Ingénierie des Architectures Logicielles
Technologies & Méthodes

Travaux Dirigés

Prof. V. Englebert
Version 1.31
Université de Namur

26 mars 2019

Les énoncés utilisent parfois des noms de sociétés ou de personnes fictives. En aucun cas, ce document ne prétend désigner une personne, société, entreprise ou organisation réelle et toute ressemblance ou homonymie serait purement fortuite. Ces noms peuvent être modifiés sur simple demande des parties concernées.

1. http://creativecommons.org/licenses/by-nc-sa/3.0/fr/

Table des matières

1 Méthodes et Scénarios **4**
1.1 Scrabble™ . . . 4
1.2 Master-Data & Plexitude . . . 4
1.3 BlindJobs . . . 5

2 Indoor Architectures (design patterns) **7**
2.1 Exercice (Abstract Factory) . . . 7
2.2 Exercice (Singleton) . . . 7
2.3 Exercice (Singleton contextuel) . . . 7
2.4 Exercice (State) . . . 8
2.5 Exercice (Mediator & Façade) . . . 8
2.6 Exercice (Configuration) . . . 8
2.7 Exercice (Configuration.hiérarchique) . . . 8
2.8 Exercice (Configuration+Strategy) . . . 8
2.9 Exercice (SingletonZ) . . . 8
2.10 Exercice (Factory on singleton) . . . 8
2.11 Exercice (Singleton contextuel) . . . 9
2.12 Exercice (Factory et DB) . . . 9
2.13 Exercice (Prototype) . . . 9
2.14 Exercice (Prototype & Java) . . . 9
2.15 Exercice (Configuration & al.) . . . 10
2.16 Exercice (Singleton+Configuration+Observer) . . . 10
2.17 Exercice (Proxy distant) . . . 10
2.18 Exercice (Proxy et Java) . . . 10
2.19 Exercice (Player-Role) . . . 10
2.19.1 Question (+Observer) . . . 11
2.19.2 Question (+Bridge) . . . 11
2.20 Exercice (Flyweight) . . . 11
2.21 Exercice (Flyweight+Visitor) . . . 11
2.22 Exercice (Composite) . . . 11
2.23 Exercice (Composite+Player-Role) . . . 11
2.24 Exercice (Composite+Protype) . . . 11
2.25 Exercice (Composite & files) . . . 12
2.26 Exercice (Composite & Mereology) . . . 12
2.27 Exercice (Visitor) . . . 12
2.28 Exercice (Memento) . . . 12
2.29 Exercice (Memento pour transactions) . . . 12
2.30 Exercice (Strategy/Decorator) . . . 12
2.31 Exercice (State) . . . 13
2.32 Exercice (Memento+Command) . . . 13
2.33 Exercice (Composite+Command) . . . 13
2.34 Exercice (Command+Factory+Observer) . . . 13
2.35 Exercice (Strategy et Bridge) . . . 14
2.36 Exercice (Strategy.zip . . . 14
2.37 Exercice (Proxy+Interceptor) . . . 14
2.38 Exercice (Immutable+Prototype) . . . 14
2.39 Exercice (Graphor) . . . 14
2.40 Mises en Situation . . . 15

2.40.1 Usine d'automobiles . . . 15
2.40.2 Système de gestion des fichiers . . . 15
2.40.3 Application IB . . . 15
2.40.4 Achat en ligne . . . 15
2.40.5 Serveur web . . . 16
2.40.6 MySmart . . . 16
2.41 Énoncés mayonnaise . . . 16
2.42 Fin-Once . . . 16

3 Concurrence & Threads **17**
3.1 Occurence . . . 17
3.2 Stop & Start . . . 17
3.3 FIFO . . . 17
3.4 Winner . . . 17
3.5 Deadlock . . . 17
3.6 Orchestre . . . 17
3.7 Pool and not Poule . . . 18
3.8 Bintree . . . 18
3.9 Speedothread . . . 18
3.10 Workflow . . . 18
3.11 Petri Net . . . 19
3.12 Philosophes . . . 19
3.13 Le problème métaphysique des "philosophes et des boulets à la liégeoise congelés" . . . 19
3.14 Le verrou Mutex . . . 20
3.15 Tower Bridge . . . 20
3.16 Power Bridge . . . 20
3.17 Robots . . . 20
3.18 Pipe-Line . . . 20
3.19 Argument et synchronisation . . . 21
3.20 Tri Fusion multi-threadé . . . 21

4 Styles **21**
4.1 Bouchetoultan . . . 21
4.2 Université Futuropolis . . . 22
4.3 Povray . . . 22

1 Méthodes et Scénarios

Pour chaque énoncé, identifiez les attributs de qualité non fonctionnels et précisez la sémantique de chacun avec un (ou plusieurs) scénario tel qu'expliqué dans le cours théorique.

1.1 Scrabble^{TM}

Concevez l'architecture d'un système permettant à des candidats de former des équipes de binômes afin de jouer anonymement au Scrabble^{TM}. Le système sera déployé dans un premier temps pour un public restreint de quelques centaines de personnes. Il sera par la suite étendu pour être accessible au plus grand nombre sur différents continents. L'interaction avec le système devra être fluide et adaptée à un public de plus de 7 ans.
Solution

- anonymement
- scalability
- utilisable dans le monde entier (latence, etc)
- Interface fluide.

1.2 Master-Data & Plexitude

La Plexitude est un pays organisé en différentes structures (provinces, régions, communautés, villes, fédéral, ...). Chaque entité dispose de ses propres données. Ces données représentent des informations extrêmement complexes et volumineuses. Nous souhaiterions développer un mécanisme permettant de factoriser l'information de sorte à éviter les doublons et les incohérences entre les différents niveaux de pouvoir, sans altérer le fonctionnement des services existants et en minimisant les modifications du back-office actuel. Il va de soi que le nouveau système doit préserver la même degré de confidentialité.
Solution

- NE PAS ALTÉRER LE FONCTIONNEMENT DES SERVICES EXISTANTS.

  Source du stimulus : L'équipe de développement

  Stimulus : Ajout du mécanisme de factorisation

  Artefact : Les services existants avant la modification

  Environnement : — Les services profitent toujours du même support de la part des équipes de gestion.

  Réponse : Les services continuent à répondre aux demandes.

  Mesure : 100% de réponses correctes sans dégradation de la qualité de service.

- MINIMISER LES MODIFICATIONS DU BACK-OFFICE ACTUEL.

  Source du stimulus : Le gouvernement de la Plexitude

  Stimulus : Demande d'une nouvelle architecture avec factorisation

  Artefact : Les back-office de chaque entité.

  Environnement : — Les responsables de chaque entité acceptent de suivre les recommandations.
  — Les recommandations sont clairement énoncées et formalisées.

