

## Examen Réparti 2eme partie

17 Janvier 2019 (2 heures avec documents : tous SAUF ANNALES CORRIGÉES). Barème indicatif sur 23 points (donne le poids relatif des questions)

### Questions de cours

[5 Pts]

Répondez de façon précise et concise aux questions.

#### Question Cours (QC):

QC1) (1 point) Dans la méthode SCRUM, à quoi sert le « Backlog » et de quoi est-il constitué ?

QC2) (1 point) On considère des composants réalisant tous la même interface *IComponent* organisés sous la forme d'un arbre, suivant le DP Composite. Sur un diagramme de composant, représenter un composant *CComposite* interne (qui a des fils) et un composant *CLeaf* terminal (pas de fils).

QC3) (1 point) Définissez la notion de *Domain Specific Language (DSL)* et donner un exemple de DSL.

QC4) (1 point) Représentez et expliquez l'amélioration continue vue sous l'angle de la roue de Deming.

## 2. Problème: Conception SUBOO [Barème sur 18 Pts]

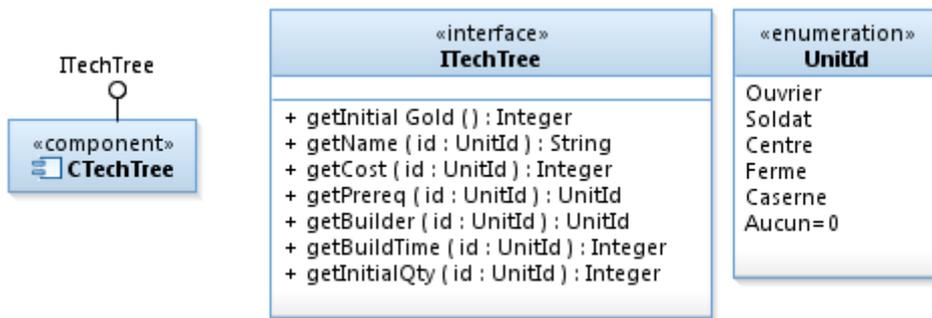
**Rappels d'analyse :** Cet énoncé se base sur une version simplifiée de l'application SUBOO qui a fait l'objet du projet de TME. Nous considérons l'application SUBOO, permettant de générer des *build order* optimisés.

### I. Composant *CTechTree* (5 points)

Les informations décrivant les unités sont présentées dans le cahier des charges sous la forme d'un tableau (NB : cette version est légèrement modifiée par rapport au projet de TME). Initialement chaque joueur dispose de 50 PO et les unités sont :

Unité	Cout PO	Prérequis	Construit par	Temps construction	Initial
Centre	300	Aucun	Ouvrier	200	1
Ferme	100	Aucun	Ouvrier	60	0
Caserne	150	Ferme	Ouvrier	100	0
Ouvrier	50	Aucun	Centre	30	5
Soldat	100	Aucun	Caserne	50	0

Ce composant bout de chaîne stocke donc les informations relatives aux unités disponibles dans le jeu et à l'état initial de la partie. On pourra faire varier sa réalisation si besoin pour traiter l'évolution des versions du jeu. Pour plus de lisibilité, on introduit une énumération pour identifier les types d'unités disponibles plutôt qu'un entier ou un String; il est donc possible pour un composant arbitraire de connaître la liste d'unités disponibles en inspectant simplement l'énumération. On propose de réaliser le composant *CTechTree* suivant :



1. (1,5 point) Sur un diagramme de classes, donner une conception détaillée possible de ce composant *CTechTree*.

On souhaite mettre en place des tests d'intégration de ce composant.

2. (0,75 point) Représentez un ou plusieurs composants (testeur, bouchon(s)) permettant de mettre en place ces tests.

3. (0,75 point) Sur un diagramme de structure interne, représentez une configuration des composants pour le test.

4. (1 points) Sur un diagramme de séquence où vous utiliserez une ligne de vie par instance identifiée en 3), montrez les échanges nécessaires pour découvrir que la construction d'une caserne nécessite une ferme (prérequis technologique de caserne) et un ouvrier (pour construire la caserne) et coûte 150 PO. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.

5. (1 point) Expliquez comment écrire en pratique le testeur et un test d'intégration qui couvre ce scénario, en supposant l'existence d'une *factory* statique **ComposantFactory** pour la construction des composants.

## II. Composant *CGameState* (5,5 points)

On souhaite à présent construire un composant représentant l'état d'une partie. L'état d'une partie se résume à une date (en secondes depuis le début de la partie), à la quantité d'or (PO) disponible à cet instant et aux instances d'unité disponibles. Chaque instance d'unité est associée à son type, ainsi qu'à l'un des trois états « En construction », « Libre » ou « Occupé », et à un compteur qui décompte le temps restant avant que l'unité ne soit libre (0 si elle est déjà libre).

1. (2,25 points) Sur un diagramme de composants, représentez un composant bout de chaîne *CGameState* et son interface offerte *IGameState* répondant à ce besoin (avec signatures complètes).

L'API doit supporter :

- La consultation et la mise à jour du total d'or disponible de façon arbitraire,
- La consultation de la date
- L'ajout d'instances d'unités arbitraires (type, état, temps restant),
- Accès en lecture : déterminer le nombre et l'état de chacune des instances d'unité
- Accès en écriture : mettre à jour l'état et le temps restant pour une instance d'unité
- Une opération *step()* qui laisse une seconde s'écouler et met à jour l'état des unités (-1 aux compteurs de temps restant des instances, retour à l'état libre si 0 atteint) et l'or disponible (+1 par ouvrier libre).

