INFO-F204 - Analyse et méthode
Christian Hernalsteen
Résumé du cours

Rodrigue Van Brande

10 octobre 2015

Table des matières

1 Le software engineering

1.1 Introduction

Dans les années 70 on s'est rendu compte qu'on avait besoin de méthode de développement. Mais plusieurs problèmes se posent : Le client peut vouloir changer quelque chose au cours du projet, on doit pouvoir estimer le temps que ça va prendre, etc... Deux plus grosses causes d'échec : Compréhension du client et le travail en équipe. Les diagrammes UML vont aider, et clarifier les choses avec le client et son équipe. Pourquoi ne peut on pas s'inspirer de l'ingénierie civile (ex :construction d'un pont) ? Car un pont on le voit, pas le logiciel et ce dernier évolue continuellement contrairement au pont. Problème : Évolution du logiciel dégradante ⇒ Chaque modification devient de plus en plus compliquée.

1.2 Méthode Waterfall

On procède par phase :

Requirements Collection Rencontre avec le client et note de tous les besoins (Risques : Documentations incomplètes, inexactes et ambiguës)

Analysis Les analystes définissent les besoins, les écrans, ... (Risques : Fournir une spécification qui ne correspond pas aux besoins du client.)

Design Architectes conçoivent l'architecture de l'application. Conception de diagrammes et choix des librairies.

Implementation Les codeurs développent leurs modules.

Testing Assemblage et livraison.

Cela a donné de bon résultats mais problèmes de communication car elle est faite par documents plutôt que par la parole. Les codeurs n'ont pas de recul et ne peuvent détecter des problèmes potentiels. Si il y a une erreur il faut corriger dans chacune des étapes. Un logiciel est long à développer et les besoins du client peuvent changer dur le développement.

Avantages : Très contrôlé, panifiable, des documents décrivant l'entièreté de l'application.

1.3 Méthodes incrémentales et itératives

On procède par incréments. On livres des morceaux de logiciel au client petit à petit. C'est un enchaînement de mini waterfall. Cette méthode permet d'avoir plus de retour du client. On ne fait pas une grosse analyse, on développe juste complètement un module qu'on va montrer au client. Il vérifie et apporte ses corrections qui seront facilement faisables. On fait cela à chaque itération. Une itération dure en moyenne entre 2 et 4 semaines.

Avantages : Le client s'implique et le projet à de grosses chances de réussites.

1.4 Bibliothèques

Avant lorsqu'on codait, on écrivait quasiment tout, mais maintenant on utilise beaucoup de frameworks et de composant. On doit trouver tout ces composants lors de l'analyse.

2 L'orienté objet

2.1 Introduction

2.2 Types de données abstraits

On sépare l'implémentation de la spécification, ce qui permet de développer plein de petits modules plutôt qu'un énorme bloc de code. On peut modifier des bouts de code à un endroit sans devoir modifier toute l'application du coup la maintenance est facile.

Code Client : Code qui dépend d'un autre

2.3 Les objets

On écrivait le code qui traitait des données. Les objets réunissent les deux dans "une boîte".

2.3.1 Ce qu'est un objet

Un objet a une réalité et est unique (!= d'une classe). Il a un état et est modifiable. Objet = Identité + état + comportement Un objet reçoit des messages (se déplacer, changer d'état, ...) et va chercher le code correspondant. L'objet cache ses données, on ne peut passer que par les messages.

2.3.2 Interaction entre objets

Si plusieurs objets, il peuvent interagir (Exemple : En passant un objet en paramètre d'un message).
Message $\neq$ Méthode $\Rightarrow$ On envoie un message à un objet, la méthode c'est le code que l'objet exécute lorsqu'il reçoit le message.
Method lookup : Ce qui trouve la méthode correspondant au message.

2.3.3 Polymorphisme

Message $\neq$ d'un appel de fonction. Lors d'un message, on dit l'objet sur lequel ça va intervenir, le a fonction peut porter sur n'importe quoi.
Polymorphisme : On envoie un même message à des objets de forme différente et la réponse pourra être différente.

2.3.4 Stockage des méthodes et classes

Classe à une réalité en mémoire, elle contient le code mais pas les données.
Un objet à aussi une réalité en mémoire, elle contient les données mais pas le code.
Un objet est une instance d'une classe.
Un objet connaît toujours sa classe, lorsqu'il reçoit un message il lui demande la méthode qui convient.

2.4 L'héritage

Héritage On décrit un objet abstrait, puis des objets qui en hérite et qui récupère tout ça.

Généraliser Opération de créer un objet plus abstrait dont on va hériter.

2.4.1 Principe de l'héritage

On dérive d'une classe parente et les sous-classes héritent de tous ses attributs et méthode.
Les sous-classes peuvent surcharger les méthode pour les adapter à leurs besoins $\Rightarrow$ Overriding.
But : Mettre de la structure qui amènera au polymorphisme et un code facilement maintenable.
Pour savoir si B doit hériter de A, il faut pouvoir se dire : B est une sorte de A.

2.4.2 Types d'héritage

Héritage simple : Une classe hérite d'une seule et une seule classe. Cet héritage à une structure d'arbre, chaque classe a un parent et des enfants.

Héritage multiple : Permet d'hériter directement de plusieurs classe. Les langages récents ne le permettent plus car plus de problème que de solution.

2.4.3 Method lookup

Lorsqu'un objet reçoit un message, il connaît sa classe et de laquelle elle hérite. Il va chercher si la méthode est dans sa classe, sinon il regarde dans celle héritée et ainsi de suite.

Redéfinition de méthode (*Overriding*) : Les sous-classes peuvent surcharger les méthodes pour les adapter à leurs besoins).

Surcharge de méthodes (*Overloading*) : est une possibilité offerte par certains langages de programmation qui permet de choisir entre différentes implémentations d'une même fonction ou méthode selon le nombre et le type des arguments fournis. La **surcharge de méthodes** ne doit pas être confondu avec la redéfinition de méthode (*Overriding* en anglais). **Exemple :** *classe.setPosition(int x, int y)* et *classe.setPosition(Vector<int> vector)*.

2.4.4 self/this et super

Mots clés spécials :

super représente la classe parente.

this représente la classe qui a reçu le message qu'on traite.

super est statique, on sait immédiatement ce que c'est.

this est dynamique, lors de la compilation on ne sait pas quel objet il représente. Car il peut représenter la classe elle même ou une classe fille.

2.5 Polymorphisme

2.5.1 Références

Référence (ou pointeur) : permet de nommer un objet et est différente de l'objet lui-même.
Si on a une référence vers un objet de type A, on peut la faire pointer sur un objet de type B.
L'objet de type B étant une sorte de A, on pourra utiliser les messages connus de A (mais pas ceux de B).
C'est grâce à ce principe qu'on va pouvoir utiliser le polymorphisme.

2.5.2 Exemple

Imaginons qu'on veuille réaliser une application pour faire des dessins. On a des boutons permettant de choisir un carré, un triangle ou un rond. On veut programmer ça, avec la pensée objet.

On a une classe FORME qui va contenir tout ce qui est commun à une forme (taille, position, couleur). On a les classes ROND, CARRE et TRIANGLE qui héritent de FORME. On a une classe DESSIN qui contient des FORMEs et qui n'hérite de personne.

