Introduction

Comme beaucoup de personnes, je pense qu'aujourd'hui, la meilleure plate-forme de développement pour les entreprises est J2EE. Elle combine les avantages du langage Java avec les leçons acquises dans le développement depuis ces dix dernières années. Elle bénéficie aussi du dynamisme des communautés *Open Source* ainsi que du JCP de *Sun* (*Java Community Process*, processus utilisé par *Sun* et de nombreux partenaires pour gérer les évolutions de java et de ces API).

Bien que prédit à un bel avenir, les promesses de cette plate-forme ne sont pas toujours honorées. Les systèmes délivrés sont souvent trop lents et compliqués, et le temps de développement est, quant à lui, fréquemment disproportionné par rapport à la complexité des demandes utilisateurs.

Pourquoi ? Parce que J2EE est souvent mal utilisé. Les publications faites sur cette plate-forme se concentrent plus sur les spécifications de J2EE que sur son utilisation réelle. Les problèmes réels des architectes ou des développeurs sont souvent ignorés.

Pour palier à ce manque, *Sun* a développé plusieurs *Blueprints*. Ce sont des documents (exemple de code, conseils, *design patterns*, FAQ, ...) qui ont pour but de faciliter le développement d'applications sur différentes plates-formes. L'un d'eux, appelé *Java PetStore*, modélise un site web marchand où les clients peuvent acheter des animaux domestiques. Ce site est devenu si populaire que les divers fournisseurs J2EE l'utilisent pour démontrer que leur produit répond bien aux spécifications.

Malheureusement, bien que reconnu par les fournisseurs, développeurs et architectes J2EE, le *Java PetStore* est une application complexe, puisque très riche : elle couvre la totalité des spécifications J2EE et utilise plusieurs *design patterns*. De plus, créée en mai 2001, elle a dû évoluer pour suivre les changements des spécifications, la rendant de plus en plus difficile à suivre par la communauté Java.

Cet ouvrage se propose de vous aider à redévelopper la quasi-totalité de l'application *Java PetStore* de manière pédagogique, en se basant sur une approche « problème » plutôt que technologique. En effet, les technologies ne servent à rien si elles n'ont pas une utilité réelle. Le choix d'une technologie est conditionné par un problème fonctionnel (« mon système a besoin d'échanger des données avec un partenaire qui utilise XML ») ou des contraintes de sécurité (« mon application doit être sécurisée par un mot de passe et échanger des données cryptées »), de montée en charge (« mon site web est visité par plus de mille personnes par heure »), de disponibilité (« mon site doit être accessible 24/7 »), de distribution (« le système de mon partenaire financier se trouve en Asie »).

Cet ouvrage vous guidera au travers de la vie mouvementée de la société YAPS (*Yet Another PetStore*) qui vend des animaux domestiques par catalogue et qui décide d'informatiser petit à petit son système, au gré de ses besoins. Partant de demandes simples comme « j'ai besoin d'informatiser ma gestion de clients » avec comme contrainte « je n'ai ni réseau, ni base de données et ne possède que des vieux PC », vous arriverez à des demandes utilisateurs beaucoup plus ambitieuses comme « ma société ayant doublé son chiffre d'affaires depuis la création du système d'achat en ligne, je dois faire face à une forte demande de mes clients qui m'achètent des produits du monde entier ». Ce système se nomme *YAPS PetStore*.

Cette approche pédagogique s'est un peu inspirée de XP. L'*eXtreme Programming* est un ensemble de pratiques qui couvrent une grande partie des activités de la réalisation d'un logiciel : de la planification du projet au développement à proprement dit, en passant par les relations avec le client

ou l'organisation de l'équipe. Les points suivants ont été pris en compte dans la méthode d'enseignement :

- Les développements sont itératifs. Modéliser une application n'est pas une activité linéaire, il s'agit d'une tâche très complexe, qui nécessite une approche itérative. Dans une itération, on se concentre uniquement sur le développement des fonctionnalités demandées par l'utilisateur. Une fois ces fonctionnalités livrées, on repart sur une nouvelle itération qui répondra à de nouveaux besoins ou viendra corriger un bug.

