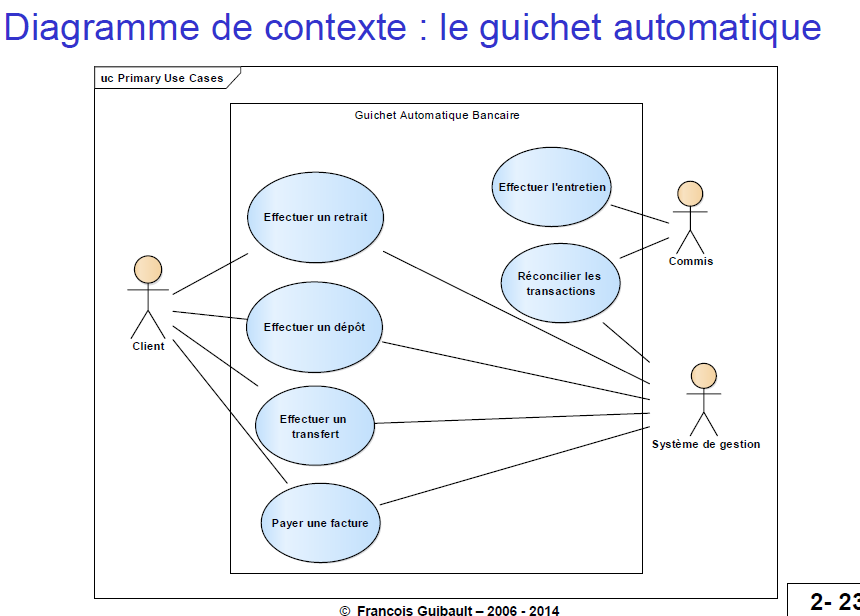
# Les Questions d’examen de LOG2410 les plus fréquemment posé par section :

## Processus de conception et cas d'utilisation

* **Est un acteur pour ce module ? Dans le contexte des cas d'utilisation, définissez en vos propres mots ce que l’on entend, d’une part, par « acteur » et, d’autre part, par « partie prenante ». Expliquez la différence entre les deux.**

Acteur: Ensemble de rôles interprétés par les utilisateurs du système lorsque ceux-ci interagissent avec le système à travers les cas d’utilisation. Il s’agit d’une entité externe au système qui interagit avec celui-ci.

Parties prenantes et intérêts: Liste des personnes qui sont impliqués dans le cas d’utilisation, ce qu’elles souhaitent et de quelle façon elles sont affectées par la réalisation du cas.

* Tracez le diagramme UML de haut niveau (diagramme de contexte) des cas d'utilisation

Le diagramme de contexte permet de visualiser les cas d’utilisation primaires d’un système.

•Ceci permet de voir rapidement les principales fonctions d’un système.

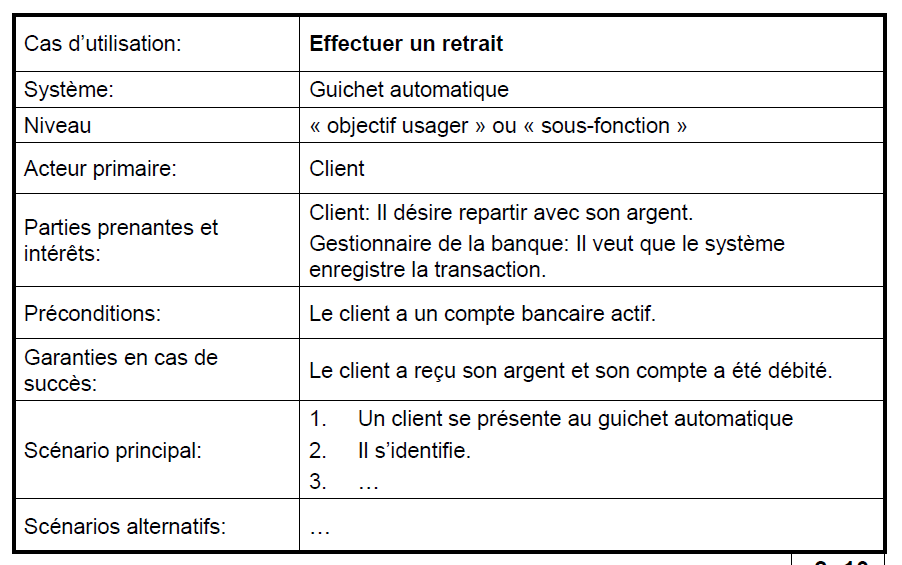
•Plus particulièrement, le diagramme de contexte définit:

•Les limites du système modélisé.

•Les principaux acteurs du modèle.

