Developpez

Le Mag

Edition de Février - Mars 2010.
Numéro 26.
Magazine en ligne gratuit.

Réalisation : Alexandre Pottiez
Rédaction : la rédaction de Developpez
Contact : magazine@redaction-developpez.com

Sommaire

Editorial

Ce mois-ci nous inaugurons une nouvelle rubrique pour le magazine, en effet la rubrique Mobiles fait sont apparition dans le magazine. Bien qu'assez pauvre dans ce numéro, elle s'étoffera bien vite à n'en pas douter, tellement cette technologie fait figure d'avenir. Bonne lecture à tous, pour ce magazine que, j'espère, vous apprécierez comme à chaque numéro.

La rédaction

Article JavaScript/AJAX

Comment créer facilement un framework JavaScript

Apprenez pas à pas à créer un framework JavaScript.

par **Teylor Feliz**
Page 17

Article C/C++/GTK/Qt

Les fonctions virtuelles en C++ : Types statiques et types dynamiques

Les fonctions virtuelles sont un des piliers de la programmation orientée objet. Cet article se propose d'explorer les fonctions virtuelles dans le langage C++.

par **3DArchi**
Page 28

Tutoriel AspectJ : création aspect LOG aspectJ

Favorisez la modularité, la réutilisabilité et la maintenance de vos applications avec AspectJ. Libérez-vous du codage 'en dur' de traces dans vos classes. Éliminez tout code 'System.out.println' ou 'log4j' dans vos classes applicatives. Mettez en œuvre AspectJ en créant un premier 'aspect' LOG. Utilisez le compilateur ajc pour compiler aussi bien des fichiers java (.java) que des fichiers AspectJ (.aj) et réaliser un tissage. Interceptez la construction d'un objet, le changement d'état d'une variable ou le lancement d'une exception. Analysez des exemples de points de jonction (JoinPoint), coupes (pointcut), greffons (advice) AspectJ

1. Introduction

Ce tutoriel est un extrait de séances pratiques de la formation SPRING (Lien1) dispensée par Objis.

Nous allons vous faire aimer Java

2. Prérequis

- Installation JDK
- Installation kit de développement AspectJ (AJDK)

3. Objectifs

- Créer et mettre en œuvre un aspect de LOG
- Comprendre le lien entre le compilateur aspectJ (ajc) et le compilateur Java (javac).
- Déclarer un aspect, une coupe, un point de jonction avec AspectJ
- Comprendre la valeur ajoutée de la programmation Aspects

4. Programme

- Contexte : application bancaire
- Partie 1 : tracer les retraits d'argent sans aspectJ.
- Partie 2 : tracer les retraits d'argent avec un aspect AspectJ de log : LogAspect.aj
- Partie 3 : nouvelle version de l'aspect de LOG.
- Partie 4 : mise en œuvre d'un profiling

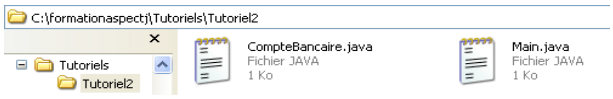
Durée totale : 40 min.

5. Contexte

Dans le cadre d'un projet d'envergure pour un établissement financier de la place, vous devez tracer les opérations de retrait d'argent de tout compte bancaire. Vous devez être capable de tracer l'état du compte AVANT le retrait et l'état du compte APRES le retrait.

6. Partie 1 : tracer sans aspectJ.

Fichiers à créer



Créez un fichier **CompteBancaire.java** qui sera la classe métier représentant un compte bancaire. Ajoutez une propriété solde ainsi que getter/setter puis constructeur. Ajoutez enfin les méthodes de retrait et dépot.

```
package com.objis.demoaspectj.banque;

public class CompteBancaire {
	private int solde;

	public CompteBancaire(int solde) {
		super();
		this.solde = solde;
	}

	public void depot (int sommeDepot){
		solde = solde + sommeDepot;
	}

	public void retrait (int sommeRetrait){
		solde = solde - sommeRetrait;
	}

	public int getSolde() {
		return solde;
	}

	public void setSolde(int solde) {
		this.solde = solde;
	}
}
```

Créez un fichier **Main.java** qui sera la classe principale de l'application (il contiendra la méthode main()). Elle va instancier un compte Bancaire et réaliser une opération de retrait.

```
package com.objis.demoaspectj;

import
com.objis.demoaspectj.banque.CompteBancaire;

public class Main {
	public static void main(String[] args) {
		// 1 : Création d'un compte avec
```

```
solde initial
                CompteBancaire monCompte = new
CompteBancaire(1000);

                // 2 : Retrait
                System.out.println("AVANT le
retrait");

                // 3 : Retrait
                monCompte.retrait(300);

                // 4 : Retrait
                System.out.println("APRES le
retrait");
        }
}
```

REMARQUE : sur les 4 étapes ci-dessus :

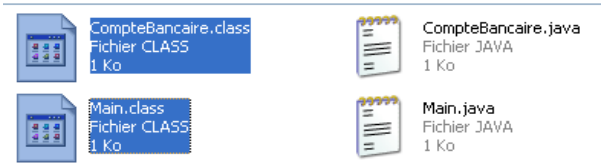
- 2 étapes ne concernent pas directement le 'métier'. En effet les étapes 2 et 4 sont liées à une préoccupation 'technique' : celle de tracer un évènement (ici le retrait).
- 2 étapes concernent le métier. En effet les étapes 1 et 3 sont liées à des préocupation directement liées au métier bancaire : créer un compte et effectuer un retrait.

INFO : Le 'tisseur d'aspects' AspectJ va nous permettre par la suite d'isoler cette préocupation technique dans un fichier distinct : le fichier LogAspectj.aj qui représente l'aspect LOG.

Compilez les classes avec le compilateur javac (issu du kit de développement Java).

```
javac CompteBancaire.java Main.java
```

Vous obtenez ceci :



Rangez les classes générées (Main.class et CompteBancaire.class) respectivement dans les répertoires com/objis/demoaspectj/ et com/objis/demoaspectj/banque.

Lancez l'exécution de la classe principale : Main

```
java - cp .;lib\aspectjrt.jar
com.objis.demoaspectj.Main
Avant le retrait
Après le retrait
```

Nous obtenons bien les traces qui précèdent et suivent l'appel à la méthode retrait().

Pour cela nous avons dû écrire 'en dur' dans la classe cliente les lignes suivantes :

- System.out.println("AVANT le retrait") ;
- System.out.println("APRES le retrait") ;

Nous aurions également pu écrire ces lignes dans la méthode retrait() de la classe appelée (CompteBancaire).

Et s'il était possible d'afficher les mêmes traces sans écrire ces lignes ni dans la classe appelante (ici Main), ni dans la classe appelée ?

C'est là qu'intervient un tisseur d'aspect comme AspectJ.

7. Partie 2 : tracer avec aspectJ

Dans cette partie vous allez mettre le tisseur d'aspect AspectJ en action. Vous allez supprimer tout code de Log dans vos classes et centraliser la gestion des logs dans un aspect aspectJ : LogAspect.aj

Vous allez utiliser le compilateur ajc (surcouche du compilateur javac) pour compiler aussi bien l'aspect LogAspect.aj que les classes Main.java et CompteBancaire.java.

Modifiez le contenu de la classe Main de la façon suivante :

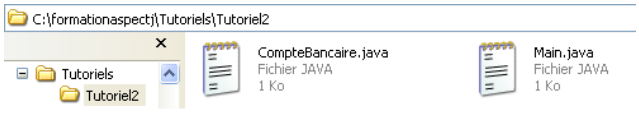
```
package com.objis.demoaspectj;

import
com.objis.demoaspectj.banque.CompteBancaire;

public class Main {
        public static void main(String[] args) {
                // 1 : Création d'un compte avec
solde initial
                CompteBancaire monCompte = new
CompteBancaire(1000);

                // 2 : Retrait
                monCompte.retrait(300);
        }
}
```

Comme vous le constatez, il n'y a aucune ligne associée au Log . C'est un aspect LogAspect.aj que nous allons créer qui va 'intercepter' toute demande de retrait.



Créez un fichier LogAspect.aj et ajoutez le contenu suivant :

```
package com.objis.demoaspectj.aspects;

public aspect LogAspect {
        pointcut logRetrait()
        : execution(*
com.objis.demoaspectj.banque.CompteBancaire.retra
it(..));

        before() : logRetrait() {
                System.out.println("Avant le
retrait");
        }

        after() : logRetrait() {
                System.out.println("Après le
retrait");
```

```
        }
}
```

Explications :

1. vous déclarez un aspect à travers le mot clé 'aspect'. Ici l'aspect LogAspect ;
2. vous déclarez une coupe nommée logRetrait() à travers le mot clé 'pointcut'. Une coupe est un ensemble de point de jonction (Moments d'exécution où il se passe quelque chose qui vous intéresse. C'est l'équivalent de point d'arrêt lors d'un débogage) ;
3. vous déclarez l'ensemble des points de jonction. Ici toute méthode retrait() de la classe com.objis.demoaspectj.banque.CompteBancaire, quelque soit le nombre de paramètres de la méthode retrait (..) et quelque soit le type de retour (*) de la méthode retrait() ;
4. vous déclarez un greffon type before() : le code System.out.println("AVANT le retrait") ; sera lancé juste avant tout point de jonction (c'est à dire ici toute exécution de la méthode retrait()) ;
5. vous déclarez un greffon type after() : le code System.out.println("APRES le retrait") ; sera lancé juste après tout point de jonction (c'est à dire ici toute exécution de la méthode retrait()).

ça y est : vous avez codé votre premier aspect 100% aspectJ. Reste à le compiler.

Compiler l'ensemble des classes Main.java, CompteBancaire.java et LogAspect.aj en utilisant le compilateur ajc (aspectj compiler) installé avec le kit de développement AspectJ (AJDK) :

```
ajc Main.java CompteBancaire.java LogAspect.aj
```

Vous obtenez ceci :

Rangez les classes générées (Main.class, CompteBancaire.class et LogAspectj.class) respectivement dans les répertoires com/objis/demoaspectj/ , com/objis/demoaspectj/banque et com/objis/demoaspectj/aspects.

Remarque : la commande suivante crée pour vous l'arborescence :

```
ajc -d . Main.java CompteBancaire.java
LogAspect.aj
```

Créez un répertoire 'lib' et ajoutez le jar aspectjrt présent dans ASPECTJ_HOME\lib . L'aspect aura besoin de ce jar à l'exécution.

Lancez l'exécution de la classe Main :

```
java - cp .;lib\aspectjrt.jar
com.objis.demoaspectj.Main
```

```
Avant le retrait
Après le retrait
```

Le résultat est le même que dans la partie 1. Mais le code de notre classe principale est plus léger. Nous nous sommes concentrés sur le métier et non sur une préoccupation de log.

8. Partie 3 : deuxième version de l'aspect

Expliquez l'effet de l'aspect suivant :

```
package com.objis.demoaspectj.aspects;

import
com.objis.demoaspectj.banque.CompteBancaire;

public aspect LogAspect2 {
        pointcut logRetrait(CompteBancaire
compte, int sommeRetrait) : call(void
com.objis.demoaspectj.banque.CompteBancaire.retra
it(int))
                                    && target(compte)
                                    &&
args(sommeRetrait);

        before(CompteBancaire compte, int
sommeRetrait) : logRetrait(compte, sommeRetrait)
{
                System.out.println("Avant le
retrait de " + sommeRetrait + " euros du compte "
+ compte);
        }

        after(CompteBancaire compte, int
sommeRetrait) : logRetrait(compte, sommeRetrait)
{
                System.out.println("Après le
retrait de " + sommeRetrait + " euros");
        }
}
```

Analysez le résultat :

```
java - cp .;lib\aspectjrt.jar
com.objis.demoaspectj.Main
Avant le retrait de 300 euros du compte
com.objis.demoaspectj.banque.CompteBancaire@9304b
1
Après le retrait de 300 euros
```

Vous découvrez ici une technique permettant de passer à un greffon le contexte d'exécution du point de jonction.

Expliquez l'effet du code suivant :

```
// Intercepter le constructeur de la classe
CompteBancaire
pointcut constructeur() : call
(CompteBancaire.new(..));

before () : constructeur(){
        System.out.println("Avant methode
constructeur");
}

after () : constructeur(){
        System.out.println("Après methode
constructeur");
}
}
```

Ajoutez ce code à l'aspect.

Exécutez.

9. Partie 4 : Aspect Profiling

Analysez le code suivant

```
package com.objis.demoaspectj.aspects;

public aspect ProfilingAspect {

        pointcut publicOperation() :
execution(public * *.*(..));

        Object around() : publicOperation() {

    long debut = System.nanoTime();

    Object ret = proceed();

    long fin = System.nanoTime();

    System.out.println(thisJoinPointStaticPart.ge
tSignature() + " a pris " + (fin-debut) + "
nanoseconds");

    return ret;
        }
}
```

- A quoi sert cet aspect ?
- Où est la coupe ?
- Combien y a-t-il de greffons ? de quel type ?

Comprendre thisJoinPointStaticPart

Remarque : la variable thisJoinPointStaticPart est une des 3 variables disponibles dans chaque greffon.

Cette variable apporte des informations à propos du point de jonction courant.

Exemples d'info :

- signature de la méthode ;
- l'objet this ;
- les arguments de la méthode.

Mise en oeuvre

A VOUS DE JOUER : utilisez le compilateur ajc pour compiler cet aspect avec le programme principal.