— Le gouvernement répond dans les 24H à toutes les demandes des différentes entités concernant la nouvelle architecture.

Réponse : Chaque entité fournit une API Façade offrant les fonctionnalités requises pour le développement du nouveau service.

Mesure : — Chaque API fournie par les entités est testée avec succès par le service informatique fédéral selon les spécifications données en début (cf. environnement).
— Le temps d'adaptation de chaque back-office (en dehors de la façade) a nécessité moins de 3 jour/personne.

Source du stimulus :

Stimulus :

Artefact :

Environnement :

Réponse :

Mesure :

1.3 BlindJobs

La société BlindJobs vient en aide aux personnes souffrant d'un déficit visuel (handicap visuel, cécité, etc) afin de les aider à s'intégrer dans la société, en particulier, en les mettant en contact avec des employeurs. Notre outil est actuellement un système développé en langage C sous AIX[2] et offrant uniquement des fonctionnalités à nos employés afin de les aider efficacement lors de leurs rencontres avec les clients (c-à-d. les personnes handicapées ou les entreprises). Nous souhaiterions ouvrir cette plate-forme vers nos clients afin qu'ils puissent directement l'utiliser via le Web, leur smartphone ou des guichets aménagés de logiciels spécifiques. L'utilisabilité de notre site est évidemment un enjeu crucial, le nouveau système devra être interopérable avec l'ancien, et être développé exclusivement avec des outils open source. L'outil permettra de rechercher des offres d'emploi compatibles avec certains handicaps et sur base de compétences professionnelles, de soumettre des CV, et de développer des réseaux sociaux sur base des caractéristiques trouvées dans le CV. Il est également important que le nouveau système n'altère pas l'intégrité des données de l'ancien système. Enfin, la confidentialité et le respect de la vie privée sont des préoccupations importantes pour nous.
Solution

— UTILISABILITÉ PAR DES PERSONNES HANDICAPÉES
  — quels types d'handicap, compétence cognitive, âge, etc
  — comment confectionner le panel de test ?

Source du stimulus : Un panel représentatif de personnes handicapées

Stimulus : Un jeu d'interactions représentatif de l'utilisation du système.

Artefact : Le site Web / Smartphone (répliquer le scénario si besoin)

Environnement : Le système fonctionne sans problème particulier.

Réponse : le système se présente et répond de manière adaptée au profil des utilisateurs handicapés

Mesure : — Pas de déviance par rapport à un panel d'utilisateurs non handicapés

2. https://fr.wikipedia.org/wiki/AIX.

  - Taux de satisfaction semblable à celui d'un panel d'utilisateurs non handicapés

— LE NOUVEAU SYSTÈME DEVRA ÊTRE INTEROPÉRABLE AVEC L'ANCIEN

Source du stimulus : Le site WEB

Stimulus : des requêtes

Artefact : l'ancien système

Environnement : fonctionnement normal

Réponse : réponses aux requêtes

Mesure : aucune déviance observée

— UNIQUEMENT DES OUTILS OPEN SOURCE

Source du stimulus : Direction

Stimulus : Le système doit être open source

Artefact : Le site WEB, Apps

Environnement : néant

Réponse : tous les développements sont open source

Mesure : Toutes les librairies utilisées ou autres artefacts (graphique, etc) ont une licence open source non virale.

— PRÉSERVER L'INTÉGRITÉ DES DONNÉES DE L'ANCIEN SYSTÈME.

Source du stimulus : Direction

Stimulus : Demande de préserver l'intégrité des données.

Artefact : Ancien système

Environnement : Le système est équipé d'espaces de stockage suffisants.

Réponse : Backup journalier et incrémental avec journalisation des requêtes du jour.

Mesure : Le backup et le journal sont réalisés chaque jour.

Source du stimulus : Responsable qualité

Stimulus : Données corrompues suite à une requête

Artefact : Le système

Environnement : Les backups et journaux sont disponibles.

Réponse : Le système est arrêté, le backup est restauré et le journal expurgé des requêtes incriminées est rejoué.

Mesure : L'interruption de service dure moins d'une heure.

— CONFIDENTIALITÉ.

Source du stimulus :

Stimulus :

Artefact :

Environnement :

Réponse :

Mesure :

— RESPECT DE LA VIE PRIVÉE.

Source du stimulus :
Stimulus :
Artefact :
Environnement :
Réponse :
Mesure :

2 Indoor Architectures (design patterns)

Les codes sources java des patterns sont disponibles sous l'URL suivante : `https://github.com/VincentEnglebert/teaching/tree/master/ialtem/patterns/src`.

2.1 Exercice (Abstract Factory)

Une simple fenêtre **Window** est composée de boutons, de libellés et de zones d'édition de texte. Dans la plateforme Java il existe deux types de bibliothèques graphiques pour la création des interfaces utilisateurs à savoir **Swing** et **SWT**.
Implémentez une solution basée sur l'**Abstract Factory** qui permet de créer ces deux types de fenêtres chacune avec ses composants correspondants aux librairies Swing ou SWT.

Question

Comment pouvez-vous utiliser les facilités offertes par le langage Java (ou autre) pour renforcer les qualités attendues de ce pattern (les modes private/protected/public, visibilité des packages, visibilité par défaut, classes publiques ou privées, classes vs interfaces, ...).

2.2 Exercice (Singleton)

Complétez votre solution de l'exercice 2.1 afin que le choix et la création de la factory se fasse dans un **Singleton**.

2.3 Exercice (Singleton contextuel)

Le singleton a vocation à retourner une seule instance pour toute la portée d'une application. Nous pourrions envisager que ce singleton ne plus si unique que cela, mais unique seulement à un certain contexte. Nous pourrions imaginer que dans une application décomposée en trois couches, chacune étant implémentée dans des packages Java différents, chaque couche puisse disposer de son propre singleton.
Comment, sans modifier la signature des méthodes du pattern, serait-il possible que la méthode `getInstance()` retourne des instances différentes selon la nature de l'appelant ?

Indice : Si vous créer un objet de type `Exception`, vous avez accès au stacktrace depuis **main** jusqu'à l'appel.

Nota Bene : Ce type de question n'est pas représentatif de ce qui sera demandé à l'examen.