•Les cas d’utilisation primaires.

* **Choisissez l’un des cas d’utilisation que vous avez identifié en b) et spécifiez-le en version étendue, en vous limitant au scénario principal**



* **Relation entre les cas d’utilisation**

On ajoute des relations entre les cas d’utilisation pour améliorer la compréhension et réduire les duplications

•Relations d’utilisation

* Association stéréotypée « include».

La relation la plus commune et la plus utile. La relation include permet de modéliser des sous-fonctions qui peuvent être réutilisées.

•Relations d’extension

* Association stéréotypée « extend».

La relation d’extension permet d’ajouter des étapes à un cas existant sans modifier le cas original.

•Relation de généralisation.

La relation de généralisation permet de hiérarchiser les cas des plus abstraits vers les plus concrets.

* **Dans un diagramme de concepts, faut-il plutôt sur spécifier ou sous spécifier les concepts ? Faut-il sur spécifier ou sous spécifier les relations ? Justifiez chaque réponse (à noter qu’aucun point ne sera accordé pour une réponse non justifiée).**

L’analyse orientée objet est la décomposition du problème en concepts individuels. Le modèle conceptuel est une représentation des éléments du monde réel, pas de composantes logicielles. Il permet de clarifier la terminologie & le vocabulaire.

Pour aborder l’analyse de systèmes logiciels complexes, une stratégie courante consiste à subdiviser l’espace du problème pour mieux appréhender la complexité:

-Approche de «diviser pour régner»,

-Une distinction fondamentale entre l’approche d’analyse orientée objet et l’approche structurée est le fait que l’approche orientée objet utilise les concepts (objets) comme base de division alors que l’approche structurée utilise les fonctions.

Construire un modèle conceptuel ressemble à un processus de cartographie, on doit :

–Utiliser les noms existants,

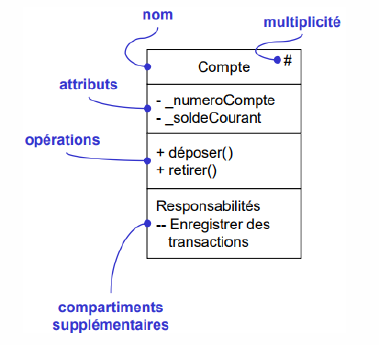
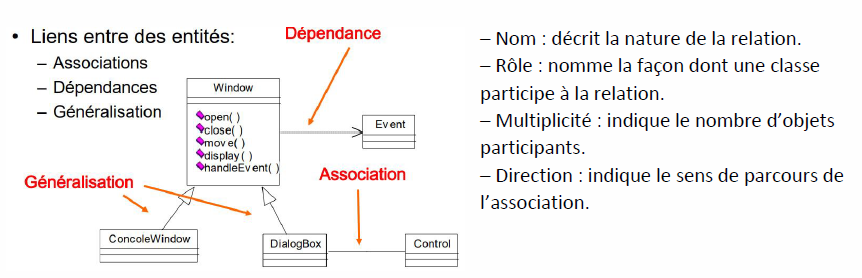
–Exclure les détails inutiles,

–Ne pas ajouter d’éléments inexistants.

– Un concept: la référence à un élément du monde réel, issu du domaine du problème. Le terme concept n’est pas utilisé ni défini en UML. Une classe est la description d'une collection d'objets qui partagent les mêmes attributs, les mêmes opérations, les mêmes relations et la même sémantique.