On crée une méthode dessiner qui ne fait rien à la classe FORME (elle est juste la pour dire qu'elle existe). Et on la surcharge dans les classes TRIANGLE, CARRE, ROND. Comme ça on peut utiliser (FORME f).*dessiner*(), ce qui va appelé la méthode de la classe correspondante (soit de TRIANGLE, soit de CARRE, soit de ROND).
Délégation : Le fait qu'un objet en utilise un autre.

2.5.3 Le C++

Le C++ laisse beaucoup de liberté, et il y a plein de piège dans son orienté objet. Les fonctions ne sont pas virtuelle par défaut (si on surcharge une méthode de A dans B, et qu'on envoie le message à une référence A qui pointe vers un objet B c'est la méthode de A qui sera appelé.) Pour résoudre le problème, il faut rajouter **virtual** devant la méthode de A.

Modificateur de visibilité : Lors de l'héritage on peut préciser si il est public privé ou protégé. Les autres types d'héritage que public masquent les attributs et méthodes de A. ⇒ Cela rompt l'orienté objet.

C++ permet de créer des objets sur la pile et pas seulement par référence. ⇒ Empêche tout un tas de bonnes choses de l'orienté objet.

this est implicite et il n'existe pas de mot clé **super**, on nomme explicitement la classe.

3 Héritage avancé

3.1 Introduction

3.2 Overriding de méthodes trouvées en Framework

Le super ne doit être appelé que dans des méthode redéfinies. Et ne doit servir que à appeler la méthode de la classe parent.

3.3 Classes abstraites

Une classe abstraite est une classe qu'on ne peut instancier directement. Cela permet d'implémenter des interfaces. Elle fournit une abstraction qui permettra de mettre des choses en commun entre ses filles. Une classe abstraite l'est si elle a au moins une méthode abstraite. (**virtual** foo() = 0;).

Si la classe fille ne veut pas être abstraite elle doit redéfinir toutes les méthodes virtuelles pures (=abstraites) de la mère. Si elle le fait, on appelle ça une **concrétisation**.

3.4 Où placer les méthodes

Quand on ajoute une méthode à une classe, on le met le plus haut possible dans la hiérarchie des classes. L'implémentation doit aussi se baser le plus possible sur les méthodes déjà existantes.

4 Aperçu d'UML

4.1 Introduction

UML = **U**NIFIED **M**ODELLING **L**ANGUAGE
Il s'agit d'une boîte à outils, le processus est propre à celui qui l'utilise.
C'est un langage de modélisation.
Modéliser : Avant d'implémenter on va faire des plans de construction.
Cela aide à réfléchir et à communiquer.

4.2 Bits de l'UML

Il a été fait pour être automatisable et lisible par un humain.
Il est générique, il peut s'appliquer à tout type d'application.

4.3 Concepts

4.3.1 Les vues

Il existe 5 vues dont chacune est définie par un certain nombre de diagrammes :

Use case montre les fonctionnalités du système tel qu'elles sont perçues par un acteur externe, qui peuvent être des utilisateurs ou d'autres systèmes (Diagrammes : Use case, Activity).

Logical view définit les fonctionnalités du système, les informations manipulés, ... (Diagrammes : Class Diagram, State, Sequence, Collaboration, Activity)

Component view indique comment le code est mis en boite. Les classes, les bibliothèques, fichiers de configurations, BDD, ... (Diagrammes : Diagramme de composants)

Deployment view indique là où les composants vont s'exécuter. Le déploiement du système dans l'architecture physique avec les ordinateurs et les appareils.

Concurrency view décrit comment les composants interagissent entre eux. Point de vue dynamique du système. (Diagramme de séquence)

4.3.2 Les diagrammes

Il y a au total 9 diagrammes.

Use case Basique. Ce sont des dessins. On définit ce que les acteurs peuvent faire.

Class On représente ici les classes à un haut niveau. On n'utilise que le nom des classes et les liens qui les unissent.

State Représente les états d'un programme. On a un état initial (un rond noir) Puis des états reliés par des flèches.

Sequence Représente des séquences de communication entre objets. Le temps se lit du haut vers le bas.

Collaboration Correspond au sequence diagram, met on met en évidence la structure plutôt que le temps.

Object Donne un exemple pour aider à la compréhension.

Activity Représente le coté comportemental de l'application. On y montre des étapes.

Component Identifie les composants et les relient.

Deployment Constitué de gros cube. Un gros cube est une unité de traitement ou de stockage, un serveur ou une bdd.

5 Meta Model UML

5.1 Introduction

Tout les composants qu'on peut utiliser en UML ont été écrits en UML.
Un méta-model est un modèle qui en décrit un autre.
Chaque diagramme est une instance du méta-model.

5.2 L'élément

Tout ce qu'on manipule en UML est un élément.
Chaque élément peut être contenu dans un paquet.

5.3 Mécanismes

Stéréotype permet d'étendre UML. (format : « stéréotype »). On veut par exemple montrer que certaines classe servent à la gestions des freins, on leur ajoute le stéréotype «gestion des freins».
Tagged values est une association entre un nom et une valeur. (Exemple : créateur : machin)
Notes donne des bouts de texte, des commentaires.
Contraintes entouré par des accolades. Permet de représenter des contraintes sur les attributs, valeurs, associations, ...
Dépendances dit qu'un élément dépend d'un autre (flèche en pointillé)
Types prédéfinis sont les types qu'on connaît (bool, string, int, ...)
Multiplicité représente un nombre lors des associations.
Package rassemble des éléments.

5.4 Diagrammes des classes

On y représente les classes utilisées dans le projet. Représente les différentes informations que l'application va devoir manipuler.

5.4.1 Représentation d'une classe

Représenté par un grand rectangle contenant en haut le nom de la classe, puis les attributs, puis les méthodes.

5.4.2 Règles de représentation

Classe commence par une majuscule.
Méthode doit être un nom, non une action ou un verbe.
Attribut commence par une minuscule et ont un type ou une visibilité et peut avoir une valeur par défaut.
Méthodes ont une signature (type de retour, le nom, 0 ou plusieurs paramètres) et une visibilité.

5.4.3 Relations de base

4 types de relations/associations entre les classes.

Usage Utilisation

Inheritance L'héritage

Refinement La même classe avec plus de détails.

Réalisation Les classes abstraites qu'on réalise.

Pour la relation d'usage on peut (pas obligatoire) avoir une direction et une multiplicité.

Relations ternaires : Relations entre trois classes. **Role name** : Le nom du rôle joué par la classe dans la relation (Exemple : maître ou esclave.)

5.4.4 Relations étendues

Association Une association représente une relation quelconque entre deux classes ; en général, les objets d'une classe se servant de ceux d'une autre classe. (Exemple : une personne utilise un ordinateur.) Un simple trait entre les deux classes, peut être composé d'une flèche pour indiquer une direction.

Agrégation : Une agrégation permet de définir une entité comme étant liée à plusieurs entités de classe différente ; décrit une association de type « fait partie de », « a ». (Exemple : Une flotte constituée de plusieurs bateau. Une flotte n'est plus une flotte sans bateau.) Représenté par un diamant vide du coté de la classe qui agrège (ici la flotte).

Composition : Comme l'agrégation mais en plus fort. Elle fait partie entière de l'objet. Si on détruit le tout, on détruit tous les éléments. (Exemple : Si on détruit le livre, les pages n'ont plus de raison d'être, contrairement aux bateaux sans la flotte.) Représenté par un diamant noir du coté de la classe principale (ici le livre).