- Être proche des besoins utilisateurs. XP préconise d'avoir le client, ou un représentant, à ses côtés. Ce n'est, bien entendu, pas notre cas, mais les besoins utilisateurs ont cependant une place cruciale dans le cycle de développement tout au long de ce livre.

- Une fois l'application livrée au client (ici la société YAPS), elle peut être utilisable de suite. Avec des cycles d'itération court, l'application pourra être livrée rapidement et permettre au client de l'utiliser au quotidien. Ceci évite les « effets tunnel », c'est-à-dire lorsque les analystes et les développeurs passent des mois sans aucun contact avec les utilisateurs et délivrent alors une application qui ne répond plus à leurs besoins.

- Simplicité. XP prône le développement de la chose la plus simple qui puisse fonctionner (*the simplest thing that could possibly work*). Cela permet de rester proche des demandes utilisateurs sans perdre de temps à développer des fonctionnalités qui ne seront jamais utilisées (*You Ain't Gonna Need It*).

- Pour qu'un projet XP puisse itérer dans de bonnes conditions, il lui faut des retours utilisateurs (*feedback*). Cela veut dire que durant le développement de l'application, les utilisateurs font des remarques afin d'améliorer l'application et être sûr qu'elle répondra bien à leurs besoins. Une fois de plus, ce n'est pas possible dans notre cas, alors nous nous appuierons sur des plates-formes de test (*xUnit*) qui viendront confirmer les besoins utilisateurs en faisant office de test de recette.

- Remanier (*refactoring*) le code existant pour qu'il s'intègre facilement avec les nouveautés fonctionnelles et technologiques.

Java PetStore

Java PetStore est une application J2EE que *Sun* a créé en mai 2001 pour son programme de *Blueprints*. C'est un site web marchand où l'on peut choisir des animaux domestiques, les rajouter dans un panier, puis payer électroniquement. Ce *Blueprint* a permis de documenter les meilleures pratiques (code java, *design pattern*, architecture) pour développer une application J2EE.

Le *Java PetStore* est aussitôt devenu un standard de fait puisque les constructeurs de serveur d'application l'ont utilisé pour démontrer la compatibilité de leur produit avec les spécifications J2EE. En fait, *Oracle* fut le premier à l'utiliser pour ces tests de montée en charge. Bien que *Java PetStore* ait été développé à des fins éducatives, *Oracle* déclara que cette application fonctionnait deux fois plus rapidement sur son serveur d'application que sur ceux de BEA ou IBM. La communauté s'enflamma et tous les vendeurs commencèrent à utiliser le *Java PetStore* pour démontrer leurs meilleures performances.

Figure 1 - Page d'accueil de Java PetStore

Cette anecdote contribua à augmenter la popularité de ce *Blueprint* qui rentra très vite dans le langage commun. Tout le monde commença à l'utiliser pour illustrer une nouvelle technologie, une nouvelle idée ou implémentation. Aujourd'hui, on dénombre ainsi plusieurs dérivés du *Java PetStore* :

PetShop	Version de *Microsoft* utilisant le framework .NET.
xPetStore	En utilisant les tags *xDocLet*, cette équipe *Open Source* nous offre deux versions de *PetStore* : une avec EJB et l'autre avec *Servlet* uniquement.
Flash Petsore	Version de *Macromedia* utilisant la technologie Flash sur les pages web.
Presto	Implémentation de *Prevayler* qui utilise leur framework se basant uniquement sur des objets java (POJO).
JPetStore	Implémentation d'*iBatis* utilisant leur couche de persistance.
Spring Petstore	Utilisation du framework *Spring*.
openMDX Petstore	Plate-forme *Open Source* MDA

OXF Petstore	Version d'*Orbeon* utilisant *Open XML*

Toutes ces applications ne peuvent nous faire oublier les véritables sites *PetStore* qui vendent vraiment des articles pour vos animaux domestiques : http://www.petstore.uk.com/, http://www.maltepoopetstore.com/

Cet ouvrage se propose de vous faire développer de manière incrémentale le *Java PetStore*. Il reste à signaler qu'à la fin du livre, vous n'aurez pas la totalité de l'application. En effet, la version du *Blueprint* possède beaucoup de composants, de classes ou de technologies que nous n'utiliserons pas pour simplifier le développement.

