Systèmes d'Information Avancés (et répartis)

Université Lyon 1
M1 Miage soir

L. Médini, avril 2008

Plan des cours

- Langages de structuration de l'information
- Protocole HTTP et programmation serveur
- Architectures réparties
- Paradigmes de programmation avancés
  - Retour sur les patrons de conception
  - Conteneurs d'objets
  - Métaprogrammation par annotations
  - Programmation orientée aspects
- Objets distribués (EJB)
- Web services (SOA, WSDL, SOAP, UDDI)

Plan du cours

- Langages de structuration de l'information
- Protocole HTTP et programmation serveur
- Outils de programmation avancés
  - Retour sur les patrons de conception
  - Inversion de contrôle et conteneurs d'objets
  - Les patterns MVC et les containers légers
  - Rappels sur les annotations en Java
  - Programmation orientée aspects
- Systèmes d'information distribués
  - Appel de méthodes distantes (CORBA, RMI)
  - Framework Java EE 5
- Objets transactionnels distribués

Retour sur les design patterns

- Composantes d'un patron
  - Nom : évocation de la solution
  - Problème à solutionner
  - Contexte d'application du patron et limites de la solution
  - Forces/contraintes de la solution par rapport au contexte
  - Solution mise en oeuvre (avec variantes éventuelles)

Patterns à utiliser

- DAO
  - Traitements d'accès aux données regroupés dans des objets spécialisés
- Observateur
  - Notification des changements d'états d'un objet observé à des objets observateurs (permet un faible couplage)
- MVC
  - Permet de séparer les différentes couches de l'application
  - Plusieurs versions en fonction des priorités de l'application
- Facade, singleton...

L'inversion de contrôle (IoC)

- Problème
  - Réduire les dépendances (couplage) entre des objets dont l'implémentation peut varier
  - Diminuer la complexité de gestion du cycle de vie de ces objets (patterns singleton et factory)
- Principe
  - S'appuie sur le pattern d'indirection
  - Le contrôle du flot d'exécution d'une application n'est plus géré par l'application elle-même mais par une structure externe (conteneur)

L'inversion de contrôle (IoC)

- Autres noms
  - Recherche / Injection de dépendances
  - Injection de code
- Exemples
  - Interface à modèle événementiel (Swing)
  - Serveur Web
  - Frameworks et conteneurs d'objets (servlets, EJB)

L'inversion de contrôle (IoC)

- Fonctionnement
  - Les constructeurs, destructeurs et certaines méthodes des objets sont appelés par un **conteneur**
  - Le conteneur gère les services extérieurs et isole les objets de l'application
  - Le conteneur est lui-même paramétré plutôt que programmé
  - Le conteneur peut servir de pattern façade pour isoler les couches de l'application

Différentes variantes

- Recherche de dépendances (EJB)
  - Un objet accède à un autre en interrogeant le conteneur
  - Exemple : appel à un annuaire JNDI pour trouver un EJB

```
ctx = new InitialContext(proprietes);
ref = ctx.lookup("MonEJB");
home = (MonEJBHome) javax.rmi.PortableRemoteObject.narrow(ref,
MonEJBHome.class);
monEJB = home.create();
```

  - Avantage : mécanisme d'indirection *via* un annuaire
  - Inconvénients
    - Nécessite un appel explicite au conteneur
    - Méthode d'appel générique qui nécessite un transtypage

Différentes variantes

- Injection de dépendances (Conteneurs légers, Spring)
  - Rendre l'inversion de contrôle transparente pour les objets
  - Initialisation directe des objets à partir d'un référentiel de dépendances
  - Les liens des objets entre eux et avec le conteneur devient implicite
  - 3 méthodes d'injection
    - Par constructeur
    - Par accesseurs
    - Par interface

Différentes variantes

- Injection de dépendances (Conteneurs légers, Spring)
  - Injection par constructeur
    - Passage d'une référence aux objets connus dans le constructeur
    - Exemple

```
public ObjetA {
  ObjetB objb;
  public ObjetA (ObjetB o) { (...) objb = o; (...) } (...) }
```

    - Avantage : conforme aux bonnes pratiques de la POO
    - Inconvénients
      - Paramètres du constructeur non nommés mais ordonnés
      - Devient fouillis quand il y a de nombreux paramètres
      - Pas d'héritage de cette configuration entre les objets

Différentes variantes

- Injection de dépendances (Conteneurs légers, Spring)
  - Injection par modificateurs
    - Utilise les modificateurs (setters) des attributs des objets
    - Exemple

```
public ObjetA {
  ObjetB objb;
  public setObjb (ObjetB o) { objb = o; } (...) }
```