Généralisation : Héritage, flèche pointant sur la classe parente.

Raffinement : Relation d'une classe vers elle même. On représente ça avec « refine»

Réalisation : Dit qu'une classe en réalise une autre.

6 L'ingénierie des besoins

6.1 Introduction

On doit capturer les besoins du client. On doit savoir comment l'informatique va résoudre ses problèmes. C'est souvent à cause de cette étape que des projets tombent à l'eau.

6.2 Les besoins

Les besoins décrive le système et ses contraintes.
C'est ce que le programme doit faire mais pas comment il va le faire.

6.2.1 User et System Requirements

User requirements Ce que l'utilisateur veut.

System requirements Un peu plus bas, plus formel.

Les users requirements doivent être dans le jargon du client.

6.2.2 Besoins fonctionnels ou non

Besoins fonctionnels Les services que l'application doit rendre.

Besoins non-fonctionnels ergonomie, robustesse, sécurité, ...

Domain requirements Vient du domaine dans lequel l'application sera utilisée. Ces besoins ne seront pas donné par le clients car ils sont évidents pour lui.

6.3 Écrire les besoins

Il faut écrire les besoins pour que d'autres personnes puissent les lire. Ces documents vont devoir être simple et clair.
Il faut numéroter les besoins avec des titres,sous-titres, ... et éviter les gros pavés de texte.
Il faut se définir un standard. Il faut aussi éviter le jargon informatique car le client va le lire aussi.

6.4 Les System requirements

Il faut être plus rigoureux que pour les user requirements. UML peut intervenir ici. On définit les terminologie, on remplace le jargon par des définitions, ... On a aussi une table des matière.
Software requirements document : Il s'agit du document qui regroupe les user requirements et les system requirements. Il existe des standards pour ce genre de document.

6.5 Croiser les besoins

Différentes personnes pensent différemment. Pour chaque besoins, il faut chercher les informations. On va souvent tomber sur des contradictions. Certaines personnes pourrait vouloir freiner le projet.
Il ne faut pas non plus oublier d'aller voir les utilisateurs.

6.6 Aperçu d'un SRD

On commence par l'introduction. On y explique les buts à atteindre.
Le SRD va évoluer au fur et à mesure du projets.
Il faut donc un gestionnaire de version pour avoir un historique des versions et on peut récupérer les anciennes.

6.7 UML Use case

La vue des use case est centrale en UML car ce sont les besoins qui définissent le programme. On l'utilise pour exprimer des interactions, pas des contraintes donc elle est mieux pour les besoins fonctionnels.

6.7.1 Les acteurs

On représente ça par un bonhomme dans tous les cas. Il ne s'agit pas spécialement d'une personne. C'est les acteurs qui déclenche les use case.

6.7.2 Les use cases

Il s'agit d'un ensemble d'action que fait le système. Ils sont initiées par un acteur et lui fournissent une réponse.
On ne dit pas ce qui se passe derrière, seulement les informations qui entrent et qui sortent.
On peut décrire l'use case sous forme de texte structuré ou sous forme de diagrammes de séquence ou des diagrammes d'état.
On peut décrire les exceptions, les erreurs, les scénarios alternatifs, ...

6.7.3 Description de l'use case

Généralement sous forme de texte.
- Acteurs ;
- Pré-conditions (ce qui est vrai avant que le use case se déclenche) ;
- Post-conditions (ce qui est vrai après le use case) ;
- Cas basique (décrit chaque étape de l'interaction entre le système et l'utilisateur) ;
- Alternative flow ;
- Special requierement (Besoins spéciaux du use case) ;
- Relation entre use cas.

6.7.4 Relations

Généralisation entre les acteurs Permettre à des acteurs d'hériter d'autre.

Généralisation d'use-case Permet de spécialiser un use case.

Include Permet d'inclure le contenu d'un use case dans un autre. (on représente ca avec le stéréotype « include ».

Extensibilité Revient à décrire un use case en laissant un vide.

6.8 Modélisation dynamique

Ce qu'on a vu avant = Modélisation statique => On définit les choses.
Maintenant on s'intéresse au côté dynamique comme l'envoi de messages entre objets, leur vie, ...
On ne modélise pas tout mais seulement le nécessaire.
UML offre 4 diagrammes : Sequence digrams, collaboration digrams, state diagrams, activity diagrams.

6.8.1 Sequence diagrams

Les diagrammes de séquence et de collaboration montrent tous les deux de la collaboration mais celui de séquence se concentre sur le temps alors que celui de collaboration se concentre sur la structure de communication.
Le séquence diagramme représente l'interaction entre plusieurs objets comme une séquence de message se lisant du haut vers le bas.
On peut utiliser des branchements, des boucles, des conditions, ...
L'échange de message se fait de manière synchrone (flèche pleine), asynchrone (flèche pas pleine) et la réponse (flèche pointillée).
On peut envoyé un message new peut être envoyé à un objet qu'on crée. Quand on a fini on le delete en plaçant une croix à la fin de sa ligne de vie.

6.8.2 Collaboration diagrams

Parfois on a trop d'objet et le diagramme de séquence deviendrait trop complexe.
On utilise donc le diagramme de collaboration où la notion de temps disparaît
On peut numéroter les messages, pour savoir dans quel ordre ils sont envoyé.
On peut mettre des conditions et des descriptions.
La visibilité permet de savoir comment un objet sait qu'un autre existe.

Association Le réceptionniste est un attribut

Global Variable global

Local Variable local

Parameter Objet recu en paramètre.

Self Le pointeur this.

7 Le test

7.1 Introduction

On va faire des test unitaire pour tester le programme. Ils testent une partie du programme de manière indépendante des autres.
On prend une classe , on l'isole et on vérifie qu'elle fait bien ce qu'elle est censé faire.
Il existe différentes approches de test :

Test exploratoire On joue avec l'application, on utilise l'interface, on a rien prévu pour le test. Utiliser des outils qui injectent des données à l'application.

Test unitaire On ne teste pas l'application mais les composants. On va créer un code supplémentaire pour tester.

Stub ou Mock Remplacer une classe par une autre, minimaliste et ne fournissant que l'interface ainsi éventuellement des outillages de test.

7.2 Conditions

Invariant Expression booléenne qui est toujours vraie. On commence donc par les vérifier. Si un des invariant est faux, c'est qu'il y a une erreur.

Pré-condition et Post-conditions Expression assurées être vraies avant ou après la méthode.

7.3 Assertions

:

Assertion Permet d'exprimer une expression booléenne qui doit être vraie à cet endroit.
Cela permet de formaliser et d'écrire des pré-conditions, invariants et post-conditions.
Avec une assertion, si l'invariant est violé on sait où et l'erreur ne se propage pas.

7.4 Test unitaire

On écrit en général les test unitaires sous la forme d'objet. On les appelles des testssuites. Un avantage des tests unitaires est qu'on peut les relancer très facilement. On peut rejouer les test même sur ce qu'on a pas modifié. On vérifie qu'on a pas cassé quelque chose qui marchait déjà. C'est un test de non-régression.

7.4.1 Rédaction de tests

Les développeurs doivent écrire du code et des tests en même temps mais parfois on écrit d'abord les tests puis le code.
Extreme programming : Technique de programmation agile, très fortement itérative qui se base la dessus. Écrire d'abord les test permet de tester la spécification et non l'implémentation.
Avant on écrivait beaucoup de documentation, qui finissait par ne plus être à jour. La meilleure source d'information sur une application est le code. Il faut investir du temps au niveau du code, de son auto-documentation, ses tests, sa lisibilité, ...
Les tests servent maintenant de documentation car ils sont toujours à jour.