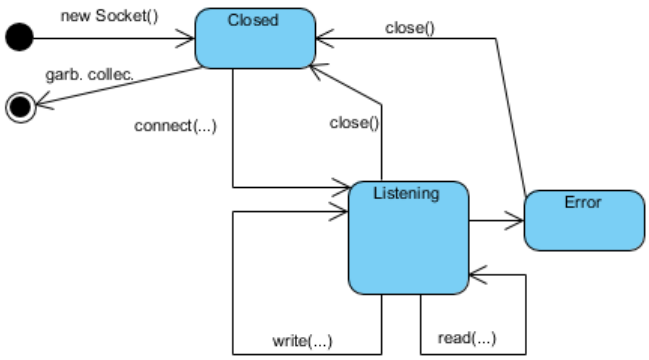

FIGURE 1 – Machine à états d'une connexion TCP

2.4 Exercice (State)

TCP est un protocole de transfert fiable via le réseau. Dans ce contexte, l'utilisation d'une connexion TCP sur un port spécifique peut être sprécifié avec la machine à états de la figure 1.
Implémentez cet énoncé par un programme Java avec le design pattern **State** en vous référant à l'API Java.

2.5 Exercice (Mediator & Façade)

Les patterns ***Mediator*** et ***Façade*** présentent des similitudes dans la mesure où les classes `médiateur` et `façade` ont vocation à devenir les seuls points d'accès aux autres classes du pattern. Discutez les autres points de similitude ou de différenciation.

2.6 Exercice (Configuration)

Le modèle **ISO** est une norme de communication réseau qui impose l'organisation de chaque entité communicante en 7 couches. Les couches doivent pouvoir être utilisées du haut vers le bas et réciproquement (c-à-d. envoi et réception). Chaque couche offre donc des primitives `send(...)` et `receive(...)` (entre autres).
Proposez une implémentation du design pattern **Configuration** pour permettre la configuration des trois premières couches du modèle ISO (couches Physique, Données et Réseau).

2.7 Exercice (Configuration.hiérarchique)

Repensez l'application du pattern ***Configuration*** de sorte qu'une configuration puisse être organisée de manière hiérarchique à la façon d'une « matriochka » et que chaque couche puisse profiter du même pattern.

2.8 Exercice (Configuration+Strategy)

Intégrez le pattern ***Strategy*** au pattern ***Configuration*** de sorte à ce que la configuration des composants puisse varier selon le contexte.

2.9 Exercice (SingletonZ)

Comment pourrait-on généraliser le pattern ***Singleton*** pour en gérer plusieurs selon un certain contexte ? Exemple : un serveur a été conçu pour ne supporter qu'une session à la fois et la session est définie comme un objet singleton. Cependant l'évolution du système nécessite maintenant de gérer plusieurs sessions en parallèle, à raison d'une session par adresse IP du client.

2.10 Exercice (Factory on singleton)

Modifiez le pattern ***Abstract Factory*** afin de pouvoir trouver aisément une factory particulière, conçue comme un ***Singleton***.

2.11 Exercice (Singleton contextuel)

Proposez une implémentation du pattern ***Singleton*** de sorte à produire un singleton en fonction du contexte de l'appelant. On pourrait imaginer que le singleton retourné soit spécifique à la classe de l'objet qui appelle la méthode statique.
Nota Bene : *Cet énoncé est spécifique au langage Java et nécessite l'emploi d'exceptions Java. Il n'est pas représentatif des questions posées lors de l'examen.*

2.12 Exercice (Factory et DB)

Le modèle de données de notre système est constitué de ces différentes classes : `client`, `facture` et `produit`. Nous souhaiterions gérer la persistance de ces classes avec différents frameworks (JPA, Hibernate, JDBC, ...) ou de manière « transient[3] » de la manière la plus transparente possible. Malheureusement, chaque framework nécessite de modifier certaines méthodes. Expliquez comment le recours au pattern ***Abstract Factory*** permet de remédier élégamment à cette situation.

2.13 Exercice (Prototype)

Concevez des classes selon le pattern ***Prototype*** afin de concevoir le décor d'un jeu vidéo. Les acteurs seront amenés à évoluer dans une ville composée de quartiers, composés de bâtiments, composés d'étages, composés de pièces, etc.
Comparez la version avec prototypes avec la vesion avec constructeurs.

2.14 Exercice (Prototype & Java)

Java n'est pas un langage orienté prototypes. Comment pourrait-on mettre en œuvre ce paradigme en Java de manière générique (réutilisable) avec le design pattern ***Prototype*** ?
D'autres langages comme Groovy, Ruby, Python ou Scala proposent-ils des facilités pour mettre en œuvre ce pattern ?

Question (Proto et types)

Complétez si nécessaire la solution précédente afin d'expliciter le concept de « type de prototype ».

3. C-à-d. uniquement en mémoire vive.

Question (Proto et méthodes)

Complétez si nécessaire la solution précédente afin de pouvoir doter un prototype de méthodes que l'on puisse adapter ou modifier au runtime après clonage.
Indices : songez aux méthodes anonymes, aux lambda expressions [4] ...

Question (Proto et héritage)

Complétez le mécanisme précédent afin d'introduire de l'héritage multiple entre « types de prototypes ».

Question (Proto et surcharge)

Complétez le mécanisme précédent afin qu'un appel de méthode non implémenté puisse avoir un comportement par défaut et qui puisse éventuellement être « surchargé » (c-à-d. implémenté autrement par le programmeur).

2.15 Exercice (Configuration & al.)

Montrez comment le pattern ***Configuration*** peut aider à construire d'autres patterns.
textbfIndice : songez à l'***Observer***, au ***Mediator***, à l'***Interceptor***, ...

2.16 Exercice (Singleton+Configuration+Observer)

Combinez les design patterns ***Singleton***, ***Configuration*** et ***Observer*** afin de pouvoir changer une configuration dynamiquement et en informer des observateurs.

2.17 Exercice (Proxy distant)

Concevez une implémentation du pattern ***Proxy*** afin de permettre un accès transparent à des objets de la classe Client depuis une application distante (via internet) :

```
class Client{
  private final String name=null;
  private float amount=0.0;
  public Client(String name}{
    assert name!=null;
    this.name=name;
  }
  public String getName() {...};
  public float applyTax(float tax){...};
}
```

2.18 Exercice (Proxy et Java)

Certains langages de programmation procurent des facilités pour générer des classes proxy à la volée depuis une classe donnée. Examinez les langages Java, Scala, Ruby, Groovy et Python pour découvrir leurs facilités.

4. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

2.19 Exercice (Player-Role)

Questions

- L'objet Player doit-il être détruit lorsqu'il ne joue plus aucun rôle ?
- Comment pouvez-vous implémenter efficacement la relation entre les rôles et le player ?
- Des rôles distincts peuvent avoir des informations d'état en commun. Comment gérer cela efficacement ?
- Comparez les pros/cons de ce pattern avec l'alternative qui consiste à faire implémenter tous les rôles comme interface Java par une même classe.
- Faut-il donner accès à l'objet de type Player à l'applicatif ? Et si non, comment procéder ?
- Comment empêcher l'applicatif de poursuivre à utiliser un rôle qui a été « détaché » de son player ?

