Préparation à l'examen d'*Architecture Logicielle*

Wery Benoît

8 décembre 2017

Chapitre 1

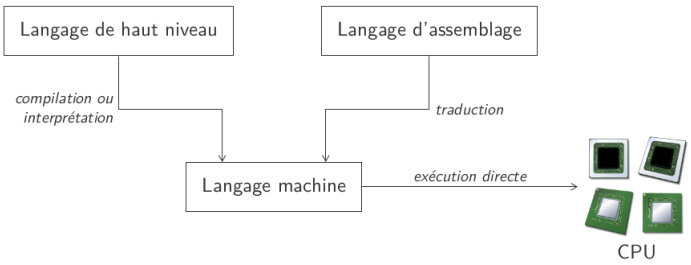
Vrai ou Faux

1. *L'assembleur est un langage de programmation de bas niveau dont les instructions dépendent du type de microprocesseur/microcontrôleur.* Vrai

   Le **langage d'assemblage** est un langage bas niveau qui est **traduit** en langage machine au moyen d'un assembleur.

   Puisque chaque famille de processeurs utilise un jeu d'instructions différent, le langage assembleur, qui est une **traduction exacte** du langage machine, est spécfifique à chaque architecture de processeur.

   Contrairement aux langages de haut niveau (qui dépendent du compilateur utilisé), un programme en assembleur sera donc TOUJOURS traduit en un même code machine (= directement exécutable par processeur).



2. *L'architecture d'un système logiciel décrit les spécifications des différentes procédures/fonctions/-méthodes présentes dans le code.* Faux

   L'architecture d'un système logiciel **traduit sa structure** : éléments softwares, relations entre eux, propriétés,...

   Il s'agit d'une **représentation abstraite** du système, une **vue globale** de ses composants, qui représentent les unités fonctionnelles, et des connecteurs qui marquent les interractions entre ceux-ci (= COMMENT).

   Les spécifications/méthodes sont décrites par la documentation du code et les fonctions à développer (càd ce que le système doit faire = QUOI) sont définies par l'*Analyse Fonctionnelle*.

3. *L'analyse fonctionnelle d'un système logiciel identifie la structure à donner au système.* Faux

   L'analyse fonctionnelle définit les fonctions à développer, c'est à dire **ce que le système doit faire** (QUOI), contrairement à *l'architecture* qui identifie la structure du système, c'est-à-dire COMMENT faire ces fonctions dans le système.

4. *L'architecture d'un système logiciel identifie la structure à donner au système.* Vrai

5. *On peut décrire l'architecture d'un système logiciel de manière graphique avec des boites et des flèches.* Vrai

   Les boîtes représentent les composants du système (qui traduisent ses unitées fonctionnelles) tandis que les flèches indiquent les interactions entre eux.

   Dans l'exemple ci-dessous, on a diminué le niveau d'abstraction de l'architecture en **identifiant la structure** interne d'un des composants.

   De plus, cette vue globale peut également servir de **pan de travail** en y spécifiant l'attribution des composants aux différentes équipes de la team.

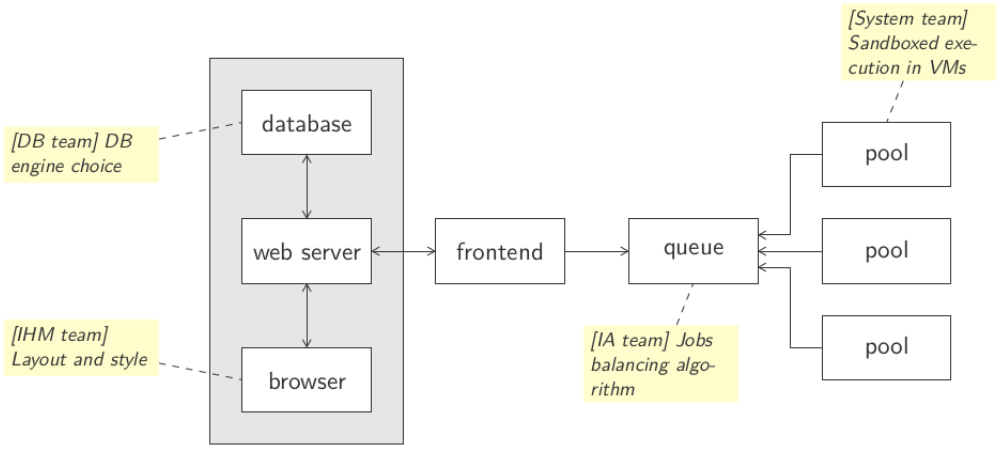


FIGURE 1.1 – Architecture simplifiée de la Plateforme Pythia [1]

6. *Parmi les parties prenantes autour d'un système logiciel, il revient au développeur d'intégrer le système dans l'entreprise.* Faux

   C'est le rôle du *business manager*

   Les parties prenantes du système logiciel sont les **catégories d'acteurs** qui interviennent par rapport à celui-ci, et ce à différents niveaux :
   - les **développeur** : écrivent le code
   - le **business manager** : intègre le système dans l'entreprise
   - l'**utilisateur final** : utilise et interagit avec le système
   - le **gestionnaire** de l'infrastructure : installe et déploie le système

7. *Parmi les parties prenantes autour d'un système logiciel, il revient au gestionnaire de l'infrastructure de déployer le système.* Vrai

8. *Le choix d'architecture d'un système logiciel peut avoir une influence sur sa qualité.* Vrai

   Une architecture donnée assure au système une série de critères de qualité. (**Lien fort** entre archi et qualité).

   Parmi les critères de qualité, on retrouve : tolérance aux pannes, compatibilité, maintenabilité, disponibilité, sécurité, fiabilité, exensibilité,...

   Les différentes parties prenantes du projet ne souhaiteront pas forcément les mêmes critères.