    - Avantages
      - Apparition explicite des noms des dépendances
      - Les modificateurs peuvent effectuer des opérations complexes
    - Inconvénient : non conforme à la POO (l'initialisation « sort » du constructeur)

Différentes variantes

- Injection de dépendances (Conteneurs légers, Spring)
  - Injection par interface
    - Chaque objet implément autant d'interfaces que de dépendances
    - Chaque interface définit une méthode publique d'injection
    - Exemple

```
public class ObjetA implements InjectObjetB, ... {
  ObjetB objb;
  public void injectObjetB(ObjetB o) { objb = o; } (...) }
```

    - Mêmes avantages et inconvénients que les setters
    - Inconvénient supplémentaire
      - forme du code imposé assez lourde
      - lisibilité délicate

Utilisation

- Gestion du cycle de vie des objets
  - Génération d'événements
    - Appel de méthodes spécifiques des objets par le conteneur pour contrôler leurs changements d'états
    - Méthodes formalisées par des interfaces ou par le paramétrage du conteneur
    - Exemples : doGet(), doPost(), EJBCreate(), EJBActivate()...

Utilisation

- Gestion du cycle de vie des objets
  - Gestion des singletons
    - Nombre d'instances créées géré par le conteneur
    - Déclaration d'une classe comme singleton dans les paramètres de configuration du conteneur
    - Permet de s'abstraire d'un type d'implémentation particulier pour les singletons (exemple : classe abstraite, constructeur privé)
    - Mécanisme de création de singletons générique
    - Transtypage des classes ainsi générées nécessaire

Fonctionnement des conteneurs

- Instanciés par le framework avant la / les objets de la /des application(s)
- Peuvent permettre l'accès à un contexte d'application
- 2 techniques (non incompatibles)
  - Inversion de contrôle par configuration (statique)
    - Utilise des fichiers de configuration (XML)
  - Inversion de contrôle dynamique
    - Le conteneur est un objet d'une application (framework)
    - Il possède des méthodes appelées lors du déroulement de l'application

Fonctionnement des conteneurs

- Conteneurs légers (Spring, Pico, NanoContainer)
  - Spécifiques à une application
  - Ne contiennent que les services nécessaires
- Conteneurs lourds (Catalina, conteneurs d'EJB)
  - Sont eux-mêmes des patterns singletons
  - Peuvent gérer les objets de plusieurs applications
  - Possèdent un large éventail de fonctionnalités

Extension de la notion d'IoC

- Pourquoi / comment réaliser de l'injection de dépendances sur des valeurs et non sur des objets ?

Métaprogrammation par annotations

- Position du problème
  - Générer automatiquement certains éléments récurrents des programmes
    - Documentation
    - Fichiers de configuration
    - Métadonnées utilisées pour la compilation
    - Traitements spécifiques
  - Pour cela, il faut un outil pouvant intervenir
    - Sur les fichiers sources
    - Sur les fichiers compilés (.class)
    - Lors de l'exécution (réflection)

Métaprogrammation par annotations

- Principe
  - Programmation déclarative
    - le concepteur décrit ce qu'il veut obtenir
    - Un outil interprète ces annotations et le réalise
  - Avantages de la génération automatique
    - Gain de temps
    - Pas d'erreur de programmation
    - Pas de maintenance de « side files » qui doivent être synchonisés avec le code source (l'information est directement stockée dans les fichiers sources)

Métaprogrammation par annotations

- Fonctionnement
  - Des annotations dans le code
    - Programmation déclarative (indépendante de l'EDI et du code Java)
    - Exemple : les tags Javadocs
  - Un outil capable de réaliser des tâches spécifiques à la lecture de ces tags
    - Générer de la documentation (Javadoc, XDoclet)
    - Gérer les fonctions transverses (persistance des données, transactions, sécurité : AOP)
    - Gérer le cycle de vie d'objets complexes (EJBGen)
    - Permettre l'introspection durant l'exécution du code

Métaprogrammation par annotations

- Définition
  - Annotation : mot-clé « Interface » préfixé par '@'
    ```
    public @interface MonAnnotation {
    ```
    - Remarque : toute annotation hérite implicitement de java.lang.annotation.Annotation
  - Attributs : méthode avec type de retour et nom
    ```
    /** Message décrivant la tâche à effectuer.*/
     String att1();
    }
    ```
  - Remarque : la portée des attributs d'une annotation est implicite et toujours *public*

Métaprogrammation par annotations

- Utilisation dans le code
  - Nom de l'annotation préfixé par '@'
  - Devant l'élément concerné
    - Types d'éléments affectés : package, class interface, enum, annotation, constructeur, méthode, paramètre, champ d'une classe, variable locale
    - Plusieurs annotations différentes sont autorisées sur un même élément (mais jamais deux fois la même)
  - Exemple avec une annotation simple (« marqueur »)
    ```
    @MonAnnotation
    public class MaClasse {
        /* ... */
    }
    ```