Ci-dessous la liste des principales différences entre la version finale de cet ouvrage et la version originale de Sun :

- Le site final de *YAPS PetStore* ne possède pas de site pour les fournisseurs.
- La console d'administration n'offre pas des services de statistique des ventes.
- Simplification de l'arborescence des répertoires de développement.
- Simplification des fichiers Ant.
- Utilisation de serveur d'application et base de données *Open Source*.
- *YAPS PetStore* n'utilise pas le framework WAF mais juste des *servlets*.
- Utilisation d'*includes* JSP et non les décorateurs.
- Aucun *style sheet* (fichier css) n'est utilisé pour les pages web.
- Le site n'est affiché qu'en une seule langue, l'anglais.

Sous le succès des *Web Services*, *Sun* développa un nouveau *Blueprint* couvrant cette technologie au lieu de l'intégrer dans le *Java PetStore*. L'*Adventure Builder* est né en 2004. Ce site vous permet de personnaliser un séjour pour vos vacances, en utilisant principalement des *Web Services*. L'application *YAPS PetStore* utilisera certains de ces *Web Services*.

UML

UML est utilisé tout au long de ce livre pour vous aider dans vos développements. Par exemple, grâce aux cas d'utilisation, le périmètre du système à modéliser est délimité par les besoins des utilisateurs (les utilisateurs définissent ce que doit être le système).

UML (*Unified Modeling Language*), traduisez « langage de modélisation unifié », est né de la fusion des trois méthodes qui ont le plus influencé la modélisation objet au milieu des années 90 : OMT, *Booch* et OOSE. Issu d'un travail d'experts reconnus (James Rumbaugh, Grady Booch et Ivar Jacobson), UML est le résultat d'un large consensus qui est vite devenu un standard incontournable. Fin 1997, ce langage est devenu une norme OMG.

UML est un langage de modélisation objet et non une démarche d'analyse. Il représente des concepts abstraits de manière graphique. Un modèle est une abstraction de la réalité. L'abstraction est un des piliers de l'approche objet. Il s'agit d'un processus qui consiste à identifier les caractéristiques intéressantes d'une entité en vue d'une utilisation précise.

UML est donc un langage universel et visuel qui permet d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation. Et comme UML n'impose pas de méthode de travail particulière, il peut être intégré à n'importe quel processus de développement logiciel de manière transparente.

UML décrit neuf types de diagrammes qui seront utilisés tout au long des chapitres. Ils pourront tous être présents en même temps, ou seulement quelques-uns, selon les besoins. Ces diagrammes peuvent être regroupés en deux vues : statique et dynamique.

Vues statiques du système

Cette vue s'intéresse à la structure du système sans se soucier de son comportement dans le temps.

Diagrammes de cas d'utilisation
Les *use cases* permettent de structurer les besoins utilisateurs et les objectifs d'un système. Ils se limitent aux préoccupations « réelles » des utilisateurs et ne présentent pas de solutions d'implémentation. Les cas d'utilisation identifient les utilisateurs du système (acteurs) et leur interaction avec ce dernier.

Diagrammes de classes
Un diagramme de classes est une collection d'éléments de modélisation statiques (classes, interfaces, paquetages...), qui montre la structure d'un modèle. Il fait abstraction des aspects dynamiques et temporels. Ce diagramme se concentre sur les relations entre classes (association, héritage…).

Diagrammes d'objets
Ce type de diagramme UML montre des objets (instances de classes dans un état particulier) et des liens (relations sémantiques) entre ces objets. Ils s'utilisent pour montrer un contexte.

Diagrammes de composants
Les diagrammes de composants permettent de décrire l'architecture physique et statique d'une application en terme de modules : fichiers sources, librairies, exécutables, etc. Les composants peuvent être organisés en paquetages, qui définissent des sous-systèmes.

Diagrammes de déploiement
Les diagrammes de déploiement montrent la disposition physique des matériels qui composent le système et la répartition des composants sur ces matériels. Les ressources matérielles sont représentées sous forme de nœuds qui peuvent être connectés entre eux à l'aide d'un support de communication.