9. *Le style d'architecture en couches permet de diminuer la complexité et la modularité pour augmenter la réutilisabilité et la maintenance.* Vrai

   Rappel : le syle d'architecture d'un système peut se définir par rapport aux données à traiter, par rapport à l'organisation hiérarchique des composants ou encore par les invocations implicites de comp. Parmi les différents syles, on retrouve :

   (a) arch. **centrée sur les données** : traitement de grandes quantités de données, opérations de lecture/écriture, analyses,... *ex : Google Analytics*

   (b) arch. **flot de données** : processus définit par des étapes (successives ou simultanées) qui traitent des flux (paquets) de données,... *ex : lecteur multimédia*

   (c) arch. **en couches** : organisation hiérarchique, une couche délivre des services aux couches supérieures et est cliente des couches inférieures *ex : OS*

   (d) arch. **en niveaux** : composants séparés physiquement en différents niveaux *ex : protocole TCP/IP, client-serveur*

   (e) arch. par **invocation implicite** : les composants sont appelés en réaction à des évènements *ex : modèle MVC*

   Dans le cas du système en couches, chaque couche peut être utilisée **indépendamment** des autres. Il suffit de connaître les IN/OUT d'une couche pour pouvoir utiliser ses services -> chaque couche offre un niveau d'abstraction supplémentaire aux couches sup. facilitant ainsi le développement dans celles-ci (= COMPLEXITE)

   De cette façon, l'implémentation interne d'une couche peut changer (pour autant que les IN/OUT restent les mêmes) sans impacter les couches clientes (= MAINTENABILITE) et elle peut facilement être utilisée par de nouvelles couches (= REUTILISABILITE).

10. *Le style d'architecture en niveaux augmente le couplage pour diminuer la cohésion.* Faux

    Le style en niveaux a pour objectifs un découplage des composants et une augmentation de la cohésion.

    En effet, en séparant physiquement les composants, lorsqu'un crash survient sur un composant les autres n'en seront pas affectés directement MAIS la procédure générale peut ne plus fonctionner (manque fonctions du à l'absence d'un composant). Il est alors facile d'identifier l'élément fautif.

11. *Le choix d'architecture peut avoir une influence sur la sécurité du système logiciel.* Vrai

L'architecture logicielle influence fortement les **propriétés** du système dont sa sécurité. Par exemple, il est plus difficle d'assurer la sécurité d'un système s'il est distribué (modèle en niveaux) que s'il est sur une unique machine.

Comme un choix d'architecture est quasi définitif, il y a des points de **non retour** lors du développement du système, la phase d'analyse du projet est donc primordiale.

Remarque : il est *impossible* d'optimiser toutes les propriétés d'un système (performance, sécurité, disponibilité, maintenabilité, fiabilité, tolérance aux pannes, comptabilité,...) car l'amélioration d'un critère se fait bien souvent au détrimant d'un autre. Des choix doivent donc être faits (= COMPROMIS) quant aux critères les plus pertinents suivant l'application du système.

Afin d'améliorer la qualité d'un logiciel, une solution consiste à créer plus de tests unitaires pour couvrir le code au plus possible et effectuer soi-même d'avantage de "*tests à la main*" pour évaluer tous les scénarios possibles d'utilisation.

Il est à noter que la *qualité logicielle* est une notion très vague. Il faut se baser sur des critères (ex, norme ISO/CEI 9126 : capacité fonctionnelle, fiabilité, facilités d'utilisation, performance, maintenabilité) et définir la façon de les mesurer pour avoir une évaluation qui a du sens.

12. *Un design pattern est un template de code applicable automatiquement étant donné la spécification d'une procédure/fonction/méthode.* Faux

Un design pattern est un **modèle de conception** général qui répond à une problématique récurrente en développement. Il s'agit d'une desciption de solution dont l'implémentation doit être adpatée aux cas particulier.

Un pattern n'est donc pas automatiquement applicable.

Remarque : les patterns sont regroupés selon 3 catégories :

(a) **construction** : concerne l'instanciation des classes

(b) **structuraux** : concerne l'organisation des classes entre elles

(c) **comportementaux** : concerne la communication entre les objets

13. *Le design pattern du GoF Singleton permet de créer des instances d'une classe une à la fois.* Faux

Le pattern Singleton a pour objectif de garantir qu'une classe ne puisse être instanciée au plus que une seule fois.

Pour ce faire, on rend le constructeur privé (donc accessible uniquement depuis l'intérieur de la classe elle-même) et passe par une méthode pour instancier la classe. Cette méthode garantit qu'il n'existe que une seule instance au plus.

14. *Le design pattern du GoF Builder est de type construction.* Vrai

L'objectif du pattern Builder est de déléguer la construction d'un objet à une autre classe afin de séparer la construction de l'implémentation.

Il est utilisé, par exemple, dans le cas d'une classe ayant beaucoup de paramètres dans son constructeur ou plusieurs possibilités d'instanciation avec différentes règles.

Pour ce faire, le pattern utilise une classe abstraite qui définit un comportement commun à la création des objets et des classes concrètes, qui étendent celle-ci, pour définir les différents cas possibles de création. Une autre classe, quant à elle, passe par un de ces *builders* concrets pour instancier la classe abstraite.

15. *Pour appliquer le design pattern du GoF Facade, il faut impérativement rendre toutes les procédures/fonctions/méthodes des sous-systèmes à cacher privées.* Faux

Le pattern façade est un **point d'entrée** pour un sous-système, il facilite l'accès à toutes ses fonctionnalités. La façade délègue les requêtes du client aux objets appropriés. Cependant, il s'agit d'une *alternative simplifiée* pour utiliser le système, elle ne bloque pas l'accès individuel aux différentes classes (**pas d'encapsulation** !)

L'intérêt du pattern est de diminuer le couplage entre le client et les classes du sous-système. La façade fait le lien avec les différentes interfaces du sous-système de sorte que l'implémentation des classes ou du client peut changer sans impacter le fonctionnement.

Le client peut donc au choix utiliser une des façades du système pour accéder à ses fonctionnalités ou instancier directement une partie du système.

