Travail pratique #3 - Une évaluation du document d'architecture

INF8301 – Ingénierie en qualité logicielle

Simon Delisle - 1538886
Félix Gingras Harvey - 1490242
François Pierre Doray - 1529405
Alexandre Vanier - 1525768

«La musique sans support physique»

Facteurs de qualité importants pour la discipline et pour le client

Flexibilité

Notre projet nécessite l'utilisation de plusieurs technologies nouvelles dans l'industrie du jeu vidéo et des interfaces naturelles. Étant donné que nous ne pouvons pas aisément juger du potentiel et des limites de chaque technologie en avance, le document d'architecture doit être structuré de manière à faciliter les changements et offrir une certaine souplesse au design.

La flexibilité est importante pour nous et notre client, car l'évolution des prototypes et possiblement celle des requis seront de potentielles causes de changement technologies et peut-être de design, d'où l'importance, par exemple, de présence de façades face aux parties risquées comme les librairies propres à certains capteurs, permettant ainsi leur remplacement.

Testabilité

La testabilité du logiciel est un facteur très important, car sans celle-ci certains requis pourraient ne pas être testés exhaustivement ou alors les résultats de tests pourraient causer de faux positifs ou de faux négatifs. Il est très important dans tout processus d'avoir une idée précise de la qualité de notre produit et il faut être assuré dans la mesure du possible que les défauts sont le moins possible présents. Pour cela, il faut maximiser le facteur de qualité "testabilité".

Dans le cadre de notre projet, ce facteur est très important, car une défaillance non prévue lors d'une présentation peut faire la différence entre une première place et une moins bonne place à la compétition Laval Virtual. Il est donc très important que toutes les phases du processus prennent en compte la testabilité, qui est un excellent de mesurer la qualité obtenue pour faire les ajustements nécessaires.

Pour le cas de l'architecture, la testabilité peut être maximisée principalement en s'assurant de minimiser le couplage. En effet, il est important de pouvoir tester les parties du logiciel indépendamment, ce qui est facilité par une architecture découplée. Aussi, une définition claire des interfaces des paquetages (par des façades, par exemple) permettra de faire des stubs de ces paquetages plus facilement ce qui peut largement aider pour effectuer les tests.

Compréhensibilité

Afin que les membres de l'équipe puissent travailler ensemble sur le projet, il est primordial que tous aient une même compréhension du document d'architecture. Sinon, il est possible que les idées soient perdues simplement parce certains pensaient que les autres les avaient comprises en lisant le document d'architecture. Également, une mauvaise

compréhensibilité pourrait mener à du temps perdu à implémenter des fonctionnalités non conformes aux attentes ou incompatibles entre elles.

Il est primordial que le client comprenne bien notre architecture s'il souhaite nous conseiller et nous faire profiter de son expérience antérieure de la compétition à laquelle nous participons.

Maintenabilité

Notre projet doit être maintenable, car il pourrait servir dans le futur afin de faire rayonner le programme de génie logiciel. Notre client souhaite en effet présenter notre projet aux journées portes ouvertes de Polytechnique dans les années à venir. Il doit être possible de réutiliser notre projet en entier ou par petit morceau (exemple retirer des instruments de musique si les capteurs nécessaires pour ceux-ci ne sont pas disponibles).

Faisabilité

L'utilisation de technologies nouvelles est un très grand risque pour notre projet. Il est difficile de savoir si les technologies utilisées sont suffisamment évoluées pour obtenir les résultats attendus avant d'avoir fait des tests avancés. Nous avons tout de même la contrainte de réaliser le produit d'ici le mois d'avril. Le client doit donc savoir que notre architecture isole les parties les plus risquées de manière à permettre la réalisation des parties à faible risque de manière indépendante. Nous serons ainsi assurés d'avoir un produit fonctionnel (répondant à toutes les parties à faible risque) à la date voulue.

Modèle de qualité pour le document d'architecture

Flexibilité

Question #1

Question

L'architecture prévoit-elle un niveau d'abstraction adéquat entre les couches afin de réduire le couplage?

Métrique

Présence de façade entre les couches.

Explications

On souhaite vérifier le respect de la pratique «Use abstraction to implement loose coupling between layers» proposée par Microsoft, afin de juger si notre application permet le changement facile de composantes risquées, soit celles du plus bas niveau comme le capteur utilisé.

Question #2

Question

L'architecture propose-t-elle une séparation et distribution suffisante des responsabilités dans des composantes séparées?

Métrique

Nombre de responsabilités par classe.

Explications

On souhaite vérifier le respect des principes:

- «Separation of concerns»
- «Single Responsibility principle

Et celui des pratiques:

- «Separate the areas of concern»
- «Do not overload the functionality of a component»

Plus les fonctionnalités seront séparées, plus il sera facile d'en modifier, retirer ou optimiser seulement une, augmentant donc la flexibilité du design.

Testabilité

Question #1

Question

Est-il possible de tester indépendamment chaque classe sans être contraint par un couplage trop élevé entre les classes?

Métrique

Nombre de liens d'agrégation/composition sur le nombre total de classes (nombre moyen de liens entre classes).

Explications

Le couplage a une importance élevée pour la testabilité, car s'il est trop élevé il peut être difficile de tester le comportement d'une classe de façon indépendante. Il faut donc chercher à le diminuer dans une mesure raisonnable.