Vues dynamiques du système

Un logiciel repose principalement sur le comportement; par conséquent, la vue dynamique est très importante. Elle s'intéresse aux changements dans le temps du système.

Diagrammes de collaboration
Les diagrammes de collaboration montrent des interactions entre objets (instances de classes) et acteurs.

Diagrammes de séquence
Les diagrammes de séquences permettent de représenter des collaborations entre objets selon un point de vue temporel, on y met l'accent sur la chronologie des envois de messages. Contrairement au diagramme de collaboration, on n'y décrit pas le contexte ou l'état des objets, la représentation se concentre sur l'expression des interactions. Ils peuvent illustrer un cas d'utilisation.

Diagrammes d'états-transitions
Ce diagramme sert à représenter des automates d'états finis, sous forme de graphes d'états, reliés par des arcs orientés qui décrivent les transitions. Les diagrammes d'états-transitions permettent de

décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs.

Diagrammes d'activités
UML permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation, à l'aide de diagrammes d'activités (une variante des diagrammes d'états-transitions). Une activité représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles.

4+1 Vues

Demandez à dix architectes de vous donner une définition du mot architecture et vous obtiendrez dix définitions différentes. Dans le livre « *Software Architecture and the UML* » de Grady Booch vous avez la liste suivante :

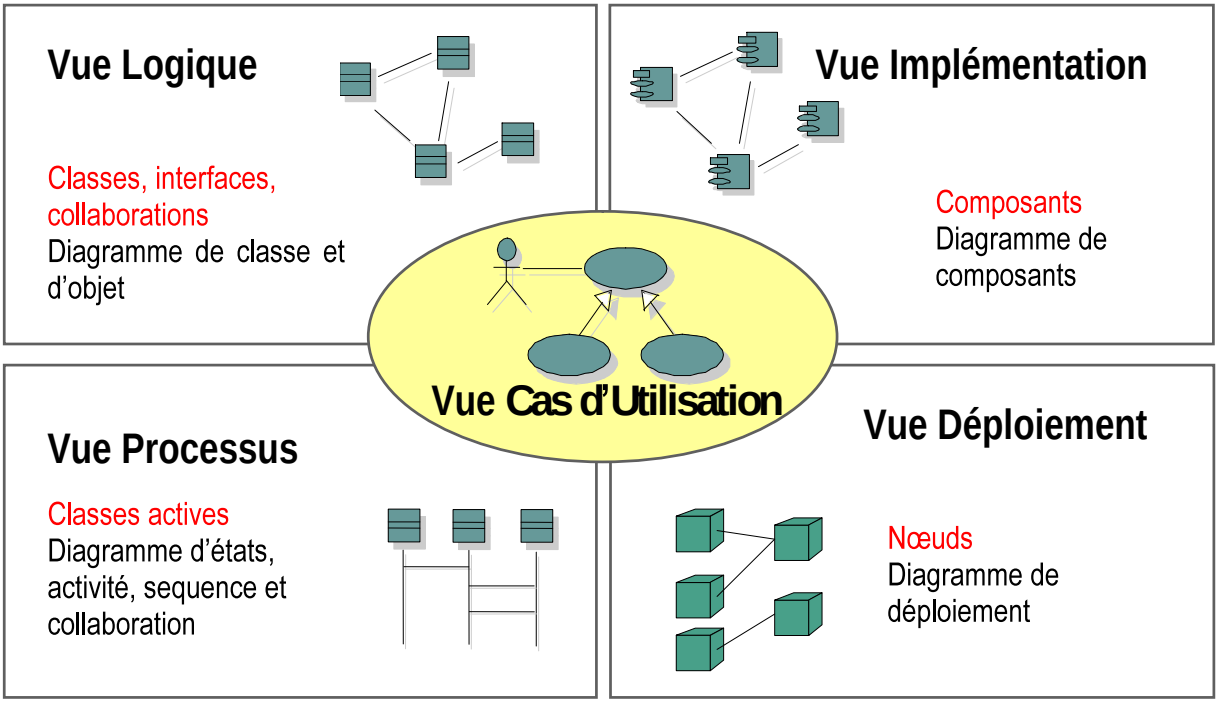
- Architecture et conception, c'est la même chose.
- Architecture et infrastructure, c'est pareil.
- <ma technologie préférée> est l'architecture.
- Une bonne architecture est le fruit du travail d'un seul architecte.
- L'architecture ne sert à rien, un Blueprint est suffisant.
- L'architecture, c'est juste de la structure.
- L'architecture système précède l'architecture logicielle.
- L'architecture est une science.
- L'architecture est un art.
- Une architecture ne peut être ni validée ni mesurée.