16. *Le design pattern du GoF Template permet d'implémenter un algorithme incomplet avec des « trous » à remplir (hooks).* Vrai

Le pattern Template ou Strategy permet de définir une **famille d'algorithmes** et de choisir dynamiquement (lors de l'exécution) celui à utiliser.

Pour cela, le pattern utilise une classe abstraite (la *Stratégie*) qui définit une méthode avec les parties communes d'un algorithme et des trous, représentant les parties variables, qui devront être remplis par des classes concrètes qui étendent cette *Stratégie*. Pour choisir l'algorithme a utiliser, le client possède une variable du type de la classe abstraite et il instancie une des classes conrètes au choix.

17. *La programmation impérative met l'accent sur le comment un programme fonctionne.* Vrai

Un programme, en prog. impérative, possède à tout instant un **état** qui est définit par la valeur en mémoire de ses variables (à l'image des objets en POO). Il est constitué d'une **séquence d'instructions** exécutées de façon "linéaire" et qui modifient l'état du programme.

Il s'agit de l'opposé de la programmation déclarative.

Exemples : Fortran, Algol, Pascal, Basic, C,...

18. *La programmation déclarative met l'accent sur le comment un programme fonctionne.* Faux

La programmation déclarative se concentre sur **ce que** le programme doit faire et ne se soucie pas de comment il va y parvenir, on fournit donc seulement une "description" des résultats que l'on attend.

Ce paradigme est également marqué par l'**absence d'effets de bord**,c'est-à-dire qu'une fonction ne modifie pas un état autre que celui de ses valeurs de retour.

Il s'agit de l'opposé de la programmation impérative.

Exemples : XML, SQL, Prolog, ...

19. *La programmation fonctionnelle est plus proche de l'impérative que de la déclarative.* Faux

En programmation fonctionnelle, l'exécution d'un programme est une séquence d'évaluations de **fonctions mathématiques**. Le résultat d'une telle fonction ne dépend **que de ses entrées** et retourne toujours une même valeur de sortie pour une entrée donnée.

Les variables n'ont **pas d'état** et les données ne sont pas mutables, les seules valeurs fournies sont passées en paramètres aux fonctions.

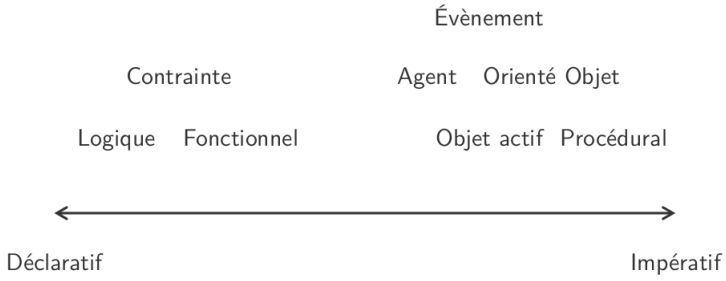


FIGURE 1.2 – Vue générale des paradigmes

Remarque : Le point d'entrée du programme est une fonction main qui produira un résultat en appelant d'autres fonctions.

20. *La programmation fonctionnelle est plus éloignée de l'impérative que de la déclarative.* Vrai

21. *Un bon système distribué doit être plus fiable que le même système en centralisé unique.* Vrai

Un système distribué étant plus difficile à mettre en place, si celui-ci n'est pas plus fiable que son équivalent en centralisé unique alors il n'a que peu d'intérêt.

22. *Dans une architecture client-serveur, on a toujours un unique serveur et un ou plusieurs clients.*

23. *Une architecture de type broker peut être vue comme orientée service.*

24. *Dans le cadre de services web, UDDI est un langage de description de services.* Faux

L'UDDI (***U**niversal **D**escription, **D**iscovery, and **I**ntegration*) est un annuaire de services web qui permet de les localiser sur le réseau. Il se base sur l'utilisation du langage XML pour définir la description de ces services, en reprenant quatre informations essentielles :

(a) businessEntity : fournisseur

(b) bussinessService : informations "non techniques"

(c) bindingTemplate : informations pour accéder au service

(d) tModel : modèle technique

25. *Dans le cadre de services web, WSDL est un langage de description de services.* Vrai

Le WSDL (***W**eb **S**ervices **D**escription **L**anguage*) est le langage qui permet de décrire la signature précise de chaque service invoquable, c'est une grammaire XML.

Il décrit une interface d'accès à un service web qui décrit comment communiquer pour l'utiliser ("QUOI, Où, COMMENT").

Remarque : XML est une grammaire soit un *métalangage* (de balisage). Un tel langage permet de décrire la syntaxe (mise en forme/structure) ET la sémantique (contenu/significaiton), il s'agit donc d'une sorte de super-langage permettant la définition formelle d'autres langages de programmation.

26. *Le standard REST définit les règles précises à appliquer pour obtenir des services web RESTful.* Vrai

Une **architecture REST** est une architecture de développement web qui définit une collection de principes à respecter pour créer des service RESTful (ce n'est donc pas d'une technologie en soi).

Une **application** est dite **RESTful** si elles respectent les 6 contraintes : *uniform interface, stateless, cacheable, client-server, layered system, code on demand.*

27. *L'architecture d'un compilateur est typiquement orientée flux de données.* Vrai

En architecture orientée **flux** de donées, celles-ci subissent un ensemble de transformations en passant successivement par plusieurs composants indépendants.

On retrouve principalement trois architectures de ce type :

(a) batch séquentiel : Exécution séquentielle de sous-systèmes indépendants qui s'échangent des *batchs* de données. Les transformations sont successives et chaque composant doit attendre que le précédent ait fini sont traîtement pour pouvoir commencer le sien.
   ✔ Faible couplage des sous-systèmes qui n'ont connaissance que du format des données E/S.

(b) :

(c) :

28. *L'architecture centrée données consiste en un data store passif et des clients actifs.*

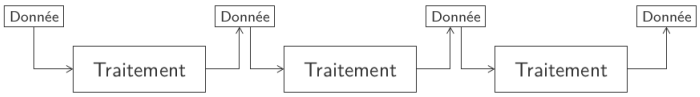


FIGURE 1.3 – Batch séquentiel - flux de données
[?]