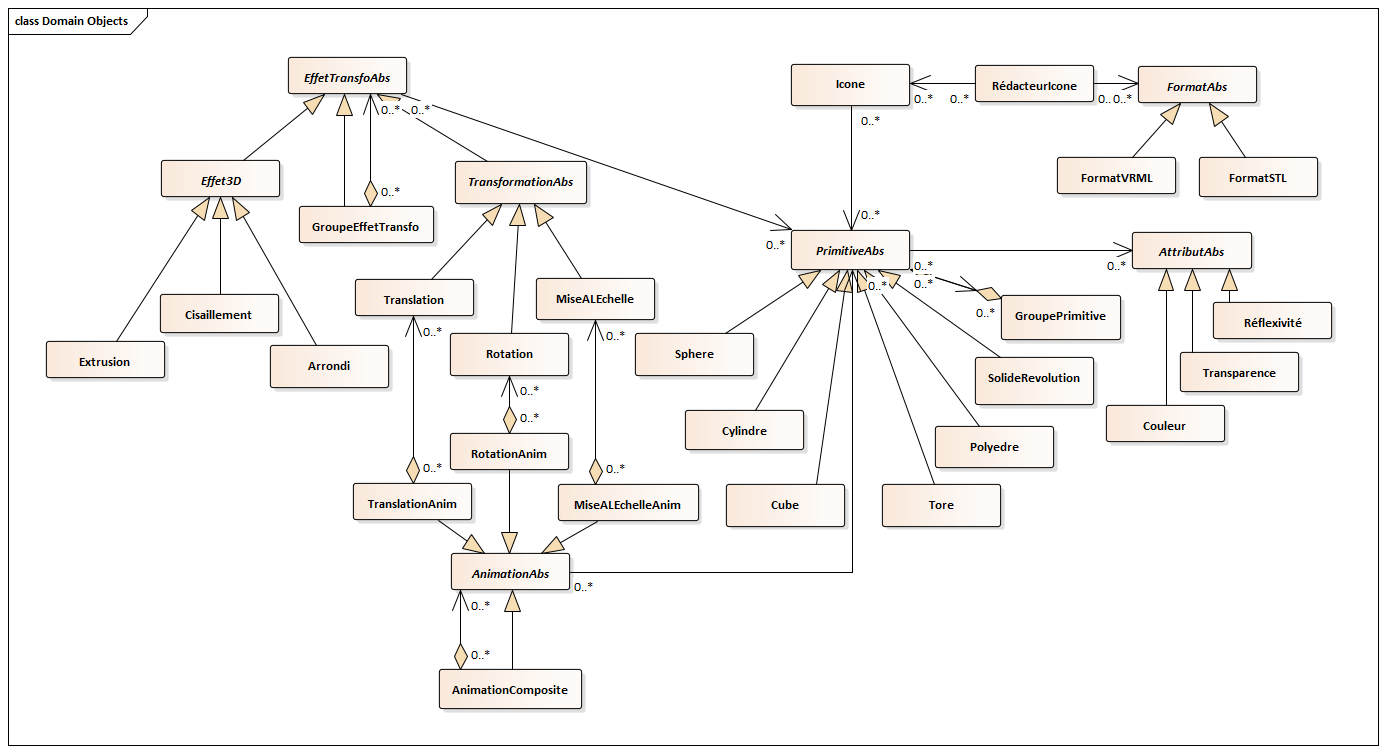
– Une classe: l’implémentation logicielle d’un concept, avec des attributs et des méthodes. Le terme classe est défini en UML

– Un type: similaire à une classe, mais sans les méthodes. Un type, en UML, est indépendant du langage.



L’erreur la plus courante lors de la construction d’un modèle conceptuel est probablement le fait de représenter sous forme d’attribut quelque chose qui aurait dû être représenté comme un concept:

* **Proposez un diagramme de concepts**



* **Dans son livre "UML 2 et les design patterns", Larman décrit en détail les patrons GRASP. Parmi ceux-ci, les patrons Faible couplage et Cohésion élevée jouent tous les deux un rôle important afin d'assurer la qualité globale d'une conception logicielle. Décrivez chacun de ces patrons et donnez une conséquence positive pour chacun.**

Les patrons GRASP (General Responsability Assignment Software Patterns) décrivent des principes fondamentaux d’assignation des responsabilités à des objets. La découverte des responsabilités se fait souvent durant la construction des diagrammes d’interaction. Les patrons fournissent des solutions à des problèmes courant de conception.