Il n'existe pas une architecture type mais plutôt plusieurs types d'architecture. Il faut être à l'écoute des besoins utilisateurs et de leurs contraintes pour choisir la plus adéquate. Nous n'essaierons donc pas de redéfinir ce terme mais plutôt de savoir le reconnaître tout au long de ce livre. Lorsque la société YAPS aura besoin de nouvelles fonctionnalités ou que nous devrons faire face à de nouvelles contraintes, nous serons obligés de faire évoluer le système et souvent de changer son ossature.

Dans une équipe, un architecte a des relations avec plusieurs intervenants :

- Utilisateurs finaux.
- Clients.
- Chefs de projet.
- Ingénieurs système.
- Développeurs.
- Architectes d'autres équipes.
- Equipe de maintenance.

Chaque intervenant possède une vue différente sur l'architecture. Et pourtant une architecture adaptée est la clé de voûte du succès d'un développement car elle décrit des choix stratégiques qui déterminent en grande partie les qualités du logiciel (adaptabilité, performances, fiabilité...). Le travail d'un architecte est donc de prendre tous ces différents points de vue et de réaliser l'architecture adéquate satisfaisant tous les intervenants. Ce livre utilisera le principe des « 4+1 » vues pour décrire une architecture. Chacune de ces cinq vues est une description simplifiée d'un système à partir d'un angle différent.



Chaque vue est agrémentée de diagrammes UML synthétisant la vision d'un intervenant. Ce sont ces différentes vues qui vous seront données tout le long de ce livre pour vous permettre de développer les versions du *YAPS PetStore*.

Bien sûr, les applications n'ont pas toutes besoin d'utiliser la totalité de ces vues. Par exemple, les applications monoprocesseurs ne nécessitent pas de vue déploiement, les monoprocessus, quant à elles, n'ont pas besoin de la vue processus.

La vue logique

Cette vue de haut niveau se concentre sur l'abstraction et l'encapsulation, elle modélise les éléments et mécanismes principaux du système. Elle identifie les éléments du domaine, ainsi que les relations et interactions entre ces éléments :

- Les éléments du domaine sont liés au(x) métier(s) de l'entreprise.
- Ils sont indispensables à la mission du système.
- Ils gagnent à être réutilisés (ils représentent un savoir-faire).

La vue des composants

Cette vue de bas niveau (aussi appelée « vue de réalisation »), montre :

- L'allocation des éléments de modélisation dans des modules (fichiers sources, bibliothèques dynamiques, bases de données, exécutables, etc...).
- Identifie les modules qui réalisent (physiquement) les classes de la vue logique.
- L'organisation des composants, c'est-à-dire la distribution du code en gestion de configuration, les dépendances entre les composants...
- Les contraintes de développement (bibliothèques externes...).
- l'organisation des modules en "sous-systèmes", les interfaces des sous-systèmes et leurs dépendances (avec d'autres sous-systèmes ou modules).

La vue des processus

Cette vue est très importante dans les environnements multitâches ; elle montre :

- La décomposition du système en termes de processus (tâches).
- Les interactions entre les processus (leur communication).
- La synchronisation et la communication des activités parallèles (*threads*).

La vue de déploiement

Cette vue très importante dans les environnements distribués, décrit les ressources matérielles et la répartition du logiciel dans ces ressources :

- La disposition et nature physique des matériels, ainsi que leurs performances.
- L'implantation des modules principaux sur les noeuds du réseau.
- Les exigences en termes de performances (temps de réponse, tolérance aux fautes et pannes...).

La vue des besoins des utilisateurs

Cette vue (dont le nom exact est « vue des cas d'utilisation »), guide toutes les autres. C'est la fameuse « +1 » des « 4+1 » vues.