29. *L'architecture blackboard consiste en un data store passif et des clients actifs.*

30. *Le cloud computing consiste simplement à installer des logiciels sur des serveurs plutôt que des desktop stations ou laptops afin de les rendre accessibles à tout le monde via internet.*

31. *Selon la définition du NIST, l'une des caractéristiques du cloud est d'avoir une très grande élasticité et une adaptation très rapide.*

32. *Avec l'IaaS, le client doit gérer de lui-même l'installation et la mise à jour de son application.*

33. *Google docs est un service dans le cloud de type PaaS.*

34. *La complexité software de Halstead offre une mesure de la structure d'un code.* ???

    La complexité de Halstead se base sur l'implémentation **effective** d'un programme. Elle utilise le nombre d'opérateurs et d'opérandes uniques/totaux pour en déduire une série de propriétés : *taille du vocabulaire*, *longueur du programme*, *volume d'information (V en bits)*, *difficulté (D)*, *efforts (E)*, *temps d'implémentation (T)*, ...

    Ce modèle simpliste peut facilement être mis en place et permet principalement une **comparaison relatives** entre plusieurs programmes.

    Cependant, il ne s'agit donc pas de prédictions mais bien de mesures obtenues sur base du programme développé. De plus, les valeurs sont fortement dépendantes du langage utilisé (ex : les opérateurs varient selon que l'on soit en bas niveau vs haut niveau, d'un langage à l'autre,...)

35. *La complexité cyclomatique offre une mesure de la structure d'un code.* ???

    La complexité cyclomatique mesure le nombre d'**instructions de décision**, autrement dit le nombre de chemins possibles dans une fonction.

    Ce métrique, facile à calculer et à mettre en oeuvre, permet d'évaluer la facilité de maintenance d'un code (au plus la complexité est grande, au plus le code devient difficile à comprendre) ainsi que d'identifier les zones pour lesquelles les efforts de tests seront les plus importants (là où la comp. est grande).

Cependant, ce modèle n'évalue pas la complexité des données.

Chapitre 2

Questions ouvertes

Petite question rapide à répondre en une ou deux minutes maximum, sans devoir donner de détails, juste pour s'assurer que vous avez compris le concept de la question.

1. *Définissez ce qu'est un design pattern, comment le caractériser et à quoi il sert.*

   Un design pattern est un **modèle de conception** général qui répond à une problématique récurrente en développement. Il s'agit d'une desciption de solution dont l'implémentation doit être adpatée aux cas particulier.

   Les patterns sont donc utilisés pour simplifier la vie des programmeurs, ils représentent un gain de temps (ne pas réinventer la roue) et une fiabilité puisqu'ils s'agit de modèles testés et approuvés avec le temps.

   Un pattern est définit par :
   - un **nom**
   - une **description du problème** auquel il s'applique
   - la **solution** : générique ! ! Son implémentation est à adapter au cas par cas
   - les **conséquences** de son utilisation : peut avoir des répercutions sur le reste du code, forcer des choix d'implémentation allieurs,...

   Il existe 23 patterns *classiques*, définis par le GoF, regroupés selon trois catégories :
   (a) **construction** : concerne l'instanciation des classes
   (b) **structuraux** : concerne l'organisation des classes entre elles
   (c) **comportementaux** : concerne la communication entre les objets

2. *Que sont les concurrency patterns ? Donnez quelques exemples.*

   Ce sont des patterns utilisés pour de la **programmation concurrente**. Ils apportent entre autre des modèles pour : la synchronisation, la communication, le stockage de données, les caches,...

3. *Décrire ce qu'est le test driven development (TDD).*

   Le cycle TDD définit une méthode de développement qui consiste à être **guidé par l'écriture de tests** et intégrer des phases de refactoring.

La figure ci-dessous définit les trois grandes étapes de chaque tour de cycle. La deuxième étape consiste à écrire un code permettant de passer le(s) test(s) -> on ne parle encore d'optimisation. C'est seulement à l'étape 3, lorsque le code est fonctionnel, que l'on peut penser à des améliorations pour éliminer les *bad smells*.

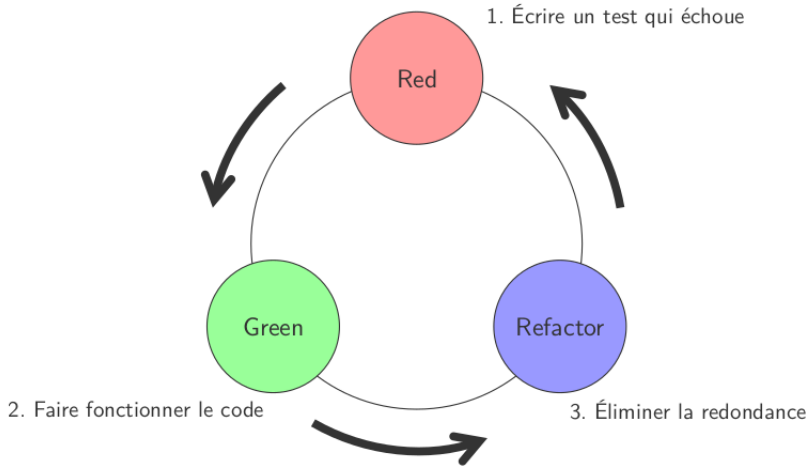


FIGURE 2.1 – Le cycle TDD selon Kent Beck

4. *Définissez ce qu'est un bad smell et donnez un exemple.*

Un *bad smell* est une **faiblesse de codage** dans un programme, il ne s'agit pas d'un bug ! ! !

Exemples de bad smells :

- **Large class (the blob)** : une classe (God class) monopolise tout le traitement et les autres stockent uniquement des données -> va à l'encontre du principe de POO "une classe = une responsabilité". La *God Class* utilise des détails d'implémentation des autres classes -> va à l'encontre de l'encapsulation.
- **code dupliqué** : du code identique ou très similaire à différents endroits dans le programme -> difficile à maintenir. Il faut isoler les parties communes dans des métodes.
- **Long method** : méthode dont le coprs possède beaucoup d'instructions -> limite la réutilisabilité et difficile à comprendre. Une méthode ne devrait avoir qu'une seule fonctionnalité.
- ...