7.4.2 Méthodologie de rédaction de tests

Si on a un bug trouvé par le client, c'est qu'il est passé à travers les tests, donc aucun ne le couvre.
On rédige alors un test qui échoue sur le bug et seulement ensuite on va pouvoir le corriger.
Si en cherchant le bug on place des printf, c'est que ça devrait se retrouver dans le test.

7.5 Mise en place de tests unitaires

On va tester une classe ou un petit morceau de programme.
Il existe des frameworks de test unitaires dans presque tout les langages. (CppUnit pour C++)
En CppUnit, un testcase hérite de CppUnit : :TestCase. La classe du test devrai avoir le nom "NomTest".
Un échec est une assertion qui rate, mais on l'avait anticipé. Une Erreur est un bug non-détecté par une assertion.

7.5.1 Qualité des tests unitaires

Un bon test unitaire doit avoir certaines propriétés.
Un test unitaire doit être déterministe : A chaque lancement du test, on doit avoir le même résultat. Sinon on peut perdre confiance dans les tests. Le lancement doit être automatisé.
Les test sont une source de documentation. Ils doivent donc être lisible. Ils doivent être le moins sensible possible aux changements du code.
Ils ne doivent pas tester des choses évidentes mais traiter les choses complexes et difficiles en premier.

7.5.2 Frameworks de test

Les éléments clé d'un framework de tests sont le TestCase (Cas de test), les mécanisme pour les exécuter, les fixtures (un contexte, les objets cobaye dont on a besoin, les TestSuites (un ensemble de TestCase) et le TestRunner qui va déclencher les tests.

On commence par déclarer un TestRunner. Un Outputter prend le flux de sortie et le met ailleurs. On enregistre les test auprès du TestRunner. Puis on le lance et l'Outputter contient maintenant les résultats.

TestFixture : Une fixture est le contexte partagé par tout les testcase d'une suite de test.
Entre chaque test, le contexte est réinitialisé à la fixture de départ.

8 Étude de cas de développement

8.1 Analyse

Requirements du client : Une description du jeu.

La première chose à faire est d'identifier les acteurs et de faire des use case. Dans ce cas on a deux joueurs qui interagissent avec le programme. On a donc un seul acteur joueur, celui qui joue son tour. Ensuite on fait un diagramme de classes. On fait parler tout le monde sans critique, on amène de l'information sans la classer ni rien.

8.2 Approche itérative

L'approche itérative propose des sous-versions du logiciel après chaque étape montrable au client. Cela permet d'avoir des retours plus vite. Mais cela peut mener à développer des choses qu'on va bazarder plus tard mais c'est normal.

Refactoring : Changer le code pour que l'application fasse non pas des choses en plus mais différement. A chaque itération, on modifie ce qu'on a déjà fait et on arrive à la fin avec un système qui marche.

8.3 Direction et début

Avant de commencer la première itération, on fixe le scope. (Le jeu sera t'il en résaux ? Avec parties sauvegardées ? ...).

On va proposer les objets. Classe Game qui gère les règles. La classe Player représente un joueur et s'occupe de l'input. Compartment est une case de la grille, gérée par Game. Figure est ce qu'on met dans les Compartments. On ajoute un objet Driver qui va gérer le jeu et l'affichage.

Sequence Diagram montre une petite partie : Game crée des Player, Driver passe son temps à demander à game si il a fini. Game demande à un joueur chacun son tour quel mouvement il peut faire.

8.4 Extention

Notre prochaine itération vise à avoir un Board, de pouvoir l'imprimer et de pouvoir ajouter des joueurs.

Plateau de jeu : matrice 3x3, statique, de caractère. Chaque carac est soit un O, soit un X, soit un espace. Le board init à plein d'espace au début. On le référence avec des set et get. Dans ces méthodes on a une assertion inRange, qui vérifie que la colonne est entre A et C et la ligne entre 1 et 3.

Pour le test, on fait plein de get et set et de vérifier qu'on récupère bien ce qu'on met dans la matrice. On teste également qu'au début, la matrice est vide. La dernière vérification teste que inRange("D","3") renvoie bien faux.

9 Implementation and Design Issues

9.1 Aspect du code

On va voir quelques standards à appliquer quand on code.

9.1.1 Mise à niveau

Cohérence : Le premier principe de lisibilité est la cohérence. Il faut éviter qu'on sache reconnaître quel développeur a écrit quel code.

Couplage : Interdépendante entre le code du projet. Quand un élément change, il faut modifier tout ce qui y est couplé. Quand le couplage au sein de l'application est trop important, la maintenance est difficile.

Cohésion : est le couplage interne. Ce qu'on a mis dans un objet a une bonne raison d'y être. Quand on a un faible couplage, on a une très forte cohésion. => Ca c'est bien.

Standard de codage : Ensemble de règles qui définissent comment on écrit du code. Il faut toujours suivre le standard de codage du projet sur lequel on travaille.

9.1.2 Noms

C'est le seul endroit ou on peut donner une signification humaine au code. Prendre le temps de bien choisir un nom fait gagner énormément de temps. Le nom d'une méthode doit expliquer ce qu'elle fait. Un objet peut changer son état , changer l'état d'un de ses arguments, ou alors ne rien faire et retourner un résultat. Chacun de ces types de méthode utilise des noms prédéfinis.

Quand on **change l'objet** , on utilise un verbe(insert, clear, ...).

Quand on **modifie un paramètre** on peut utiliser des noms constitués d'un verbe et d'une préposition or ou to. (displayOn, printTo,...)

Quand on ne fait que **retourner une valeur** , on utilise un nom(left,size,color,...)

Les **méthodes d'accès** ont généralement des noms venant par paire : getWinner et setWinner.

Les **méthodes de test** vérifient quelque chose sur l'objet. On a des nom en is.. et has.. (isEmpty, ...)

Les **méthodes de conversion** sont généralement en as..(asString, ...)

Pour **les classes** , les noms commencent par une majuscule et sont proches de ce que l'homme comprend (ex : Voitre, Roue, ...)

Quand une classe est **abstraite** on place Abstarct quelque-part dans le nom.

Lorsqu'on utilise un **design pattern** , on utilise son nom dans le nom de la classe.

9.1.3 Formattage

Tout le monde doit utiliser une même configuration d'indentation. SI quelqu'un fait un commit après que son éditeur ait réindenter le fichier, tout le fichier sera réécrit au lieu des seules modifications apportées.

9.2 Encapsulation

Le principe de Parnas : Le développeur d'un composant ne doit fournir à l'extérieur que le strict minimum nécessaire pour que l'objet soit utilisable. Ni plus , ni moins.

Quand un bout de code en appelle un autre en connaissant tout son intimité, c'est un couplage fort. On est sans doute très dépendant de plein de petits comportements de l'objet appelé. Si on appelle la méthode d'un objet de manière polymorphique, on ne connaît rien de l'objet appelé. Ce couplage est bien plus faible.

La loi de Demeter : Si un bout de code qui en utilise un autre sait que cet autre en utilise un troisième, alors le premier et le troisième sont liés. On évite ça en s'assurant que l'interface publique d'un objet n'expose rien d'interne.! La loi de demeter dit qu'on ne peut envoyer un message qu'à soit même, un objet qu'on crée, ses attributs, les arguments qu'on reçoit ou super. Cette règle est très tricte, on ne peut pas passer "à travers" un objet.