2.19.1 Question (+Observer)

Combinez les patterns ***Player-Role*** et ***Observer*** afin que chaque rôle puisse être notifié chaque fois que l'entité principale est modifiée (par l'un des rôles par exemple).

2.19.2 Question (+Bridge)

Comment le pattern, ***Bridge*** permettrait-il de détecter l'usage d'objets rôle alors qu'ils ont été détachés de l'objet principal.

2.20 Exercice (Flyweight)

Imaginez la structure de données la plus économique pour un logiciel de traitement de texte permettant de gérer du texte structuré (caractères, paragraphes, pages) avec des mises en formes complexes (polices, attributs, alignement, etc) en utilisant le pattern ***Flyweight***.

2.21 Exercice (Flyweight+Visitor)

Dans l'exercice 2.20, utilisez le pattern ***Visitor*** pour ajouter des fonctionnalités telles que *Imprimer*, *Compter nombre de mots*, *Vérification grammaticale*.

2.22 Exercice (Composite)

Utilisez le pattern ***Composite*** afin de représenter l'arbre syntaxique du langage dont la BNF est donnée ci-après :

```
langage ::= Affect*
affect ::= Var '=' E
E ::= nombre | E '+' E | E '-' E | E '*' E | E '/' E | '(' E ')' | Var
```

Question 1

Comment peut-on vérifier que l'arbre syntaxique est bien construit quelque soit l'usage de l'API que vous fournissez à l'utilisateur de cette structure de données.

Question 2

Utilisez le pattern ***Visitor*** pour évaluer ce langage (selon votre sémantique), pour suggérer du refactoring ($x + x + x \rightarrow 3 \times x$, $x \times 0 \rightarrow 0$, ...), ou pour normaliser le langage en produisant un autre programme où par exemple les expressions composées seraient distribuées.

2.23 Exercice (Composite+Player-Role)

Dans le pattern ***Composite***, la nature « feuille » ou « composée » d'un noeud est une propriété intrinsèque. Lorsqu'une feuille change et devient composée, il faut alors détruire la feuille, créer un autre type de noeud et transférer les informations. Pouvez-vous améliorer cette situation en intégrant le ***Payer-Role*** ?

2.24 Exercice (Composite+Protype)

Comme dans l'exercice 2.23, tentez de corriger la situation mais en utilisant le pattern ***prototype***.

Indice Référez-vous à l'exercice 2.14.

2.25 Exercice (Composite & files)

Utilisez le pattern ***Composite*** pour représenter le contenu d'un fichier structure XML ou JSON.

2.26 Exercice (Composite & Mereology)

Après vous être renseigné sur la méréologie, comment pourriez-vous améliorer le pattern ***Composite*** pour affiner sa sémantique ?

Indice Visitez la page `en.wikipedia.org/wiki/Mereology`.

2.27 Exercice (Visitor)

Utilisez le pattern ***Visitor*** avec le pattern ***Configuration*** afin de vous assurer qu'une configuration établie répond bien aux prescriptions initiales (invariant d'assemblage).

2.28 Exercice (Memento)

Utilisez le pattern ***Memento*** dans la solution de l'exercice 2.22 afin de pouvoir éditer la structure de données et de pouvoir la restaurer à des états antérieurs.

2.29 Exercice (Memento pour transactions)

Utilisez le pattern ***Memento*** pour gérer l'*isolation* de transactions concurrentes accédant à des ressources partagées.

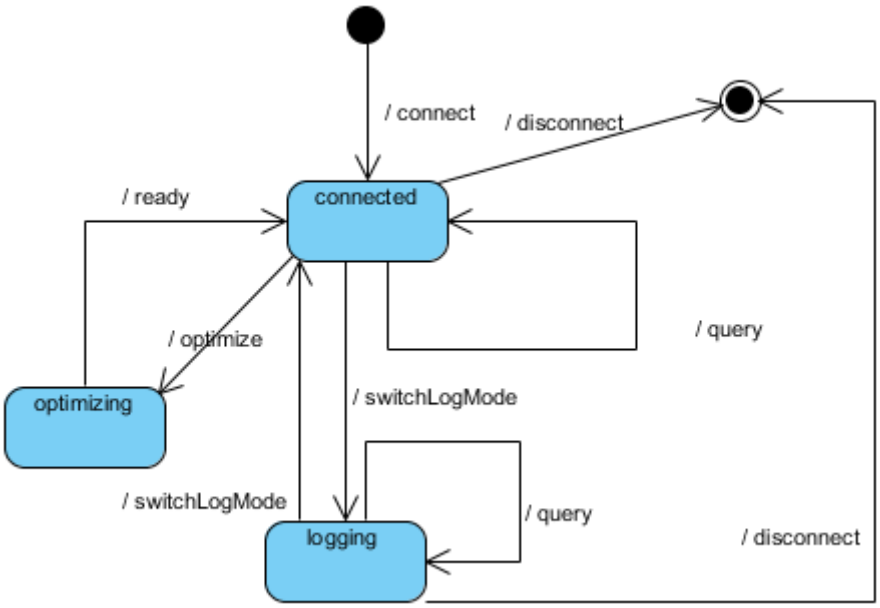

FIGURE 2 – Machine à états de la classe Server

Principe Les clients utilisent un gestionnaire de transactions. Ce gestionnaire a pour objet de créer et de commiter les transactions si les conditions le permettent (aspect ignoré dans cette question) ou de les avorter sinon. Lorsqu'une transaction accède à une ressource pour la première fois, cette dernière crée un objet memento qui sera l'état de la ressource tel que manipulé par cette transaction de manière exclusive. D'autres transactions pourront utiliser cette même ressource, mais avec leurs mementos respectifs. Ceci garantit donc le principe d'isolation entre les transactions, elles ne peuvent prendre connaissance des effets d'autres transactions, car elles restent cantonnées à des mementos spécifiques. Lors du commit, l'état de la ressource est officiellement promu à celui du memento associé à la transaction commitée. Si la transaction est avortée, le memento est simplement ignoré et "détruit" (cela dépend du langage de programmation et de la technologie utilisée).

2.30 Exercice (Strategy/Decorator)

Dans un diagramme de classes représentant des graphes au sens conceptuel (noeud/arc), si l'on souhaite que les noeuds puissent être représentés par une forme laissée au choix de l'utilisateur, nous pourrions mettre en oeuvre ce besoin avec le pattern ***Strategy*** ou ***Decorator***.

1. Concevez le diagramme de classes correspondant à chaque option.
2. Comparer les avantages et désavantages de chaque solution.