Métaprogrammation par annotations

- Utilisation dans le code
  - Exemple avec des attributs
    ```
    @MonAnnotation (att1 = "mavaleur")
    public void MaMethode() {
        /* ... */
    }
    ```
  - Remarque : la méthode standard `String value();` permet d'omettre le nom de l'attribut à l'appel de l'annotation

Métaprogrammation par annotations

- Les annotations standard
  - @Deprecated
  - @Override
  - @SuppressWarnings (String_ou_tab warnings)
- Permettent d'interagir avec le compilateur
- Version : API Java 2 SE 5.0

Métaprogrammation par annotations

- Les méta-annotations standard
  - Qualifient les annotations non standard
  - @Documented
  - @Inherit
  - @Retention (duree_de_vie)
    - @Retention (RetentionPolicy.SOURCE)
    - @Retention (RetentionPolicy.CLASS)
    - @Retention (RetentionPolicy.RUNTIME)

Métaprogrammation par annotations

- Les méta-annotations standard
  - @Target (type_d_element_ou_tab)
    - @Target (ElementType.ANNOTATION_TYPE)
    - @Target (ElementType.CONSTRUCTOR)
    - @Target (ElementType.FIELD)
    - @Target (ElementType.LOCAL_VARIABLE)
    - @Target (ElementType.METHOD)
    - @Target (ElementType.PACKAGE)
    - @Target (ElementType.PARAMETER)
    - @Target (ElementType.TYPE)

Métaprogrammation par annotations

- Utilisation des annotations non standard
  - L'outil Annotation Processing Tool (APT)
    - Génération de messages (notes, warnings, errors)
    - Génération de fichiers (textes, binaires, sources et classes Java)
    - Syntaxe proche de celle de javac (ligne de commande + options)

Métaprogrammation par annotations

- Utilisation des annotations non standard
  - L'outil Annotation Processing Tool (APT)
    1. Détermine les annotations présentes dans le code source
    2. Recherche les AnnotationProcessorFactories que vous avez écrites
       1. Demande aux factories les annotations qu'elles traitent
       2. Demande aux factories qui traitent des annotations présentes dans le code de fournir un AnnotationProcessor
    3. Exécute les AnnotationProcessor
    4. Si ces processeurs ont généré de nouveaux fichiers sources, APT reboucle jusqu'à ce qu'il n'y ait plus de nouveaux fichiers générés

Métaprogrammation par annotations

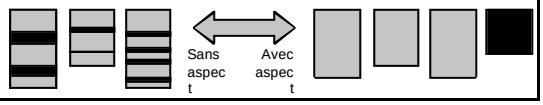
- Utilisation des annotations non standard
  - L'outil Annotation Processing Tool (APT)
    - APT relie chacune des annotations figurant dans le code à l'*AnnotationProcessor* la concernant
    - Chaque *AnnotationProcessor* comporte une méthode *process()* qui lui indique quoi faire de l'élément annoté et qui est exécutée par APT
    - Les *AnnotationProcessor* sont générés par des *AnnotationProcessorFactory* et reliés aux annotations par la méthode getProcessorFor()
    - Documentation : http://java.sun.com/javase/6/docs/technotes/guides/ap/

Métaprogrammation par annotations

- Utilisation des annotations non standard
  - Introspection
    - Dans le code de l'application
    - Permet d'accéder aux annotations dont la rétention est RUNTIME
    - Depuis Java SE 5
    - Interface *java.lang.AnnotatedElement* (implémentée par *AccessibleObject*, *Class*, *Constructor*, *Field*, *Method* et *Package*)
    - Méthodes *getAnnotation()*, *getAnnotations()*, *getDeclaredAnnotations()*, *isAnnotationPresent()*

Programmation orientée aspects (AOP)

- Position du problème
  - Gérer les fonctionnalités transverses d'une application (accès aux données, transactions sécurité)
    - En regroupant (*vs.* dispersant) le code lié à ces fonctionnalités
    - En les séparant de la logique métier
    - Avantages : productivité, maintenance, réutilisabilité



Programmation orientée aspects (AOP)

- Limites du paradigme objet
  - Données encapsulées dans les classes
    - Traitements similaires sur des données différentes
  - Différentes considérations (aspects) de l'application représentés au même niveau d'abstraction
  - « Pollution » du modèle métier par les fonctionnalités transverses
  - Même avec des patrons appropriés (façade, observateur), il reste beaucoup de code dans les classes

Programmation orientée aspects (AOP)