9.3 Commentaires

Avant on apprenait à mettre beaucoup de commentaire. Maintenant on dit que moins il y a de commentaire mieux c'est. MAIS il faut commenter.

Le code auto-documenté est basé sur le fait que si on est tenté de mettre un commentaire dans son code pour expliquer ce qu'il fait c'est que le code est peut être trop compliqué.

Un puriste préfère un code lisible sans commentaires qu'un code aussi lisible mais avec commentaires.

9.4 Gestion des booléens et valeurs de retour

Parfois passer des paramètres un peu interne peut entrainer un couplage trop important. (setStatus(ON|OFF)). Il faut alors faire deux méthodes , makeOn et makeOff.

Si parfois on est tenté d'avoir des méthodes qui retournent plusieurs choses, il faut alors découper la méthode pour en avoir deux ou plus.

Une autre solution est de renvoyer un objet avec de beaux getters. Ex : Au lieu de renvoyer un font et un color, on crée un objet Style qui contient ça.

9.5 Duplication de code

: Quand on modifie un bout de code, il faut ensuite aller regarder partout où il a été copié et corriger partout. Si on copie/colle quelque chose, c'est que quelque chose cloche.

9.6 Magic numbers

: Parfois on a des nombres qui se promène dans le code (ex : des 0.21 pour le TVA). Il faut remplacer ces nombres magiques par des constantes nominées ou des méthodes qui retournent des choses.

9.7 Code gardé

: Un code gardé n'est exécuté que dans certain cas. C'est une grosse fonction placée dans un gros if.

Pour le résoudre on inverse la condition du if et on met un return. Cela permet de gagner de l'indentation et c'est plus clair.

10 Distribution et synchronisation en UML

10.1 Diagramme d'état

Un diagramme d'état représente des états, reliés par des transitions. Sur chaque transition, il y a un label "Déclencheur / Action" (ex : "Activation du backup / Activation de l'utilitaire graphique"). Un état peut avoir une activité. Quand le système rentre dans cet état, il exécute l'activité et n'en sort que quand elle est finie. (do/activité dans la description de l'état).

10.2 Diagrammes de composants et de déploiement

Représente la structure physique de l'application.

Le composants sont nommés et on un type (source, objets, exécutable, ...). Le diagramme de déploiement est le mapping entre les composants et les noeuds d'exécution de l'application. Un noeud de stockage ou d'exécution est représenté par un gros cube, contenant des composants.

10.3 Diagramme d'activité

Comme un diagramme d'état mais plus adapté à des choses avec des conditions et plus d'activités.

On a un état de début , un état de fin, des activités, puis des branch (if, avec une condition sur chaque sortie. Une seule est prise.), des join (des entrées, et on revient sur le même chemin).

Un fork lance toutes les actions filles en parallèle. Le join permet de synchro des choses.

Une sous-activité permet d'avoir un diagramme simple avec des activité haut niveau. Si on l'ouvre, on tombe sur un autre diagramme.

Une swimlane permette de séparer deux choses indépendant. (comme ce qui se passe sur un client et un serveur.)

10.4 Processus et threads

Un objet actif représente un thread ou un processus. Pour représenter un objet actif en UML, on lui fait un gros bord. Un stéréotype (= Permet de classifier des éléments UML.) peut être apposé sur quelque-chose qui représente un thread ou un processus.

Deux objets actifs peuvent communiquer de manière synchrone (bloquant) ou asynchrone.

Si plusieurs actifs appelle un passif, on peut avoir des problèmes. Pour régler ça, on utilise des stéréotypes. sequential : Les appelants doivent sérialiser leurs appels. guarded : Même chose. Sauf que c'est l'objet qui s'assure de ne traiter les choses qu'en série. concurrent : Il peut accepter tout en même temps et il sait s'arranger pour que ca marche correctement.

Un contrainte avec un égal (location=airport) est une tagged value.

11 Cycle de vie

11.1 Question - 2 points

11.1.1 Présentez le cycle de vie "Waterfall"

On procède par phase :
- **Requirements Collection**
  Rencontre avec le client et note de tous les besoins (Risques : Documentations incomplètes, inexactes et ambiguës)
- **Analysis**
  Les analystes définissent les besoins, les écrans, ... (Risques : Fournir une spécification qui ne correspond pas aux besoins du client.)
- **Design Architectes**
  On conçoit l'architecture de l'application. Conception de diagrammes et choix des librairies.
- **Implementation**
  Les codeurs développent leurs modules.
- **Testing**
  Assemblage et livraison.

Cela a donné de bon résultats mais problèmes de communication car elle est faite par documents plutôt que par la parole. Les codeurs n'ont pas de recul et ne peuvent détecter des problèmes potentiels. Si il y a une erreur il faut corriger dans chacune des étapes. Un logiciel est long à développer et les besoins du client peuvent changer durant le développement.
Avantages : Très contrôlé, panifiable, des documents décrivant l'entièreté de l'application.

11.1.2 Expliquez la différence entre ce cycle de vie et le développement itératif

Le cycle Waterfall a donné de bon résultats mais problèmes de communication car elle est faite par documents plutôt que par la parole. Les codeurs n'ont pas de recul et ne peuvent détecter des problèmes potentiels. Si il y a une erreur il faut corriger dans chacune des étapes. Un logiciel est long à développer et les besoins du client peuvent changer dur le développement.
Avantages : Très contrôlé, panifiable, des documents décrivant l'entièreté de l'application.

Tandis que le développement itératif. On procède par incréments. On livres des morceaux de logiciel au client petit à petit. C'est un enchaînement de mini waterfall. Cette méthode permet d'avoir plus de retour du client. On ne fait pas une grosse analyse, on développe juste complètement un module qu'on va montrer au client. Il vérifie et apporte ses corrections qui seront facilement faisables. On fait cela à chaque itération. Une itération dure en moyenne entre 2 et 4 semaines.
Avantages : Le client s'implique et le projet à de grosses chances de réussites.

12 OOP

12.1 Question - 2 points

12.1.1 Expliquez ce qu'est l'overloading

C'est le fait de d'implémenter deux méthodes de même nom mais de signature différente (paramètres différents).
Exemple :
- bool maFonction(val1) ;
- bool maFonction(val1, val2) ;

12.1.2 Expliquez ce qu'est l'overriding

C'est le fait de re-implémenter une méthode fille qui a déjà été implémenté dans sa classe parente, et donc de remplacer l'appel de l'ancienne méthode par la nouvelle. On peut appeler la méthode parente en appliquant super.NomDeLaMethode(...)

12.1.3 Expliquez l'utilité de ces deux concepts

L'overloading et l'overriding sont, chacune, une des particularités du polymorphisme. On peut par exemple imaginer surcharger les opérateurs +,-,*, ... pour divers classes avec l'overloading ou modifier une méthode d'une classe parente dans la classe fille.

12.2 Question - 2 points

12.2.1 Définissez le concept de polymorphisme

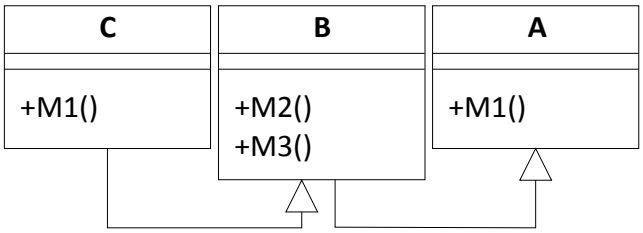
Polymorphisme : On envoie un même message à des objets de forme différente et la réponse pourra être différente.