Résultat attendu :

```
java - cp .;lib\aspectjrt.jar
com.objis.demoaspectj.Main
Avant le retrait
Après le retrait
void
com.objis.demoaspectj.banque.CompteBancaire.retra
it(int) a pris 2250064 nanoseconds
void com.objis.demoaspectj.Main.main(String[]) a
pris 90539444 nanoseconds
```

Expliquez

10. Conclusion

Dans ce tutoriel, vous avez mis en oeuvre AspectJ à travers la création d'un aspect LogAspect.aj

Retrouvez l'article d'Objis en ligne : Lien2

Les livres Java

Programmation GWT 2

Développer des applications RIA et Ajax avec Google Web Toolkit

Pour peu qu'on en maîtrise les prérequis d'architecture, le framework GWT 2 met à la portée de tous les développeurs web la possibilité de créer des applications web 2.0 interactives et robustes avec autant d'aisance qu'en Flex ou Silverlight. Publié en licence libre Apache, le Google Web Toolkit génère depuis Java du code Ajax (JavaScript et XHTML/CSS) optimisé à l'extrême. La référence sur GWT 2 : une autre façon de développer pour le web. La version 2 de GWT est une révolution en termes de productivité, de simplicité et de robustesse. C'est ce que montre cet ouvrage de référence sur GWT 2, qui fournit au développeur une méthodologie de conception et de bonnes pratiques d'architecture. Il détaille les concepts qui sous-tendent le framework pour en donner les clés d'une utilisation pertinente et optimisée : performances, séparation en couches, intégration avec l'existant, design patterns, sécurité... De la conception à l'optimisation et aux tests, toutes les étapes du développement sont déroulées, exemples de code à l'appui. S'adressant tant au développeur qu'à l'architecte, l'ouvrage dévoile les coulisses de chaque API au fil d'un cas d'utilisation simple et décortique les bibliothèques tierces principales telles que GWT-DnD, les API Google Calendar, Google Maps, SmartGWT et Ext-GWT...

Critique du livre par benwit

Pour diffuser des ressources sur une technologie, on peut soit traduire une ressource existante dans une autre langue, soit créer une nouvelle ressource de toute pièce. Sami Jaber a choisi, pour notre plus grand bonheur, la seconde solution. L'originalité de son ouvrage consiste à apporter des informations complètement inédites sur GWT 1.0 et 2.0.

Sur la forme, c'est un ouvrage technique qui se picore davantage qu'il ne se lit de bout en bout.

Sur le fond, c'est assez riche. Jugez plutôt :

1) Des chapitres "classiques" qu'on retrouve plus ou moins dans les autres livres GWT.

L'introduction intéressera particulièrement les décideurs puisqu'elle brosse une vue d'ensemble de GWT et le situe

face aux autres frameworks Ajax.

Les chapitres "L'environnement de développement", "Les contrôles", "Le plug-in Eclipse pour Gwt" s'adresseront davantage aux développeurs débutants qui n'auraient pas préalablement lu de livre sur le sujet.

Le chapitre "Les bibliothèques tierces" est un peu tiède à mon goût, à la fois pas assez exhaustif (comment pourrait-il l'être ?) et trop détaillé pour un simple survol (Qu'est ce qu'apporte un seul exemple ?). Notez bien mon questionnement car à la place de l'auteur, je ne sais pas comment j'aurai fait ? Il ne pouvait pas passer dessus et il ne pouvait pas trop détailler non plus pour des contraintes éditoriales évidentes ; il a donc opté pour une solution intermédiaire. Une fois le livre terminé, ma perplexité subsiste. Les librairies présentées sont intéressantes aujourd'hui mais comparées au potentiel de GWT 2, elles paraissent bien fades.

- Smart GWT étant un wrapper de code JS, on perd des avantages de GWT (comme le debug pas à pas, la prévention contre les fuites mémoires...)
- GXT quand à elle est une librairie full GWT mais tous ceux qui ont regardé son code vous diront qu'elle est loin déjà d'avoir intégré toute la puissance de GWT 1.0 (deferred binding). On a donc hâte qu'elle soit réécrite en tirant toutes les spécificités supplémentaires introduites par GWT 2.0.
- Gears est aujourd'hui abandonnée au profit d'une solution plus standard du W3C.
- Quant aux autres, plus particulières, on a qu'un souhait : que leur intégration future à GWT se fasse le plus tôt possible.

Le chapitre "Les services RPC" ressemble pour l'essentiel à ce qu'on trouve ailleurs. Certes, l'auteur parle de deRPC, la nouvelle version du RPC, du mode hybride compatible RPC 1.X et RPC 2.X, de bonnes pratiques et du contournement de la règle SOP (Exemple du proxying) mais il m'a un peu laissé sur ma faim. J'aurais aimé dans ce chapitre une section "sous le capot" telle qu'on les trouve dans d'autres chapitres. J'aurais également aimé qu'il détaille la connexion d'un client RPC à un serveur non RPC.

2) Des chapitres qui détaillent des aspects de GWT comme aucun ouvrage ne l'a fait jusqu'à présent.

Le chapitre "L'intégration de code Javascript" est passionnant. En suivant la problématique, l'approche historique et les débats de l'équipe de développement GWT, on a l'impression d'être avec eux dans les coulisses.

Le chapitre "La création de composants personnalisés" est assez étoffé. On y apprend surtout pourquoi Javascript a des fuites mémoires et comment GWT s'y prend pour les éviter. A conseiller aux développeurs Javascript !!!

Le chapitre "L'intégration J2EE" fournit des pistes à ceux qui doivent s'interfacer avec du Spring et des EJB. Certains reprocheront que la liaison avec leur librairie préférée n'est pas abordée mais vu la richesse de J2EE, le choix de l'auteur se limitant aux deux principales est pertinent.

Les chapitres "La liaison différee" et "Sous le capot de GWT" détaillent les mécanismes utilisés par GWT pour générer du code multinavigateur et optimisé. Très intéressant !!!

Le chapitre "L'internationalisation" est très complet. Je note particulièrement la question des formes plurielles et des outils pour en écrire le moins possible (Feignant ? Non, productif !).

Dans le chapitre "Les design Pattern GWT", il est question à la fois des patterns MVC, MVP pour séparer les responsabilités et de bonnes pratiques concernant l'historique, les traitements longs et la sécurité. Cette dernière étant détaillée (injection sql, XSS, CSRF, authentification), elle pourrait peut-être faire l'objet d'un chapitre séparé ?

3) Et "must have", des chapitres complètements inédits sur la version 2.0 qui est sortie en même temps que l'ouvrage.

Le chapitre "Les modèles de placement CSS" intéressera tous ceux qui ont galéré pour faire la mise en page de leur application GWT. La nouvelle solution est beaucoup plus propre et est donc à essayer d'urgence !

Le chapitre "Le chargement à la demande" décrit la nouvelle fonctionnalité qui permet de partitionner le code javascript généré. Sur le papier, elle semble intéressante mais reste à voir en pratique si elle n'apporte pas davantage de complexité. Mais bon, l'ami Sami vous aura prévenu !

Le chapitre "La gestion des ressources" répond également à une problématique d'optimisation des temps de chargement. On y apprend comment mettre en cache tout types de ressources et même comment faire de l'injection de styles CSS (Il faut dire que l'extension du mécanisme ImageBundle a permis d'introduire des directives dans les fichier CSS).

Le chapitre "L'environnement de tests" présente les styles de test introduits dans la nouvelle version. A se demander comment on faisait avant ?

Le chapitre "La création d'interface avec UIBinder" sera utile à tous ceux qui préfèrent coder leur vues en XML.

Comme je vous le disais dans ma critique du livre d'Olivier Gérardin, les francophones ont dû attendre plus de trois ans les livres en français sur GWT mais une fois de plus, cela valait le coup d'attendre. Le hasard faisant bien les choses, je suis ravi de l'ordre de parution puisque après le premier livre en français sur GWT (plus didactique), les développeurs pourront continuer avec celui-ci (plus technique), le premier livre mondial sur GWT 2.0.

Bonus : trop rare pour ne pas être souligné, un site dédié est disponible à l'adresse http://www.programmationgwt2.com/web/guest (Lien3) pour remonter les coquilles, vos opinions ou simplement discuter avec l'auteur.

Apache Maven

Maven, l'outil open-source de gestion et d'automatisation de développement Java, a le vent en poupe.
Les raisons : il systématise, rationalise et simplifie le développement collaboratif de projets Java, faisant gagner aux entreprises comme aux développeurs du temps et de l'argent !
Les auteurs, membres de l'équipe de développement Maven, aidés par toute la communauté francophone, ont imaginé de présenter Maven 2 sous un angle original et didactique, à travers un projet fictif, inspiré de leurs expériences sur le terrain, dont ils détaillent toutes les phases successives.
Ce projet évolue au fil des besoins et de la contribution de développeurs aux profils différents, vous familiarisant avec les concepts fondamentaux de Maven et leur mise en oeuvre pratique, mais aussi avec les fonctionnalités plus avancées.
Vous profitez également des recommandations et bonnes pratiques pour optimiser votre utilisation de Maven.
Vous découvrez ainsi de manière ludique et grâce à des exemples concrets le potentiel de Maven, et tous les avantages qu'il peut apporter à vos propres projets.

Critique du livre par romaintaz

On peut s'en douter en lisant le titre, cet ouvrage va nous faire découvrir Maven, l'outil de construction de projets Java d'Apache.
Il ne faut pas s'attendre ici à trouver LA bible sur Maven, mais plutôt d'apprendre à utiliser cet outil au sein d'une organisation.

Pour ce faire, Nicolas de Loof et Arnaud Héritier nous racontent l'histoire d'une équipe de développeurs, qui va être confrontée à différentes situations - qui ne manqueront pas de nous rappeler nos propres expériences.

Le livre se divise en trois grandes parties.
La première se destine à nous familiariser avec Maven et ses principaux concepts.
La seconde aborde des concepts plus poussés, en particulier concernant l'utilisation de Maven au sein d'une entreprise.
Enfin la dernière partie regroupe différents sujets, par exemple des cas atypiques d'utilisation de Maven.

1re partie : premiers pas avec Maven

Cette première partie se destine à nous familiariser avec Maven et ses principaux concepts.
Les auteurs en profitent pour poser le décor de leur histoire. Nous suivrons donc l'équipe de développement d'un petit projet nommé *noubliepaslalistedescourses* !
Autrefois construite par un simple fichier .bat, cette application va rajeunir en étant migrée vers Maven.
C'est ici que l'on constate les premiers bénéfices de l'outil d'Apache : gestion simplifiée des dépendances, cycle de vie de construction d'un projet, création des livrables (fichier JAR final par exemple), etc.
Les auteurs s'attaquent à d'autres problématiques, telles que la gestion des versions du JDK, l'intégration d'autres langages (Groovy par exemple).
Un chapitre est également consacré à la politique de tests, en mettant en lumière les différents types de tests (unitaires, fonctionnels, d'intégration, etc.) et même l'intégration continue.

On le voit dès cette première partie : ce livre n'est pas là uniquement pour nous parler de Maven, mais bien de tout ce qui a trait à l'environnement de développement.

2e partie : Maven en entreprise

Cette seconde partie aborde des questions plus poussées des aspects de Maven qui prennent tout leur sens dans un environnement d'entreprise.
Ainsi, la première problématique abordée est la gestion avancée des dépendances, et en particulier l'introduction des dépôts d'entreprises (les *repositories*).
Par la suite, on s'attaque aux concepts des projets multimodule et de l'héritage, très utiles dès que les projets prennent de l'importance !
Il est également question de la construction de livrables plus complexes, en particulier les WAR, les EJB ou les EAR.

Un chapitre est dédié à l'intégration de tout ceci au sein du principal outil du développeur : l'IDE (l'environnement de développement).
Qu'il s'agisse d'Eclipse, de NetBeans ou encore d'IntelliJ, ils sont tous étudiés par les auteurs du livre, qui nous montrent aussi bien la facilité d'utilisation de Maven directement dans l'IDE, que les petits plus qu'ils ont à offrir au développeur pour faciliter son travail quotidien.

Cette seconde partie s'achève sur un point crucial de la vie d'un projet : la livraison !
Cette étape, trop souvent bâclée, peut faire échouer un projet. Les auteurs nous donnent donc les moyens proposés par Maven pour simplifier, automatiser et fiabiliser ce processus.

3e partie : encore plus loin avec Maven

Cette dernière partie aborde différents sujets qui n'ont pas encore été vus dans les chapitres précédents.
Tout d'abord, les auteurs nous expliquent comment il est possible de lancer des tâches par Maven alors qu'aucun plugin spécifique n'existe pour cela.
Il est ainsi question de développement de plugin Maven, mais aussi de l'intégration d'Ant au sein de Maven.

Un autre sujet connexe est abordé ici : la surveillance de la qualité du code.
Là aussi, Maven peut apporter des solutions - essentiellement grâce à des plugins ou à Sonar - et les auteurs ne manquent pas de nous le montrer.
C'est également le bon moment pour introduire la fonctionnalité de génération de site - et de rapports techniques - offerte par Maven !

Par la suite, un chapitre est entièrement consacré au démarrage d'un nouveau projet.
C'est ici l'occasion de voir les recommandations que l'on peut faire sur de nouveaux projets, et non plus de parler de migration de projets existants vers Maven.
On passe ainsi en revue certaines bonnes pratiques à mettre en place dès le début d'un projet, on apprend

également à utiliser les archétypes de Maven, ces plugins qui nous permettent de créer le squelette d'un projet en fonction de son type.

Maintenant que nous avons fait le tour des principales fonctionnalités de Maven, les auteurs se posent une question un peu déroutante, mais tout de même intéressante : "Avons-nous fait le bon choix ?".
C'est ici que l'on fait une sorte de bilan, de savoir si finalement Maven est véritablement l'outil à utiliser au sein de nos projets, d'en définir les limitations et les problèmes.
Les auteurs nous parlent ensuite de l'écosystème autour de Maven : ses concurrents (Ant, Ivy et Gradle en première ligne), les sociétés qui gravitent autour de l'outil (en particulier Sonatype), etc.
Enfin, on termine avec quelques pages nous montrant les nouveautés de Maven 3.