- Cette vue définit les besoins des clients du système et centre la définition de l'architecture du système sur la satisfaction (la réalisation) de ces mêmes besoins.
- À l'aide de scénarios et de cas d'utilisation, cette vue conduit à la définition d'un modèle d'architecture pertinent et cohérent.
- Cette vue est la "colle" qui unifie les quatre autres vues de l'architecture.
- Elle motive les choix, permet d'identifier les interfaces critiques et force à se concentrer sur les problèmes importants.

Remaniement

Le remaniement, ou le *refactoring* en anglais, est une opération manuelle de maintenance du code. Elle consiste à retravailler le code source, non pas pour ajouter une fonctionnalité supplémentaire au logiciel mais pour améliorer sa lisibilité et simplifier sa maintenance. C'est donc une technique qui s'approche de l'optimisation du code, même si les objectifs sont radicalement différents. Bien que cette technique ne soit pas nouvelle, c'est Martin Fowler qui l'a institutionnalisé en 1999 avec son livre « *Refactoring – Improving the Design of Existing Code* ».

Au court de la vie d'un logiciel, on est amené à implémenter de nouvelles fonctions ou à corriger des bogues. Or ces modifications ne se plient pas toujours avec élégance dans l'architecture du logiciel. Le code source d'un programme tend donc à devenir de plus en plus complexe tout au long de son existence. Cela est d'autant plus vrai avec les techniques modernes de développement itératif et incrémental, où le logiciel entre en phase de modification pratiquement dés le début de son existence.

Pour améliorer le code qui s'enrichira au fur et à mesure des versions de *YAPS PetStore*, ce livre utilisera certains remaniements de code. Une fois de plus, le but n'étant pas d'utiliser la totalité du catalogue mais uniquement ceux qui nous sont utiles.

Design Pattern

Dans son livre « *A Pattern Language* » édité en 1977, l'architecte en bâtiment Christopher Alexander introduit le terme de *Pattern* (patron) : « chaque patron décrit un problème qui se produit de manière récurrente dans notre environnement ». Si ce livre est dédié à une autre profession que celle de l'architecture informatique, il faudra attendre le livre du *Gang of Four* (GoF) en 1994 pour ramener ses idées au monde de l'orienté objet. « *Design Pattern – Elements of Reusable Object-*

Oriented Software » de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides fourni un catalogue de 23 patrons de conception.

Il ne faut pas confondre ces patrons avec des briques logicielles (un *pattern* dépend de son environnement), des règles (un *pattern* ne s'applique pas mécaniquement) ou des méthodes (ne guide pas la prise de décision). Mais plutôt les voir comme une solution de conception a un problème récurrent.

Viendront alors, bien plus tard, deux livres s'inspirant du GoF mais se spécialisant à la plate-forme J2EE. « *EJB Design Pattern* » de Floyd Marinescu et « *Core J2EE Pattern* » de Deepak Alur, John Crupi et Dan Malks.

Ces catalogues ont créé un vocabulaire commun entre les développeurs, concepteurs et architectes. Il s'avère que durant le développement de l'application *YAPS PetStore*, nous viendrons à utiliser plusieurs de ces *design patterns*.

Test logiciel

Le test logiciel, lorsqu'il est correctement mis en œuvre, peut-être omniprésent tout au long du cycle de développement. Son but étant de trouver les défauts du système, il peut intervenir avant même le développement (ce que prône l'une des pratiques de l'*eXtreme Programming*) ou tout à la fin, pour les tests de recette. Il existe donc plusieurs niveaux.

Le test unitaire
Ce test contrôle le plus petit composant compilable, c'est-à-dire une classe. Il permet de savoir si elle correspond à ses spécifications ou s'il y a des erreurs de logique. Ce test est généralement fait directement par le développeur de la classe, qui conçoit lui-même le test en question. Il doit être fait de manière isolée en premier lieu (pour savoir si la classe fonctionne comme on le souhaite), puis en combinaison avec les classes qui travaillent avec elle.

Le test d'intégration
Ce test cherche à tester la stabilité et la cohérence des interfaces et interactions de l'application. Il est réalisé par l'équipe de développement plutôt que par une équipe indépendante.