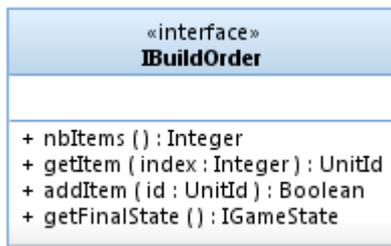
Sans que ce soit imposé on suggère d'utiliser une API relativement basique, basée sur *size()* donnant le nombre d'instances et d'un accès indicé par l'index de l'instance à l'état des instances (indices entre 0 et *size()-1*).

2. (1 point) On considère un composant **CInitialBuilder** chargé de créer l'état initial (une instance de *CGameState*) d'une partie : il offre une opération *createInitialState() :IGameState*. Représentez ce composant sur un diagramme de composants.

- (2,25 point) Sur un diagramme de séquence de niveau intégration, modélisez l'invocation de `createInitialState()` par un composant arbitraire `CComp` (jouant le rôle d'acteur). On suppose que l'état initial d'un `CGameState` est vide (aucune instance d'unité et zéro PO). On fera attention à être cohérent avec l'API proposée en 1) et celle de `ITechTree`. On utilisera des commentaires plutôt que de répéter des messages trop similaires. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.

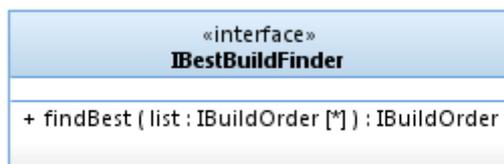
### III. Composant `CBuildOrder` (7,5 points)

On considère à présent la représentation et la manipulation d'un *build order* (BO), c'est-à-dire une séquence ordonnée de constructions d'unités. L'objectif est de pouvoir mesurer le temps que prend un BO, c'est-à-dire la date à laquelle on arrive à exécuter la dernière instruction du BO.

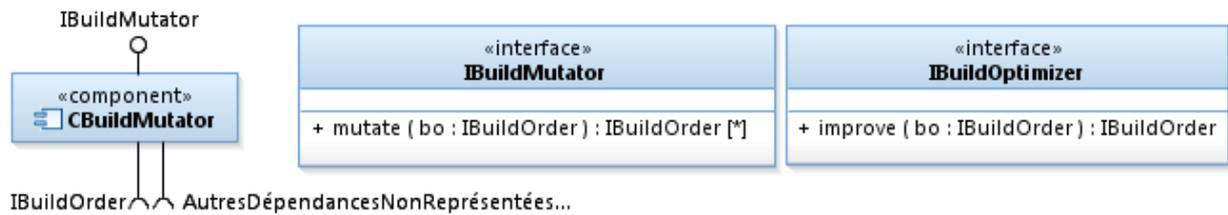


Pour cela, le composant `CBuildOrder` démarre vide à partir de l'état initial du jeu (un `IGameState` cf. questions précédentes). L'invocation à « `addStep(id : UnitId) : Boolean` » vérifie que l'état actuel contient bien au moins une unité du type du « constructeur » et une unité du type « prérequis ». Si cette condition n'est pas satisfaite, l'opération rend `false`. Si les prérequis sont satisfaits, on fait progresser l'état du jeu (à l'aide de `step()`) jusqu'à ce que les ressources en or soient suffisantes pour acheter l'objet, qu'au moins une unité de type « constructeur » soit libre, et que le prérequis éventuel soit dans un état différent de « en construction ». La nouvelle unité est alors ajoutée à l'état courant, dans l'état « en construction », avec un temps d'attente égal à son temps de construction. On mobilise également une instance qui est le « constructeur » de l'unité, qui passe à l'état « Occupé » pendant le temps de construction.

- (1 point) Sur un diagramme de composants, représentez ce composant `CBuildOrder`.
- (1 point) On souhaite comparer des BO qui ont été complètement construits. Dans un premier temps la métrique utilisée sera simplement la date de l'état final du BO. Représentez sur un diagramme de composants `CBestBuildFinder` qui offre ce service.



- (1,5 points) Sur un diagramme de séquence de niveau interaction entre composants, représenter un scénario où un composant arbitraire `CComp` (jouant le rôle d'un acteur) recherche le meilleur parmi deux BO à l'aide du composant `CBestBuildFinder`. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.
- (1 point) On vous fournit un composant `CBuildMutator` capable de proposer des variations autour d'un BO « de base ». Il offre l'interface `IBuildMutator` (cf. figure), et dépend d'autres composants qui ne seront pas détaillés dans cet énoncé.



En appui sur les composants développés dans cette partie III, on veut réaliser un composant d'optimisation *CBuildOptimizer* satisfaisant l'interface *IBuildOptimizer*. Représentez ce composant *CBuildOptimizer* sur un diagramme de composants.

5. (1,25 point) Sur un diagramme de séquence de niveau interaction entre composants, représenter un scénario où un composant arbitraire *CComp* (jouant le rôle d'un acteur) traite l'optimisation d'un BO donné. On ne détaillera pas les interactions qui ont déjà été décrites dans d'autres séquences de cet énoncé. On fera attention à bien porter des valeurs (arguments, valeur de retour) sur tous les messages.
  6. (1,75 point) Pour chacun des composants suivants, dites combien d'instances du composant seront présentes dans une configuration nominale du système SUBOO : *CTechTree*, *CTesterTechTree*, *CTechTreeBouchon*, *CGameState*, *CInitialBuilder*, *CBuildOrder*. Justifiez vos réponses succinctement.
-