5. *Définissez ce qu'est le refactoring et à quel moment il peut être utilisé dans le processus de développement.*

Le refactoring consiste à transformer du code tout en **préservant son comportement**, il s'agit donc d'**améliorer la qualité** d'un code déjà fonctionnel.

Cette technique s'inscrit dans la logique d'*optimisation du code*, dont un principe fondamental est le suivant : *"l'optimisation ne doit intervenir qu'une fois que le programme fonctionne et répond aux spécifications fonctionnelles."*

Dans le cas d'un cycle TDD (Test-Driven Development, voir Q3), le réfactoring du code a donc lieu après que celui-ci ait passé une batterie de tests. L'idée est de se servir de ces mêmes

tests pour s'assurer que les modifications n'aient pas altéré le bon fonctionnement du code.

EXEMPLES [...]

Remarque : l'étape de refactoring n'ajoute pas de fonctionnalité et ne corrige pas de bug !! On re-travaille un code dans le but d'améliorer sa lisibilité et sa maintenance mais son comportement externe reste identique.

6. *Définissez la notion de paradigme de programmation, ainsi que la programmation impérative et déclarative.*

Un paradigme est une représentation du monde, une manière de voir les choses, un modèle cohérent qui repose sur une base définie.

Dans le domaine de l'informatique, un paradigme de programmation est :
- un **style fondamental** qui traîte de la manière dont les solutions aux problèmes doivent être formulées dans un langage de prog.
- une **manière de programmer** un ordinateur, basée sur un ensemble de principes ou une théorie
- un **ensemble de concepts** particulier

Définitions des paradigmes *impératif* et *déclaratif* aux questions V/F 17 et 18.

Exemples... une façon de voir les choses du monde réel est de les modéliser au moyen d'objets/patrons avec des caractéristiques et la possibilité de les traîter, ce qui a amené au paradigme de la POO.

Dans d'autres cas, on préférera l'utilisation de fonctions mathématiqes établies auxquelles ont fournit des paramètres, d'où le paradigme de prog. fonctionnelle.

Ou encore, on peut utiliser un programme pour établir une séquence d'instructions qui changent l'état de ses variables, ce qui correspond à un modèle impératif.

7. *Définissez la notion de système distribué en reprenant rapidement les cinq buts.*

"L'architecture d'un environnement informatique ou d'un réseau est dite distribuée quand toutes les ressources ne se trouvent pas au même endroit ou sur la même machine." - *Wikipédia*

Dans une telle architecture, les composants sont répartis sur plusieurs plateformes, matérielles (machine, serverus,...) ou logicielles (machine virtuelle,...), et coopèrent via un **réseau de communication** (ethernet, wifi, mémoire partagée,...).

Les buts principaux de l'architecture distribuée sont les suivants :

(a) la TRANSPARENCE : pour l'utilisateur, le système doit apparaître comme un **unique système cohérent**, il faut que l'ensemble des machines engagées apparaissent comme une seule.

Cet aspect de "transparence" peut être considéré à différents niveaux :

- ACCÈS AUX RESSOURCES : la façon dont les ressources sont accédées doit être cachée
- LOCALISATION DES RESSOURCES CACHÉE : il ne faut pas dépendre du lieu ou sont les ressources
- PRÉSENCE DE DIFFÉRENTES TECHNOLOGIES : différents OS, langages, frameworks,... doivent pouvoir "cohabités" dans le système sans l'impacter
- MIGRATION DES RESSOURCES : une ressource doit pouvoir être changée d'emplacement
- RÉPLICATION DES RESSOURCES : une même ressource peut se trouver sur plusieurs composants en même temps
- CONCURRENCE ENTRE PLUSIEURS UTILISATEURS : un utilisateur utilise le système comme s'il était le seul, il n'a pas conscience de la présence des autres et les ressources doivent donc pouvoir être partagées entre eux
- PANNE ET DÉFAILLANCE DE RESSOURCES : un problème technique ne doit pas être ressenti par l'utilisateur
- PERSISTENCE : cacher le fait qu'une ressource se trouve en mémoire ou sur le disque

(b) l'OUVERTURE : le système offre des services en suivant un **standard** afin de simplifier sa construction et les changements futurs. De cette façon, une machine pourra facilement être rajoutée au système, peut importe ses caractéristiques, tant qu'elle respecte le standard. On obtient ainsi un ensemble **hétérogène** de machines

(c) la FIABILITE : un système distribué étant plus difficile à mettre en oeuvre, il doit être plus efficace/fiable que son équivalent en centralisé unique

(d) la PERFORMANCE : l'architecture distribuée doit augmenter les performance du système. Grâce à l'hétérogénétié du système, il est possible de combiner des machines spécialisées dans le traîtement de certaines tâches

(e) l'EVOLUTIVITE : le système doit pouvoir évoluer facilement, il faut pouvoir ajouter des ressources et ce de façon dynamique, pendant l'exécution. Une machine ajoutée au système doit être reconnue directement par les autres et intégrée à l'ensemble des opérations. Pour cela, elle envoie au réseau ses propres caractéristiques, suite à quoi elle se voit attribuer les tâches en fonctions de ses "compétences".