2.31 Exercice (State)

L'implémentation de l'interface Server doit vérifier la machine à états décrite dans la figure 2 — la méthode `ready()` est privée et réservée à l'implémentation. Utilisez le pattern ***State*** afin de mettre en oeuvre proprement ce problème.

```
interface Server {
  public void connect();
  public void disconnect();
  public void query();
```

```
  public void optimize();
  public void switchLogMode();
}
```

Question 1

Si vous examinez l'implémentation de la méthode `query()` :

- Quelles difficultés avez-vous rencontrées ?
  - Pensez-vous que les implémentations de ces deux méthodes diffèrent fortement ?
  - Comment pourrait-on procéder pour éviter ce problème ?
- Quels ont été les avantages de ce pattern ?
- Que faire si une méthode est invoquée dans un état qui ne devrait pas le permettre ?

Question 2

Complétez la solution précédente de sorte que certains objets puissent être notifiés des changements d'état des objets Server.

Question 3

Discutez l'alternative qui consisterait à implémenter les comportements des différents états comme des variations supportées par le pattern ***Bridge***

2.32 Exercice (Memento+Command)

Soit une classe C qui est conforme au pattern ***Memento***, utilisez le pattern ***Command*** pour permettre un nombre indéfini de Undo/Redo sur C.

2.33 Exercice (Composite+Command)

Utilisez le pattern ***Composite*** afin d'organiser les commandes du pattern ***Command*** en hiérarchies, de sorte à avoir des transactions imbriquées.

2.34 Exercice (Command+Factory+Observer)

Combinez les patterns ***Command***, ***Abstract Factory*** et ***Observer*** de sorte à faciliter la notification d'acteurs extérieurs lorsque certaines opérations sont réalisées (sauver, quitter, etc), et ce dans des contextes différents (en production versus en développement par exemple).

2.35 Exercice (Strategy et Bridge)

Quelle différence observez-vous entre les patterns ***Strategy*** et ***Bridge***.

2.36 Exercice (Strategy.zip

Vous disposez d'une interface `Compress`, mettez-la en œuvre en utilisant le pattern ***Strategy*** de sorte à pouvoir choisir entre différents algorithmes dynamiquement : *zip*, *7zip*, *gz*, *arc*, ...

```
interface Compress {
  public File compress(File file);
}
```

2.37 Exercice (Proxy+Interceptor)

Complétez la solution de l'exercice 2.17 avec le pattern ***Interceptor*** afin que l'on puisse échanger les informations entre le proxy et le serveur de manière confidentielle, et en choisissant la technologie de chiffrement dynamiquement.

2.38 Exercice (Immutable+Prototype)

Combinez les patterns ***Immutable*** et ***Prototype*** afin d'avoir des prototypes immutables. Comment peut-on dès lors les modifier ? Quel intérêt peut-on trouver à ce type d'objet ?

2.39 Exercice (Graphor)

L'application Graphor est un outil de dessin vectoriel permettant de dessiner des modèles de type UML, BPMN, ER, etc. Il doit pouvoir supporter les fonctionnalités suivantes :

1) Un même graphe doit pouvoir être édité simultanément selon plusieurs mises en formes [5].
2) Les formes vectorielles peuvent être assemblées librement et être connectées entre elles par des arcs.
3) Une forme peut être visible de manière détaillée (son contenu est affiché) ou condensée (représentée par un pictogramme).
4) Une forme peut être ajoutée à une librairie pour être réemployée ultérieurement.
5) Une forme sélectionnée change d'apparence et est « auréolée » d'une ombre.
6) Les formes doivent pouvoir être déplacées (avec leur contenu) par un drag&drop.
7) Les formes contenues dans une autre doivent pouvoir être déplacées par un SHIFT-drag&drop (par exemple, dans un diagramme de classes, déplacer une classe d'un package vers un autre package).
8) Sauver et charger les graphes dans différents formats (XML, sérialisation Java, etc).
9) Charger et re-charger des mises en forme pour un type de graphe.
10) Pouvoir copier/coller une partie d'un graphe dans un autre graphe.
11) Pouvoir faire des undo/redo à volonté sur les opérations d'édition.
12) Pouvoir charger dynamiquement des plugins afin d'exécuter des opérations sur le graphe courant (cela nécessite une connaissance plus approfondie de Java).
13) Les graphes devront pouvoir être imprimés.
14) Les actions d'édition seront proposées aussi bien dans le menu général (en haut) que dans le menu contextuel, accessible par un click droit.

Concevez l'ossature de l'application comme un diagramme de classes en utilisant les design patterns les plus appropriés pour chaque exigence.

2.40 Mises en Situation

Pour chaque énoncé précisez quel est le design pattern le plus approprié et modélisez votre choix par un diagramme de classes.

5. Une mise en forme consiste en un agencement spécifique des formes dans un vue (*layout* en anglais). Cet agencement résulte d'un placement manuel par l'utilisateur selon certaines préférences.

2.40.1 Usine d'automobiles

Une usine qui fabrique plusieurs types de véhicules (automatique, manuel, diesel, essence, électrique) décide de concevoir son usine avec des robots collaboratifs (cobot [6]) car ils sont rentables, sûrs, faciles à programmer, flexibles, simples à déployer et facilitent l'automatisation. Ainsi, chaque pièce d'une voiture est construite par un robot générique, indépendamment du type de voiture. Ce robot est contrôlé par plusieurs composants logiciels adaptés aux différents types de véhicules à produire. L'application du robot utilise des composants logiciels (par ex. des objets) contrôleur, capteur et actionneur de type diesel pour fabriquer des pièces pour des automobiles de ce type mais les véhicules électriques sont par contre montés par le même robot mais dirigé par un contrôleur, capteur et actionneur de type électrique.

2.40.2 Système de gestion des fichiers

En informatique un répertoire ou un dossier est une sorte de classeurs dans lesquels on peut ranger d'autres dossiers ou des fichiers. Cette structure est hiérarchique et la base est appelée racine.

2.40.3 Application IB

Dans l'application de commerce en ligne **IB**, une grande variété d'articles est mise à disposition des clients. Les clients sélectionnent les produits à acheter en les plaçant dans un « panier ». Une fois que le panier est validé, il existe plusieurs méthodes de paiement au choix du client : paiement par carte de crédit (Mastercard, Visa CARD), PayPal, Bitcoin...

2.40.4 Achat en ligne

Dans le contexte des applications d'achat en ligne, le client peut choisir plusieurs items distincts et de natures différentes : Livres, CDs, produits alimentaires, vêtements, électroménagers, produits immatériels (mp3, mp4, etc).
Chaque type d'item possède des caractéristiques différentes. Par exemple, un livre est identifié par son ISBN et possède un titre, un ou plusieurs auteurs et une maison d'édition. les produits alimentaires sont décrits par leur nom, leur type (fruit, végétaux, viandes), le pays d'origine. Les articles des vêtements sont définis par leur genre (prêt à porter, de luxe) leur sexe (Homme ou Femme), leur saison et leur taille. Finalement, les produits immatériels (mp3) sont caractérisés par leur nom, leur durée, etc.
Pour tous ces types d'articles, nous souhaitons pouvoir opérer des traitements de masse comme des opérations de d'inventaire, de production de catalogue papier, de génération de pages HTML, de réapprovisionnement, ...