Le test système
On teste ici la fiabilité et la performance de l'ensemble du système, tant au niveau fonctionnel que structurel, plutôt que de ses composants. On teste aussi la sécurité, les sorties, l'utilisation des ressources et les performances.

Le test de recette
Ce test doit confirmer que l'application répond d'une manière attendue aux requêtes qui lui sont envoyées. Ce sont les utilisateurs qui doivent faire ce test, et surtout pas les développeurs. Il permet d'adapter l'application aux attentes des futurs clients.

Le test de régression
Ce test fait suite à une modification de l'application (du fait d'une mise à jour ou de l'aboutissement du test de recette) : il faut alors vérifier que l'application n'a pas perdu de fonctionnalités suite à ces modifications.

Durant le développement de l'application *YAPS PetStore*, vous pourrez utiliser les tests unitaires si vous le désirez. Cela ne vous est pas demandé, mais peut vous permettre d'obtenir un code plus robuste. Par contre, chaque chapitre sera agrémenté de tests de recette simulant une recette

utilisateur. Vous aurez ainsi le code Java que le client souhaite voir fonctionner. Au fur et à mesure que l'application se complexifie, les tests de recette, eux, s'enrichissent.

Outils

Plusieurs outils seront utilisés dans vos développements. Certains ne le seront que ponctuellement, alors que d'autres feront partie intégrante des cycles de livraison de l'application. Tous ont la particularité d'être gratuit et parfois même *Open Source* (logiciel libre), c'est-à-dire qu'il ne vous coûtera rien de les utiliser, et vous pourrez même participer à leur développement s'il vous en prenait l'envie.

Ne vous affolez pas si vous ne connaissez pas ces outils. Nous aurons l'occasion de revenir dessus à chaque fois que nous les utiliserons, ou, tout simplement, à leur installation.

JDK

Le *Java Development Kit*, communément appelé JDK, est le kit de développement proposé gratuitement par *Sun*. Il comprend plusieurs outils, parmi lesquels :

- javac: le compilateur Java.
- java: un interpréteur d'applications (machine virtuelle).
- javadoc: un générateur de documentation.
- jar: un compresseur de classes Java.

xUnit

Comme le prône l'*eXtreme Programming*, l'utilisation de tests unitaires est essentielle à la constitution d'un code robuste. De plus, cela permet de remanier le code sans introduire de nouveaux bogues. Pour vous aider à écrire les classes de test unitaire ou pour faire fonctionner les tests de recette utilisateur, nous utiliserons plusieurs framework : JUnit pour tester les classes java, HttpUnit pour tester les composants vivant dans un serveur web et J2EEUnit pour les serveurs d'application.

Vous l'aurez compris, les produits xUnit seront utilisés tout le long des développements. Les classes de test de recette feront office de *feedback* utilisateur en vous informant si oui ou non votre application répond aux besoins spécifiés.

Ant

Ant est devenu au monde Java ce que *MAKE* est devenu au monde du C : un outil incontournable pour automatiser des traitements répétitifs en mode batch. Il est simple d'utilisation, bâti sur des technologies ouvertes (Java et XML), extensible, et supporté par de nombreux logiciels.

Ant sera utilisé pour compiler les applications java et les déployer, ainsi que pour la création et la mise à jour de la base de données.

MySQL

Le logiciel MySQL est un serveur de base de données relationnel, multi-threadé, multi-utilisateur et robuste. Il dispose de deux licences : *Open Source*, sous les termes de la licence GNU (*General Public License*) ou bien, une licence commerciale achetée auprès de MySQL AB. Nous utiliserons la version libre et ses différents outils comme l'éditeur de requête SQL.

Tomcat

Tomcat est un conteneur web J2EE, hébergeant des applications web composées de *servlets* et/ou JSP. Il fait également office de serveur web grâce à un mécanisme dédié au service de fichiers, c'est-à-dire à des pages HTML statiques. La popularité de *Tomcat* est telle qu'il est utilisé comme implémentation de référence des spécifications *Servlet*.