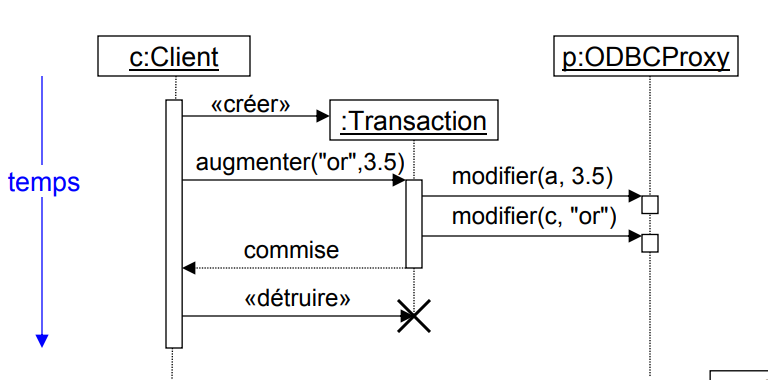
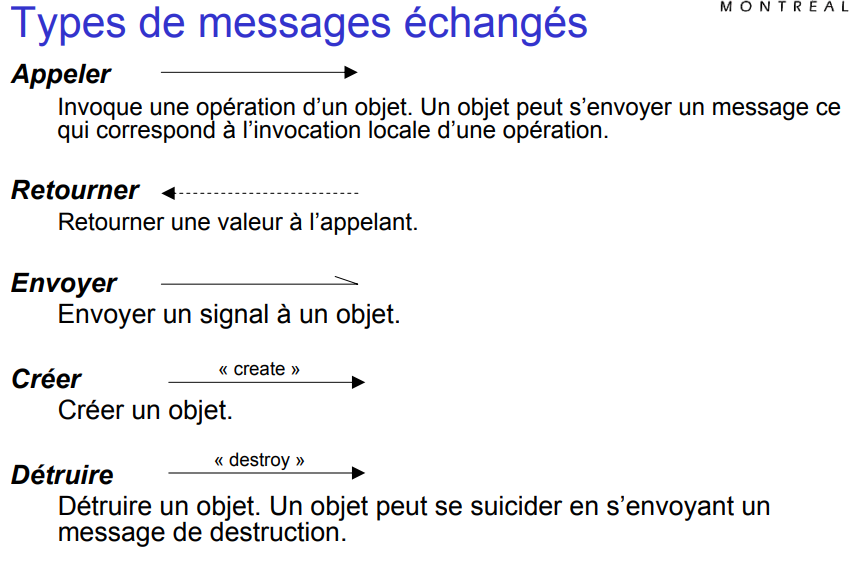
GRASP: des patrons pour l'assignation de 
responsabilités 
Neuf patrons d'assignation de responsabilité: 
Expert: donner la responsabilité à l'objet le mieux en mesure d'y satisfaire, 
• Créateur: donner la responsabilité de créer un objet à un autre objet directement impliqué 
dans une relation avec l'objet créé, 
• Couplage faible: assigner les responsabilités de façon à ce que le couplage reste faible, 
• Cohésion élevée: assigner les responsabilités de façon à ce que la cohésion reste élevée 
• Contrôleur: assigner la responsabilité de gérer les messages d'évènements du système 
une classe spécialisée du modèle, 
• Polymorphisme: utiliser des interfaces pour traiter des altematives qui dépendent du 
des objets, 
• Indirection: assigner des responsabilités à un objet intermédiaire pour éviter que des 
classes soient couplées, 
• Pure Fabrication: assigner un ensemble de responsabilité à une classe créée 
artificiellement, 
• Variation protégée: identifier des points d'instabilité ou de variation et développer des 
interfaces stables autour de ces points. 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GRASP - (General Responsability Assignment Software Patterns) décrivent des principes fondamentaux d’assignation des responsabilités à des objets. Ils sont utilisés comme guide pour assigner correctement les responsabilités aux différentes classes lors du passage vers la conception** | | | | |
| **Patrons** | **Intention** | **Applicabilité** | **Conséquences** | **Exemple** |
| Expert | Donner la responsabilité à l’objet le mieux en mesure d’y satisfaire, | La prise en charge d’une responsabilité requiert de l’information distribuée parmi plusieurs objets ou classes, ce qui donne lieu à plusieurs « experts partiels » qui collaborent pour remplir complètement la responsabilité. | +Favorise l’encapsulation, puisque les objets utilisent leur propre information pour satisfaire la responsabilité,  +Supporte un couplage faible entre les classes, ce qui mène à des systèmes plus robustes et plus faciles à maintenir,  + Distribue le comportement parmi les classes qui possèdent l’information nécessaire: ce qui encourage des définitions de classes plus cohérentes et plus faciles à comprendre. |  |
| Créateur | Donner la responsabilité de créer un objet à un autre objet directement impliqué dans une relation avec l’objet créé | Le patron Créateur guide dans l’assignation de la responsabilité de création d’objets, une tâche très commune dans les systèmes orientés-objets. L’idée de base du patron Créateur consiste à trouver une classe qui a déjà d’excellentes raisons d’être associée à la classe à créer. | + Supporte un couplage faible entre les classes, ce qui mène à moins de dépendances et plus de possibilités de réutilisation. Le couplage n’est probablement pas augmenté, puisque le Créateur a déjà d’excellentes raisons d’être connecté à la classe des objets qu’il crée. | Transaction: le patron Créateur suggère de chercher une classe qui contient, enregistre, etc. des transactions. Puisque Compte enregistre des transactions, Compte pourrait être un bon candidat: |
| Couplage Faible | Assigner les responsabilités de façon à ce que le couplage reste faible | Le couplage est une mesure du niveau de connexion d’une classe avec les autres classes du système. Une classe ayant un Faible couplage n’est pas connectée à un « trop » grand nombre d’autres classes. Le Faible couplage est un principe de conception qui doit toujours être présent à l’esprit des concepteurs, il s’agit d’un patron d’évaluation d’une conception. | Le Faible couplage mène à la conception de classes qui sont plus indépendantes, ce qui réduit l’impact de changements, et qui sont plus réutilisables, ce qui améliore les possibilités de gains de productivité.  – Le couplage ne peut pas être considéré de façon isolée, mais doit être évalué en même temps que d’autres principes comme Expert et Forte Cohésion dans l’évaluation d’une conception,  – Le couplage n’est peut-être pas important si la réutilisation n’est pas un objectif.  – Le cas extrême du Faible couplage, où une classe n’est connectée à pratiquement aucune classe n’est pas désirable, puisqu’il mène à des classes ayant une très faible cohésion, qui assume toutes les responsabilités à l’interne. |  |
| Couplage Élevée | Assigner les responsabilités de façon à ce que la cohésion reste élevée | La cohésion est une mesure du niveau de lien qui existe entre les différentes fonctions d’une classe. Si toutes les responsabilités d’une classe sont reliées les unes aux autres, la cohésion de la classe est forte. La Cohésion élevée est un principe de conception qui doit toujours être présent à l’esprit des concepteurs, il s’agit d’un patron d’évaluation d’une conception | + La cohésion élevée d’une classe augmente la clarté et la facilité de compréhension de la classe,  + La cohésion élevée facilite la maintenance et les améliorations,  + Aide à maintenir le couplage bas,  + La granularité fine de l’assignation des responsabilités augmente la possibilité de réutilisation, étant donné qu’une classe ayant un haut niveau de cohésion peut être utilisé dans un but bien précis. |  |