Question #2

Question

Est-ce que l'architecture est conçue de manière à ce que les tests soient indépendants des capteurs?

Métrique

Le ratio de classes testables sans capteur.

Explications

Il est beaucoup plus complexe de tester des classes avec les capteurs, car cela dépend de beaucoup de facteurs physiques. Il faut donc essayer de minimiser dans la conception le nombre de classes qui nécessitent une utilisation directe du capteur. Même les classes qui ont une très forte relation avec les capteurs peuvent être testées sans capteur si elles sont capables, par exemple, de lire des données enregistrées d'un capteur (stub) plutôt que de communiquer directement avec le capteur.

Compréhensibilité

Question #1

Question

Est-ce que les diagrammes d'architectures sont faits dans un langage compréhensible par tous?

Métrique

Proportion de paquetages illustrés par un diagramme de classes respectant l'UML.

Explication

On souhaite vérifier si tous les diagrammes contenus dans notre document d'architecture sont facilement traçables et si tous les membres de l'équipe on la capacité de bien comprendre ceux-ci.

Maintenabilité

Question #1

Question

L'architecture permet-elle d'ajouter ou de retirer des composants sans modifier les autres composants?

Métrique

Uniformité des paquetages de même type.

Explication

On souhaite pouvoir ajouter ou retirer des paquetages au projet (exemple: des instruments ou des modes de jeu) en limitant les modifications aux autres paquetages. Si les paquetages des différentes couches en savent trop sur les classes des autres couches, cela peut se révéler difficile à faire. Il est préférable que les composantes de même type (dans notre cas: mode de jeu, instrument / cas plus général: commandes, formats de fichiers…) aient une

interface identique. Ainsi, les paquetages des autres couches peuvent simplement manipuler des conteneurs (listes) de paquetages, sans être couplés à leurs spécificités.

Nous souhaitons vérifier si les principes suivants sont respectés :

- «Keep design patterns consistent within each layer.»
- «Do not mix different types of components in the same logical layer.»

Faisabilité

Question #1

Question

L'architecture est-elle partitionnée de manière à ce que les exigences du SRS ayant une faible incertitude puissent être implémentées indépendamment des exigences plus risquées?

Métrique

Nombre moyen de paquetages à implémenter pour réaliser une exigence.

Explications

On souhaite vérifier le respect du principe:

- «Separation of concerns».

Question #2

Question

Combien de connaissances doivent être acquises pour implémenter chaque classe décrite dans le document d'architecture?

Métrique

Diversité des connaissances requises pour implémenter chaque classe.

Explications

On souhaite vérifier le respect des principes:

- «Do not mix different types of components in the same logical layer.»
- «Keep crosscutting code abstracted from the application business logic as far as possible»
- «Understand how components will communicate with each other.»

Métriques pour évaluer la qualité

Flexibilité

Tableau 1: Ratio des séparations de couches utilisant les façades

a) QME Name	Ratio des séparations de couches utilisant les façades.
b) Target entity	Document d'architecture
c) Objectives and property to quantity	L'objectif est de cerner la proportion des couches de paquetages utilisant une façade de communication avec la couche supérieure. Ici ne sera pas comptée la couche la plus haut niveau, c'est-à-dire la logique de jeu (Unity) qui ne communique pas avec une couche supérieure.
d) Relevant Quality measures(s)	Flexibilité de l'architecture logicielle.
e) Measurement method	Pour chaque séparation entre deux couches de paquetages • Noter la présence d'un paquetage faisant office de façade
f) List of sub properties related to the property to quantity (optional)	Liste: paquetage, couche de paquetages
g) Definition of each sub property (optional)	Paquetage: Un paquetage est un ensemble logique de classes pour lequel une description textuelle et un diagramme de classe ont été faits dans le document d'architecture. Couche de paquetages: ensemble logique de paquetages se rattachant à un niveau d'interaction (comme Modèle, Vue ou Contrôleur).
h) Input for the QME	Document d'architecture logicielle
i) Unit of measurement for the QME	Ratio
j) Numerical rules	Le ratio Q de la métrique est obtenu avec Q = Sf/St

	• Sf : Séparation entre 2 couches de paquetage possédant une façade. • St : Total des séparations entre 2 couches.
k) Scale type	Ratio
l) Context of QME	Pour que notre projet ait une bonne *flexibilité*, il faut que l'on puisse facilement changer certaines parties risquées ou certaines technologies sans impact sur le reste de l'application. C'est pourquoi il est important pour une couche de paquetages donnée de présenter une interface facilement utilisable pour les couches de plus haut niveau. C'est pourquoi la présence d'une façade pour chaque séparation de couches de paquetages favorise la flexibilité.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement Constraints (optional)	Le document architecture logiciel doit être complété avant de prendre cette mesure. Les paquetages doivent être groupés en groupes.

Tableau 2: Nombre de responsabilités par classe

a) QME Name	Nombre de responsabilités par classe
b) Target entity	Document d'architecture
c) Objectives and property to quantity	L'objectif est d'évaluer le nombre de fonctionnalités qu'une classe est destinée à remplir dans le document d'architecture.
d) Relevant Quality measures(s)	Flexibilité de l'architecture logicielle.

e) Measurement method	Pour chaque classe définie dans le document d'architecture:
	• Catégoriser la classe de la manière suivante :
	  o Mauvais : On détecte plus qu'une responsabilité principale.
	  o Moyen : On détecte une responsabilité avec quelques utilités connexes mineures.
	  o Excellent : On détecte strictement une responsabilité sémantiquement indivisible.