Avant de conclure cet ouvrage, les auteurs décident de nous offrir leurs précieuses recommandations vis-à-vis de l'utilisation de Maven, leurs *10 commandements*.

Mon avis

Apache Maven est un livre plutôt atypique.
Il s'apparente plus à un roman qu'à un livre technique traditionnel, et ne doit pas être pris pour une sorte de "bible" sur l'outil d'Apache.
Le fait que Nicolas de Loof et Arnaud Héritier aient abordé le sujet de Maven en racontant la vie et l'évolution d'une équipe de développement est une excellente idée, car cela nous montre un usage très réel et concret de cet outil.
Cela nous permet également de voir tous les à-côtés qui ne sont habituellement pas traités par des ouvrages informatiques sur un outil.
Ici, au-delà de Maven, les auteurs évoquent le TDD (Test Driven Development ou Développement piloté par les tests), les différentes façons de tester son application, le suivi de qualité, l'intégration continue, l'aspect livraison, les bonnes et mauvaises pratiques en général, etc.
C'est forcément un autre bon point pour ce livre !

Alors, certes, mon niveau en Maven qui est plutôt bon, ne m'a pas permis de sortir de cette lecture avec beaucoup de nouvelles informations.
Toutefois, cela m'a conforté dans mes idées, m'a rassuré sur mes bonnes pratiques, et m'a appris quelques astuces plutôt sympathiques.
Le fait que ce livre aborde vraiment tous les aspects de la vie d'un projet est un énorme plus, et pourra, à mon avis, intéresser des personnes n'ayant pas un profil de développeur !

Selon moi, ce livre doit être vu comme un indispensable complément à une "bible" Maven, par exemple *Maven: The Definitive Guide*, d'autant que ce dernier vient tout juste d'être traduit en français.

Vous trouverez une critique complète de cet ouvrage, ainsi que tous les détails relatif à ce livre à cette adresse : http://linsolas.developpez.com/critiques/apache-maven/ (Lien4)

Retrouvez ces critiques de livre sur la page livres Java : Lien5

Facebook dévoile sa ré-écriture de PHP, HipHop traduit PHP en C++ puis le compile avec g++

L'information vient d'être officiellement confirmée par Facebook, suite a de nombreuses rumeurs sur la toile qui anticipaient son annonce. Il se murmurait ici et là qu'une équipe du réseau social aurait travaillé sur un nouveau compiler PHP JIT (Just In Time) qui permettrait des augmentations de vitesse allant jusqu'à 80%.

Le projet serait assez similaire à l'Unladen Swallow de Google, qui avait consisté en une ré-écriture du compiler de Python.

Le projet des équipes de Facebook s'appelle HipHop et il est disponible depuis cet après-midi en open-source. Il résulte d'un travail acharné et secret de deux longues années.

Mais contrairement à ce qui était attendu, HipHop consiste en fait en une ré-écriture du runtime de PHP. Le code source de PHP est traduit en C++ puis compilé avec g++.

Un ingénieur ayant travaillé sur le projet, Haiping Zhao, déclare : "Avec HipHop, nous avons réduit l'usage du CPU sur nos serveurs web d'environ 50%".

Ceci devrait permettre d'alléger le datacenter de Facebook.

"HipHop exécute le code source d'une manière sémantique et en sacrifie certaines fonctionnalités peu utilisées, comme eval(), pour de meilleures performances. HipHop nous permet de corriger la logique de l'assemblage final d'une page en PHP", continue-t-il.

Il conclut en indiquant que PHP et C++ partagent pratiquement la même syntaxe, mais que C++ est bien moins gourmand en ressources système.

Commentez cette news en ligne : Lien6

PHP : Sortie de la version du Zend Framework 1.10.0 beta1

Zend Technologies annonce la sortie de Zend Framework 1.10 (Lien7)

Le Framework PHP Zend voit la prochaine version de la branche 1 passer en bêta.

Vous pouvez la télécharger à cette adresse (Lien8).

Cette version bêta signifie que toutes les nouvelles fonctionnalités sont prêtes et que toutes les nouvelles API intégrées sont considérées comme finales.

Vous trouverez entre autre dans cette version :

- **Zend_Barcode**, contributed by Mickael Perraud
- **Zend_Cache_Backend_Static**, contributed by Pádraic Brady
- **Zend_Cache_Manager**, contributed by Pádraic Brady
- **Zend_Exception - previous exception support**, contributed by Marc Bennewitz
- **Zend_Feed_Pubsubhubbub**, contributed by Pádraic Brady
- **Zend_Feed_Writer**, contributed by Pádraic Brady
- **Zend_Filter_Boolean**, contributed by Thomas Weidner
- **Zend_Filter_Compress/Decompress**, contributed by Thomas Weidner
- **Zend_Filter_Null**, contributed by Thomas Weidner
- **Zend_Log::factory()**, contributed by Mark van der Velden and Martin Roest (of ibuildings)
- **Zend_Log_Writer_ZendMonitor**, contributed by Matthew Weier O'Phinney
- **Zend_Markup**, contributed by Pieter Kokx
- **Zend_Oauth**, contributed by Pádraic Brady
- **Zend_Serializer**, contributed by Marc Bennewitz
- **Zend_Service_DeveloperGarden**, contributed by Marco Kaiser
- **Zend_Service_LiveDocx**, contributed by Jonathan Marron
- **Zend_Service_WindowsAzure**, contributed by Maarten Balliauw
- **Zend_Validate_Barcode**, contributed by Thomas Weidner
- **Zend_Validate_Callback**, contributed by Thomas Weidner
- **Zend_Validate_CreditCard**, contributed by Thomas Weidner
- **Zend_Validate_PostCode**, contributed by Thomas Weidner
- Additions to Zend_Application resources, including Cachemanager, Dojo, Jquery, Layout, Log, Mail, and Multidb (contributed primarily by Dolf Schimmel)
- Refactoring of Zend_Loader::loadClass() to conform to the PHP Framework Interop Group reference implementation, which allows for autoloading PHP 5.3 namespaced code
- Updated Dojo version to 1.4

Vous pouvez faire vos remontées de bug dans le bugtracker (Lien9).

Les prochaines versions sont attendues pour :
* 1.10.0rc1: 20/01/2010
* 1.10.0: 26/01/2010

Commentez cette news en ligne : Lien10

Les derniers tutoriels et articles

Les filtres PHP : une fonctionnalité importante de sécurité

Cet article est la traduction de « PHP Filters: An Important Security Feature » (Lien11) de Marc Plotz.
Vous avez sûrement remarqué que j'ai mentionné dans un de mes articles précédents que la plus grande faiblesse de PHP réside dans sa simplicité. Mais ne vous méprenez pas une seule seconde - je suis un développeur PHP, et je le serai jusqu'au jour où je mourrai. Mais il y a du bon code, et il y a du mauvais code. Cet article va vous apprendre comment faire en sorte que votre code tombe dans la bonne catégorie.

1. Introduction

Peut-être que certaines des discussions les plus amusantes que j'ai vues dans les forums de développeurs sont les discussions pour savoir si PHP est un "vrai" langage de programmation ou non. Apparemment, il est dit que PHP n'aura jamais la puissance de Java, parce que PHP est un langage faiblement typé. Eh bien, oui. Ce que vous n'avez pas besoin de vous rappeler cependant, c'est que PHP n'a jamais été conçu pour être un clone de Java. PHP n'est pas un système à l'état solide. Il dure une fraction de seconde - alors que la page se charge - et alors il cesse de fonctionner. C'est tout. C'est la raison pour laquelle il existe des méthodes comme GET, POST et les sessions en PHP : dans un système non à l'état solide, dont vous avez besoin pour transporter l'information d'une page à l'autre. En effet, PHP fait ce pour quoi il a été conçu, car c'est un langage faiblement typé, si vous codez pour qu'il le soit. Il appartient à la personne de la conception et la mise en œuvre du système de décider dès le début s'il va faire les choses correctement ou non. La même chose s'applique à vos techniques de validation.

La validation est peut-être la chose la plus importante que vous puissiez faire sur un site Web. Oublier de valider absolument toutes les parties de votre site Web ou une application qui interagit avec un utilisateur est probablement l'erreur la plus commune que vous puissiez faire. Je sais de par ma propre expérience que la validation peut être une douleur. Habituellement dans mon esprit ce grand nombre de SWITCH commence à émerger chaque fois que quelqu'un commence à parler de validation. Si cela vous arrivait, asseyez-vous et détendez-vous : PHP dispose en interne des fonctions de validation toutes prêtes pour que vous puissiez les utiliser à cette fin.

Les filtres PHP sont une extension de PHP qui vous aideront à facilement - et de manière fiable - valider les variables et les chaînes de caractères, de sorte que nous puissions espérer, qu'une chose comme cela ne se reproduise plus :

```
<?php
include($_GET['filename']);
?>
```

ou, pire encore,

```
<? php
mysql_query ( "INSERT INTO table (champ) VALUES
({$_POST [ 'value']})");
?>
```

2. Le filtrage des variables

Pour utiliser l'extension de filtre pour filtrer les variables, vous utilisez la fonction filter_var(). Essayons de valider le texte suivant comme un entier, par exemple.

```
<?php
$variable = 1122;
echo  filter_var($variable, FILTER_VALIDATE_INT);
```

Le résultat du echo est "1122" parce que le type de variable a été trouvé pour être un entier. Si la variable d'entrée était "A344" rien n'aurait été imprimé à l'écran car la validation a échoué. Ok, ok, je vois que vous vous dites que c'est un tour assez propre. Mais il y a plus. Supposons que nous voulons nous assurer que notre variable est un entier et a une valeur de plus de 5 et moins de 10. Comment ferions-nous cela?

```
<? php
$variable =6;
$minimum_value = 5;
$valeur_finale = 10;

echo filter_var($variable, FILTER_VALIDATE_INT,
array ( "options" => array ( "min_range" =>
$minimum_value ", max_range" =>
$valeur_finale)));
?>
```

Donc, la variable devrait être dans les limites - comme elle l'est dans l'exemple ci-dessus - le numéro 6 sera affiché à l'écran.

PHP fournit également une très bonne façon de vérifier les valeurs float - particulièrement utile pour ceux d'entre nous qui construisent des paniers d'achat et ont la nécessité de vérifier que les valeurs ont deux décimales. L'exemple ci-dessous affiche "31.53 est un nombre à virgule flottante valide".

```
<? php

$num = 31.53;

if (filter_var($num , FILTER_VALIDATE_FLOAT) ===
false)
(
echo $num. "n'est pas valide!";
)
 else
(
  echo $num. "est un nombre à virgule flottante
valide";
```

```
)

?>
```

N'avais-vous jamais essayé de valider une URL ? Si non, il est préférable que vous lisiez la RFC1738 - Uniform Resource Locators (Lien12) d'abord, puis ouvrez votre éditeur PHP et écrivez une classe qui décrit essentiellement les 2000 lignes de la RFC, non ?

Eh bien, non. En fait, PHP peut le faire automatiquement avec le filtre d'URL.

```
<?php

$url = "http://www.somewebsite.domain";

if(filter_var($url, FILTER_VALIDATE_URL) ===
FALSE)
{
echo $url." n'est pas une URL valide<br />";
}
else
{
echo $url." est une URL valide<br />";
}

?>
```

"http://www.somewebsite.domain est une URL valide" est la réponse que je reçois.

Maintenant passons à quelque chose qui m'énerve : la validation des adresses e-mail. C'est l'une de ces choses qui nécessite une expression régulière, non ? Faux. FILTER_VALIDATE_EMAIL le fait de manière simple, sans une goutte de sueur. Allons-y :

```
<?php

$email = "marc@somehost.com";

if(filter_var($email, FILTER_VALIDATE_EMAIL) ===
FALSE)
{
 echo $email." est invalide";
}
else
{
 echo $email." est valide";
}

?>
```

Maintenant, ne croyez-vous pas que cela en vaille la peine ? La validation d'email peut être un véritable casse-tête, surtout pour les débutants, donc à mon avis, c'est une petite bénédiction.

Mais il y a plus intéressant. La nécessité de supprimer les balises HTML dans une chaîne. Que pensez vous de cela ?

```
<?php
$string = "<p>text</p>";

echo filter_var($string, FILTER_SANITIZE_STRING);

?>
```

Le résultat est qu'il va simplement afficher "text" sans les balises.

3. Conclusion

Ce que nous avons étudié ici sont quelques exemples de ce que nous pouvons faire avec PHP FILTERS. Bien sûr, il est important de valider votre code - nous le savons tous. Mais le faire en réalité, c'est une autre histoire. Je suppose que cet article vous donne juste, que vous soyez un débutant ou un expert, un moyen d'être sûr que quelque chose est fait pour aider votre code dans son voyage du mauvais côté au bon. Que votre code soit le meilleur qu'il puisse être !

Retrouvez l'article de Marc Poltz traduit par Joris Crozier en ligne : Lien13

Utiliser des outils en ligne de commande avec PHP

Apprenez à mieux intégrer les scripts avec des outils de ligne de commande. Examinons avec les commandes shell_exec(), exec(), passthru() et system() comment passer des informations en toute sécurité à la ligne de commande, et comment en récupérer.

1. Introduction

Si vous avez déjà travaillé avec PHP, vous savez qu'il est un excellent outil pour la création de pages Web riches en fonctionnalités. Comme tout langage de script en général, PHP:

- Est facile à apprendre.
- A beaucoup de frameworks puissants et efficaces comme CakePHP et CodeIgniter, faisant de vous un programmeur aussi productif que n'importe quel autre.
- Peut communiquer avec MySQL, PostgreSQL, Microsoft ® SQL Server, et même Oracle.
- S'intègre facilement avec les frameworks JavaScript comme script.aculo.us et jQuery.