| Contrôleur | Assigner la responsabilité de gérer les messages d’évènements du système à une classe spécialisée du modèle, | Le patron Contrôleur s’applique dans pratiquement tous les systèmes orientés objet qui doivent traiter des événements externes impliquant par exemple un usager ou des systèmes logiciels ou physiques externes (senseurs, commutateurs, etc.). Dans tous les cas où une conception orientée objet est utilisée, une classe Contrôleur doit être choisie pour traiter les événements. Le patron Contrôleur constitue un guide pour choisir une classe acceptable. | + Centralise la gestion des événements dans une classe bien identifiée,  + Augmente les possibilités de réutilisation en s’assurant que les processus liés au domaine d’application sont traités au niveau de la couche du modèle et non de la couche de présentation,  + Permet de s’assurer de la cohérence séquentielle des événements en fournissant une classe unique pour traiter les événements, qui peut s’assurer qu’ils se produisent dans le bon ordre.  – Il est facile d’assigner trop de responsabilités au Contrôleur, ce qui mène à une classe ayant peu de cohésion, et difficile à comprendre,  – Lorsqu’un Contrôleur assume trop de responsabilités, on peut le décharger en créant des Contrôleurs spécialisés, ou en s’assurant que le Contrôleur délègue la gestion de chaque événement à un objet spécialisé. | Deux approches sont envisageables pour solutionner le problème d’un Contrôleur trop complexe:  1. Déléguer le traitement de certains événements à d’autres contrôleurs: patron « Chain of responsability »,  2. Traiter chaque événement système comme un objet autonome: patron « Command » |
| Polymorphisme | Utiliser des interfaces pour traiter des alternatives qui dépendent du type des objets, | Un autre principe fondamental de l’approche orientée-objet.  Dès que le comportement d’objets reliés les uns aux autres doit varier en fonction du type concret de l’objet.  Lorsqu’il serait nécessaire de tester le type concret de l’objet et d’utiliser des énoncés conditionnels ou des énoncés switch/case | + Permet d’ajouter facilement les extensions requises pour de nouvelles variations.  + De nouvelles implémentations peuvent être ajoutées sans affecter les clients.  – Permet d’ajouter des interfaces ou des points de variation en prévision de développements futurs qui ne se réaliseront peut-être pas.  – Ajouter seulement les points de variation pour des alternatives qui sont confirmées par la conception du système et éviter d’ajouter des points de variation superflus. |  |
| Indirection | Assigner des responsabilités à un objet intermédiaire pour éviter que des classes soient couplées, | Vieil adage:  « La plupart des problèmes en informatique peuvent être solutionnés par un niveau supplémentaire d’indirection. »  Et son corolaire:  « La plupart des problèmes de performance peuvent être solutionnés en éliminant une couche d’indirection. » | + Permet de réduire le couplage entre des classes ou des composants,  – Augmente le nombre de classes,  – Réduit la performance. |  |
| Pure Fabrication | Assigner un ensemble de responsabilité à une classe créée artificiellement, | Lorsque l’on veut regrouper dans une classe un ensemble de méthodes qui sont liées entre-elles, mais qui ne correspondent pas directement à un concept du domaine d’application. Lorsqu’un ensemble de comportements doivent être extrait d’une classe et regroupé afin de maintenir une cohésion élevée. | + Permet de factoriser dans une nouvelle classe un ensemble de comportements qui méritent d’être regroupés,  + Permet de maintenir une cohésion élevée de la classe d’où sont extraits les comportements et de la nouvelle classe créée,  – Créer trop de classes strictement basées sur le comportement tend vers une décomposition plus fonctionnelle qu’orientée-objet,  – Une surabondance de classes fabriquées tend à éloigner les données et les comportements qui leur sont associés, ce qui contredit le patron Expert et affecte négativement le couplage. |  |
| Variation protégée | Identifier des points d’instabilité ou de variation et développer des interfaces stables autour de ces points. | Il s’agit d’un principe de conception fondamental, très important dans n’importe quel contexte de résolution de problème. Principe de base derrière des principes tels que l’encapsulation des données, les interfaces, le polymorphisme et les indirections. | Simplifie l’ajout d’extensions nécessaires pour de nouvelles variations,  + De nouvelles implémentations peuvent être ajoutées sans affecter les clients,  + Réduit le couplage,  + Permet de réduire l’impact ou le coût de changements,  – Favorise l’ajout de points d’évolution pour des alternatives qui ne se concrétiseront peut-être pas,  – Favorise l’ajout de flexibilité non requise. |  |