JBoss

JBoss est un serveur d'application *Open Source* certifié J2EE. Historiquement spécialisé sur les EJB, son architecture modulaire lui permet de répondre aux autres spécifications J2EE : *JBoss Server* est le conteneur d'EJB, JBossMQ gère les messages (JMS), JBoss MX la messagerie électronique, JBoss TX les transactions JTA/JTS, JBoss SX la sécurité, JBoss CX la connectivité JCA et JBossCMP la persistance CMP. Pour les JSP/*servlets*, JBoss intègre *Tomcat*.

IDE

Avec le JDK et un simple traitement de texte, vous pouvez développer n'importe quelle application. Cependant, il est important d'avoir un outil intégré pour vous permettre d'accélérer vos développements. Vous pourrez ainsi utiliser *Eclipse* ou *NetBean* pour les outils *Open Source*, ou n'importe quel autre IDE.

Méthode de travail

Vous l'aurez déjà sans doute compris, vous allez devoir développer tout au long des chapitres de ce livre. Vous aurez cependant deux manières pour le faire. La première, développer entièrement chaque chapitre. L'expression des besoins et l'analyse sont suffisamment précises pour que vous puissiez y arriver. Si vous n'y arrivez pas ou si vous accrochez sur une des technologies, utilisez la deuxième manière : téléchargez les sources du chapitre qui vous permettront de démarrer et développez les classes manquantes. En effet, à chaque chapitre, vous aurez à votre disposition un fichier compressé contenant la quasi-totalité des classes de l'application ainsi que les différents scripts, charge à vous de les compléter.

Tout au long de vos développements, vous aurez à votre disposition les classes de test correspondant à la recette utilisateur. Ces classes vous permettent de savoir si votre application répond aux expressions des besoins ou non. Exécutez les autant de fois que nécessaire jusqu'à ce que tous les tests réussissent.

Enfin, si vous n'arrivez vraiment pas à développer l'application d'un chapitre ou que vous êtes curieux de savoir comment d'autres pourraient le faire, vous pouvez télécharger la totalité du code.

Versions

Les outils, API et les spécifications Java sont constamment en train d'évoluer. Le tableau ci-dessous liste les versions utilisées dans ce livre pour développer le *YAPS PetStore*.

Outils	Version	API	Version
JDK	1.4	EJB	2.1
JUnit		Servlet	2.4
HTTPUnit		JSP	2.0
J2EEUnit		TagLib	
Ant		JDBC	
MySQL		HTML	1.0/1.1
Tomcat			
JBoss			

Référence

- *Java 2 Platform, Standard Edition (J2SE)*
  `http://java.sun.com/j2se/`

- *Java 2 Platform, Enterprise Edition (J2EE)*
  `http://java.sun.com/j2ee/`

- *BluePrints - Guidelines, Patterns, and code for end-to-end Java applications*
  `http://java.sun.com/blueprints/enterprise/index.html`

- *Java PetStore*
  `http://java.sun.com/developer/releases/petstore/`

- *Designing Enterprise Applications with the J2EE Platform*
  Inderjeet Singh, Beth Stearns, Mark Johnson, Enterprise Team. Addison-Wesley. 2002.

- *JCP – Java Community Process*
  `http://www.jcp.org/`

- *Extreme Programming*
  `http://www.extremeprogramming.org/`

- *Extreme Programming Explained: Embrace Change*
  Kent Beck. Addison-Wesley. 1999.

- *UML*
  `http://www.uml.org/`

- *The Unified Modeling Language User Guide*
  Grady Booch, Ivar Jacobson, James Rumbaugh. Addison-Wesley. 1998.

- *Refactoring*
  `http://www.refactoring.com/`

- *Refactoring – Improving the Design of Existing Code*
  Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. Addison-Wesley. 1999.

- *Design Pattern*
  `http://c2.com/cgi/wiki?CategoryPattern`

- *Design Pattern – Elements of Reusable Object-Oriented Software*
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley. 1995.

- *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, 2nd Edition, Volume 1*
  Mark Grand. Wiley. 2002.

- *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*
  Deepak Alur, Dan Malks, John Crupi. Prentice Hall. 2003.

- *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*
  Floyd Marinescu. Wiley. 2002.