	Ensuite, effectuer le calcul défini en j.
f) List of sub properties related to the property to quantity (optional)	Liste: classe, fonctionnalité de base
g) Definition of each sub property (optional)	Classe: Élément listé à la section «classe» de la présentation d'un paquetage dans le document d'architecture. Responsabilité : Une utilité qui se doit d'être affectée à une composante du produit afin de respecter les exigences. Par exemple: Faire jouer une chanson.
h) Input for the QME	Document d'architecture logicielle
i) Unit of measurement for the QME	Aucune (ratio)

j) Numerical rules	Le ratio Q de la métrique est obtenu avec
	Q = (Cc) / Ct
	
	• Cc : Catégorie de la classe discrétiser de la manière suivante:
	  ○ Mauvais = 0
	  ○ Moyen = 0.5
	  ○ Excellent = 1
	• Ct : Total des classes dans le document d'architecture.

k) Scale type	Ratio
l) Context of QME	Afin d'avoir une grande flexibilité dans le document d'architecture et accessoirement dans le futur produit, il est important de bien séparer les responsabilités des classes en lien avec les exigences. Ainsi un changement de requis causant le rejet ou l'ajout de responsabilité à affecter dans le produit causera un minimum de complications, autant au niveau du document d'architecture que dans les modifications du logiciel.

	C'est pourquoi il est recommandé d'avoir un minimum de responsabilité par classe et ainsi avoir une grande flexibilité dans le design.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement Constraints (optional)	Le document architecture logiciel doit être complété avant de prendre cette mesure.

Testabilité

Tableau 3: Nombre de liens d'agrégation/composition sur le nombre total de classes

a) QME Name	Nombre de liens d'agrégation/composition sur le nombre total de classes
b) Target entity	Document d'architecture
c) Objectives and property to quantity	L'objectif est d'évaluer le couplage afin de déterminer si les classes sont facilement testables sans nécessiter trop de stubs.
d) Relevant Quality measures(s)	Testabilité de l'architecture logicielle.
e) Measurement method	Pour chaque classe définie dans le document d'architecture: o Calculer le nombre de liens d'agrégation et de composition. Ensuite, il suffit de faire un total du nombre de liens dans toutes les classes en additionnant tous les nombres obtenus. Finalement, il faut faire la formule définie en j.
f) List of sub properties related to the property to quantity (optional)	Liste: lien d'agrégation, lien de composition, classe.
g) Definition of each sub property	Classe: Élément listé à la section «classe» de la présentation d'un paquetage dans le document d'architecture.

(optional)	Lien d'agrégation : Lien entre deux classes où un objet d'une classe est contenu dans l'autre. Cet objet peut être manipulé par la classe, mais a une existence indépendante et n'est pas détruit lorsque les objets de l'autre classe le sont. Ces relations sont typiquement représentées en UML par un lien avec un losange vide. Lien de composition : Lien d'agrégation où l'objet contenu ne possède pas d'existence indépendante et est détruit en même temps que la classe qui le contient. Ces relations sont typiquement représentées en UML par un lien avec un losange plein.
h) Input for the QME	Document d'architecture logicielle
i) Unit of measurement for the QME	Nombre moyen de liens de couplage par classe
j) Numerical rules	Le nombre moyen de liens de couplage par classe Q est défini par : Q =(Cc)/Ct • Cc : Nombre de liens de couplage pour chaque classe. • Ct : Total des classes dans le document d'architecture.
k) Scale type	Ratio
l) Context of QME	Le couplage est un problème récurrent lorsque vient le temps de tester un logiciel, car un couplage élevé signifie que l'on doit créer beaucoup de stubs afin de tester les classes indépendamment. Un couplage fort rend aussi la phase des tests d'intégration beaucoup plus complexe. Un moyen simple de réduire le couplage est de limiter le nombre de liens d'agrégation et de composition (principale source de couplage). C'est pourquoi une métrique telle que celle décrite ci-dessus est très pertinente pour améliorer la testabilité d'une architecture.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement Constraints (optional)	Le document architecture logiciel doit être complété avant de prendre cette mesure.

Tableau 4: Nombre de classes nécessitant l'utilisation d'un capteur

a) QME Name	Nombre de classes nécessitant l'utilisation d'un capteur
b) Target entity	Document d'architecture
c) Objectives and property to quantity	L'objectif est d'évaluer le nombre de classes ayant besoin d'un capteur afin d'évaluer leur impact sur la testabilité.
d) Relevant Quality measures(s)	Testabilité de l'architecture logicielle.
e) Measurement method	Pour chaque classe définie dans le document d'architecture: • Vérifier si la classe nécessite l'utilisation d'un capteur. Ensuite, additionner 1 au nombre de classes si la classe contient un capteur.
f) List of sub properties related to the property to quantity (optional)	Liste: classe, capteur.
g) Definition of each sub property (optional)	Classe: Élément listé à la section «classe» de la présentation d'un paquetage dans le document d'architecture. Capteur : Composante électronique physique utilisée pour percevoir l'environnement. Dans le cas de notre projet, réfère à la Kinect, à la PS Eye ou à la caméra d'Intel.
h) Input for the QME	Document d'architecture logicielle
i) Unit of measurement for the QME	Aucune.
j) Numerical rules	Le nombre de classes utilisant un capteur Q est défini par : Q =(Cc) • Cc : o 1 si la classe utilise un capteur o 0 si la classe n'utilise pas de capteur