12.2.2 Expliquez l'utilité du polymorphisme dans les langages orienté objet

Le polymorphisme est utile dans le cas où l'on a plusieurs sous classes qui héritent d'une classe virtuelle (virtuelle pure). Les méthodes de ces sous classes peuvent être redéfinies (doivent être dans le cas d'une virtuelle pure). Lorsqu'un message est envoyé, et suivant la redéfinition de la classe, le résultat ne sera pas le même.

12.3 Question - 2 points

Étant donné le diagramme de classes suivant et que les méthodes sont définies de la façon suivante :
— A - M1() : Print "A-M1"
— B - M2() : Super M1()
— B - M3() : Self M1()
— C - M1() : Print "C-M1"



12.3.1 Sachant que a est une référence vers la classe A, b est une référence vers la classe B et c une référence vers la classe C, identifiez les assignations qui sont valides

a) **a = b** Valide

b) **b = a** Erreur, c'est **b** qui hérite de **a**. Donc **b** est une sorte de **a** et pas l'inverse. (Exemple : Une **b**aleine est un **a**nimal, mais un **a**nimal n'est pas forcément une **b**aleine. L'**a**nimal pourrait être autre chose...)

c) **b = c** Valide

d) **A al = new C** Erreur, A al = C mais une référence n'enregistre pas une adresse (new).

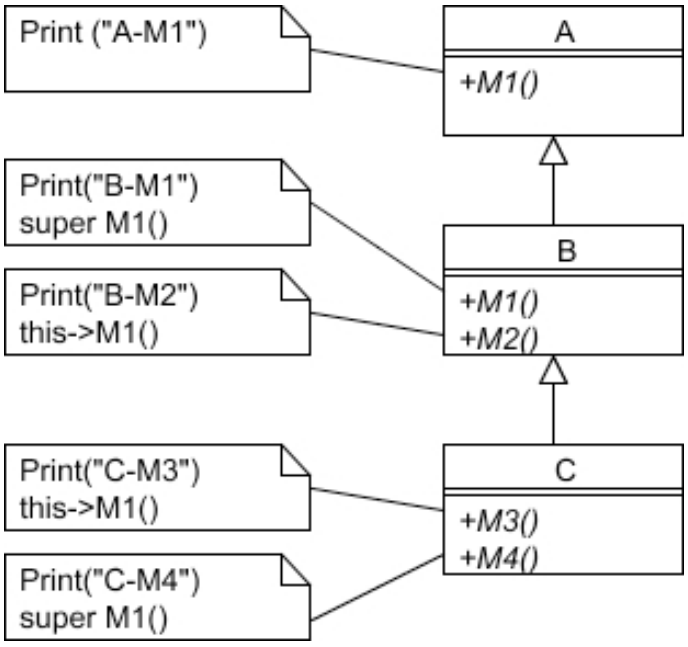
12.3.2 Déterminez ce qui est affiché lors des appels ci-dessous si a2 est une référence de classe B référençant un objet de classe C

a) **a2.M2()** B.M2() → A.M1() → Print "A-M1"

b) **a2.M3()** B.M3() → C.M1() → Print "C-M1"

12.4 Question - 2 points

Étant donné le diagramme de classes suivant et que b et c référencent respectivement des objets des classes B et C :



12.4.1 Déterminez ce qui est affiché lors des appels

a) **b.M1()** B-M1 / A-M1
b) **b.M2()** B-M2 / B-M1 / A-M1
c) **c.M3()** C-M3 / B-M1 / A-M1
d) **c.M4()** C-M4 / B-M1 / A-M1

12.4.2 Expliquez la différence entre les méthodes M3 et M4.

Dans le cas de M3, on appelle la méthode M1 sur l'objet lui-même (sur C), comme M1 n'existe pas, on va voir dans la classe parente qui est B.
Dans le cas de M4, on appelle directement la méthode de la classe parente (B).
Ici, il n'y a pas de différence étant donné que la méthode M1 n'existe pas dans la classe C ; on va donc voir dans les deux cas dans la classe parente.

12.4.3 Illustrez cette différence en étendant le diagramme UML mais sans modifier les éléments déjà existants.

En définissant C.M1() et le faisant Print("C-M1"). Ainsi C.M3() appelera C.M1() et C.M4() appelera B.M1() et on notera la différence.

12.5 Question 5 points

12.5.1 Définissez et expliquez l'utilité des concepts suivant

— **Self (this) et Super**
Self (this) référence l'objet courant de la classe.
Super référence l'objet parent de l'objet courant de la classe.
Utilité : Tout ceci sert pour le polymorphisme. Super peut servir pour redéfinir une méthode parente en l'appelant puis en changeant son résultat. Self est aussi utile pour le polymorphisme puisqu'il permet de s'assurer qu'on appellera la méthode courante si elle existe (Sinon elle se redirigea vers celle du parent).

— **Classe abstraite (Abstract class)**
Une classe est dîtes abstraite quand au moins une de ses méthodes est virtuellement pure. (type methode() = 0 ;) La classe abstraite n'est pas complète et n'est donc pas instanciable seule.
Elle sert de base aux classes filles et si la classe fille ne veut pas être abstraite elle doit redéfinir toutes les méthodes virtuelles pures de la mère. Si elle le fait, on appelle ça une concrétisation.

— **Interface**

Le principe d'une interface est donnée par l'utilisation de classe abstraite. Elle fournit une abstraction qui permettra de mettre des choses en commun entre ses filles. Cette classe ne contient que des déclarations de fonctions virtuelles et pas d'attributs, ça permet de rajouter des contraintes d'implémentation.

— **Héritage**

On dérive d'une classe parente et les sous-classes héritent de tous ses attributs et méthode. Les sous-classes peuvent surcharger les méthode pour les adapter à leurs besoins ⇒ Overriding.
But : Mettre de la structure qui amènera au polymorphisme et un code facilement maintenable. Pour savoir si B doit hériter de A, il faut pouvoir se dire : B est une sorte de A.
Il exite 2 types d'héritages :

— **Héritage simple** : Une classe hérite d'une seule et une seule classe. Cet héritage à une structure d'arbre, chaque classe a un parent et des enfants.

— **Héritage multiple** : Permet d'hériter directement de plusieurs classe. Les langages récents ne le permettent plus car plus de problème que de solution.

— **Polymorphisme**

Polymorphisme : On envoie un même message à des objets de forme différente et la réponse pourra être différente.

13 UML

13.1 Question - 2 points

13.1.1 Énumérez et expliquez le rôle des différentes vues (View) en UML. Pour chacune de ces vues donnez un exemple typique de diagramme UML utilisé.

Use case : Montre les fonctionnalités du système tel qu'elles sont perçues par un acteur externe, qui peuvent être des utilisateurs ou d'autres systèmes (Diagrammes : Use case, Activity).

Logical view : On y définit les fonctionnalités du système, les informations manipulés, ... (Diagrammes : Class Diagram, State, Sequence, Collaboration, Activity)

Component view : Comment le code est mis en boite. Les classes, les bibliothèques, fichiers de configurations, BDD, ... (Diagrammes : Diagramme de composants)

Deployment view : Indique là où les composants vont s'exécuter. Le déploiement du système dans l'architecture physique avec les ordinateurs et les appareils.