2.40.5 Serveur web

Le processus de traitement typique d'un serveur Web consiste à recevoir un URI du navigateur, à lui faire correspondre un fichier sur disque, à ouvrir le fichier et à envoyer son contenu au navigateur (client).
Nous souhaitons que dans notre serveur web, chacune de ces étapes puisse être remplacée ou modifiée, par exemple en remplaçant la manière dont les URI sont traduites en noms de fichiers, en (dé)chiffrant le contenu des fichiers, en adaptant le contenu des fichiers au type de navigateur (station ou smarphone), etc.

6. https://www.sirris.be/fr/cobots

2.40.6 MySmart

La société **MySmart** est en train de faire un gros coup de Kick-starter et s'apprête à vendre des objets connectés (bracelets, smartphones,etc) qui peuvent être assemblés sur mesure à la demande du client.
Le bon fonctionnement de ces objets est rendu possible par un contrôleur logiciel fait maison qui pilote l'ensemble des connecteurs logiciels liés aux composants électroniques : un ensemble de capteurs à savoir capteur luminosité, capteur de température, capteur cardiaque et capteur d'activités pour mesurer le nombre des pas, de minute d'activités et nombre de calories brûlés. Le kit contient aussi un accéléromètre, une boussole et une puce WIFI pour offrir différentes possibilités de fabrication.
Proposer alors un diagramme de classes le plus adéquat du logiciel embarqué dans ce type d'objets connectés.

2.41 Énoncés mayonnaise

Pour chaque exercice :

1. Citez les design patterns qui se prêtent le plus à mettre en œuvre chaque énoncé.
2. Produisez le diagramme de classes qui intègre ces patterns avec un maximum de détails (associations, cardinalités, attributs, méthodes, etc). Indiquez quelles classes représentent chaque pattern.
3. Produisez un « sequence diagram » qui décrit les comportements représentatifs de ces solutions.

2.42 Fin-Once

Fin-Once développe et vend une application pour des entreprises dans le monde de la finance. La création du schéma de la BD est effectuée automatiquement pour insérer de nouvelles tables, vues et triggers. Mais une entreprise cliente peut vouloir stocker les données de ses factures dans différents types de bases de données. En effet, dans un environnement Windows une BD **MySQL** sera mise en place alors que pour les systèmes d'exploitation Unix une BD **PostgreSQL** sera préférée (par exemple).
Une fois le schéma créé, pour chaque table présente dans la BD, il existe une classe correspondante dans le programme java dont les objets représenteront les lignes stockées dans ces tables et utilisées à un moment t. Un objet d'audit doit être notifié chaque fois que des données dans la BD sont modifiées.
Le système informatique de la société assemble différents composants bien coordonnés : La BD (MySQL ou PostgreSQL), un serveur d'authentification pour gérer les identités des utilisateurs et leurs droits d'accès ainsi qu'un serveur transactionnel qui coordonne les transactions à exécuter sur la BD.

3 Concurrence & Threads

3.1 Occurence

Une application reçoit une liste de fichiers et un mot et doit rechercher les occurrences de ce mot dans les différents fichiers.

Exemple :

```
>java Search unamur exam.txt lesson.txt agenda.txt
```

Écrire une architecture qui initie les recherches en parallèle par fichier dans un thread différent. Le nombre de fichiers est inconnu.

3.2 Stop & Start

Écrire une architecture qui initie l'exécution de deux threads et attend la terminaison de ceux-ci. Généraliser à N threads.

3.3 FIFO

Écrire une file FIFO partagée par plusieurs threads avec stage d'attente quand la file est vide (ou pleine).

3.4 Winner

Un programme doit réaliser une tâche, cette dernière peut être réalisée par trois algorithmes différents aux performances aléatoires [7]. Implémentez ce programme de sorte qu'il puisse exploiter le premier résultat obtenu et interrompre proprement les autres tâches.

3.5 Deadlock

Illustrer par un programme comment un deadlock peut survenir entre deux threads.

3.6 Orchestre

Un orchestre est composé de N musiciens et d'un chef d'orchestre. Le chef d'orchestre lit une partition et précise quelle note chaque musicien doit interpréter à un moment précis et quelle est sa durée. Écrivez un simulateur en Java de cette situation où chaque acteur est un thread. Vous pourrez utiliser l'API Java Sound pour avoir une mise en oeuvre réaliste, mais dans un premier temps, on se contentera de logger des messages d'information par chaque acteur.

3.7 Pool and not Poule

Concevez l'architecture qui permet de confier la gestion de requêtes à un pool de threads de taille fixe. Un agent émet des requêtes de manière sporadiques, et les confie à un pool qui traite ces requêtes le plus vite possible en exploitant le parallélisme entre les threads du pool. Les threads doivent être créés une fois pour toute en début de programme. Que se passe-t-il si la fréquence d'émission des requêtes est telle que l'agent émet plus de requêtes que le pool ne peut en traiter.

7. Par exemple, réseaux neuronaux, algorithmes génériques, recuit simulé, etc.

3.8 Bintree

Soit un arbre binaire défini par la classe :

```
class ArbreBinaire {
        public ArbreBinaire gauche;
        public ArbreBinaire droite;
        public Node noeud;
        ...
}
```

Rechercher la présence d'une valeur dans un tel arbre revient à comparer le noeud principal et sinon à parcourir les deux sous arbres. Écrivez le programme qui implémente une telle recherche en confiant à deux threads la recherche dans les sous-arbres.

- Lorsqu'un thread a trouvé une valeur, il est inutile de laisser l'autre thread poursuivre sa recherche. Comment pourrait-on le stopper ?
- Lorsqu'un thread a terminé sa recherche en vain, aménagez votre algorithme afin que ce thread puisse être affecté à une autre recherche dans une autre partie de l'arbre.
- Généralisez votre solution afin de confier la recherche à un pool de threads de taille fixe.
- Comment pourrait-on comparer cet algorithme parallèle avec la version séquentielle ?

3.9 Speedothread

Soient trois threads P, Q et R. Chacun boucle en écrivant son nom à l'écran. Comment s'assurer que P imprime toujours moins de fois son nom que la somme des impressions de Q et R ?

3.10 Workflow

Soit le flux de documents suivant :

> *L*a tâche CHECK reçoit des documents et vérifie leur conformité. Si le document est conforme, il est confié à la tâche DISPATCH, sinon à la tâche TRASH. La tâche DISPATCH envoie le document vers les tâches BILL, SEND ou ORDER selon l'indication présente dans le document dans le champ STATUS (choix exclusif). La tâche ORDER envoie pour chaque document reçu un document à SEND et BILL. Lorsque ces deux tâches ont terminé de traiter 2 documents apparentés, la tâche BACKUP écrit dans un fichier qu'un document a été traité.