k) Scale type	N/A
l) Context of QME	L'utilisation d'un capteur peut rendre la phase de tests plus difficile, car les données reçues par les capteurs ne sont pas toujours fiables et on doit ajouter des cas de tests afin de tenir compte des conditions environnementales et matérielles. Un nombre élevé de classes utilisant le capteur nécessitera beaucoup plus de tests, il est donc mieux d'essayer d'adapter l'architecture afin que l'utilisation du capteur soit faite dans le moins de classes possible.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement Constraints (optional)	Le document architecture logiciel doit être complété avant de prendre cette mesure.

Compréhensibilité

Tableau 5: Proportion de paquetages illustrés par un diagramme de classes respectant l'UML

a) QME Name	Proportion de paquetages illustrés par un diagramme de classes respectant l'UML.
b) Target entity	Document d'architecture
c) Objectives and property to quantity	L'objectif est d'évaluer la compréhensibilité du document d'architecture. C'est-à-dire si les diagrammes sont traçables et lisibles par tous. Ce qui doit être mesuré est le nombre de diagrammes qui respectent le standard UML sur le nombre de paquetages.
d) Relevant Quality measures(s)	Compréhensibilité de l'architecture logicielle.
e) Measurement method	Pour chaque paquetage du document d'architecture : • Ajouer 1 si le paquetage est illustré par un diagramme UML. Diviser le résultat par le nombre de paquetages.
f) List of sub properties related to the property to quantity (optional)	Liste: diagramme, UML.
g) Definition of each	Diagramme de classes: diagramme de classes tel que défini par UML

sub property (optional)	(voir http://www.uml-diagrams.org/class-diagrams.html) UML : langage de modélisation
h) Input for the QME	Document d'architecture logicielle
i) Unit of measurement for the QME	Aucune (ratio)

j) Numerical rules	• Q = D/Dt
	  ○ D : Nombre de diagrammes respectant l'UML
	  ○ Dt : Nombre de paquetages

k) Scale type	Ratio
l) Context of QME	Pour que tous les membres de l'équipe puissent bien comprendre le document d'architecture, il est nécessaire d'avoir une bonne compréhensibilité. En effet, si un membre de l'équipe comprend l'architecture d'une façon différente, la qualité du produit final en sera affectée. L'UML étant un langage standardisé, il minimise les ambiguïtés.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement Constraints (optional)	Le document architecture logiciel doit être complété avant de prendre cette mesure. UML doit être la norme retenue pour les diagrammes.

Maintenabilité

Tableau 6: Uniformité des paquetages de même type.

a) QME Name	Uniformité des paquetages de même type.
b) Target entity	Document d'architecture

c) Objectives and property to quantity	L'objectif est d'évaluer la maintenabilité du document d'architecture. C'est-à-dire si c'est possible d'ajouter ou d'enlever des composants sans trop affecter les autres composantes. Ce qui doit être mesuré est la présence d'interface commune entre les composantes de même type et l'absence d'exposition de l'implémentation dans ces interfaces.
d) Relevant Quality measures(s)	Maintenabilité de l'architecture logicielle.
e) Measurement method	Regarder l'ensemble des diagrammes et vérifier les interfaces et si le code doit être exposé pour pouvoir utiliser le composant. Donner une valeur à la mesure à l'aide du barème suivant : • Mauvais - Les paquetages de même type n'ont pas une interface identique et elle dévoile leur implémentation. • Moyen - Les paquetages de même type ont une interface similaire et elle ne dévoile par leur implémentation. • Excellent - Les paquetages de même type ont une interface identique qui ne dévoile pas leur implémentation.
f) List of sub properties related to the property to quantity (optional)	Liste: diagramme
g) Definition of each sub property (optional)	Diagramme: les différents diagrammes d'architectures utilisés dans le document d'architecture.
h) Input for the QME	Document d'architecture logicielle
i) Unit of measurement for the QME	Mesure qualitative. Choix parmi: Excellent, moyen, mauvais.
j) Numerical rules	Aucune règle numérique, voir les étapes en e)
k) Scale type	Ordinal
l) Context of QME	Il faut permettre une bonne maintenabilité afin que le projet

	puisse évoluer et que d'autres personnes puissent l'utiliser en ajoutant ou en retirant des composantes.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement Constraints (optional)	Le document architecture logiciel doit être complété avant de prendre cette mesure.