L'évolutivité doit permettre de faire face à deux principaux problèmes :

- SIZE SCALABILITY : augmenter le nombre de ressources pour éviter que le système ne surcharge
- GEOGRAPHICAL SCALABILITY : augmenter le nombre de ressources pour diminuer les délais de communication

Finalement, les avantages et inconvénients d'une telle architecture sont les suivants :

✔ économie de ressources : possibilité d'acheter des machines supplémentaires moins chères pour augmenter les capacités du système

✔ augmentation de la vitesse et de la fiabilité : si une machine tombe en panne, le reste du système est établit pour ne pas perdre de donnée et éventuellement prendre les relais et continuer de fonctionner normalement

✔ croissance incrémentale : ajout simple de machine

✘ complexité logicielle et interopérabilité (hardware ou software) : si le système est mal conçu, il sera fermé à l'extension. Ex : si trop de dépendance avec le hardware -> problèmes de compatiblités pour ajouter des machines

✘ nécessité d'avoir un réseau de communication

✘ sécurité plus difficile à garantir sur l'ensemble du système

8. *Définissez la notion de middleware. Dans quel type d'architecture les retrouve-t-on ?*

"Un middleware est un logiciel tiers qui crée un réseau d'échange d'informations entre des applications." - *Wikipédia*

Il permet à des apps hétérogènes, qui ne sont pas prévues pour se "parler", de communiquer ensemble via une même technique d'échange d'informations.

Il supporte l'architecture distribuée : les applications peuvent donc se trouver sur des machines différentes et s'échanger des informations quels que soient les ordinateurs impliqués et les caractéristiques matérielles/logicielles.

Le middleware agit donc comme un buffer entre les applications et le réseau.

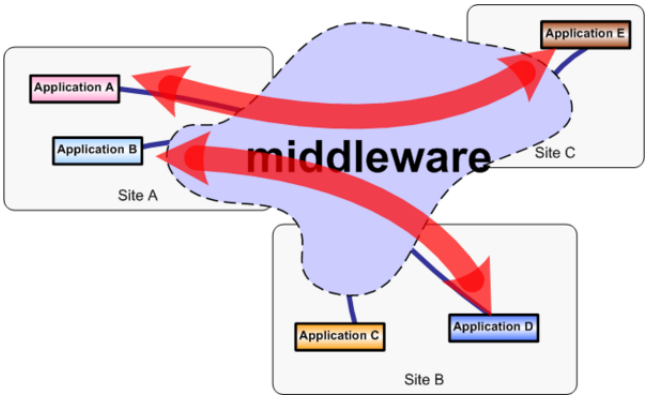


FIGURE 2.2 – Schématisation d'un middleware en réseau distribué [2]

L'infrastructure middleware peut être décomposée en plusieurs couches :

(a) couche de **transport** : assure l'envoie des requêtes et le déplacement des données

(b) couche d'**application serveurs** : sécurité, répertoire de services (liste des services proposés par la machine)

(c) couche de **gestion des messages** par brokers : manipulation et routage des messages qui structurent l'info

(d) couche **Business Process Orchestrators (BPO)**

9. *Donnez la différence entre un client léger et lourd, dans une architecture client-serveur.*

L'architecture client-serveur, de type distribuée, est constituée de deux sous-systèmes :

(a) le **client** : émet des requêtes pour bénéficier des services du serveur

(b) le **serveur** : reçoit les requêtes, les traite et envoie une réponse au client

Ce modèle d'architecture peut être implémenté de différentes façons, selon l'importance que l'on donne au client :

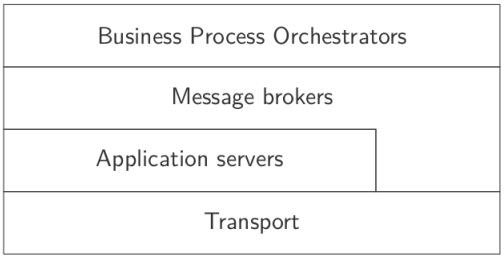
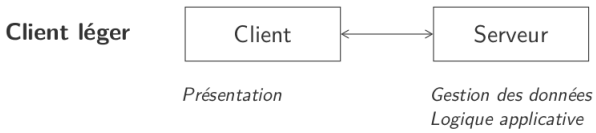
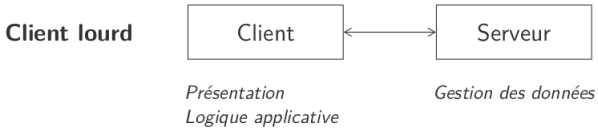


FIGURE 2.3 – Niveaux d'abstraction du middleware



- CLIENT LÉGER : le client a peu de responsabilités, il interragit avec le serveur qui gère les données ET la logique applicative du système.
- CLIENT LOURD : la logique applicative est migrée du côté client, qui a désormais le plus grand rôle, laissant au serveur uniquement la gestion des données. Il faut imposer au client certaines capacités minimums pour faire fonctionner le système.



10. *Qu'est-ce-que CORBA et quel type d'architecture supporte-t-il ?*

11. *Quelles sont les différentes étapes de l'appel d'un service web ?*

L'architecture orientée-services (SOA) est un modèle d'architecture dans lequel les composants fournissent des services à des consommateurs (autres services où utilisateurs). Un service consiste en une **fonction bien définie** (avec ses limitations fonctionnelles et un rôle spécifique), **indépendante** et **mise à disposition** selon un "contrat" qui définit son utilisation.

Un service n'a pas d'état, il ne retient pas d'information sur les données qu'il traîte. Ainsi par exemple, un service de login permet d'authentifier un user mais il n'a pas conscience des users connectés, cela relève de la responsabilité de l'application.

Les données et les services, qui représentent la logique du système, sont **distribués**. Les services communiquent entre eux via des protocoles d'échanges de messages qui **suivent des standards**, ce qui assure le découplage et les rend facilement **réutilisables** pour former de nouveaux processus.

Les caractéristiques que doit satisfaire une telle architecture sont les suivantes :

(a) avoir des FRONTIÈRES BIEN DÉFINIES : 1 service = 1 fonctionnalité -> ensemble d'opérations définies

(b) AUTONOMIE des services : les services doivent être indépendant, ce qui induit un faible couplage

(c) les services partagent leur CONTRAT : description du service -> ses fonctions et son utilisation

(d) les services sont COMPATIBLES SUIVANT DES POLITIQUES : fonctionnalités du services changent selon contexte (mise en page suivant navigateur web, gratuit/payant suivant utlisateur,...)