Cependant, parfois, vous voulez faire plus, ou vous êtes obligé de faire plus ; Par là, je veux dire si vous devez travailler directement avec le système de fichiers du serveur où est exécuté PHP, vous finissez par avoir besoin de travailler avec des fichiers sur le système, pour voir quels processus sont en cours d'exécution, ou executer d'autres tâches.

Au départ, vous avez du contenu en utilisant des commandes PHP comme file() pour ouvrir des fichiers. Cependant à un certain moment le seul moyen de faire

quelque chose est d'être capable d'exécuter des commandes shell sur le serveur et récupérer une partie de la sortie. Par exemple, vous pourriez avoir besoin de savoir combien de fichiers existent dans un répertoire donné ou vous pouvez avoir besoin de connaître le nombre de lignes qui ont été écrites pour un groupe de fichiers log ou encore vous pouvez avoir le besoin de faire certains traitements sur ces fichiers en le copiant dans un autre répertoire ou utiliser rsync pour les transporter vers un autre emplacement.

Dans l'article « Command-line PHP? Yes, you can!! » (Lien14), Roger McCoy montre comment utiliser PHP directement depuis la ligne de commande - pas de navigateur web nécessaire. Dans cet article, j'aborde le sujet d'un autre point de vue, en vous montrant comment intégrer dans vos scripts des commandes shell et d'y récupérer les données retournées dans vos interfaces et processus. Ce tutoriel se base sur la distribution Linux ® Berkeley Software Distribution (BSD), mais vous pouvez utiliser n'importe quel autre système sur base d'UNIX ® Nous supposons que vous travaillez sur un système basé sur la pile Linux-Apache-MySQL-PHP (LAMP). Vos connaissances peuvent varier si vous utilisez une autre variante UNIX, car la disponibilité des commandes varie d'une installation à l'autre. Comme beaucoup d'entre vous développent aussi sur Mac OS X, qui exploite une variante de BSD, nous gardons l'échantillon de commandes aussi générique que possible afin d'assurer la portabilité.

2. Vue d'ensemble de la ligne de commande

L'interface en ligne de commande (CLI)/Server Application Programming Interface (SAPI) PHP a été publiée à titre expérimental depuis la version 4.2.0 de PHP. A la version 4.3.0 elle a été entièrement prise en charge et activée par défaut. PHP CLI/SAPI vous permet de développer en shell. En effet, il est possible de créer des outils en PHP qui s'exécutent directement depuis la ligne de commande. De cette manière, les développeurs PHP peuvent être aussi productifs que ceux sur Perl, AWK, Ruby, ou les scripts shell dans ce contexte.

Cet article présente les outils intégrés à PHP, qui vous permettent de puiser dans l'environnement du shell de base et du système de fichiers où PHP est exécuté. PHP fournit un certain nombre de fonctions pour exécuter des commandes externes, nous avons parmi elles : **shell_exec()**, **exec()**, **passthru()** et **system()**. Ces commandes sont similaires, mais fournissent des interfaces différentes pour le programme externe que vous utilisez. Chacune de ces commandes génère un processus fils pour exécuter la commande ou le script que vous désignez, et chacune d'elle capture la sortie de votre commande comme c'est fait sur la sortie standard (stdout).

2.1. shell_exec()

La commande **shell_exec()** n'est juste qu'un alias de l'opérateur guillemet oblique (`). Si vous avez fait du shell ou des scripts Perl, vous savez que vous pouvez capturer la sortie d'autres commandes à l'intérieur des opérateurs guillemets obliques. Par exemple, le listing 1 montre comment utiliser les guillemets obliques pour obtenir un compte de mots pour chaque fichier texte (.txt) dans le répertoire courant.

Listing 1. Utilisation de guillemets obliques pour le nombre de mots

```
#! /bin/sh/bin/sh
number_of_words=`wc -w *.txt`
echo $number_of_words

# résultat serait quelque chose comme:
#165 readme.txt 388 results.txt 588 summary.txt #
165 readme.txt 388 results.txt 588 summary.txt
# etc ….
```

Dans votre script PHP, vous pouvez simplement exécuter cette commande à l'intérieur de **shell_exec()** , et obtenir les résultats dont vous avez besoin, en supposant que vous ayez des fichiers texte dans le même répertoire.

Listing 2. Exécution de la même commande avec shell_exec()

```
<?php
$results = shell_exec('wc -w *.txt');
echo $results;
?>
```

Comme vous pouvez le voir dans la figure 1, vous obtenez les mêmes résultats voulus à partir du script shell. C'est parce que **shell_exec()** vous permet d'exécuter un programme externe via le shell, puis renvoie le résultat sous forme d'une chaîne.

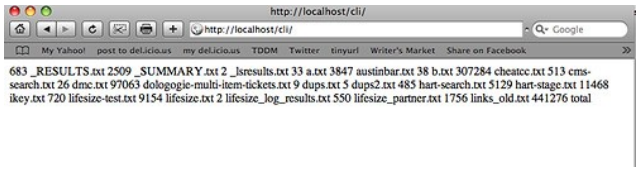


Figure 1. Résultats de l'exécution d'une commande shell par shell_exec()

Notez bien que vous obtiendrez le mêmes résultat si vous utilisez seulement les guillemets obliques, comme montré ci-dessous.

Listing 3. En utilisant uniquement les opérateurs guillemets obliques

```
<?php
$results = `wc -w *.txt`;
echo $results;
?>
```

le listing 4 montre une méthode encore plus simple.

Listing 4. Une méthode plus simple

```
<?php
echo `wc -w *.txt`;
?>
```

Il est important de noter que pratiquement tout ce que vous pouvez faire sur la ligne de commande UNIX, ou dans un script shell est autorisé ici. Par exemple, vous pouvez utiliser des pipes ("|") pour enchaîner les commandes. Vous pouvez même créer un script shell avec toutes vos opérations dans le script et il suffit de composer le script shell, avec ou sans arguments, au besoin.

Par exemple, si vous souhaitez compter seulement les

mots dans les cinq premiers fichiers texte du répertoire, vous pouvez utiliser un pipe ("|") pour raccorder l'ensemble du resultat des commandes wc et head. En outre, vous pouvez inclure la sortie du résultats dans les balises **pre** afin de les afficher dans un navigateur Web, comme illustré ci-dessous.

Listing 5. Une commande shell plus complexe

```
<?php
$results = shell_exec('wc -w *.txt | head -5');
echo "<pre>".$results . "</pre>";
?>
```

La figure 2 montre le résultat de l'exécution du script du listing 5.

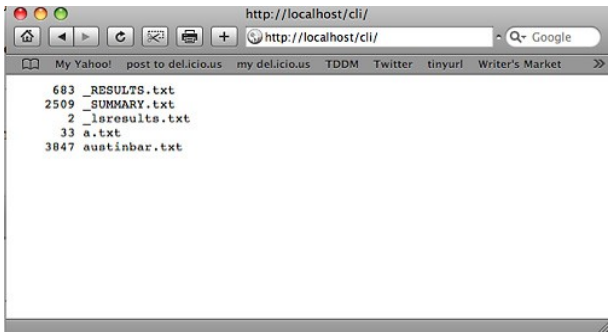


Figure 2. Résultat de l'exécution d'une commande plus complexe fantômes par l'intermédiaire shell_exec()

Plus loin dans cet article, vous apprendrez à passer des arguments à ces scripts en PHP. Pour le moment, vous pouvez considérer cela comme un moyen d'exécuter des commandes shell, tant que vous n'oubliez pas que vous verrez uniquement la sortie standard. S'il y a des erreurs dans votre script ou commande, vous ne verrez pas une erreur standard stderr sauf si vous la redirigez vers stdout.

2.2. passthru()

La commande **passthru()** permet d'exécuter un programme externe et d'afficher le résultat à l'écran. Vous n'avez pas besoin d'utiliser echo ou return pour voir ces résultats, ils s'affichent simplement dans le navigateur. Vous pouvez ajouter un argument optionnel, une variable qui contient le code de retour du programme externe, par exemple, 0 pour le succès, ce qui est une bonne manière pour le débogage.

Dans le Listing 6, j'utilise la commande **passthru()** pour exécuter le petit script *compteur des mots* déjà exécuté dans la section précédente. Comme vous pouvez le voir, j'ai également ajouter une variable $returnval qui contient le code de retour.

Listing 6. Utilisation de la commande passthru() pour exécuter le script 'compteur des mots'

```
<?php
passthru('wc -w *.txt | head -5',$returnval);
echo "<hr/>".$returnval;
?>
```

Remarquez que je n'ai pas affiché la sortie avec un echo. Le résultats finale s'affiche simplement à l'écran, comme illustré ci-dessous.

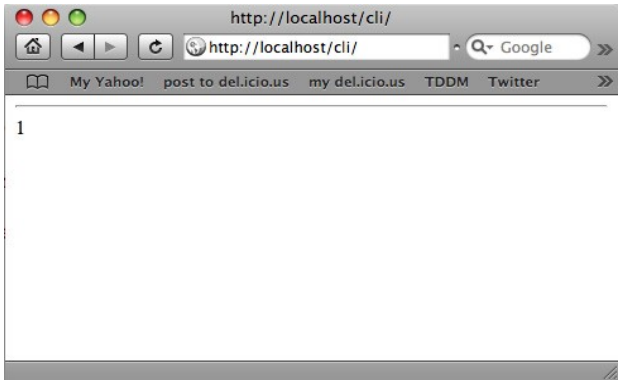
Figure 3. Résultats d'exploitation du passthru() de commande avec un code return

Dans le Listing 7, je vous présente une petite erreur dans le code en supprimant l'undescore avant le 5 dans la commande head du script.

```
<?php
//ci-dessous nous introduisons une erreur en
enlevant le - de la commande head

passthru('wc -w *.txt | head 5',$returnval);
echo "<hr/>".$returnval;
?>
```

Notez que le script ne s'exécute pas comme prévu. Comme le montre la figure 4, vous obtenez un écran blanc, et une valeur de retour de 1. Ce code de retour indique généralement qu'une erreur quelconque s'est produite. Être capable d'essai pour ce code de retour, il est plus facile de comprendre ce qui doit être réparé.



2.3. exec()

La commande **exec()** est similaire à **shell_exec()** sauf qu'elle retourne la dernière ligne de la sortie et éventuellement, renvoie un tableau contentant la sortie complète de la commande et le code d'erreur. Le listing 8 est un exemple de ce qui se passe si vous exécutez exec() sans stocker le résultat dans un tableau de données.

Listing 8. Exécution de la fonction exec() sans stockage du résultat dans un tableau

```
<?php
$resultats = exec('wc -w *.txt | head -5');
echo $resultats;

#n'affichera que la dernière ligne du résultat,
exemple:
#3847 fichier.txt
?>
```

Pour stocker le résultat dans un tableau, ajoutez le nom du tableau comme second argument de **exec().** Je le fais dans le listing 9 en utilisant $data comme nom du tableau.

Listing 9. Stockage du résultat de la commande exec() dans un tableau

```
<?php
$results = exec('wc -w *.txt | head -5',$data);
print_r($data);

#pourrait afficher les données stockées sous la
forme:
#Array ( [0]=> 555 text1.txt [1] => 283
text2.txt)
?>
```

Après avoir stocké le résultat dans un tableau, vous pouvez modifier chaque ligne. Par exemple, vous pouvez diviser l'espace d'abord pour trouver et stocker les valeurs discrètes dans une table de base de données, ou vous pouvez appliquer un formatage spécifique ou des balises à chaque ligne.

2.4. system()

La commande **system()**, illustrée dans le listing 10, est hybride. Comme **passthru()** elle affiche tout ce qu'elle reçoit directement des programmes externes. Comme **exec()** elle retourne également la dernière ligne et rend le code de retour disponible.

Listing 10. La commande system()

```
<?php
system('wc -w *.txt | head -5');

#pourrait afficher:
#123 file1.txt 332 file2.txt 444 file3.txt
#etc...
?>
```

3. Quelques exemples

Maintenant que vous avez appris comment utiliser toutes ces commandes PHP, vous aurez probablement des questions. Par exemple, quelle commande utiliser et quand ? Cela dépend entièrement de vous et surtout de vos besoins.

La plupart du temps, j'utilise la commande **exec()** à cause des données fournies dans un tableau sans aucun traitement. Sinon, j'utilise **shell_exec()** pour des commandes simples, surtout si je ne me soucie pas de la sortie. Si j'ai juste besoin d'exécuter un script shell, j'utilise **passthru()**. Souvent, j'utilise ces fonctions pour différentes raisons et parfois de façon interchangeable. Tout dépend de mon humeur et ce que j'essaie de faire.

Vous vous posez peut être la question : "A quoi sert tout cela ?". Au cas où vous êtes en panne d'idées, un projet annulé, de même si aucune bonne manière d'utiliser les commandes shell ne s'est présentée, je vous propose ici quelques idées.

Si vous écrivez une application qui offre la possibilité de sauvegarder ou de transférer des fichiers, il est plus intelligent d'utiliser **shell_exec()** ou une des autres commandes citées dans cet article pour lancer un script shell rsync. Vous pouvez écrire le script shell qui contient les commandes rsync nécessaires, puis d'utiliser passthru() pour l'exécuter sur la base d'une commande utilisateur ou d'une tâche planifiée.

Par exemple, un utilisateur avec les privilèges appropriés dans votre application, comme admin, peut avoir besoin de transférer 50 fichiers PDF à partir d'un serveur vers un autre. L'utilisateur devrait naviguer vers l'emplacement approprié dans votre application, cliquer sur Transfert, sélectionner les fichiers à transférer, puis cliquer sur Envoyer. Dans son action, le formulaire aurait un script PHP qui exécute votre script rsync via **passthru()** avec une variable optionnelle de retour afin de savoir si un problème est survenu, comme indiqué ci-dessous.