Faisabilité

Tableau 7: Nombre moyen de paquetages à implémenter pour réaliser une exigence

a) QME Name	Nombre moyen de paquetages devant être implémentés pour réaliser une exigence
b) Target entity	SRS Document d'architecture
c) Objectives and property to quantity	L'objectif est de trouver les requis qui nécessitent d'implémenter un grand nombre de paquetages pour être atteints. Ce qui doit être mesuré est le nombre moyen de paquetages qui doivent être implémentés pour qu'une exigence tirée du SRS soit réalisée.
d) Relevant Quality measures(s)	Faisabilité de l'architecture logicielle.
e) Measurement method	Pour chaque requis défini dans le SRS: • Compter le nombre de paquetages dans lequel du travail devra être fait pour implémenter ce requis spécifique. • Ajouter 1 à une variable «nombre de requis contenus dans le SRS».
f) List of sub properties related to the property to quantity (optional)	Liste: requis, paquetage
g) Definition of each sub property (optional)	Requis: Un requis est un élément de niveau 3 du SRS, c'est-à-dire une section du SRS dont la forme est #.#.#. Paquetage: Un paquetage est un ensemble de classe pour lequel une

	description textuelle et un diagramme de classe ont été faits dans le document d'architecture.
h) Input for the QME	Spécification des requis logiciels (SRS) Document d'architecture logicielle
i) Unit of measurement for the QME	Paquetages / exigence.
j) Numerical rules	La proportion Q de la métrique est obtenue avec Q = Somme_Paquetages / R, où • Somme_Paquetages: Somme pour chaque requis des paquetages nécessaires à sa réalisation. • R: Nombre de requis contenus dans le SRS.
k) Scale type	Ratio
l) Context of QME	Pour que notre projet ait une bonne *faisabilité*, il faut que chaque membre de l'équipe ait à se familiariser avec un nombre restreint de technologies. L'apprentissage serait autrement beaucoup trop long. Étant donné que nos différents requis sont associés à des technologies de pointe différentes, il est souhaitable d'associer chaque requis à un nombre restreint de paquetage. On réduit ainsi l'apprentissage nécessaire pour travailler dans un paquetage donné. Également, un nombre de paquetages faible par requis minimise le risque associé à la non-faisabilité potentielle de certains requis: on saura rapidement quels requis sont infaisables s'il est possible de les développer indépendamment des autres.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement Constraints (optional)	Le document architecture logiciel doit être complété avant de prendre cette mesure. Les classes doivent être groupées en paquetages. Le SRS doit être disponible et à jour.

Tableau 8: Diversité des connaissances requises pour implémenter chaque classe

a) QME Name	Diversité des connaissances requises pour implémenter chaque classe.
b) Target entity	Document d'architecture
c) Objectives and property to quantity	L'objectif est d'évaluer la quantité de connaissances qui doivent être acquises par un développeur avant d'être en mesure d'implémenter une classe définie dans le document d'architecture logicielle. Ce qui doit être mesuré est le nombre de SDKs et/ou de techniques spécialisées dont un développeur doit avoir une connaissance solide pour implémenter chaque classe.
d) Relevant Quality measures(s)	Faisabilité de l'architecture logicielle.
e) Measurement method	Pour chaque requis défini dans le SRS: • Faire une liste des SDKs avec lesquels la classe doit interagir directement. Cela signifie que des fonctions ou des types de données du SDK sont utilisées directement dans la classe. • Faire une liste des SDKs avec lesquels la classe doit interagir indirectement. Cela signifie que la classe ne communique pas directement avec le SDK, mais dépend de ses détails d'implémentation. • Faire les mêmes évaluations avec les techniques spécialisées (tel que défini en g). Donner une valeur à la mesure à l'aide du barème suivant: • Excellent: Chaque classe nécessite une connaissance du fonctionnement d'au plus 1 SDK ou technique spécialisée pour être implémentée. Il est possible d'utiliser les données fournies par un autre SDK ou technique spécialisée, à condition que cela ne requière pas de comprendre comment elles sont produites/capturées. • Moyen: Chaque classe nécessite une connaissance pointue d'au plus 1 technique spécialisée ou SDK, mais peut aussi nécessiter une connaissance modérée du fonctionnement d'un autre SDK ou technique spécialisée. • Mauvais: Il y a des classes dont l'implémentation nécessite de comprendre les entrailles de plus d'un SDK ou des techniques spécialisées.
f) List of sub properties related to	Liste: SDK, technique spécialisée, classe.

the property to quantity (optional)	
g) Definition of each sub property (optional)	SDK: Kit de développement logiciel développé par des entités externes à notre équipe. Par exemple, le SDK permettant d'extraire des squelettes de la Kinect ou le SDK permettant de connaître les positions de doigts à partir d'une caméra Senz3D. Technique spécialisée: Technique prenant des données en entrée et produisant des données en sortie qui n'a pas été vue dans le cursus commun de génie logiciel de Polytechnique. Classe: Élément listé à la section «classe» de la présentation d'un paquetage dans le document d'architecture.
h) Input for the QME	Document d'architecture logicielle
i) Unit of measurement for the QME	Mesure qualitative. Choix parmi: Excellent, moyen, mauvais.
j) Numerical rules	Aucune. Suivre le barème décrit en e.
k) Scale type	Ordinal.
l) Context of QME	Notre projet nécessite l'utilisation d'un grand nombre de technologies inconnues. Tel que mentionné plus tôt, il est impensable que chaque développeur acquiert une connaissance pointue de chaque technologie. Il est donc souhaitable que chaque classe nécessite un nombre limité de connaissances préalables pour être implémentée afin d'augmenter la faisabilité. Cela permettra aux classes d'être implémentées par le développeur disposant des meilleures connaissances sur le sujet sans qu'il ait à faire de recherches trop poussées sur d'autres sujets (faisabilité). Par le fait même, nous aurons une grande confiance au code écrit par un expert du sujet et dans le cas où une technologie doit être changée, les changements seront limités à quelques classes.
m) Software Life Cycle process(es)	SYSTEM/ SOFTWARE ARCHITECTURAL DESIGN, révision du document d'architecture.
n) Measurement	Le document architecture logiciel doit être complété avant de

Constraints (optional)	prendre cette mesure. On doit avoir une bonne idée des solutions techniques disponibles afin d'associer de manière réaliste les SDKs et techniques spécialisées nécessaires à chaque classe.

