Celà dit, une fois qu'un réplicateur crée une reine, elle devient sa reine, et chaque réplicateur créé par cette reine lui sera affilié. Il faut donc rajouter à chaque réplicateur cette information. (Un pointeur serait le bienvenu).

★ Enfin, créer une reine est coûteux en énergie (1000/categorie). Vérifiez bien qu'un réplicateur peut créer sa reine avant de la créer….

11. Ajoutons un constructeur à notre reine

❑ Ceci permettra que son id contienne celle de son créateur (une reine créée par Cortex s'appellera ReineCortex).

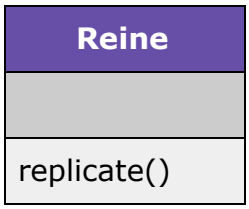
★ Note: Une reine n'a pas de reine.

Il ne reste plus qu'à pouvoir créer des réplicateurs, et ça c'est le boulot de la reine.

12. Créons la méthode replicate

❏ Celle-ci renvoie un Replicator tout neuf.

★ Attention, créer un réplicateur est coûteux. (100 d'énergie).



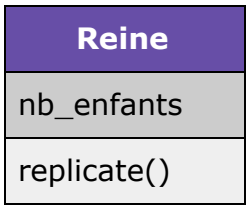
Il va donc falloir que la reine absorbe quelques choses si besoin.

★ Le nom de chaque nouveau réplicateur sera le nom de sa reine suivi par un numéro de série. Le numéro de série est donc le nombre de réplicateur créés par cette reine.

Exemple : ReineCortex crée trois réplicateurs, ils se nommeront ReineCortex1,ReineCortex2 et ReineCortex3.

13. Ajoutons un numéro de série incrémental

❏ Cell-ci nous permettra d'identifier les nouveaux réplicateurs.



★ Le numéro est donc propre à chaque reine, ce qui veut dire qu'il ne peut pas être static.

★ Il doit donc être implémenté comme attribut de Reine, et bien s'incrémenter à chaque fois qu'elle crée un réplicateur.

Note : la fonction to_string existe, mais suivant l'IDE et le compilateur que vous utilisez, elle peut bugger (sur codeblocks par exemple…) au cas où, voici une réécriture de la fonction, à mettre dans votre .cpp:

```
#include <sstream>
template <typename T>
std::string to_string(T value)
{
	std::ostringstream os ;
	os << value ;
	return os.str() ;
}
```

Enfin, on aimerait connaitre le nombre total de réplicateurs créés depuis le début.

14. Créons donc un attribut static

❏ Celui-ci sera dans notre classe Replicator, et nous renseignera sur le nombre total de réplicateurs.

Replicator
categorie id Energie nombre
absorber() agir() ping() creer_reine() legion()

❏ Cette information peut être obtenue en demandant à n'importe quel réplicateur leur nombre (méthode int legion()). Attention aux constructeurs et destructeurs !

La reine est cependant peu mobile, elle a besoin de ses réplicateurs pour se nourrir. Il faut donc permettre aux réplicateurs de pouvoir donner de leur énergie à leur reine.

16. Codons la méthode nourrir_reine()

Cette méthode de chaque réplicateur lui permet de transférer une partie de son énergie à sa reine.

Replicator
categorie id Energie nombre
absorber() agir() ping() creer_reine() legion() nourrir_reine()

Pour les plus avancés, on va maintenant créer un monde (une succession de choses.)

17. Créez un vector de choses

❑ Ce vector sera à remplir de choses. (voir vector, push_back(), pop_back()...)

★ A chaque fois qu'un réplicateur va vouloir absorber une chose, on la retire des choses disponibles sur le monde. Attention aux cas où le monde est vide !

Enfin, chaque reine doit pouvoir savoir quels réplicateurs elle a créé.

18. Ajoutons un attribut vector

- ❏ Ce vector de pointeurs sur réplicateurs, appelé cohorte, contiendra l'adresse de tous les réplicateurs créés par cette reine.

19. Ajoutons une méthode ping_cohorte()

Cette méthode nous permettra de pinger tous les réplicateurs créés par la reine.

Pour aller plus loin :

Nous avons la base de nos réplicateurs, ils peuvent consommer et se reproduire. Libre à vous d'implémenter un monde autour de ça.

Idées :

Faire une carte avec un visuel des réplicateurs. On commence avec un réplicateur et beaucoup d'objets. Le réplicateur doit aller vers les objets pour les consommer. Faire cela lui coûte de l'énergie. Une fois qu'il en a accumulé suffisamment, il peut créer une reine immobile. Celle-ci va utiliser l'énergie que ses réplicateurs lui apportent pour continuer à produire d'autres réplicateurs. Ajouter une IA de recherche des objets sur le monde, et observer les réplicateurs se multiplier.

- ★ PS: Cela constitue une base tout à fait acceptable pour un éventuel projet C++...