Un "*service web*" est un protocole qui permet à des applications, aux fonctionnalités diverses, d'échanger des données entre elles, il implémentate une architecture SOA . La figure [**?**] décrit la procédure d'appel d'un service web.

Différentes technologies sont utilisées par les services webs :

(a) HTTP(S) (***Hyper**Text **T**rasnfert **P**rotocol **S**ecure*) : communications

(b) XML (*e**X**tensible **M**arkup **L**anguage*) ou JSON (***J**ava**S**ript **O**bject **N**otation*) : construction messages

(c) WSDL (***W**eb **S**ervices **D**escription **L**anguage*) : description précise des signatures (*quoi*, *où*, *comment*) des services invoquables

(d) UDDI (***U**niversal **D**escription, **D**iscovery, and **I**ntegration*) : annuraire des services disponibles, protocole d'enregistrement des services

(e) SOAP (***S**imple **O**bject **A**ccess **P**rotocol*) : protocole d'invocation des messages

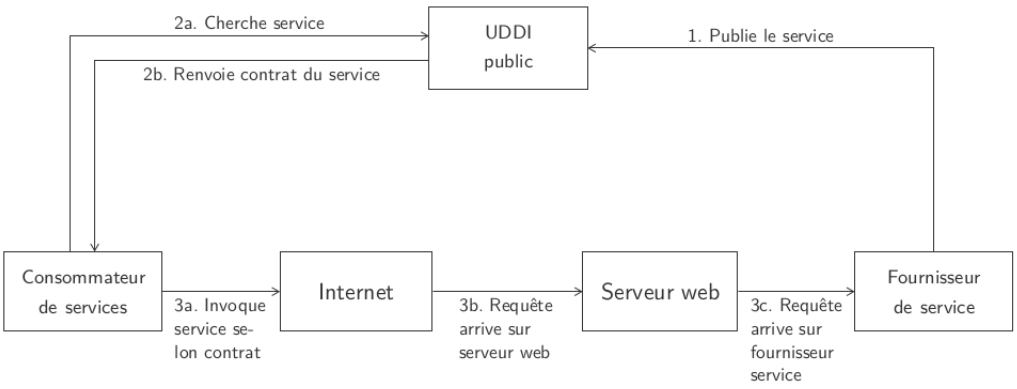


FIGURE 2.4 – Protocole d'appel d'un service web [**?**]

Description des étapes :

(a) PUBLICATION DU SERVICE : le fournisseur de service rapporte à l'annuaire sa présence. Autrement dit, il se fait connaître sur le réseau.

(b) RECHERCHE DE SERVICE et ENVOIE DU CONTRAT : Le consommateur interroge le répertoire de services pour prendre connaissance des services disponibles et connaître leurs contrats, c'est-à-dire la façon de les utiliser. Cette étape est nécessaire uniquement lors de la première utilisation d'un service (ou de temps en temps par après pour vérifier d'éventuelles mises à jour).

(c) INVOCATION DU SERVICE : le consommateur fait appel au service selon les conditions définies par le contrat. La requête passe par le réseau Internet et arrive au serveur qui contient le code source pour exécuter le service.

(d) RÉPONSE DU SERVICE : le service effectue ses opérations et retourne une réponse au consommateur.

12. *Expliquez brièvement les six contraintes d'une architecture REST ?*

REST (***RE****presentation* ***S****tate* ***T****ransfer*) est une architecture dédiée au développement de services web. Il s'agit d'une **collection de principes/contraintes/conventions** qui ne constitutent pas des standards (ce n'est pas une technologie en soi !) Autrement dit, leur utilisation n'est pas obligatoire pour l'implémentation d'un tel service.

L'architecture REST est définie par 6 principes :

(a) UNIFORM INTERFACE : interface uniforme entre le client et le serveur où les ressources sont identifiées par un **URI** (***U****niform* ***R****essource* ***I****dentifier*). Les requêtes reposent sur l'utilisation du **protocole HTTP** avec une hiérarchie des ressources et des *query string* uniquement en tant que filtres (*ex : GET http ://www.example.com/customers/1234/orders ?offset=0& limit=25*).

(b) STATE LESS : le service n'a pas d'état, il ne conserve pas l'état des ressources, il n'existe donc pas de session entre le serveur et le client. C'est la **requête** (donc responsabilité côté client/application) qui **contient toute l'information** nécessaire à son traitement.

Ce principe permet de facilement déployer le service sur différents serveurs, le client pouvant alors être indifféremment redirigé vers l'un d'entre eux. En revanche, cela implique une mise en place correcte de la sémantique de l'application, puisque le serveur perd une partie du contrôle.

(c) CACHEABLE : possibilité de mettre une réponse en cache (côté serveur ou client) pour **augmenter les performances**. Ces informations en mémoire évitent de formuler des requêtes inutilement ou de traiter des requêtes identiques, mais avec un risque que celles-ci soient obsolètes (à contrôler).

(d) CLIENT-SERVEUR : **séparation précise des responsabilités** du serveur et du client, ce qui leur permet d'évoluer indépendamment.

(e) LAYERED SYSTEM : définitions de **couches**, de telle sorte qu'une couche n'ait accès qu'à des composant d'une couche avec laquelle elle communique directement.

(f) CODE ON DEMAND : le serveur envoie le code (JavaScript,...) au client de telle sorte qu'il puisse faire varier les fonctionnalités du service en fonction du client. **Variation dynamqiue du service** en fonction du contexte d'utilisation.

Lorsque les 6 contraintes sont implémentées, on parle alors d'architecture *RESTful*.

Les avantages d'une telle architecture sont les suivants :

✔ répartition des requêtes et meilleure tolérance aux pannes, dus à l'absence d'état et donc la duplication du service

✔ soulagement

13. *Exposez brièvement les différences entre IaaS, PaaS et Saas.*

14. *Exposez brièvement les différences entre IaaS, PaaS et Saas.*

15. *Comment se calcule la complexité Fan-in Fan-out et quels sont ses avantages et inconvénients ?*

    La complexité "Fan-in Fan-out" est un *métrique* qui se base sur le flux des données locales (in=params et out=valeurs retour).