Concurrency view : Décrit comment les composants interagissent entre eux. Point de vue dynamique du système. (Diagramme de séquence)

13.2 Question – 2 points

13.2.1 Expliquez la différence entre une Association, une Agrégation et une composition dans les diagrammes de classes. Donnez, pour chacune de ces relations, un exemple en justifiant clairement pourquoi cet exemple est une utilisation adéquate de cette relation.

Association : Une association représente une relation quelconque entre deux classes ; en général, les objets d'une classe se servant de ceux d'une autre classe. (Exemple : une personne utilise un ordinateur.) Un simple trait entre les deux classes, peut être composé d'une flèche pour indiquer une direction.

Agrégation : Une agrégation permet de définir une entité comme étant liée à plusieurs entités de classe différente ; décrit une association de type « fait partie de », « a ». (Exemple : Une flotte constituée de plusieurs bateau. Une flotte n'est plus une flotte sans bateau.) Représenté par un diamant vide du coté de la classe qui agrège (ici la flotte).

Composition : Comme l'agrégation mais en plus fort. Elle fait partie entière de l'objet. Si on détruit le tout, on détruit tous les éléments. (Exemple : Si on détruit le livre, les pages n'ont plus de raison d'être, contrairement aux bateaux sans la flotte.) Représenté par un diamant noir du coté de la classe principale (ici le livre).

13.3 Question – 3 points

13.3.1 Quel diagramme UML permet de communiquer les mêmes informations que le diagramme de collaboration (collaboration diagram) ?

Le diagramme de séquence.

13.3.2 Présentez ce diagramme et expliquez la différence entre celui-ci et le diagramme de collaboration.

Le diagramme de séquence met l'accent sur le classement des messages par ordre chronologique. Ce diagramme met l'accent sur le temps alors que le diagramme de collaboration se soucie plutôt de l'espace.

13.4 Question – 3 points

13.4.1 Nommez et expliquez les trois relations pouvant relier des uses cases (Cas d'utilisation). Donnez un exemple illustrant ces 3 relations.

???

13.5 Question – 2 points

13.5.1 Décrivez la différence entre modélisation *statique* et *dynamique*

La modélisation statique montre plus la structure du système avec le découpage du code ou le lien entre classes, ... tandis que la modélisation dynamique montre plutôt l'évolution du système avec l'envoi des messages entre objets, leur durée de vie, changements d'état, ... On modélise uniquement le nécessaire.
UML offre 4 diagrammes dynamique : Sequence diagrams, Collaboration diagram, State diagrams et Activity diagrams

13.6 Question – 5 points

13.6.1 Le programme suivant résout le problème des tours de Hanoï avec un nombre de disques égal à 2. Décrivez comment se déroule son exécution à l'aide d'un diagramme de séquence puis de collaboration (en vous concentrant uniquement sur les classes Hanoi et Tour)

```
/***********************************/

class Tour {
    stack<unsigned int> s;

public:
    Tour(unsigned int);
    void push(unsigned int);
    unsigned int pop();
};

Tour::Tour(unsigned int n=0) {
    for (unsigned int i=n; i>0; --i)
        s.push(i);
}

void Tour::push(unsigned int i) {
    s.push(i);
}

unsigned int Tour::pop() {
    unsigned int res = s.top();
    s.pop();
```

```
    return res;
}

/**********************************/

class Hanoi {
    void hanoi(unsigned int, Tour&, Tour&, Tour&);

public:
    Hanoi(unsigned int);
};

void Hanoi::hanoi(unsigned int i, Tour& src, Tour& dst, Tour& aux) {
    if (i == 1) {
        unsigned int x = src.pop();
        dst.push(x);
    }
    else {
        hanoi(i-1, src, aux, dst);
        unsigned int x = src.pop();
        dst.push(x);
        hanoi(i-1, aux, dst, src);
    }
}

Hanoi::Hanoi(unsigned int n) {
    Tour a(n), b, c;
    hanoi(n, a, c, b);
}

/**********************************/

int main() {
    Hanoi(2);
    return 0;
}
```

13.7 Question – 12 points

13.7.1 Un aéroport est un ensemble de bâtiments : Terminaux, Hangars, Tour de contrôle, ... Plusieurs compagnies aériennes sont basées dans un aéroport, chacune d'entre elles ayant son ou ses hangars, son ou ses comptoirs dans le bâtiments principal, ... Une compagnie possède donc du personnel au sol pour réaliser différentes tâches (secrétariat, accueil, réparations, cuisine, ...) ainsi que du personnel volant (pilotes, steward, hôtesse). Une compagnie aérienne possède une flotte d'avions se caractérisant, entre autre, chacun par un ID, un nombre de place disponible, nombre de moteurs, type de moteurs, ...
Un client peut acheter une place à bord d'un vol, il aura alors le choix de choisir la classe dans laquelle il veut voyager (1ère, 2ème ou 3ème), un type de siège (siège-lit, ultra-confort en 1ère, confort ou standard en 2ème, tabouret ou siège plastique en 3ème), s'il désire ou non un repas et si il veut un siège coté fenêtre, coté allée centrale ou si cela lui est égal.
Un vol représente donc énormément de données : personnel volant, passagers, destination, départ, heure de départ, temps de vol, terminal de départ, ...
Les compagnies aériennes tiendront donc pour chaque jour, un liste des vols qu'elles proposent. Et pour chaque vol, la liste des passagers (avec référence à leur réservation). On vous demande de nous fournir un diagramme de classes qui reprendra un maximum de détails décrits dans ce texte ainsi que d'autres informations qui vous semblent pertinentes. Soyez attentif à montrer que votre design suit le plus possible les principes de l'orienté objet. Toute hypothèse sur ce texte devra être décrite dans votre réponse. Pensez à justifier vos éventuels choix. De plus, on vous demande dans un deuxième temps, de détailler au mieux l'objet avion d'un point de vue OO ainsi que de mentionner toutes les relations que cet avion peut avoir avec d'autres objets.

13.8 Question – 8 points

13.8.1 Modéliser à l'aide des diagrammes de collaborations et de séquences (avec les notations UML exactes) la phase de décollage pour un avion. Pour décoller l'avion doit attendre son tour (dépendant de son heure de départ) et avoir l'autorisation de la tour de contrôle pour décoller (celle-ci vérifie que la piste est libre, que le chemin mentant à la piste est libre, ...) L'avion va ensuite après une série de contrôle (moteur, ...) se mettre en mouvement afin de se placer sur la piste pour décoller, mettre les gaz, placer correctement les ailes, prendre de la vitesse et enfin décoller. Veuillez à choisir des noms de méthodes explicites et adaptés.

14 Exigence

14.1 Question - 3 points

14.1.1 Qu'est-ce qu'un SRD ?

C'est un document de spécificité des besoins, il reprend les besoins d'un utilisateur pour la création d'un programme sous forme d'un rapport écrit.

14.1.2 À quoi sert un SRD ?

Il reprend ce que les développeurs du système doivent implémenter. Il contient les exigences de l'utilisateur et du système.

14.1.3 À quoi faut-il faire attention lorsque l'on rédige un SRD ?