- Écrivez le programme Java qui simule ce workflow avec des threads, chaque tâche étant exécutée par une même personne qui ne peut traiter qu'un document à la fois.
- Y a-t-il un risque qu'une tâche ne traite jamais un document ? Si oui, corrigez votre solution.
- Peut-on fournir une estimation de la borne supérieure de la taille mémoire (ram, disque, peu importe) nécessaire à toute exécution de ce type de workflow ?
- Peut-on concevoir une architecture plus générique dont les composants peuvent être réutilisés pour d'autres énoncés de ce type.
- Comment qualifieriez-vous le style de votre architecture ?

3.11 Petri Net

Les réseaux de Petri[8] sont souvent utilisés pour décrire ou formaliser des workflows, leur formulation permet en effet de représenter l'important parallélisme des workflows. Dans un réseau de Petri, les transitions sont normalement tirées une à la fois.

- Écrivez le programme java, qui sur base d'une structure abstraite de réseau de Petri, permet de simuler des exécutions d'un réseau de Petri, une transition est tirable à la fois.
  - Quelles sont les ressources partagées entre les threads ?
- Augmentez le parralélisme de la simulation en autorisant le tirage de plusieurs transitions simultanément lorsqu'il n'y a pas de conflits.
  - Quels types de conflits peuvent-ils survenir du fait de la concurrence ?
  - Quelles sont les ressources partagées entre les threads ?
- Qu'apportent les threads à la formulation de la solution ?
- Comment devrions-nous faire sans les threads ?
- Votre solution est-elle sujette à des deadlocks que l'on pourrait éviter en adoptant une autre stratégie ?

3.12 Philosophes

Cinq philosophes se trouvent autour d'une table ronde. Chacun des philosophes a devant lui un plat de spaghetti. À gauche de chaque assiette se trouve une fourchette.
Un philosophe n'a que trois états possibles :

1. penser pendant un temps indéterminé.
2. être affamé (pendant un temps déterminé et fini sinon il y a famine).
3. manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation : quand un philosophe a faim, il va se mettre dans l'état "affamé" et attendre que les fourchettes soient libres pour manger, un philosophe a besoin de deux fourchettes : celles à sa droite et à sa gauche. Si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

- Proposez une modélisation naïve de ce problème à base de threads.
- Trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

Source : Dîner des philosophes, `http://fr.wikipedia.org/w/index.php?title=Dîner_des_philosophes&oldid=95577115` (Page consultée le octobre 14, 2013).

3.13 Le problème métaphysique des "philosophes et des boulets à la liégeoise congelés"

Lors de la fête du XV août en Outremeuse, 999 philosophes liégeois sont installés autour d'une table et ont devant eux un stock de boulets à la liégeoise dans leur assiette respective. Chacun prend un boulet et s'il est congelé, il tente de le réchauffer avec ses mains pendant un certain temps et le dépose dans l'assiette de son voisin de droite avant d'attraper des gelures, sinon il le mange.
Transcrivez ce problème en une application Java qui émule son fonctionnement.
Toute ressemblance avec un autre problème impliquant des philosophes est purement fortuite.

8. Petri est un nom propre, il s'agit de Monsieur Carl Adam Petri `https://fr.wikipedia.org/wiki/Carl_Adam_Petri`.

3.14 Le verrou Mutex

Implémentez une classe qui joue le rôle d'un verrou mutex en Java. Vous aurez certainement besoin d'utiliser des méthodes `synchronized`, ainsi que les méthodes `wait()`, `notify()` et `notifyAll()`.

3.15 Tower Bridge

On vous demande de simuler en Java l'accès concurrent par des voitures à un pont à une seule voie. Ainsi, les voitures ne peuvent avoir accès de manière concurrente au pont que si elles roulent dans le même sens (plusieurs voitures peuvent emprunter le pont en file indienne). Mais avant toute chose, essayez de bien identifier quels peuvent être les problèmes de *safety* et de *liveness* dans ce problème.

3.16 Power Bridge

Un pont supporte une charge maximale de 30 tonnes. Ce pont est traversé par des camions dont le poids est de 15 tonnes ainsi que par des voitures dont le poids est de 5 tonnes. On vous demande de gérer l'accès au pont de sorte que :

1. La charge maximale du pont soit respectée.
2. la priorité soit donnée aux camions : lorsqu'une voiture et un camion demandent l'accès au pont, le camion doit être choisi en priorité, sous réserve que la capacité maximale du pont soit respectée.

Écrire un programme qui simule les règles de partage du pont ci-dessus. Votre programme modélisera les camions et voitures sous la forme de threads.

3.17 Robots

Un certain nombre de robots se trouvent sur une grille, et se déplacent de manière aléatoire et instantanée. Modélisez ce problème au moyen d'une classe `Grille` qui joue le rôle de la ressource et d'une classe `Robot` jouant le rôle du thread. Nous faisons l'hypothèse que lorsqu'un robot décide de se rendre à une position, il n'abandonne pas sa décision. Quel type de problème peut survenir ici ? Illustrez. Comment le résoudre ?

3.18 Pipe-Line

On vous demande d'écrire un programme qui convertit des nombres en base 10 vers la base 2. Pour ce faire, on vous demande d'utiliser un *pipe-line* de threads. Votre programme doit être constitué de 8 threads :

— Le premier thread lit depuis le clavier le nombre à convertir, puis, transfère cette information vers le thread suivant.
— Six threads affichent 0 ou 1 selon que le nombre reçu par le thread soit divisible ou non par deux, et transfèrent le nombre divisé par 2 au thread suivant (sauf si le nombre divisé est égal à zéro).
— Le dernier thread du pipe-line doit afficher une erreur si le nombre à convertir dépasse la capacité du pipe-line.

Tel que le pipe-line est décrit ici, le résultat de la conversion doit normalement afficher le nombre en base 2 à l'envers.

3.19 Argument et synchronisation

Lors d'un appel de méthode synchronisée, l'évaluation des arguments de la méthode est-elle réalisée avant ou après l'obtention du verrou ? Que dit la documentation de Java sur ce point ? Écrivez un programme Java qui met en évidence la sémantique déduite du premier point.

3.20 Tri Fusion multi-threadé

Implémentez un tri par fusion avec plusieurs threads. Le tri par fusion consiste à scinder récursivement le tableau en deux parties, de trier chacune de ses parties, et ensuite de les interclasser. Le but ici sera de faire le tri des deux parties du tableau de manière parallèle.

