

LIFAP4 : un corrigé des exercices types « Conception »

Vous avez la charge du développement des applications de conception de modèles de Duplo. Les Duplo sont une version simplifiée des Lego mais le principe est le même : différentes pièces sont assemblées pour former un jouet.

Les pièces de Duplo diffèrent par leur forme en 2D, leur épaisseur, et leur couleur (voir l'image ci-dessous). Pour l'affichage de la notice de montage, il peut être pratique d'avoir une représentation 3D de la pièce. Nous supposons ici que vous disposez des classes d'un moteur 3D : *Maillage3D* spécifiant tout pour l'affichage 3D ; *Translation3D/Rotation3D* spécifiant la position et orientation d'un objet 3D.

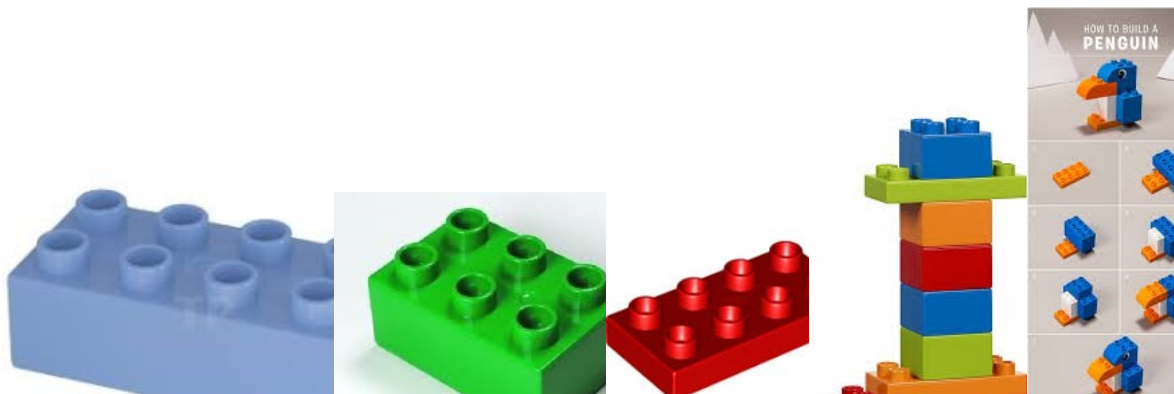
Un jouet est un assemblage dans un ordre précis de différentes pièces. Une version simple de l'assemblage consiste à ajouter une unique pièce à chaque étape. Une version plus évoluée doit permettre de construire des morceaux complexes du jouet séparément et de les assembler ultérieurement. Précisez clairement laquelle des solutions vous proposez car cela change la structure de données utilisée.

Vous devrez concevoir deux applications. La première est celle utilisée par les concepteurs des jouets devant leur station de travail. Elle doit permettre de gérer la conception de toutes les boîtes de jouets du catalogue de la marque en proposant une interface pratique, ergonomique et adaptée. A partir des données de votre application, on doit pouvoir fabriquer en usine une boîte qui va contenir le sachet de pièces et le mode d'emploi de fabrication montrant les différentes étapes de montage. La deuxième application est à destination des enfants ; elle leur permettra de consulter la notice de montage sur leur tablette mobile.

Concevez le diagramme des classes de ces deux applications. Vous êtes libre de constituer les classes de votre choix, avec les données de votre choix, voire d'ajouter toutes données qui vous sembleraient pertinentes, comme des identifiants.

Pour chaque module du logiciel, vous donnerez les classes : données et fonctions membres avec leurs paramètres (mode et type). Soyez pertinent sur les mutateurs et accesseurs, réfléchissez plutôt en « actions » de haut niveau. Pour des raisons pratiques de présentation sur votre feuille, il vous est possible d'écrire les fonctions membres en dehors du diagramme. Précisez alors clairement à quel module/classe elles appartiennent.

Remarque : on ne demande pas d'algorithme détaillant chaque partie mais plutôt que chaque module prévoit les fonctions nécessaires.



Un exemple de 3 pièces, de jouets et de notice de montage.

De nombreuses règles, remarques, explications à propos de ce corrigé

- Les couleurs :
 - en blanc, les classes noyaux des applications ;
 - en bleu les classe codant les applications, qui comportent les éléments de l'IHM ;
 - en mauve, les classes génériques, souvent des classes outils, essentiellement des « conteneurs », ici ceux de la STL.
- Dans les 2 classes IHM/App : il manque de nombreux éléments graphiques/widget. Ces classes seront sûrement éclater en plus qu'une classe dans la pratique. Mais il est important d'avoir au moins une classe de présente !
⇒ Ceux sont souvent ces classes qui manquent dans les réponses aux examens
- Il est important ici d'avoir des identifiants/clés/références pour identifier les objets. Dans le cas de ce sujet, des pièces de Duplo identique ne doivent pas avoir leurs données dupliquées. On les assemble en prenant des instances. Cette factorisation des données permet d'éditer les propriétés d'une pièce sans devoir les changer partout, mais uniquement à un endroit.
⇒ Ce point est souvent une lacune dans les réponses aux examens.
- Les fonctions importantes qui sont souvent non présentent dans vos conceptions sont
 - Les constructeurs sont importants car ils sont là pour initialiser les données !
 - Sauvegarde/chargement depuis un fichier ou depuis/vers l'écran. Notion de sérialisation des données. Dans ce corrigé, nous les avons regroupés avec les operator<< et operator>> qui peut afficher à l'écran ou sauver dans un fichier texte.
 - Les fonctions de recherche, d'ajout d'un élément. Ici il s'agit de pouvoir ajouter une étape au montage qui est le minimum indispensable.
- Il ne faut mettre des accesseurs et des mutateurs pour toutes les données membres. Il est important de raisonner en fonctions (actions) explicites qu'offre une classe. Un mutateur set/get ne doit pas être juste un contournement du coté privé d'une donnée. Il faut que la fonction apporte une réelle action, ou au moins une vérification. Il ne faut pas de set/get en masse !
⇒ Ce point est souvent une lacune dans les conceptions de LIFAP4
- Dans ce corrigé, l'assemblage est linéaire, une pièce l'une après l'autre. Ce qui se traduit par un tableau dans la classe Montage : `noticeMontage: vector<Assemblage>` ; Pour faire un assemblage de sous-partie indépendamment, il faut transformer ce tableau en arbre. Chaque sous branche est alors une sous partie du montage total.
- Les classes Transformation3D/Vec3D sont écrites explicitement dans la partie Core. Elles sont différentes de la partie du moteur 3D dans le namespace M3D. Ceci pour garantir l'indépendance du noyau par rapport à n'importe quel moteur 3D.
- Le bon conteneur (list, vector, deque, map) dépend de la complexité du problème. En LIFAP4, nous n'insistons pas énormément sur ce point mais il faut le garder en tête. L'avantage d'une map (dictionnaire) est sa bonne complexité (log(n)) pour ajouter et retrouver des éléments.
- Il était possible d'avoir un catalogue de jouet chargé en mémoire. Il faut alors ajouter une classe Catalogue et changer dans les 2 classes d'applications les données membres.