Listing 11. Exemple de script PHP qui exécute un script rsync via passthru()

```
<?php
passthru('xfer_rsync.sh',$returnvalue);

if ($returnvalue != 0){
    //Nous avons un problème
    //ajouter un code sur l'erreur ici
}else{
    //tout est OK
    //aller sur une autre page
}
?>
```

Si vous avez une application qui a besoin de lister les processus ou les fichiers, ou d'avoir certaines données sur ces processus ou ces fichiers, vous pouvez utiliser l'une des commandes décrites dans cet article pour faire cela. Par exemple, une simple commande grep peut vous aider à trouver des fichiers correspondants à certains critères de recherche. L'utilisation de cette commande avec **exec()** et le stockage du résultat dans un tableau pourrait permettre de construire un tableau HTML ou un formulaire pour exécuter d'autres commandes.

Jusqu'à présent, j'ai abordé les évènements générés par les utilisateurs : si l'utilisateur appuie sur un bouton ou clique sur un lien, un script PHP fonctionne. Vous pouvez également obtenir des effets intéressants en exécutant des scripts autonomes PHP avec crontab ou un autre ordonnanceur. Par exemple, si vous avez un script de sauvegarde, vous pouvez l'exécuter indépendamment via un cron ou vous pouvez l'insérer dans un script PHP et puis l'exécuter. Pourquoi procéder de la sorte ? Cela paraît redondant et inutile, non ? Eh bien non pas si vous considérez que vous pouvez exécuter le script de sauvegarde grâce à **exec()** ou **passthru(),** puis faire un traitement en vous basant sur le code de retour. Si vous obtenez une erreur, vous pouvez écrire une entrée dans un journal d'erreur ou une base de données, ou encore envoyer une alerte e-mail. Si le script réussit, vous pouvez vider la sortie brute du script base de données (par exemple, rsync a un mode verbose utile dans le diagnostic des problèmes postérieurs).

4. Sécurité

Quelques mots sur la sécurité : Si vous acceptez que les utilisateurs saisissent et transmettent des informations au shell, le mieux serait de désinfecter ces données. Enlevez

toute commande que vous pensez sensible car pouvant désactiver certaines choses, comme par exemple sudo (exécutér avec les privilèges super-utilisateur) ou rm (supprimer). En fait, le mieux serait d'interdire aux utilisateurs de saisir des commandes librement en ne leur autorisant le choix que dans une liste d'alternatives possibles.

Exemple, si vous exécutez un programme de transfert qui accepte une liste de fichiers comme argument, vous pourriez avoir une liste de cases à cocher pour tous vos fichiers. Les utilisateurs peuvent sélectionner, dé-sélectionner les éléments de la liste et cliquer sur Envoyer (submit) pour exécuter le script shell rsync. Ils ne seront pas autorisés à saisir dans une liste les noms de fichiers ou à utiliser une expression régulière.

Pour assurer un maximum de sécurité, faites passer toutes les commandes saisies par les utilisateurs aux fonctions **escapeshellarg** et **escapeshellcmd** qui permettent d'effacer, de convertir, de transformer les caractères spéciaux, afin d'éviter des attaques du type injection.

5. Autre ressources

- Exemple "CLI" sous windows (Lien15)
- La fonction escapeshellarg (Lien16)
- La fonction escapeshellcmd (Lien17)

Retrouvez l'article de Thomas Myer traduit par Yannick Komotir en ligne : Lien18

Créer une galerie d'images avec :target

Cet article est la traduction de : Making an image gallery with :target (Lien19).

1. Utilisation de la pseudo-classe :target

L'un des nouveaux sélecteurs du CSS3 est la pseudo-classe :target, qui peut être utilisée pour appliquer des règles à un élément ayant un identifiant spécifique, que ce soit une ancre ou un id.

Par exemple, supposons que vous ayez un titre de section avec pour id 'chapitre_2' :

```
<h3 id="chapitre_2">Le titre du chapitre 2</h3>
```

Vous pouvez créer un lien direct vers cet élément en utilisant son identifiant à la fin de l'adresse URL :

```
http://www.exemple.com/index.html#chapitre_2
```

Ensuite, avec le sélecteur :target, vous pouvez appliquer un background à cet élément pour indiquer clairement où vous êtes arrivé sur la page :

```
h3:target { background-color: #ffffff; }
```

Vraiment utile, vous ne trouvez pas ? Ce n'est pas une fonction puissante, mais néanmoins utile. elle peut se rendre encore plus utile avec un peu d'ingéniosité. Que diriez vous, par exemple, d'une galerie d'images 100% CSS ?

2. Création d'une galerie d'images

Regardez cet exemple (Lien20) (avec un navigateur supportant :target ; Mozilla Firefox, Webkit ou Opera feront l'affaire). En cliquant sur les liens, vous naviguez entre les différentes images, tout ceci en un minimum de code et sans JavaScript ni PHP.

La première étape est la création d'une liste d'images et leurs noms, ceux-ci étant des liens. Par exemple :

```
<li id="one">
        <p><a href="#one">Image 1</a></p>
        <img src="../images/one.jpg">
</li>
```

Chaque élément de la liste a besoin d'un *id*, cela permettra l'utilisation d'une ancre, le lien href pointe vers l'id du li qui le contient. Cela permet à :target de fonctionner, c'est magique ! Toutes les images sont positionnées les unes par dessus les autres. En utilisant le sélecteur, nous changeons uniquement la valeur de la propriété z-index, et l'image cible se positionne au-dessus des autres images :

```
img { position: absolute; }
li:target img { z-index: 100; }
```

Facile ! Bien sûr, ceci est un exemple très basique, avec plus d'ingéniosité cela pourrait devenir un outil très utile.

Mise à jour : je viens de me rendre compte que Daniel Glazman a proposé un exemple assez similaire juste avant moi, voir CSS-only tabs (Lien21).

Retrouvez l'article de Peter Gaston traduit par ritter jack en ligne : Lien22

Comment créer facilement un framework JavaScript - Partie 1

Traduction de l'article « How to Easily Create a JavaScript Framework, Part 1 » (Lien23) de Teylor Feliz paru sur AdmixWeb.

1. Introduction

Actuellement, JavaScript est l'un des langages de programmation les plus utilisés et les plus populaires sur internet, car une grande majorité des navigateurs sont compatibles et l'utilisent. JavaScript s'est donc rapidement répandu, car il est simple d'utilisation, précis, et possède un vaste champ d'action. La plupart des programmeurs ont eu l'habitude de croire que JavaScript était un langage futile, mais l'émergence d'AJAX (Lien24) sur le marché a prouvé le contraire, en nous démontrant les capacités et les fonctionnalités de JavaScript. Depuis cette découverte, les programmeurs peuvent créer des applications Web qui ressemblent à des applications de bureau, ce qui est très utile car les données peuvent être modifiées plus rapidement.

Cependant, l'utilisation du DOM (Document Object Model) dans les navigateurs, comme Internet Explorer, rend parfois la tâche difficile. C'est la raison pour laquelle les frameworks JavaScript fleurissent sur le marché, rendant ainsi le scripting multi-navigateurs possible. Au cours de ces cinq dernières années, un engouement pour de nombreux frameworks JavaScript tels que Prototype (Lien25), Jquery (Lien26), YUI (Lien27) et Dojo (Lien28) ont été utilisés ces cinq dernières années par beaucoup de développeurs à travers le monde pour créer d'étonnantes applications Web. Maintenant, je vais vous apprendre pas à pas à créer un framework JavaScript, en utilisant de simples effets DOM et quelques fonctions utiles en AJAX. Nous espérons que vous pourrez utiliser ces informations et apprendre à les mettre en œuvre par vous-même.

2. Eviter les variables globales

Avant de commencer à créer le framework VOZ, vous devez savoir qu'il faut éviter d'utiliser des variables globales dans toutes les pages car JavaScript utilisera la dernière déclaration de la variable avec sa valeur associée sans avertir que cette variable a déjà été déclarée auparavant dans le code.
Par exemple :

```
var x = 23; // Déclaration et utilisation de la variable x avec initialisation de sa valeur à 23
var x = 44; // Affectation de la valeur 44 à la variable sans avertir que la variable a déjà été déclarée
x;  // La valeur actuelle est 44
```

Une bonne méthode pour éviter les variables globales est de créer une fonction anonyme qui restreint la portée de la variable à la fonction.
Par exemple :

```
(function(){

// Les variables et les fonctions se trouvent ici

})();
```

Les fonctions anonymes sont très utiles lorsque nous travaillons avec des frameworks JavaScript, car nous pouvons les utiliser comme paramètre de nos fonctions. Nous verrons cela plus tard.

3. Création de l'objet principal/framework

Il est désormais admis qu'en JavaScript il y a 2 façons de créer des objets : en utilisant un constructeur ou en utilisant un objet littéral.

En utilisant un constructeur :

```
// Définition de l'objet/Classe Utilisateur avec deux paramètres "prenom et nom".
function Utilisateur(prenom, nom){
  this.prenom = prenom;
  this.nom = nom;
}
// Pour utiliser la classe Utilisateur avec new
var monUtilisateur = new Utilisateur('Carl','Anderson');
monUtilisateur.prenom // Cela renvoi "Carl"
```

En utilisant un objet littéral :

La deuxième manière, la plus recommandée par les experts comme Douglas Crockford (Lien29), est d'utiliser un objet littéral. C'est cette dernière que nous allons utiliser pour créer notre framework, parce que c'est la plus élégante et la plus efficace façon de créer des objets.
Par exemple :

```
monUtilisateur = {
  prenom: 'Carl',
  nom: 'Anderson'
};
monUtilisateur.prenom // Renvoi "Carl"
```

Bien, c'est suffisant pour la théorie, passons à l'action et créons notre framework. Tout d'abord, choisissez un nom court et attractif pour votre framework. Dans cet exemple, nous utiliserons le nom "VOZ".
Par exemple :

```
// On protège la fonction principale du framework pour éviter des erreurs imprévisibles avec d'autres frameworks ou variables.
(function (){
// Création du framework VOZ
var VOZ = {}
})();
```

Maintenant notre framework est prêt mais nous devons encore créer un certain nombre de propriétés et méthodes. Pour l'instant, nous allons ajouter les méthodes "getById", "addClass" et "on".

```
(function(){

var VOZ = {
        // Le tableau elems va contenir tous les éléments les uns derrière les autres dans l'ordre
        elems :[],
        // Méthode pour récupérer les éléments par Id
        getById:function(){},
        // Méthode pour ajouter une classe CSS
        addClass:function(){},
        // Méthode pour ajouter un événement sur nos éléments
        on:function(){},
        // Ajoute du texte dans un élément
        appendText:function(){}
        // Affiche/Masque
        toggleHide:function(){}
}
})() ;
```

4. La navigation DOM partie 1

Savoir utiliser DOM (Document Object Model) est vital pour un programmeur JavaScript, car cela donne la capacité de modifier, supprimer ou ajouter de nouveaux éléments à des emplacements spécifiques sur la page Web.

Pour l'instant nous allons voir la méthode "getById;" cependant, ce n'est pas la méthode "document.getElementById" normale, car celle ci prend plus d'un paramètre pour trouver les éléments par leur ID.
Par exemple :

```
// Récupère tous les éléments par ID
// Peut prendre plus d'un paramètre
getById:function(){
    var tempElems = []; // tableau temporaire pour sauvegarder les éléments trouvés
    for(var i = 0; i<arguments.length; i++){
        if(typeof arguments[i] === 'string'){ // Vérifie que le paramètre est une chaîne
                tempElems.push(document.getElementById(arguments[i])); // Ajoute l'élement à tempElems
        }
    }
    this.elems = tempElems; // Tous les éléments sont copiés dans la propriété elems
    return this; // Renvoie this dans l'ordre d'appel
},
```

Nous avons ainsi une méthode pour trouver des éléments par ID, créons à présent la méthode suivante qui ajoute un nom de classe à nos éléments, appelée "addClass" :

```
// Ajoute une nouvelle classe à un élément
// Cela ne supprime pas les autres classes, elle en ajoute simplement une nouvelle
addClass:function(name){
    for(var i = 0; i<this.elems.length; i++){
        this.elems[i].className += ' ' + name; // C'est ici qu'on ajoute la nouvelle classe
    }
    return this; // Renvoie this dans l'ordre d'appel
},
```

La dernière méthode que nous verrons dans cette partie 1 est la fonction "on" qui est utilisée pour ajouter des événements à nos éléments, comme par exemple 'click', 'mouseover', 'mouseout', etc.

```
// Ajoute un événement aux éléments trouvés par les méthodes : getById et getByClass
//-- Action est un type d'événement comme 'click', 'mouseover', 'mouseout', etc
//-- Callback est la fonction à exécuter lorsque l'événement est déclenché
on: function(action, callback){
    if(this.elems[0].addEventListener){
        for(var i = 0; i<this.elems.length; i++){
            this.elems[i].addEventListener(action,callback,false);//Ajout de l'événement du W3C pour Firefox,Safari,Opera...
        }
    }
    else if(this.elems[0].attachEvent){
        for(var i = 0; i<this.elems.length; i++){
            this.elems[i].attachEvent('on'+action,callback);// Ajout de l'événement pour Internet Explorer :(
        }
    }
    return this; // Renvoie this dans l'ordre d'appel
},
```

Pour rattacher du texte à nos éléments, nous pouvons utiliser cette méthode :

```
// Ajout du texte sur les éléments
// text est la chaîne à insérer
appendText:function(text){
    text = document.createTextNode(text); // Crée un nouveau noeud texte avec la chaîne fournie
    for(var i = 0; i<this.elems.length; i++){
       this.elems[i].appendChild(text); // Ajoute le texte à l'élément
    }
    return this; // Renvoie this dans l'ordre d'appel
},
```