4 Styles

Pour chaque énoncé, discutez plusieurs architectures candidates en adoptant différents styles parmi les plus adaptés. Discutez les avantages et désavantages de chaque architecture.

4.1 Bouchetoultan

Le Royaume du Bouchetoultan veut doter son infrastructure autoroutière de senseurs afin d'avoir un tableau de bord de la circulation en temps réel et agir dans les meilleures conditions en cas de problèmes. Ces senseurs permettent de détecter la vitesse approximative des voitures, de prendre des photos à la demande, de déterminer les conditions météorologiques (température, pluie, taux d'humidité, taux de luminosité) ainsi que le passage d'objets dans le sens contraire des flux autorisés (par configuration). Le faible prix de ces senseurs permet d'en placer à des distances très rapprochées. Le réseau comporte environ 900Km d'autoroute, les senseurs coûtent environ 250 euros pièce, et seraient placés tous les 250 mètres environ. Chaque senseur est programmable, dispose d'un stack TCP/IP et d'une API permettant d'exploiter toutes les mesures. Le problème du raccordement est considéré comme résolu puisque notre infrastructure est parcourue d'un backbone. Le senseur dispose également d'une API d'auto-diagnostic permettant de détecter certaines mesures devenues inopérantes, ou d'anticiper la survenue de défaillances techniques de certains composants.
L'exploitation de ces résultats devra permettre :

- Un affichage contextuel plus précis sur les panneaux d'affichage surplombant l'autoroute ?
- Une information préemptive par la voie FM sur les autoradios.
- La transmission d'alertes, selon leur nature, vers différents acteurs (police, pompiers, centres techniques, ...).
- La collecte de statistiques.
- Une maintenance dans les 24H de l'infrastructure.
- L'activation de panneaux de signalisation électroniques (Sens interdit, Stop, Verglas, ...).
-

Si le système peut fonctionner en mode dégradé, il ne peut par contre en aucun cas inférer des informations erronées et causer des réactions inappropriées.

4.2 Université Futuropolis

L'université Futuropolis (dénommée UF) comprend plusieurs facultés et services administratifs (financier, logistique, bâtiment, gestion du personnel, ...). Chaque service disposait de ses

propres informaticiens et a développé des outils spécifiques en fonction de leurs besoins, sans veiller à l'interopérabilité. En cela, l'UF a continué à fonctionner selon ses vieilles habitudes, en échangeant les informations sur format papier. En outre, certaines informations sont nécessairement répliquées et sujettes à des erreurs. Face à de nouveaux défis, l'UF a décidé de recentrer toutes les activités IT dans un seul service dédié et de poursuivre les nouveaux développement de manière plus intégrée tout en maintenant les anciens systèmes en attendant leur reconversion. Quel(s) style(s) architecturaux préconiseriez-vous pour la nouvelle architecture ?

Hypothèses :

- Les systèmes legacy ont tous été développés en interne, par des personnes compétentes, et toujours en fonction ;
- Les services ne souhaitent pas nécessairement abandonner leurs anciennes prérogatives ;
- Les délais pour mettre en place la nouvelle architecture sont flexibles ;

Question Discutez la pertinence de votre choix si les hypothèses précédentes sont invalidées une à la fois.

Remarque Toute ressemblance entre cet énoncé et une quelconque université serait purement fortuite. Cette situation est en fait rencontrée dans la plupart des grandes organisations.

4.3 Povray

PovRay (the Persistence of Vision Ray-Tracer)[9] est un programme qui crée des images en utilisant une technique de rendu appelée *ray-tracing*. Il fonctionne comme un compilateur : il lit un fichier texte contenant des informations décrivant les objets et les lumières dans une scène et génère une image de cette scène du point de vue d'une caméra (sa position étant aussi donnée dans le fichier texte). Le ray-tracing n'est en aucun cas un processus rapide, mais il produit des images de très bonne qualité avec des réflexions réalistes, des effets d'ombre et de perspective. Ces logiciels sont par exemple utilisés pour produire des films d'animation ; la production d'une quantité énorme d'images est alors requise.
Le but de l'exercice est de concevoir une architecture permettant de répartir la charge de calcul d'une tâche PovRay en exploitant le parallélisme intrinsèque à la tâche et de diminuer drastiquement le temps de calcul. L'architecture sera en outre capable de supporter la soumission massive de tâches sans faillir. La réalisation de cette architecture mise sur la disponibilité d'un grand nombre de ressources (i.e. des Workers).
Nous utiliserons cette terminologie dans la suite de l'énoncé.

GRID : Point d'entrée du système pour les clients.

Client : Application permettant de soumettre des tâches au GRID, de les monitorer et de récupérer les résultats.

Worker : C'est une ressource mise à disposition du système pour assumer une partie de la charge.

Tâche : Demande de traitement pour un rendu de PovRay.

Requête : Demande plus générique, pouvant comprendre une tâche, des méta-données, etc.

9. www.povray.org

Système : Ensemble des composants logiciels nécessaires à la réalisation des objectifs.

Les exigences du problèmes sont formulées ci-après.

1. Les clients peuvent se trouver sur n'importe quel poste de travail sur internet (ordinateur au bureau, au domicile, appareil mobile, ...).
2. Le client ne doit pas rester connecté en permanence durant le calcul.
3. La fiabilité du système doit être maximale : la fiabilité du système ne doit pas dépendre de la panne d'un ordinateur, quel qu'il soit !
4. Le système repose sur la mise à disposition de ressources (CPU, disque, bande passante) par des tiers dont on n'a aucune garantie de fiabilité. En particulier, ces ressources (par ex. des workers) peuvent être mises à disposition du système et s'en retirer sans préavis. On mise néanmoins sur un effet de masse pour garantir un certain niveau de disponibilité dans l'ensemble.
5. Les workers peuvent être hétérogènes. Certains peuvent être sous la responsabilité de l'administrateur du GRID, d'autres non. Tous les workers disposent néanmoins du logiciel PovRay installé correctement et seront dotés des agents spécifiques nécessaires au bon fonctionnement du système.
6. Certaines requêtes peuvent mentionner le souhait d'avoir une garantie sur l'intégrité des tâches et des résultat tout au long du processus de construction.
7. Certaines tâches peuvent requérir un caractère confidentiel, mentionné dans la requête.
   (a) La tâche initiale ne peut circuler en clair sur le réseau.
   (b) Des workers qui ne seraient pas contraints par un SLA ad-hoc ne devraient pas être en mesure de reconstituer le tout.

Remerciements

Je remercie Fabian Gilson et Maouaheb Belarbi pour leur aide et contribution. Je remercie également mes étudiants qui par leur curiosité, esprit critique ou questions m'ont inspiré dans l'élaboration de ce document. Je remercie plus particulièrement Kodjo ADEGNON (18-19).