Mesure du document d'architecture

Flexibilité

Ratio des séparations de couches utilisant les façades

Toutes les séparations de couches de paquetages définies dans le document d'architecture contiennent une façade de communication. Cela nous donne donc un ratio Q = 1, ce qui est la valeur recherchée dans ce cas. Il est à noter ici que des efforts spécifiques à ce facteur de qualité ont été faits lors de l'élaboration du design, d'où notre bon ratio.

Nombre de responsabilités par classe

Classes catégorisées comme non excellentes :

- InstrumentChooser: Surveille les gestes de changement d'instrument et instancie les bons instruments. (Moyen)
- ModeChooser: Surveille les gestes de changement de mode et exécute le changement de mode. (Moyen)
    - Ces deux classes ont une responsabilité commune qui est de surveiller les gestes de l'utilisateur. Cette responsabilité devrait se trouver dans une classe à part afin d'unifier l'analyse des gestes.
- AssistedController : Gère le mode assisté (Mauvais)
    - Le mode assisté requiert la gestion de musique, de partitions, et de choix de notes ou de sons en temps réel. Ces responsabilités devront être séparées pour améliorer la flexibilité.
- NoteComponent : Gère la rotation des notes selon les impacts avec les mains et gère la logique du son de la note. (Moyen)
    - La logique des sons pourrait être dans une classe dédiée.

Avec un total de 30 classes définies dans le document d'architecture, nous obtenons un ratio

Q = (3 * 0.5) + (1 * 0) + (26 * 1) = 27.5/30 = 0.92

Qui se situe près du ratio visé de 1.

Il est à noter ici qu'il y a une partie subjective et possiblement variable dans la définition de responsabilité, et surtout avec quelle granularité nous les définissons. Il y a donc une certaine incertitude sur cette mesure.

Testabilité

Nombre moyen de liens de couplage par classe

Nombre de liens de couplage
Paquetage GameLogic : 2
Paquetage Rendering : 0
Paquetage GestureRecognizer : 0
Paquetage Drum : 0
Paquetage Guitare : 0
Paquetage Piano : 0
Paquetage Skeleton : 0
Paquetage SensorLibInterop : 0
Paquetage SensorLib : 0
Paquetage HandTracker : 3
Paquetage IntelWrapper : 0
Paquetage KinectWrapper : 2

Nombre total de classes : 35

Q = Nombre moyen de liens de couplage par classe = (2+3+2)/35 = **0.2**

Nombre de classes utilisant un capteur

Classes qui utilisent un capteur directement :
IntelHandTracker
HandParameters
KinectWrapper
KinectSensor
KinectData

Q = Nombre total = 5

Compréhensibilité

Proportion de paquetages illustrés par un diagramme de classes respectant l'UML.

Tous les diagrammes de notre document d'architecture respectent le standard UML. Nous avons compté 13 diagrammes qui le respectent pour un total de 13 paquetages. En appliquant la règle, nous obtenons un ratio **Q = 1**, ce qui est la valeur recherchée dans ce cas.

Maintenabilité

Uniformité des paquetages de même type.

En regardant notre document d'architecture, on peut constater qu'aucun paquetage n'a une interface identique. Tous les instruments ont des méthodes différentes pour les

démarrer / arrêter. Il n'est pas possible d'avoir une liste d'instruments ou de modes de jeu traités uniformément.

De plus, on retrouve des interfaces qui exposent leur implémentation. En effet, pour le détecteur de doigts de Intel, il faut appeler des fonctions pour ré exécuter les calculs et fournir de nouvelles données. Il serait très difficile d'utiliser un autre capteur qui n'a pas ces requis. Également, pour la batterie, il faut penser à appeler une fonction qui change l'orientation de la caméra en lui fournissant la position de la tête du joueur. Il n'y a pas de raison que cela soit exposé de l'extérieur. Par contre, le paquetage «Skeleton» permet d'éviter d'exposer l'implémentation du paquetage «Kinect» dans le cas du drum et de guitare.

Suite à cette analyse, on peut dire la maintenabilité est de niveau «Mauvais».

Faisabilité

Nombre moyen de paquetages à implémenter pour réaliser une exigence.

Les paquetages nécessaires à chaque exigence sont mentionnés ci-dessous.