- Le paradigme aspect
  - Pas contradictoire, mais complémentaire au paradigme objet
  - En POA, une application comporte des classes et des aspects
    - Une classe est un élément du domaine à modéliser
    - Un aspect est une fonctionnalité à mettre en oeuvre dans l'application
  - Chaque aspect permet d'obtenir une « vision » différente de l'application
    - Métier, données, sécurité…

Programmation orientée aspects (AOP)

- Concepts de base / glossaire
  - Tangled code
    - Code embrouillé, code spaghetti.
  - Crosscutting concerns
    - Aspects de la programmation qui concernent plusieurs classes, et qui donc transcendent le modèle objet (synchronisation, logging, persistance…)
    - Mélange, au sein d'un même programme, de sous-programmes distincts couvrant des aspects techniques séparés (Wikipédia)

Sources
http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_aspec
http://www.dotnetguru.org/articles/dossiers/aop/guid/AOP15.ht

Programmation orientée aspects (AOP)

- Concepts de base / glossaire
  - Weaver (tisseur d'aspects)
    - Infrastructure mise en place pour greffer le code des aspects dans le code des classes
    - Selon les tisseurs cette greffe peut avoir lieu
      - directement sur le code source donc avant la compilation
      - durant la compilation
      - après la compilation sur le code compilé mais avant l'exécution
      - pendant l'exécution

Sources
http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_aspec
http://www.dotnetguru.org/articles/dossiers/aop/guid/AOP15.ht

Programmation orientée aspects (AOP)

- Concepts de base / glossaire
  - Joinpoint (point de jonction)
    - Endroit du code où il est autorisé d'ajouter un aspect (avant, autour de, à la place ou après l'appel d'une fonction)
    - Dans 80% des cas, liés aux méthodes
    - Parfois liés aux classes, interfaces, attributs, exceptions…
  - Pointcut (point de coupe)
    - Endroit du code où est effectivement inséré le greffon

Sources
http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_aspec
http://www.dotnetguru.org/articles/dossiers/aop/quid/AOP15.ht

Programmation orientée aspects (AOP)

- Concepts de base / glossaire
  - Crosscut (coupe)
    - Ensemble ou sous-ensemble des points de jonction liés à un aspect dans une application
    - Permet de définir la structure transversale d'un aspect
  - Advice (greffon)
    - Fragment de code qui sera activé à un certain *point de coupe* du système
    - 4 types : *before*, *after returning*, *after throwing*, *around*

Sources
http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_aspec
http://www.dotnetguru.org/articles/dossiers/aop/quid/AOP15.ht

Programmation orientée aspects (AOP)

- Outils et frameworks OA
  - Disponibles dans de nombreux langages
    - Java, C++, PHP, Python, CommonLisp, Ruby…
  - Outils Java (tisseurs)
    - AspectJ
      - De loin le plus connu
      - Tisseur d'aspect à la complilation
      - Génère du bytecode
      - Utilisation de fichiers XML ou d'annotations Java
      - Plugin Eclipse
  - Frameworks Java : JBoss, Spring

Programmation orientée aspects (AOP)

- POA et méthodes de conception
  - Au départ, surtout un paradigme de programmation
  - Bien isoler les responsabilité des objets
  - Mise en relation et amélioration des patterns du GoF
    - Importances des rôles dans les patterns
    - Exemple principal : pattern Observateur
      - Pattern disséminé dans le code
      - Notification = crosscut
  - Implémentation avec AspectJ

Source :
http://hannemann.pbwiki.com/f/OOPSLA2002.pdf

Conclusion

- Différents paradigmes de conception
  - Objet : bonnes pratiques
  - Aspects : encore en évolution
  - Distribué ?
- Outils de programmation
  - Intégrés aux frameworks
  - Bibliothèques de composants
- Outils de déploiement…

Références utilisées pour ce cours

- Design patterns
  - http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html
  - http://hillside.net/patterns/onlinepatterncatalog.htm
  - http://www.martinfowler.com/
  - http://liris.cnrs.fr/yannick.prie/ens/07-08/SIMA/CM-patterns.pdf
- Containers
  - http://www.dotnetguru.org/articles/dossiers/ioc/ioc.htm
  - http://www.picocontainer.org/
  - http://www.nanocontainer.org/

Références utilisées pour ce cours

- Annotations Java
  - http://java.sun.com/javase/6/docs/technotes/guides/ap
    GettingStarted.html
  - http://adiguba.developpez.com/tutoriels/java/tiger/anno
    ations/
- POA
  - http://hannemann.pbwiki.com/Design+Patterns
  - http://fr.wikipedia.org/wiki/Programmation_orient%C3%
    A9e_aspect
  - http://www.dotnetguru.org/articles/dossiers/aop/quid/A
    P15.htm#_Toc47186529
  - http://www.eclipse.org/aspectj/index.php