Pour afficher/masquer les éléments trouvés, nous utiliserons cette méthode :

```
// Affiche ou masque les éléments trouvés
toggleHide:function(){
    for(var i = 0; i<this.elems.length; i++){
        this.elems[i].style['display'] =
(this.elems[i].style['display']==='none' ||
'') ?'block':'none';
                // Vérifie le statut de l'élément
pour savoir s'il peut être affiché ou masqué
    }
    return this; // Renvoie this dans l'ordre
d'appel
}
```

Finalement, ajoutons cela à la fin du framework :

```
if(!window.$$){window.$$=VOZ;} //Nous créons un
raccourci pour notre framework, nous pouvons
appeler les méthodes par $$.method ();
```

Ceci permet d'utiliser notre framework avec $$ comme raccourci.
Par exemple :

```
$$.getById('myElement')
```

Maintenant notre framework ressemble à ça :

```
(function(){
var VOZ = {
    elems:[],// Tableau pour sauvegarder tous les
éléments trouvés par les fonctions getById,
getByClass
    // Récupère tous les éléments par ID
    // Peut prendre plus d'un paramètre
    getById:function(){
        var tempElems = []; // tableau temporaire
pour sauvegarder les éléments trouvés
        for(var i = 0; i<arguments.length; i++){
            if(typeof arguments[i] === 'string'){
// Vérifie que le paramètre est une chaîne
                tempElems.push(document.getElemen
tById(arguments[i])); // Ajoute l'élement à
tempElems
            }
        }
        this.elems = tempElems; // Tous les
éléments sont copiés dans la propriété elems
        return this; // Renvoie this dans l'ordre
d'appel
    },



    // Ajoute une nouvelle classe à un élément
    // Cela ne supprime pas les autres classes,
elle en ajoute simplement une nouvelle
    addClass:function(name){
        for(var i = 0;i<this.elems.length;i++){
            this.elems[i].className += ' ' +
name; // C'est ici qu'on ajoute la nouvelle
classe
        }
        return this; // Renvoie this dans l'ordre
d'appel
    },



    // Ajoute un événement aux éléments trouvés
par la méthodes : getById et getByClass
    //-- Action est un type d'événement comme
'click', 'mouseover', 'mouseout', etc
    //-- Callback est la fonction à exécuter
lorsque l'événement est déclenché
    on: function(action, callback){
        if(this.elems[0].addEventListener){
            for(var i = 0;i<this.elems.length;i+
+){
                this.elems[i].addEventListener(ac
tion,callback,false);//Ajout de l'événement du
W3C pour Firefox,Safari,Opera...
            }
        }
        else if(this.elems[0].attachEvent){
            for(var i = 0;i<this.elems.length;i+
+){
                this.elems[i].attachEvent('on'+ac
tion,callback); // Ajout de l'événement pour
Internet Explorer :(
            }
        }
        return this; // Renvoie this dans l'ordre
d'appel
    },

    // Ajout du texte sur les éléments
    // text est la chaîne à insérer
    appendText:function(text){
        text = document.createTextNode(text); //
Crée un nouveau noeud texte avec la chaîne
fournie
        for(var i = 0;i<this.elems.length;i++){
            this.elems[i].appendChild(text); //
Ajoute le texte à l'élément
        }
        return this; // Renvoie this dans l'ordre
d'appel
    },

    // Affiche ou masque les éléments trouvés
    toggleHide:function(){
        for(var i = 0;i<this.elems.length;i++){
            this.elems[i].style['display'] =
(this.elems[i].style['display']==='none' ||
'') ?'block':'none';
                        // Vérifie le statut de
l'élément pour savoir si il peut être affiché ou
masqué
        }
        return this; // Renvoie this dans l'ordre
d'appel
    }
}
if(!window.$$){window.$$=VOZ;} //Nous créons un
raccourci pour notre framework, nous pouvons
appeler les méthodes par $$.method ();
})();
```

5. Méthode de Chaînage

Le chaînage est la capacité d'appeler une méthode à partir du résultat retourné par la précédente. Cela rend le code plus compréhensible et mieux organisé.

Sans Chaînage

```
$('#element').method1();
$('#element').method2();
$('#element').method3();
```

Avec chaînage

```
$('#element').method1().method2().method3();
```

Ceci est rendu possible parce que nos méthodes retournent systématiquement l'objet (this) sur lequel nous travaillons.

6. Conclusion

Exemple du framework JavaScript VOZ (Lien30)

Visitez le lien ci-dessus pour voir la première partie de notre framework VOZ en action ! Aussi, attendez la deuxième partie de cette série d'articles, qui devrait arriver bientôt ! J'espère que vous avez aimé lire ce tutoriel et que vous l'avez trouvé facile à suivre ! N'hésitez pas à laisser vos commentaires, car j'apprécie d'avoir les impressions des autres développeurs !

Retrouvez l'article de Teylor Feliz traduit par KalyParker en ligne : Lien31

Comment créer facilement un framework Javascript - Partie 2

Traduction de l'article How to Easily Create a JavaScript Framework, Part 2 (Lien32) de Teylor Feliz paru sur AdmixWeb.

1. Introduction

La semaine dernière, j'ai parlé de l'importance du langage de programmation JavaScript, et comment il s'est popularisé parmi les programmeurs à cause de ses diverses fonctionnalités. Comme je l'ai précédemment mentionné, à l'heure actuelle, les programmeurs peuvent créer beaucoup plus facilement des applications Web avec JavaScript, c'est la raison pour laquelle on continue à l'apprendre et à l'utiliser. La semaine dernière, j'ai également expliqué étape par étape la création de notre propre framework JavaScript (Lien33), que j'ai appelé "VOZ". Dans le tutoriel de cette semaine, je vais compléter ce dernier à partir du code de la semaine dernière. Je vais principalement l'étoffer un peu, en ajoutant des méthodes utiles et d'autres méthodes supplémentaires un peu plus fun. Lisez la suite pour suivre l'évolution du framework ! N'hésitez pas à laisser vos commentaires, et amusez-vous !

2. La navigation DOM partie 2

Dans la partie précédente de la série, nous avons présenté la méthode "$$.getById()", aujourd'hui nous allons créer d'autres méthodes similaires pour utiliser DOM plus facilement. Ces techniques simples d'utilisation s'appelleront "getByName", "getByTag" et "getByClass".

Méthode "getByName"
La méthode "getByName" peut prendre plus d'un paramètre et permet d'obtenir tous les éléments ayant un nom spécifique.

```
// Récupère les élements par nom
// Peut prendre plus d'un paramètre
getByName:function(){
    var tempElems = [];// tableau temporaire pour sauvegarder les élements trouvés
            for(var i = 0;i<arguments.length;i++){
                if(typeof arguments[i] === 'string'){// Vérifie que le paramètre est une chaîne
                    var e = document.getElementsByName(arguments[i]);
                    for(var j=0;j<e.length;j++){
                        tempElems.push(e[j]);
                    }
                }
            }
            this.elems = tempElems; // Tous les élements sont copiés dans la propriété elems
            return this;// Renvoie this dans l'ordre d'appel
},
```

Méthode "getByTag"
La méthode "getByTag" a la capacité de trouver tous les éléments ayant un nom de balise spécifique.

```
// Récupère les élements par nom de balise
// Peut prendre plus d'un paramètre
getByTag:function(){
            var tempElems = []; // tableau temporaire pour sauvegarder les élements trouvés
            for(var i = 0;i<arguments.length;i++){
                if(typeof arguments[i] === 'string'){// Vérifie que le paramètre est une chaîne
                    var e = document.getElementsByTagName(arguments[i]);
                    // Recherche des éléments ayant le nom de la balise et sauvegarde dans le tableau e
                    for(var j=0;j<e.length;j++){
                        tempElems.push(e[j]); // Ajoute l'élément à tempElems
                    }
                }
            }
            this.elems = tempElems; //  Tous les éléments sont copiés dans la propriété elems
            return this;// Renvoie this dans l'ordre d'appel
        },
```

Méthode "getByClass"
La méthode "getByClass" retourne tous les éléments ayant le nom de classe spécifié. La recherche peut être filtrée dans la page entière ou sur un conteneur parent et sur un type de balise défini pas l'utilisateur.

```
// Récupère les éléments par Nom de classe
// name est le nom de la classe
// type est le nom de balise recherché
// parent est le parent du groupe spécifique
getByClass:function(name,type,parent){
  var tempElems = [];// tableau temporaire pour
```

```
sauvegarder les éléments trouvés
  var pattern = new RegExp("(^| )" + name + "( |
$)"); // Expression régulière pour savoir si
l'élément a une classe
  // Recherche des éléments par nom de balise
avec la possibilité de choisir le parent, et
sélection d'une balise
  // en particulier ou de la totalité des balises
  var e = (parent ||
document).getElementsByTagName(type || '*')
  for(var i=0;i<e.length;i++){
      if(pattern.test(e[i].className)){// Si
l'élément a un nom de classe alors...
              tempElems.push(e[i]); // Ajout de
l'élément à tempElems
          }
  }
  this.elems = tempElems; //  Tous les éléments
sont copiés dans la propriété elems
  return this;// Renvoie this dans l'ordre
d'appel
},
```

Méthode "checked"
Cette méthode coche ou décoche simplement les boutons radio ou les cases à cocher trouvés.

```
// Coche ou décoche un bouton radio ou une case à
cocher
// bol peut être true ou false;
checked:function(bol){
    for(var i=0;i<this.elems.length;i++){
        // Vérifie si l'élément est un input de
type case à cocher ou radio bouton car c'est le
même objet dans les 2 cas
        if(this.elems[i].nodeName.toLowerCase()==
='input' && (this.elems[i].getAttribute('type')
== 'checkbox'
                        ||
this.elems[i].getAttribute('type') == 'radio')){
              this.elems[i].checked = bol; //
Affecte vrai ou faux
        }
    }
    return this;
},
```

3. Méthodes supplémentaires pour le fun

Méthode "even et odd"
La méthode "even" correspond aux éléments d'un tableau dont l'indice est pair. Par exemple 0, 2, 4, 6, 8, 10...
La méthode "odd" correspond aux éléments d'un tableau dont l'indice est impair. Par exemple 1, 3, 5, 7, 9...

```
// Récupère les éléments d'un tableau dont
l'indice est pair. Exemple 0, 2, 4, 6, 8, 10...
// Méthode assez sympa pour colorer les lignes
d'une table
even:function(){
            var tempElems = [];
            for(var
```

```
i=0;i<this.elems.length;i+=2){
                tempElems.push(this.elems[i]);
            }
            this.elems = tempElems;
            return this;
        },
// Récupère les éléments d'un tableau dont
l'indice est impair. Exemple 1, 3, 5, 7, 9...
// Méthode assez sympa pour colorer les lignes
d'une table
odd:function(){
            var tempElems = [];
            for(var i=1;i<this.elems.length;i+=2)
{
                tempElems.push(this.elems[i]);
            }
            this.elems = tempElems;
            return this;
        },
```

Méthode "setOpacity"
Les méthodes "setOpacity" affectent en pourcentage ou en valeur l'opacité voulue sur un élément.

```
// Fonction assez sympa pour opacifier des
éléments
// level est le niveau d'opacité souhaité
setOpacity: function(level){
                for(var i =
0;i<this.elems.length;i++){
                    if(this.elems[0].filters)
{ // Si le navigateur est ie, alors on définit
l'opacité ainsi
                        this.elems[i].style.
filter='alpha(opacity='+level+')';
                    }
                    else{ // et pour tous
les autres navigateurs ...
                        this.elems[i].style
.opacity=level/100;
                    }
                }
                return this;// Renvoie this
dans l'ordre d'appel
            },
```

4. Conclusion

Exemple du framework JavaScript VOZ (Lien34)

Visitez le lien ci-dessus pour voir la deuxième partie de notre framework VOZ en action ! Aussi, attendez la troisième partie de cette série sur ce framework JavaScript, qui est en cours de rédaction. J'espère que vous avez trouvé ce tutoriel amusant et facile à suivre ! N'hésitez pas à laisser vos commentaires, car comme toujours j'apprécie de recevoir les impressions d'autres amoureux du Web !

Retrouvez l'article de Teylor Feliz traduit par KalyParker en ligne : Lien35

La gestion des cookies en JavaScript

Cet article est la traduction de « Cookies » (Lien36).
Dans cet article, je vais vous présenter trois fonctions permettant d'enregistrer, lire et supprimer des cookies. Ces fonctions vous permettront d'utiliser facilement les cookies sur votre site.
Tout d'abord, je vous présenterai rapidement ce qu'est un cookie ainsi que l'objet JavaScript document.cookie, suivi d'un exemple. Ensuite, je vous proposerai ces trois fonctions en expliquant leur fonctionnement.
Le script, écrit par Scott Andrew, a été copié et modifié avec son autorisation.

1. Les cookies

Les cookies ont été inventés par Netscape afin de donner une "mémoire" aux serveurs et navigateurs Web. Le protocole HTTP (Lien37), qui gère le transfert des pages Web vers le navigateur ainsi que les demandes de pages du navigateur vers le serveur, est dit *state-less* (sans état) : cela signifie qu'une fois la page envoyée vers le navigateur, le serveur n'a aucun moyen d'en garder une trace. Vous pourrez donc venir deux, trois, cent fois sur la page, le serveur considérera toujours qu'il s'agit de votre première visite.

Cela peut être gênant à plusieurs titres : le serveur ne peut pas se souvenir si vous vous êtes authentifié sur une page protégée, n'est pas capable de conserver vos préférences utilisateur, etc. En résumé, il ne peut se souvenir de rien ! De plus, lorsque la personnalisation a été créée, cela est vite devenu un problème majeur.

Les cookies ont été inventés pour remédier à ces problèmes. Il existe d'autres solutions pour les contourner, mais les cookies sont très simples à maintenir et très souples d'emploi.

2. Fonctionnement des cookies

Un cookie n'est rien d'autre qu'un petit fichier texte stocké par le navigateur. Il contient certaines données :

1. Une paire nom / valeur contenant les informations.
2. Une date d'expiration au-delà de laquelle il n'est plus valide.
3. Un domaine et une arborescence qui indiquent sur quel répertoire d'un serveur donné il y aura accès.