3.1.1 Choix du mode de jeu: GameLogic, Rendering, GestureRecognizer
3.1.2 Mode libre: GameLogic, Drum, Guitare, Piano
3.1.3 Mode assisté: GameLogic, GestureRecognizer, Rendering, Drum, Guitare, Piano.
3.1.4 Mode néophyte: GameLogic, GestureRecognizer, Rendering, Drum, Guitare, Piano.
3.2.1 Contrôles du présentateur: GameLogic
3.3.1 Détection du geste correspondant à chaque instrument: GestureRecognizer.
3.3.2 Guidage pour le choix de l'instrument: Rendering
3.3.3 Emplacement de l'instrument: Skeleton, GestureRecognizer, GameLogic, Rendering.

Piano
3.4.1 Vue à la première personne: Rendering, Skeleton.
3.4.2 Positionnement du capteur: Skeleton, SensorLib, KinectWrapper, IntelWrapper.
3.4.3 Notes jouables: IntelWrapper, HandTracker, Piano.
3.4.5 Utilisation des pédales: KinectWrapper, SensorLib, Piano.

Batterie
3.5.1 Vue à la première personne: Rendering, Skeleton.
3.5.2 Jouabilité de la batterie: Skeleton, GestureRecognizer, Drum.
3.5.3 Positionnement du capteur: Skeleton, SensorLib, KinectWrapper.
3.5.4 Pédale de la batterie: SensorLib, Skeleton, Drum.

Guitare
3.6.1 Vue à la première personne: Rendering, Skeleton.
3.6.2 Jouabilité de la guitare: Skeleotn, Guitare.
3.5.3 Positionnement du capteur: Skeleton, SensorLib, KinectWrapper.

3.7.1.1 Instruments souhaitables: Impossible à évaluer.
3.8.1.1 Nombre de joueurs: Aucun paquetage spécifique.

Un total de 21 requis du SRS ont été étudiés, mais il a été impossible de détecter les paquetages requis pour l'un d'eux. Ainsi:

Somme_Paquetages = 55
R = 20
Q = 2.75

Chaque requis nécessite du code spécifique dans une moyenne de Q = 2.75 paquetages.

Diversité des connaissances requises pour implémenter chaque classe.

On recense ici les classes pour lesquelles plus d'un SDK ou technique spécialisée doivent être connus pour permettre l'implémentation. Les classes sont groupées par paquetage.

GameLogic

- InstrumentChooser:
  - Connaissances avancées: Animation dans Unity.
  - Connaissances modérées: Kinect SDK, reconnaissance de gestes.
- ModeChooser:
  - Connaissances avancées: Animation dans Unity
  - Connaissances modérées: Kinect SDK, reconnaissance de gestes.
- AssistedController:
  - Connaissances avancées: Synthétisation sonore,
  - Connaissances modérées: Animation dans Unity.

Renderer
Les connaissances se limitent à l'animation dans Unity.

GestureRecognizer
Toutes les classes nécessitent:

- Connaissances avancées: techniques pour reconnaître des gestes.
- Connaissances modérées: Kinect SDK.

Drum

Toutes les classes nécessitent soient des connaissances de l'animation dans Unity, soit de la synthétisation sonore (un ou l'autre). Certaines classes nécessitent aussi la connaissance du Kinect SDK.

Guitare

Toutes les classes nécessitent:

- Connaissances avancées: animation dans Unity
- Connaissances modérées: Kinect SDK, synthérisation sonore.

La classe «Solo» nécessite aussi des connaissances avancées en traitement d'images 2D.

Piano

HandController

- Connaissances avancées: intelligence artificielle
- Connaissances modérées: Intel Perceptual Computing SDK, animation dans Unity, shaders dans Unity.

On note que la synthétisation du son a été bien isolée dans «NoteComponent».

Skeleton

Toutes les classes nécessitent:

- Connaissances avancées: mathématiques/géométrie.
- Connaissances modérées: Kinect SDK

SensorLib

Les algorithmes de traitement d'images sont très bien isolés dans ce paquetage.

HandTracker

HandCalibrator

- Connaissances avancées: traitement d'images 2D.
- Connaissances modérées: Intel Perceptual Computing SDK

IntelWrapper

Les classes de ce paquetage nécessitent uniquement de connaître le Intel Perceptual Computing SDK.

Kinect Wrapper

Les classes de ce paquetage nécessitent uniquement de connaître le KinectSDK.

Notre architecture obtient la valeur «Moyen». En effet, aucune classe ne nécessite des connaissances pointues de plus d'une technique spécialisée ou SDK, mais il est souvent impossible de faire totalement abstraction des autres techniques du projet.

Interprétation des résultats obtenus

Flexibilité

Au niveau de la flexibilité, nous sommes satisfaits des résultats obtenus. Pour la métrique «Ratio des séparations de couches utilisant les façades», nos efforts lors de la conception de l'architecture sont bien visibles. Sachant que les technologies risquaient de changer durant le projet, nous savions qu'il était nécessaire d'utiliser une stratégie comme le patron façade pour favoriser la flexibilité en cas de changement, entres autres, de capteurs.

Pour la métrique «Nombre de responsabilités par classe», notre résultat nous satisfait, bien que nous aurions pu faire un effort supplémentaire et nous questionner davantage sur les responsabilités de chaque classe. Des responsabilités partagées entre classes affectent grandement la flexibilité et nous nous devons d'éviter ce phénomène pour éviter les pertes de temps et les efforts supplémentaires lors d'un changement qui affecte cette responsabilité. Également une classe trop importante deviendra rapidement difficile à améliorer, optimiser ou modifier, d'où l'importance de bien diviser les responsabilités. Quelques changements au design pourraient donc améliorer ces aspects de flexibilité.