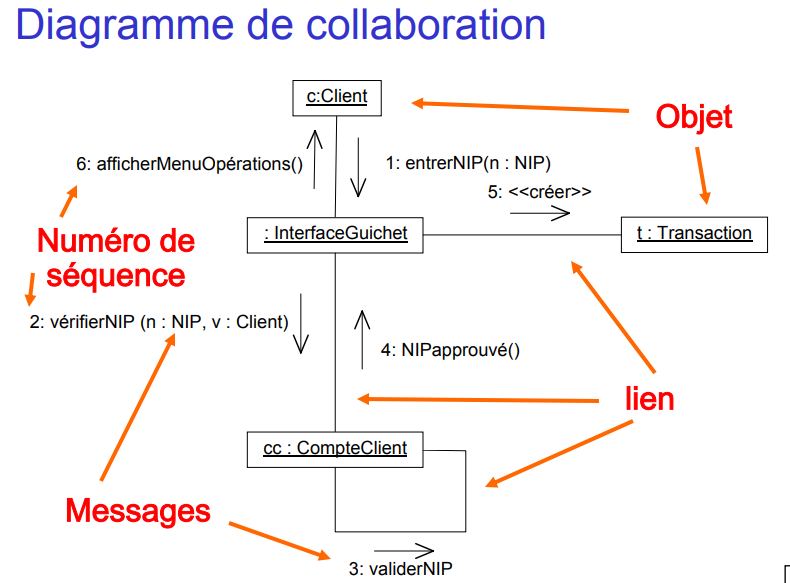
* **Diagramme d’interaction**

Deux types de diagrammes d’interaction Les deux types de diagrammes sont strictement équivalents l’un à l’autre, ils contiennent la même information. Toutefois, il ne présente pas l’information de la même façon.

->Diagramme de séquence: Se concentre sur l’ordre séquentiel des messages.



­>Diagramme de collaboration: Se concentre sur l’organisation structurelle dans laquelle les objets s’envoient et reçoivent des messages.



1. **Architecture logique**

* Indiquez deux avantages importants de la décomposition d'un logiciel en une architecture multi-niveaux.

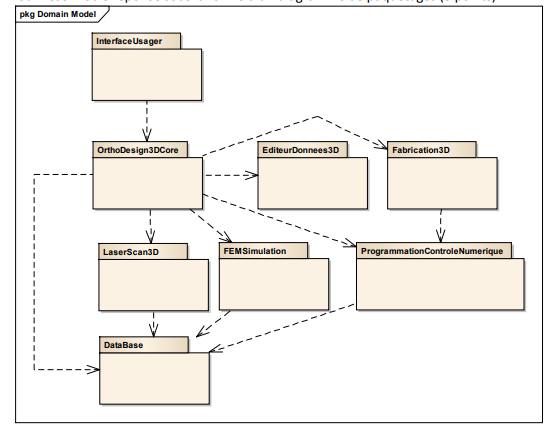
Au moins deux parmi :

o La séparation de la logique d’application dans des composantes séparées qui peuvent être réutilisé dans d’autres systèmes,

o La possibilité de répartir les niveaux sur différents nœuds de calcul, et dans différents processus,

o L’assignation de développeurs à la construction de chaque niveau: parallélisassions des efforts et spécialisation des intervenants.