Dès que vous demandez une page Web à laquelle le cookie peut être envoyé, celui-ci est ajouté dans l'en-tête HTTP. Les programmes côté serveur peuvent alors le lire et décider, par exemple, si vous avez le droit de voir la page ou si vous voulez que les liens soient jaunes avec un fond vert.

Ainsi, chaque fois que vous visitez la page d'où vient le cookie, les informations vous concernant sont disponibles. C'est parfois bien pratique, mais cela peut aussi nuire à votre vie privée. Heureusement, la plupart des navigateurs vous permettent de gérer les cookies (vous pouvez donc supprimer ceux provenant des sites publicitaires, par exemple).

Les cookies peuvent aussi être lus par JavaScript. Ils sont principalement destinés à conserver vos préférences.

2.1. Paire nom / valeur

Chaque cookie possède un couple nom / valeur qui contient les données actuelles. Le nom du cookie est destiné à vous aider, il vous suffira de chercher ce nom lorsque vous lirez les cookies de la page.

Si vous souhaitez lire le contenu d'un cookie, commencez par chercher son nom, puis la valeur qui lui est associée. Bien sûr, c'est à vous de décider quelle(s) valeur(s) peut avoir un cookie et d'écrire des scripts pour traiter cette (ces) valeur(s).

2.2. Date d'expiration

Chaque cookie doit avoir une date d'expiration au-delà de laquelle il ne sera plus valide et supprimé. Si vous ne renseignez pas cette date, le cookie sera supprimé à la fermeture du navigateur. Cette date est exprimée au format UTC (Greenwich).

2.3. Domaine et arborescence

Chaque cookie possède également un domaine et une arborescence. Le domaine indique au navigateur à quel domaine (quel site) le cookie doit être envoyé. Si vous ne l'indiquez pas, le domaine correspondra à celui qui a créé le cookie : dans cette page, par exemple, *ppk.developpez.com*.
Notez que l'utilité du domaine est d'autoriser l'envoi du cookie aux sous-domaines associés. Mon cookie ne pourra pas être lu depuis *www.developpez.com* puisque par défaut, son domaine est *ppk.developpez.com*. Si je précise le domaine *developpez.com*, alors *www.developpez.com* pourra lui aussi lire le cookie.
Je ne peux pas affecter un domaine sur lequel je ne me trouve pas : *quirksmode.org* par exemple. Dans notre cas, seul *developpez.com* est autorisé.

L'arborescence vous permet d'indiquer un répertoire sur lequel le cookie sera actif. Par exemple, si vous voulez que le cookie ne soit envoyé que vers le répertoire *tutoriels*, spécifiez /tutoriels comme arborescence. Habituellement, l'arborescence est /, ce qui signifie que le cookie est valide sur l'intégralité du domaine.
C'est comme cela que fonctionnent les scripts proposés. Les cookies créés sur cette page seront envoyés sur toutes les pages de *www.developpez.com* (bien que seule cette page contienne un script permettant de les récupérer et de les utiliser).

3. document.cookie

Les cookies peuvent être créés, lus et supprimés par

JavaScript. Ils sont accessibles à travers la propriété document.cookie. Vous pouvez traiter cette propriété comme une chaîne (bien que cela n'en soit pas réellement une). Vous n'avez accès qu'au couple nom / valeur.

Pour créer un cookie sur ce domaine avec une paire nom / valeur valant 'ppkcookie1=testcookie' et expirant le 28 février 2010, il faut faire :

```
document.cookie = 'ppkcookie1=testcookie;
expires=Sun, 28 Feb 2010 00:00:00 UTC; path=/'
```

1. D'abord, la paire nom / valeur ('ppkcookie1=testcookie').
2. Puis un point-virgule et un espace.
3. La date d'expiration dans un format correct (expires=Sun, 28 Feb 2010 00:00:00 UTC).
4. De nouveau un point-virgule et un espace.
5. Le domaine (path=/).

La syntaxe est très stricte, ne la changez pas (bien sûr, les scripts vous permettront de gérer ces valeurs) !

En revanche, même si la syntaxe donne l'impression d'écrire la chaîne dans document.cookie, dès que vous essaierez de la lire, seul le couple nom / valeur sera accessible :

```
ppkcookie1=testcookie
```

Il est possible de créer un autre cookie avec :

```
document.cookie = 'ppkcookie2=un autre test;
expires=Mon, 1 Mar 2010 00:00:00 UTC; path=/'
```

Le premier cookie n'est pas écrasé, ce qui serait le cas si document.cookie était une véritable chaîne. En réalité, le second cookie est ajouté à document.cookie. Si on le lit une nouvelle fois, on obtient donc :

```
ppkcookie1=testcookie; ppkcookie2=un autre test
```

Cependant, si l'on crée à nouveau un cookie portant le même nom que le précédent :

```
document.cookie = 'ppkcookie2=encore un autre
test; expires=Mon, 1 Mar 2010 00:00:00 UTC;
path=/'
```

cette fois-ci, le précédent cookie est bien écrasé, document.cookie renvoie désormais :

```
ppkcookie1=testcookie; ppkcookie2=encore un autre
test
```

Pour lire un cookie, vous devrez récupérer la valeur de document.cookie et la traiter comme s'il s'agissait d'une chaîne en recherchant des caractères, le point-virgule puis le nom du cookie par exemple. Je vous montrerai comment procéder dans la suite de cet article.

Enfin, pour supprimer un cookie, il suffit de fixer sa date d'expiration à une date passée. Comme cela, le navigateur détecte qu'il n'est plus valide et le supprime :

```
document.cookie = 'ppkcookie2=encore un autre
test; expires=Fri, 01 Jan 2010 00:0:00 UTC;
path=/'
```

4. Exemple

Si ces explications vous semblent encore un peu confuses, allez à la page d'exemple. Vous pourrez y gérer deux cookies, ppkcookie1 et ppkcookie2. Saisissez leur valeur dans la zone de texte.

Accéder à la page d'exemple (Lien38).

Les cookies de l'exemple sont fixés pour être valables une semaine. Si vous revisitez cette page avant ce délai, une alerte vous avertissant que le(s) cookie(s) est (sont) toujours actif(s). Essayez de retourner sur la page d'exemple.

5. Les scripts

Voici le code des scripts dont vous aurez besoin :

```
function createCookie(name,value,days) {
        if (days) {
                var date = new Date();
                date.setTime(date.getTime()+
(days*24*60*60*1000));
                var expires = ";
expires="+date.toGMTString();
        }
        else var expires = "";
        document.cookie =
name+"="+value+expires+"; path=/";
}

function readCookie(name) {
        var nameEQ = name + "=";
        var ca = document.cookie.split(';');
        for(var i=0;i < ca.length;i++) {
                var c = ca[i];
                while (c.charAt(0)==' ') c =
c.substring(1,c.length);
                if (c.indexOf(nameEQ) == 0)
return c.substring(nameEQ.length,c.length);
        }
        return null;
}

function eraseCookie(name) {
        createCookie(name,"",-1);
}
```

6.Explication du code

Les fonctions ne sont pas très compliquées. La principale difficulté réside dans l'écriture de la bonne syntaxe de création du cookie.

6.1. createCookie

Lorsque vous appelez createCookie(), vous devez lui passer trois paramètres : le nom et la valeur du cookie ainsi que le nombre de jours pendant lequel il restera actif. Voici un exemple pour lequel ppkcookie=testcookie sera actif 7 jours :

```
createCookie('ppkcookie','testcookie',7)
```

Si le nombre de jours vaut 0, le cookie sera effacé à la fermeture du navigateur. Si vous mettez un nombre négatif de jours, le cookie sera effacé immédiatement.

La fonction reçoit donc ses paramètres et commence son exécution :

```
function createCookie(name,value,days) {
```

Tout d'abord, elle vérifie que le paramètre *days* est présent. Si ce n'est pas le cas, nous n'avons pas besoin de faire de calculs sur la date :

```
if(days) {
```

Si le paramètre existe, nous créons un nouvel objet Date() contenant la date du jour :

```
var date = new Date();
```

Puis nous récupérons le temps actuel (en millisecondes) auquel on ajoute le nombre de jours (millisecondes également). Nous affectons alors à la propriété time de la variable date la valeur calculée. date a maintenant pour valeur la date en millisecondes à laquelle le cookie doit expirer :

```
date.setTime(date.getTime()+
(days*24*60*60*1000));
```

Ensuite, nous affectons cette date au format UTC/GMT à la variable expires :

```
var expires = "; expires="+date.toGMTString();
```

Si 0 est passé comme paramètre, expires n'est pas définie, ce qui provoque l'effacement du cookie à la fermeture du navigateur :

```
else var expires = "";
```

Enfin, nous écrivons le nouveau cookie dans document.cookie avec la syntaxe adéquate :

```
document.cookie = name+"="+value+expires+";
path=/";
```

Le cookie est désormais créé.

6.2. readCookie

Pour lire un cookie, nous appelons readCookie() à laquelle nous passons en paramètre le nom du cookie. Affectez le résultat de la fonction à une variable et testez si la variable a une valeur (si le cookie n'existe pas, la fonction renvoie null, ce qui peut perturber votre propre fonction), puis effectuez vos traitements :

```
var x = readCookie('ppkcookie1')
if (x) {
        // Traitement de la valeur du cookie
}
```

La fonction reçoit donc son paramètre et commence son exécution :

```
function readCookie(name) {
```

Nous allons commencer par chercher le nom du cookie suivi du signe "=". Stockons cette chaîne dans la variable nameEQ :

```
var nameEQ = name + "=";
```

Ensuite, effectuons un split(';') sur document.cookie : ainsi la variable ca contient un tableau de tous les cookies de ce domaine et de cette arborescence :

```
var ca = document.cookie.split(';');
```

Nous parcourons alors ce tableau (nous bouclons sur tous les cookies trouvés) :

```
for(var i=0;i < ca.length;i++) {
```

Appelons c le cookie en cours :

```
var c = ca[i];
```

Si le premier caractère est un espace, nous le retirons à l'aide de la méthode substring(). Nous réitérons cette étape tant que le premier caractère est un espace :

```
while (c.charAt(0)==' ') c =
c.substring(1,c.length);
```

Maintenant, c commence par le nom du cookie courant. Si c'est le nom que nous cherchons :

```
if (c.indexOf(nameEQ) == 0)
```

alors nous avons trouvé notre cookie. Nous voulons retourner la valeur du cookie, qui correspond à la partie de c qui vient juste après nameEQ. Nous retournons donc cette valeur et sortons de la fonction : sa mission est terminée.

```
if (c.indexOf(nameEQ) == 0) return
c.substring(nameEQ.length,c.length);
```

Si, après avoir passé tous les cookies en revue, nous n'avons pas trouvé le nom recherché, cela signifie que le cookie n'existe pas. Nous retournons alors null :

```
return null;
```

6.3. eraseCookie

Effacer un cookie est très simple.

```
eraseCookie('ppkcookie')
```

Il suffit de passer à la fonction le nom du cookie à effacer :

```
function eraseCookie(name) {
```

puis appelons la fonction createCookie() pour affecter au cookie une date d'expiration passée correspondant à la veille :

```
createCookie(name,"",-1);
```

Le navigateur se rend compte que le cookie n'est plus valide et l'efface immédiatement.

Retrouvez l'article de Peter-Paul Koch traduit par Didier Mouronval en ligne : Lien39

L'utilisation des composants ActiveX sans inscription préalable - (1ère partie)

Certains composants ActiveX permettent d'étendre considérablement les capacités des langages VBScript/JScript mais au prix d'un enregistrement préalable dans la base de registre. Il existe néanmoins une solution qui lève cette contrainte.

1. Introduction

VBScript comme JScript possèdent nativement la capacité de créer - ou de récupérer - des objets exposés par des composants ActiveX munis d'une interface IDispatch respectant la norme COM/OLE définie par Microsoft.

Cette norme impose notamment l'inscription préalable du composant, c'est-à-dire la déclaration de toutes ses interfaces avec leurs caractéristiques, dans la base de registre. Le client COM, en l'occurence le script, retrouve en consultant cette dernière les points d'entrée des différentes classes exposées par ce composant. Toute tentative d'instancier un objet à partir d'un composant non enregistré se traduit par un message d'erreur bien connu (VBS) : "*Un composant ActiveX ne peut pas créer un objet : ...*"

Si cette inscription s'effectue simplement au moyen de l'utilitaire **regsvr32.exe**, on peut quelques fois souhaiter s'affranchir de cette contrainte, tout particulièrement lorsqu'on ne dispose pas des droits d'administrateur. Cet article se propose de décrire une solution applicable non seulement aux langages VBScript et JScript exécutés dans le contexte Windows Script Host, mais aussi à tous les langages Active Scripting dès lors qu'ils ont la capacité de gérer les objets COM/OLE. L'exemple choisi pour illustrer cette première partie concerne un composant russe très simple - DynamicWrapperX - sans ressource typelib, qui n'expose qu'une seule classe et dont j'ai traduit par ailleurs la documentation. Ce wrapper permet aux VBScript/JScript d'appeler directement les fonctions des bibliothèques dll.

2. Les solutions identifiées

Classiquement, deux techniques existent pour contourner cette limitation :

- l'émulation de la fonction CoCreateInstance

Elle consiste à appeler directement la fonction DllGetClassObject présente dans chaque composant. Malheureusement, elle ne nous dispense pas de l'obligation d'inscrire le composant qui aura pour tâche d'assurer l'émulation qui vise justement à nous éviter d'y procéder...
C'est donc une impasse qui renvoie, comme chacun l'aura compris, à l'angoissante problématique de l'incomplétude ontologique. Une piste à explorer consisterait à la combiner avec la deuxième technique (*cf infra*), mais c'est un autre sujet.