A être clair et précis (éviter le langage informatique que le client ne comprendra pas...). Discuter régulièrement avec le client et tenir compte des changements. Décrire ce que fait le programme et non comment (pas de détails d'implémentation). Écrire des phrases courtes, utiliser des diagrammes pour la clarté. Inclure un index, une table des matières, utiliser la voix active, éviter les fautes d'orthographes, ...

14.2 Question - 2 points

14.2.1 Donnez la définition d'exigences non fonctionnelles (Non-functional requirements)

Les exigences non concernées par les fonctions spécifiques du système.

14.2.2 Donnez les 3 types principaux d'exigences non fonctionnelles, avec un exemple pour chacun d'eux.

Exigences de produit (fiabilité, portabilité, efficacité, ...)
Exemple : fiabilité, un système de contrôle de centrale nucléaire ne doit pas planter à tout moment.
Exigences organisationnelles (implémentation, livraison, ...)
Exemple : Facilité d'utilisation, un système utilisé dans des situations critiques ou par des personnes handicapées doivent être simples d'utilisation et ne pas nécessiter de manipulations précises et compliqués.
Exigences externes (sécurité, législation, ...)
Exemple : Interopérabilité, pour des systèmes s'intégrant dans d'autres existant déjà.

15 Test

15.1 Question 3 points

15.1.1 Expliquez les termes suivants et leur utilité dans le contexte de la réalisation de tests unitaires

— **Pré-condition**
Condition booléenne qui doit être vraie avant l'appel d'une méthode.
— **Post-condition**
Condition booléenne qui doit être vraie après l'appel d'une méthode.
— **Invariant**
Condition booléenne qui doit être vraie avant et après le traitement. Il peut temporairement être faux durant le traitement. Si un des invariant est faux, c'est qu'il y a une erreur.
Le test unitaire est un test où on ne teste pas l'application mais ses composants. On va créer un code supplémentaire pour réaliser cela.
Contrairement au test unitaire, le test exploratoire est le fait de jouer avec l'application. On utilise l'interface mais on a rien prévu pour tout tester.

15.2 Question 3 points

15.2.1 Expliquez en quoi consiste le test unitaire. Donnez 3 qualités que doit avoir un test unitaire. Expliquez, pour chacune de ces qualités, la raison pour la quelle cette qualité est importante.

On va tester une classe ou un petit morceau de programme. Il existe des frameworks de test unitaires dans presque tout les langages. (CppUnit pour C++) En CppUnit, un testcase hérite de CppUnit : :TestCase. La classe du test devrai avoir le nom "NomTest". Un échec est une assertion qui rate, mais on l'avait anticipé. Une Erreur est un bug non-détecté par une assertion.
Un bon test unitaire doit avoir certaines propriétés.

— Un test unitaire doit être déterministe : A chaque lancement du test, on doit avoir le même résultat. Sinon on peut perdre confiance dans les tests.
— Le lancement doit être automatisé, on peut les relancer très facilement.
— Les test sont une source de documentation. Ils doivent donc être lisible.

— Ils doivent être le moins sensible possible aux changements du code.
— Ils ne doivent pas tester des choses évidentes mais traiter les choses complexes et difficiles en premier.

16 Conception

16.1 Question - 2 points

16.1.1 Expliquez les notions de *cohésion* et de *couplage* dans la conception de l'architecture d'une application. Que faut-il atteindre, une cohésion forte ou un couplage faible ? Expliquez.

Il vaut mieux atteindre une cohésion forte et un couplage faible. Car il est plus facile de modifier une méthode précise que devoir changer tout le code complet à cause d'un modification simple du projet.
Cohésion : Le code fait quelque chose de spécifique et très précis qui demande pas de modification si jamais on venait à changer le projet dans l'ensemble. Le logiciel est plus maintenable et plus adaptable puisque les objets sont moins dépendants de la structure interne des autres objets, ceux-ci peuvent être changés sans changer le code de leurs appelants.
Couplage : Le code se coordonne et fonctionne, si l'un change, tout doit être modifier pour garder ce couplage. Quand l'interdépendance entre les codes du projet est trop importante, la maintenance est difficile.

16.2 Question - 3 points

16.2.1 Qu'est-ce que la loi de Demeter ?

Appliquée à la programmation orientée objet, un objet **A** peut appeler une méthode d'un objet **B**, mais **A** ne peut pas utiliser **B** pour accéder à un troisième objet **C** et appeler une méthode. Faire cela signifierait que **A** a une connaissance plus grande que nécessaire de la structure interne de **B**. Au lieu de cela, **B** pourrait être modifié si nécessaire pour que **A** puisse faire la requête directement à **B**, et **B** propagera la requête au composant ou sous-composant approprié. Si la loi est appliqué, seul **B** connaît sa propre structure interne. On ne peut pas passer "à travers" un objet.

16.2.2 Donnez un exemple

Store.getItem(3568).getPrice() ⇒ Store.getItemPrice(3568)

16.2.3 Expliquez pourquoi il est bon de suivre cette loi

L'avantage de suivre la règle de Déméter est que le logiciel résultat est plus maintenable et plus adaptable (Cohésion forte). Puisque les objets sont moins dépendants de la structure interne des autres objets, ceux-ci peuvent être changés sans changer le code de leurs appelants (Couplage faible). Un désavantage de la règle de Déméter est qu'elle requiert l'écriture d'un grand nombre de petites méthodes pour propager les appels de méthodes à leurs composants. Cela peut augmenter le temps de développement initial, accroître l'espace mémoire utilisé, et notablement diminuer les performances.

17 Pattern

17.1 Question 3 points

17.1.1 Expliquez ce qu'est un design pattern

Ils permettent de remplir le fossé entre " je comprends l'orienté objet " et " je réussis à bien l'utiliser ". Ce sont des descriptions de classes communiquant entre eux et qui ont été modifiées afin de résoudre des problèmes généraux, dans un contexte particulier.
Ils ne permettent pas :
— D'obtenir une solution à un problème d'implémentation.
— D'avoir une solution toute faite et prête à être utilisée.
Car ils sont beaucoup trop génériques pour ça.
Leur but est donc principalement :

— D'améliorer sa maîtrise/son contrôle de l'orienté objet.
— De mieux se comprendre/communiquer grâce à un vocabulaire précis.

Un pattern est composé :

— D'un nom.
— D'une description du problème qu'il veut résoudre.
— De la description de la solution.
— De sa conséquence.

Il y a 23 patterns de base et on peut les combiner.
Il y a 3 grands types de designs patterns :

— Les **creationnals patterns** : pattern dont le but est de créer des objets de manière propre : en limitant leurs nombre (via singleton), leur couplage...
— Les **structurals patterns** : patterns qui permettent de résoudre les problèmes structurels. Séparation entre l'interface et l'implémentation,...
— Les **behaviours patterns** : patterns qui permettent de résoudre les problèmes au niveau du comportement, des algorithmes, des responsabilités,...

17.1.2 Énumérez deux avantages de l'utilisation des design patterns dans un projet

Communication simplifiée : il suffit de dire " j'ai utilisé le pattern X ici " pour que l'interlocuteur comprenne la structure de cette partie du système, sans devoir tout expliquer dans le détail.
On est assurés que la solution fonctionne et fonctionne bien, elle a déjà été testée souvent.

17.1.3 Présentez et expliquez l'utilité du pattern observateur (Observer pattern)

Il définit une relation entre plusieurs objets de manière à ce que lorsqu'un objet change d'état, les objets dépendants sont automatiquement mis à jour.