    Deux variantes existent pour calculer cette valeur de complexité :

    (a) **Henry et Kafura** : $HK = Length * (Fan_{in} * Fan_{out})^2$

    (b) **Shepperd** : $S = (Fan_{in} * Fan_{out})^2$

16. *Définissez les notions de couplage afferent et efferent. Comment construit-on l'instabilité à partir de ces métriques.*

17. *Qu'est-ce-que la distance from main sequence et que permet-elle de mesurer ?*

18. *Définissez brièvement les principes DRY et WET.*

19. *Définissez le concept d'orthogonalité. En quoi améliore-t-il la qualité d'un logiciel ?*

20. *Expliquez brièvement le principe du code traçant.*

21. *Définissez brièvement la notion de microservice.*

22. *L'architecte est un jardinier ou un planificateur de ville, expliquez.*

23. *Quelles sont les différences entre principe et pratique.*

24. *Qu'est-ce-qu'un contexte borné et quel est le lien avec microservice ?*

Chapitre 3

Réflexion

Pour les différentes questions ouvertes, ce qui est attendu est une discussion argumentée et appuyée par des concepts vus au cours. N'hésitez pas à rappeler les définitions des concepts de base que vous utiliserez dans votre argumentation. Il n'y a pas forcément de réponse unique aux questions de réflexion, et ce qui sera évalué est l'exactitude de vos propos et l'utilisation adéquate d'arguments pertinents.

1. *Il existe un lien fort entre l'architecture et la qualité d'un système logiciel. En particulier, améliorer la qualité logicielle peut se faire en choisissant une architecture pertinente par rapport au cahier des charges du système à développer. Expliquez en donnant des exemples concrets, et argumentez.*

2. *L'architecte logiciel est le garant de l'intégrité conceptuelle du système logiciel. De quoi s'agit-il ? Quelles sont les différents éléments auquel il devra faire attention tout au long de la durée de vie du logiciel ? Illustrez vos explications avec des exemples concrets, et argumentez.*

3. *Décrivez brièvement les six principaux styles d'architecture suivant en donnant, pour chacun, les avantages et inconvénients et un exemple concret utilisant ce style. Comparez ensuite ces styles et déterminez une procédure qui permettrait à un architecte de se diriger vers le style le plus adéquat étant donné un système logiciel à concevoir. Centrée sur les données, flot de données, en couches, en niveaux, invocation implicite et MVC*

4. *Le pattern d'implémentation argue qu'il faut viser l'excellence en programmation en suivant les trois valeurs importantes que sont la communication, la simplicité et la flexibilité. En vous appuyant sur des exemples de systèmes logiciel avec un choix d'architecture le plus adapté, discutez à partir des avantages de l'architecture choisie de pourquoi elle permet de tendre vers l'excellence.*

5. *Un mauvais système logiciel peut se détériorer avec le temps avec pour conséquence qu'il deviendra cher et difficile, voir impossible, à maintenir. L'une des causes majeures est que le code*

a été sous-ingénierié. Qu'est-ce-que cela signifie-t-il et quelles sont les principales raisons pouvant mener à un tel code ? Quelles bonnes pratiques, tant au niveau du code qu'au niveau de l'architecture, peuvent aider à éviter un code sous-ingénieré ? Argumentez.

6. *Lorsqu'il s'agit de choisir un ou des langage(s) de programmation concret pour développer un système logiciel, quelles sont les questions à se poser ? Comment peut-on organiser et structurer la réflexion qui va guider vers le choix de langage ? Expliquez et argumentez.*

7. *Dans les architectures orientée-interaction le système logiciel est découpé en trois partitions principales : données, contrôle et vue. Comparez les trois grandes familles de modèles MV* (MVC, MVVM et MVP) en identifiant comment les trois partitions sont organisées et identifiez les avantages et inconvénients des différentes architectures existantes des trois familles.*

8. *Les architectures de type broker et orientée-service possèdent une série de points communs, notamment qu'elles permettent toutes deux de partager des services. Mais en quoi diffèrent-elles ? Quels sont les avantages et inconvénients de ces deux types d'architecture ? Pour quel type d'applications l'une ou l'autre sera-t-elle plus adaptée ? Argumentez.*

9. *Les architectures prédominantes ont évolué avec les changements business partant de solutions monolithiques vers des solutions actuellement orientées services. Expliquez comment cette évolution s'est passée en reprenant les points forts et faibles de chacune des architectures de la ligne du temps suivante et argumentez.*

10. *Il y a trois principales architectures orientées données que sont le batch séquentiel, les pipes et filtres et le contrôle de processus. Quels sont les points communs et différences entre ces trois architectures, les avantages et inconvénients. Illustrez votre réponse à partir d'exemples concrets. Quelles sont les questions que l'on pourrait se poser afin d'orienter son choix vers l'une des trois architectures sachant qu'on a à réaliser un système logiciel qui doit traiter des données ? Argumentez.*

11. *Afin d'évaluer la complexité d'un système logiciel, on peut procéder à des mesures sur le code de ce dernier. En particulier, on peut utiliser les métriques de Halstead, McCabe et Henry ou Kafura/Shepperd pour mesurer différents types de complexité. Quels sont les aspects de complexité mesurés par ces métriques ? Comment sont elles calculées ? Discutez de la pertinence des mesures ainsi réalisées, par rapport aux variables prises en compte et de l'utilité que l'on peut faire de ces métriques.*

12. *En quoi l'architecture en microservices permet-elle de suivre le principe de responsabilité unique (SRP). Expliquez et argumentez en mentionnant les bénéfices d'une telle architecture. En parti-*

culier, illustrez votre réponse en utilisant la notion d'architecture serverless et à l'aide d'exemples concrets.

Bibliographie

[1] Pythia-project.org. Pythia project, an online plateform to learn programming.

[2] Smile.fr. Concepts des moms et jms : qu'est-ce qu'un middleware ?