Testabilité

Les deux métriques utilisées nous donnent de très bons résultats. Pour la première métrique ("nombre moyen de liens de couplage par classe"), cela est dû autant à un effort de conception qu'à certaines limitations de la métrique. En effet, notre document d'architecture sépare les classes en des paquetages souvent très petits. Dans cette optique, il est normal que la plupart des classes n'aient pas de liens de compositions et/ou d'agrégation. Dans le cas de notre document d'architecture, une métrique testant aussi le couplage entre paquetages pourrait aussi être utilisée, mais elle serait à coup sûr moins précise, car les relations entre paquetages sont moins détaillées. Il est aussi notable dans notre cas que les liens de compositions sont quasi inexistants. Ces liens sont plus forts que les liens d'agrégation et causent un couplage plus important. Il est donc bon que nous en ayons moins.

Pour la deuxième métrique, nous avons obtenu un résultat exemplaire considérant que notre projet utilise énormément les capteurs. Cela est dû à un effort fait afin de limiter le nombre de classes utilisant directement les capteurs. Notre conception était faite de façon à ce qu'un nombre limité de classes accèdent directement aux fonctions du capteur et permettent à des classes externes de s'enregistrer auprès d'elle afin de recevoir les données en temps réel. Cela fait qu'un très petit nombre de classes utilise directement le capteur. Cette métrique nous permet de prévoir que nous aurons relativement peu de tests à faire en ce qui concerne les contraintes environnementales et matérielles.

Compréhensibilité

Nous obtenons un résultat parfait pour la métrique «Ratio du nombre de diagrammes qui respecte le UML». En effet, lors de l'écriture du document d'architecture nous avions établi les règles pour faire les diagrammes. Nous avions choisi UML, car c'est ce

qui nous est enseigné à l'école Polytechnique de Montréal. C'est très bien pour la compréhensibilité, car tous les membres de l'équipe comprennent et utilisent le standard UML. Une bonne compréhensibilité permettra d'avoir un produit de qualité, car il n'y aura pas de malentendu sur l'architecture.

Maintenabilité

Nous avons un niveau de maintenabilité «Mauvais», selon les critères définis plus haut. Nous sommes conscients de ce problème et ce serait bon de réviser l'architecture et d'adapter le code en conséquence. En effet, on aurait dû avoir des méthodes communes à tous les instruments comme «DemarrerInstrument» et «DemarrerModeAssiste». Nous allons peut-être corriger notre code afin de le rendre plus maintainable, car il est important que le programme de génie logiciel puisse le réutiliser afin de le présenter aux portes ouvertes par exemple.

Faisabilité

Nous obtenons un très bon résultat pour la métrique «Nombre moyen de paquetages à implémenter pour réaliser une exigence». Cela nous indique que lors de la conception de notre architecture, nous avons associé aux classes des responsabilités qui sont souvent des requis tirés du SRS. Cela est positif du point de vue de la faisabilité, car nous savons que pour chaque classe que nous implémentons, nous réalisons des requis spécifiques. Nous faisons donc toujours des avancées concrètes du point de vue du client et nous aurons rapidement l'assurance que certains de nos objectifs sont faisables dans le temps alloué. Également, le fait d'associer des requis spécifiques à chaque classe signifie que si nous décidons d'enlever des requis jugés non faisables au cours du développement, il suffira d'enlever des classes spécifiques. Les changements ne seront pas généralisés dans l'application.

Le résultat de la métrique «Diversité des connaissances requises pour implémenter chaque classe» est moins satisfaisant. Tel que mentionné au paragraphe précédent, nous avons assigné les responsabilités aux classes selon les requis. Lorsqu'un requis combine plusieurs technologies, nous n'avons pas pris la peine de le fractionner en plusieurs classes. Par exemple, il est dommage que les classes représentant les différents composants de la batterie soient à la fois responsable de la détection des collisions avec les baguettes, de l'animation lors d'un impact et de la production du son. Dans notre cas, Félix possède de bonnes connaissances pour moduler un son selon l'impact des baguettes alors qu'Alexandre est plus doué en animation. Ils seront forcés de travailler dans une même classe en raison de du mauvais découpage de l'architecture et risquent ainsi d'avoir à comprendre le travail de l'autre. Étant donné qu'il a été établi que les technologies utilisées nécessitaient un long temps d'apprentissage, cela diminue la faisabilité de notre projet dans le temps alloué.

L'utilisation du patron *Broker* aurait été pertinente dans ce contexte pour gérer les communications entre les composantes utilisant les différentes technologies nécessaires à la réalisation d'un requis sans avoir à se connaître entre elles. Le principe «Do not mix different types of components in the same logical layer» du guide architectural de Microsoft est également intéressant. Si nous avions créé une architecture avec couches plus claires, nous

aurions évité une dépendance envers le Kinect SDK partout à travers l'application. Un exemple de séparation par couches est:

- 1: communication avec les capteurs
- 2: extraction des données nécessaires au jeu à l'aide de traitement d'images et de filtres
- 3: logique d'application
- 4: affichage et production de sons

Temps passé sur le laboratoire

Nous avons passé 20 heures-personne à la réalisation de ce laboratoire.