* Diagramme de paquetages. Proposez une décomposition architecturale multi-niveaux du logiciel PolySonor. Fournissez votre réponse sous la forme d'un diagramme UML de paquetages



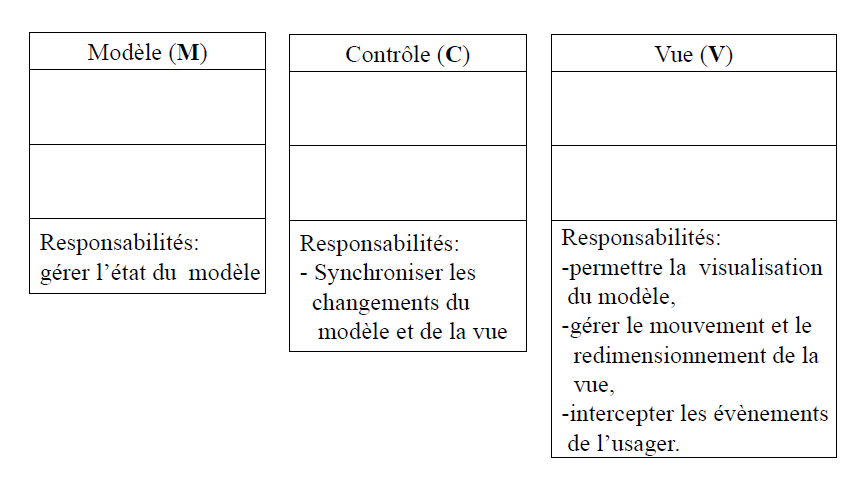
* Expliquez en quoi consiste le modèle MVC et établissez la correspondance entre le modèle MVC et l’architecture multi-niveaux proposée ci-dessus, en expliquant à quel niveau architectural appartient chaque composante du modèle MVC

Modèle-vue-contrôleur ou MVC est un motif d'architecture logicielle destiné aux interfaces graphiques lancé en 1978 et très populaire pour les applications web. Le motif est composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs.

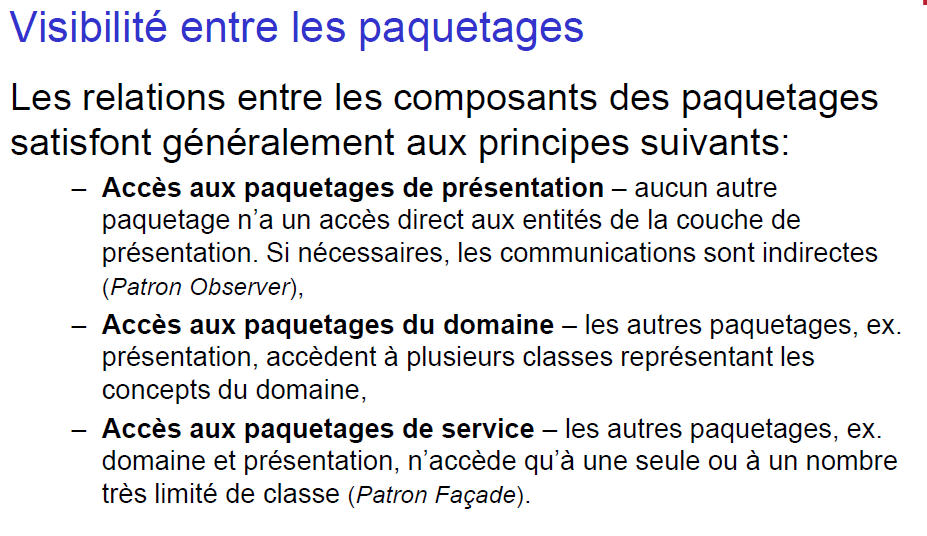
Un modèle (Model) contient les données à afficher.

Une vue (View) contient la présentation de l'interface graphique.

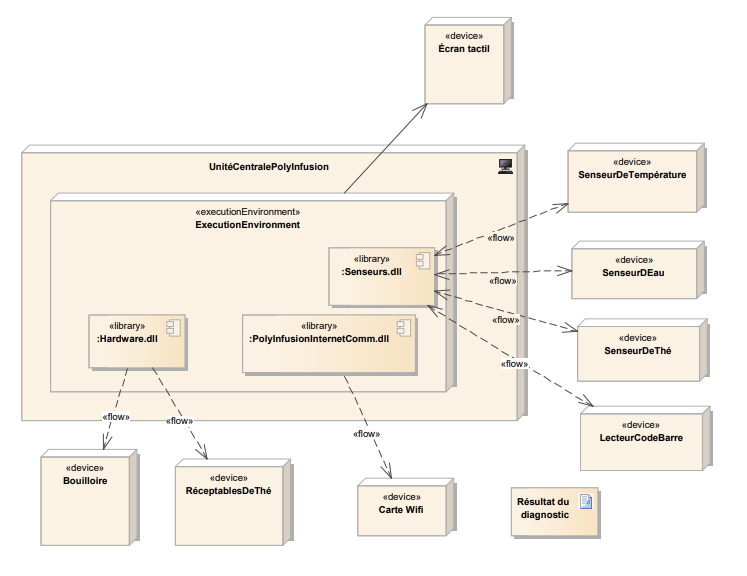
Un contrôleur (Controller) contient la logique concernant les actions effectuées par l'utilisateur.



* Expliquez les règles spécifiant quelles communications sont permises entre les composantes de l’architecture et quelles communications sont interdites



* Tracez le diagramme de composantes du système PolyIcone3D, en ne représentant que les composantes de déploiement



* Expliquer le rôle de la Façade et discutez des avantages liés à son utilisation

Une façade fournit un point d'entrée clairement identifiable et simplifie l'utilisation d'un module. L'utilisation d'une façade simplifie le remplacement d'un module par un autre en isolant les clients des détails internes du module. La façade augmente donc la modularité du système.

1. **Patrons de conception**

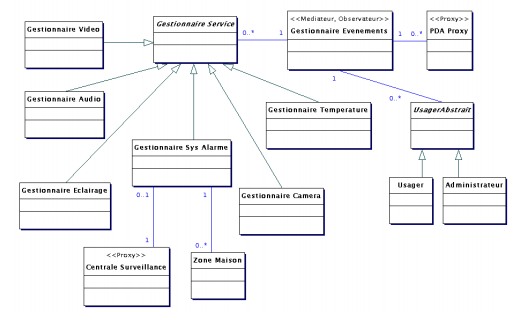
* Expliquer en vos propres mots **l’intention du patron** suggéré, faire un **diagramme de classe annoté avec les rôles identifiés** et spécifier un **avantage et inconvénient** liés à l’utilisation du patron suggéré.
  + **patron Composite**
  + **patron Visiteur**
  + **patron Template Method**
  + **patron Commande.**
  + **patron Stratégie**
  + **patron State**
  + **patron Observateur**

**VOIR DOCUMENT RÉSUMÉ**

* Dans le contexte du patron de conception Décorateur, expliquez en quoi consiste le problème « d’identité d’objet » que l’on rencontre fréquemment lors de son utilisation.

1. **Conception de classe**

* Illustrez sous la forme d’un diagramme de classe UML la relation que vous proposez d’établir entre les classes



1. **Gestion des ressources**

* Indiquez les numéros de lignes à partir desquelles des exceptions sont susceptibles de se propager en identifiant les méthodes où les opérateurs qui pourraient lancer une exception.

* Pour chacune des lignes où une exception peut se propager, indiquez dans quel état l’objet Ntuplet serait laissé et indiquez si des ressources seraient perdues.

– Assignations partielles. Cette architecture ne permet que les assignations partielles. Il est impossible d’assigner complètement un objet de la classe B dans un autre objet de la classe B (pas plus qu’un C dans un C) puisqu’on utilise un opérateur d’assignation qui reçoit une référence à un objet de la classe de base A. – Assignations mixtes. Cette architecture permet les assignations mixtes. À partir de l’opérateur d’assignation défini dans la classe de base, rien n’empêche un client d’assigner un objet de la classe B dans un objet de la classe C et vice-versa.

???

Réponse : Cela résout partiellement le problème puisqu’on peut maintenant faire des assignations complètes homogènes en passant par des références aux objets des sous-classes (B b1,b2 ; b1=b2). Toutefois, si on passe par des références à la classe de base, les assignations seront encore partielles. En outre, avec cette solution, les assignations mixtes restent permises. Ce n’est donc pas une solution valable.

* Proposez une version améliorée de la classe Ntuplet et de son opérateur d’assignation afin de corriger les problèmes identifiés en b) et justifiez votre solution À noter : vous pouvez vous contenter d’indiquer uniquement les lignes qui sont modifiées, en précisant les modifications que vous proposez.

Pour résoudre le problème, il faudrait rendre la classe de base abstraite et protéger son opérateur d’assignation de façon à éviter les assignations mixtes tout en permettant aux classes dérivées d’utiliser l’opérateur de la classe de base. Comme la classe A est nécessaire, on doit donc ajouter une nouvelle classe qui jouera le rôle de la classe de base abstraite. Les opérateurs d’assignation ne doivent pas être virtuels

**VOIR DOCUMENT RÉSUMÉ**