- L'utilisation de la technologie SxS

A partir de la version XP SP2 (et serveur 2K3), Microsoft a introduit un nouveau mécanisme permettant à une application de gérer des composants COM sans avoir besoin de les enregistrer. A l'initialisation, chaque nouveau process recherche désormais la présence, dans le répertoire de l'exécutable, d'un fichier *application manifest* contenant les métadonnées qui vont lui permettre de créer un contexte d'activation. Ce contexte fournira au client les mêmes éléments que ceux obtenus de la base de registre. Ce fichier, qui porte l'extension *.manifest*, est un simple fichier texte au format xml. Cette solution fait l'objet du présent article.

3. Mise en oeuvre

En premier lieu, un fichier application manifest doit être créé et placé dans le même répertoire que l'exécutable. Comme le script s'exécute dans le contexte Windows Script Host, l'exécutable concerné est **wscript.exe** normalement situé dans le répertoire system32. Ce fichier sera donc logiquement nommé :

wscript.exe.manifest

L'*assembly*, c'est-à-dire l'ensemble des modules formant un tout unique et indivisible, est constitué du ou des fichiers du composant et de son *assembly manifest*. Comme il s'agira d'un *private assembly*, il doit être placé dans le répertoire de l'exécutable. Pour être tout à fait exact, il peut également être placé dans un sous-répertoire selon les modalités décrites dans le chapitre "*Assembly Searching Sequence*" de la documentation en ligne fournie par Microsoft [1]. Le composant DynamicWrapperX se présente sous la forme d'un petit fichier dénommé dynwrapx.dll ainsi notre *assembly* sera constitué des 2 fichiers suivants :

dynwrapx.dll
dynwrapx.sxs.manifest

Nous possédons les binaires, il ne reste plus qu'à écrire les *manifests*.

3.1. Application manifest - wscript.exe.manifest

wscript.exe.manifest

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-
com:asm.v1" manifestVersion="1.0">
```

```
  <assemblyIdentity
      type="win32"
      name="wscript.exe"
      version="1.0.0.0"/>

  <dependency>
    <dependentAssembly>
      <assemblyIdentity
          type="win32"
          name="dynwrapx.sxs"
          version="1.0.0.0"/>
    </dependentAssembly>
  </dependency>

  <dependency>
    <dependentAssembly>
      <assemblyIdentity
          type="win32"
          name="Microsoft.Windows.Common-
Controls"
          version="6.0.0.0"
          publicKeyToken="6595b64144ccf1df"
          language="*"
          processorArchitecture="x86"/>
    </dependentAssembly>
  </dependency>

</assembly>
```

assemblyIdentity comporte les trois attributs minimums requis : *type*, *name* et *version* qui permettent d'identifier **wscript.exe**.
dependency et *dependentAssembly* définissent les composants liés à l'exécutable. Là encore, les trois attributs minimums ont été utilisés pour lier le composant DynamicWrapperX, ou plus exactement son *assembly manifest* que nous verrons ci-dessous.

La deuxième dépendance concerne la bibliothèque des contrôles XP qui permet d'appliquer le style XP à tous les contrôles visuels (les fonctions VBS *MsgBox* ou *InputBox* permettent un test simple).
Il s'agit d'un *shared assembly* que l'on retrouve dans le répertoire %windows%\WinSxS. Les plus curieux constateront que l'attribut version est inopérant et que la bibliothèque effectivement chargée sera normalement celle la plus récente, conformément aux redirections définies dans le fichier policy (répertoire %windows %\WinSxS\Policies).

3.2. Assembly manifest - dynwrapx.sxs.manifest

dynwrapx.sxs.manifest

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-
com:asm.v1" manifestVersion="1.0">

  <assemblyIdentity
      type="win32"
      name="dynwrapx.sxs"
      version="1.0.0.0"/>

  <file name="dynwrapx.dll">
    <comClass
        description="DynamicWrapperX Class"
        clsid="{89565275-A714-4a43-912E-
978B935EDCCC}"
        threadingModel="Both"
        progid="DynamicWrapperX"/>
  </file>

</assembly>
```

Concernant l'élément *assemblyIdentity*, mêmes observations que pour l'*application manifest*.

Pour l'élément *file*, seul l'attribut *name* est requis. On peut y ajouter les attributs *hash* et *hashalg* pour contrôler l'intégrité du composant mais cet exemple est, volontairement, le plus simple possible.

Un composant peut exposer plusieurs classes qui doivent être décrites dans des sous-éléments *comClass* séparés. Seul l'attribut *clsid* du sous-élément *comClass* est normalement requis, mais il est souhaitable d'y ajouter les attributs *description*, *progid* et *threadingModel* afin d'identifier clairement les différentes classes disponibles.

Après la création des *manifests*, il reste une dernière étape : choisir l'emplacement de nos différents fichiers.
La solution la plus simple mais pas nécessairement la plus commode consiste à placer tous les fichiers dans le répertoire system32. Une solution plus souple (et qui est également la moins intrusive) consiste à placer, dans un même répertoire séparé, les *manifests*, le ou les fichiers du composant, une copie de **wscript.exe** ainsi que le script. Un simple batch lancera alors le script à partir de ce répertoire.

3.3. Démonstration avec un script

Pour ne pas déroger à la tradition, nous allons illustrer cette initiation avec un script écrit en VBScript qui affichera le message *Hello world !* et que nous appellerons **hello.vbs**.

hello.vbs

```
Option Explicit
Const _
MB_ICONWARNING = &H30, _
MB_YESNO = &H4

Dim oWrap,iRep
Set oWrap = CreateObject("DynamicWrapperX")
oWrap.Register "user32.dll", "MessageBoxW",
"i=hwwu","r=l"
iRep = oWrap.MessageBoxW(0, "Hello world !",
"RegFreeSxS par omen999", MB_YESNO +
MB_ICONWARNING)
```

Si l'on souhaite regrouper l'ensemble des fichiers dans un répertoire séparé, il importe de ne pas oublier les conditions de lancement du script. Les fichiers d'extension .vbs sont habituellement associés à l'exécutable **wscript.exe** placé dans le sous-répertoire system32. Pour obtenir la lecture des fichiers *manifests*, il est nécessaire de forcer le lancement de l'hôte WSH à partir de notre répertoire. La façon la plus simple d'y procéder est de créer un petit fichier batch appelé **hello.bat**.
(La commande start /B est purement cosmétique et force la fermeture de la fenêtre de commande)

hello.bat

```
start /B wscript hello.vbs
```

4. Limitations et inconvénients

Comme indiqué ci-dessus, cette méthode n'est applicable qu'à partir de la version XP SP2. Chaque composant impose l'écriture d'un *assembly manifest* qui doit décrire l'ensemble des classes exposées, outre la mise à jour de l'*application manifest* de wscript. Un explorateur d'objets est indispensable pour déterminer les données du composant ainsi que pour créer ce fichier manuellement. Il arrive cependant que certains composants présentent des interfaces non standards qui peuvent réserver des surprises (ex : pas d'implémentation de GetTypeInfo).

Pour les moins motivés, il existe des utilitaires qui permettent de créer ces fichiers *manifests* :
- Regsvr42.exe – opensource
- Side-by-Side Manifest Maker - commercial

5. Liens

Documentation Microsoft *Assembly Searching Sequence* : Lien40

Site regsvr42 : Lien41

Site Side-by-Side Manifest Maker : Lien42

Blog décrivant un exemple javascript (russe) : Lien43

Retrouvez l'article d'omen999 en ligne : Lien44

Les fonctions virtuelles en C++ : types statiques et types dynamiques

Les fonctions virtuelles sont un des piliers de la programmation orientée objet. En favorisant l'abstraction, elles permettent la construction d'architectures logicielles stables et évolutives. Cet article se propose d'explorer les fonctions virtuelles dans le langage C++ en abordant aussi bien les problèmes syntaxiques que les conséquences sémantiques de leur utilisation.

1. Les fonctions membres en C++

Trois types de fonctions peuvent être définis dans une classe en C++ (dans cet article, les termes classes et structures sont employés indifféremment. En effet, la seule différence en ce qui nous concerne entre class et struct est la visibilité par défaut : publique pour les membres et l'héritage pour les types définis avec struct et privée pour les membres et l'héritage pour les types définis avec class. Pour tous ce que nous allons dire par la suite, la chose est strictement identique que le type soit struct ou class) :

- les fonctions membres statiques ;
- les fonctions membres normales ;
- les fonctions membres virtuelles.

1.1. Les fonctions membres statiques

Le mot-clé static utilisé en début du prototype de la fonction permet de déclarer une fonction membre statique ou encore fonction de classe :

Exemple de fonction membre statique :

```
struct my_type
{
  static void s_function(); // fonction statique
};
```

Une fonction membre statique n'est pas liée à un objet. Elle n'a donc pas de paramètre implicite this et ne peut donc accéder à d'autres membres d'instances de la classe sinon les membres statiques :

Appel d'une fonction membre statique :

```
#include <iostream>
struct my_type
{
  static void s_function(); // fonction membre
statique

  static int mi_class_variable;
  int mi_member_variable;
};
int my_type::mi_class_variable=0;
void my_type::s_function()
{
   std::cout<<"je suis une fonction membre
statique !\n";
   mi_class_variable = 5; // On peut référencer
un membre statique de la classe
   /*
   mi_member_variable = 2;// Erreur, une fonction
membre statique ne peut utiliser
                          // un membre non
statique de la classe dépendant
                          // d'une instance
   this->mi_member_variable = 2; // Erreur, une
fonction membre statique
                                 // ne peut
utiliser this ; elle n'est pas
                                 // liée à une
instance de la classe
   */
}

int main()
{
   my_type::s_function(); // appel d'une fonction
membre statique
   my_type var;
   var.s_function(); // bien qu'un objet soit
utilisé, cet appel est équivalent
                     // au précédent. var n'est
pas pris en compte dans l'appel
   return 0;
}
```

L'appel d'une fonction membre statique d'une classe ne dépend pas d'une instance de ce type.

Par conséquent, les fonctions membres statiques ne nous intéresseront pas par la suite car elles ne dépendent nullement d'un objet.

1.2. Les fonctions normales

Les fonctions normales n'ont pas de mot-clé spécifique pour leur déclaration : c'est le comportement par défaut d'une fonction membre d'une classe.

Exemple de fonction membre normale :

```
#include <iostream>
struct my_type
{
  void a_function(); // fonction membre normale
};
void my_type::a_function()
{
   std::cout<<"je suis une fonction normale !\n";
}

int main()
{
   /*
   my_type::a_function(); // Erreur : nécessite
```

```
un objet pour être appelée
   */
   my_type var;
   var.a_function(); // nécessite une instance du
type pour être invoquée.
   return 0;
}
```

Ces fonctions membres ont un pointeur this désignant l'objet au départ duquel elles ont été invoquées et peuvent accéder aux membres de cet objet :

Appel d'une fonction normale :

```
#include <iostream>
struct my_type
{
  void a_function(); // fonction membre normale
  int mi_member;
};
void my_type::a_function()
{
   std::cout<<"je suis une fonction normale !\n";
   this->mi_member =41; // OK
   mi_member++; // OK : équivalent à (this-
>mi_member)++;
}

int main()
{
   my_type var;
   var.a_function();
   std::cout<<"valeur de mi_member de var :
"<<var.mi_member<<"\n";
                        // C'est bien l'objet var
qui a été associé à l'appel de la fonction
   return 0;
}
```

1.3. Les fonctions virtuelles

Les fonctions virtuelles précèdent leur déclaration du mot clé virtual :

Exemple d'une fonction virtuelle :

```
#include <iostream>
struct my_type
{
  virtual void a_function(); // fonction
virtuelle
};
void my_type::a_function() // dans la définition
de la fonction, il ne faut
                           // pas répéter le mot-
clé 'virtual'
{
   std::cout<<"je suis une fonction
virtuelle !\n";
}

int main()
{
/*
   my_type::a_function(); // Erreur : nécessite
un objet pour être appelée
*/
   my_type var;
   var.a_function(); // nécessite une instance du
type pour être invoquée.
   return 0;
}
```

Comme les fonctions membres normales, les fonctions virtuelles ont un pointeur this et peuvent accéder aux membres de cet objet :

Appel d'une fonction virtuelle :

```
#include <iostream>
struct my_type
{
  virtual void a_function(); // fonction
virtuelle
  int mi_member;
};
void my_type::a_function()
{
   std::cout<<"je suis une fonction normale !\n";
   this->mi_member =41; // OK
   mi_member++; // OK : équivalent à (this-
>mi_member)++;
}

int main()
{
   my_type var;
   var.a_function();
   std::cout<<"valeur de mi_member de var :
"<<var.mi_member<<"\n";
                        // C'est bien l'objet var
qui a été associé à l'appel de la fonction
   return 0;
}
```

Soit en résumé :

L'appel d'une fonction membre non statique d'une classe nécessite une instance du type.

Par défaut, en C++, les fonctions membres ne sont pas virtuelles. Le mot-clé virtual est nécessaire pour définir une fonction virtuelle.

En C++, les fonctions virtuelles doivent être membres d'une classe.

Anticipons en signalant l'existence d'une catégorie particulière de fonctions virtuelles en C++ : les fonctions virtuelles pures. Une fonction virtuelle pure est une fonction virtuelle à laquelle est ajouté =0 à la fin de sa déclaration :

Exemple d'une fonction virtuelle pure :

```
struct my_type
{
  virtual void a_function()=0; // fonction
virtuelle pure
};
```

Nous reviendrons un peu plus loin sur les fonctions virtuelles pures, pour l'instant il suffit de savoir que ça existe et qu'avant d'être "pures", ce sont avant tout des fonctions *virtuelles*.

Toutes les fonctions d'une classe peuvent-elles être virtuelles ? Oui, enfin presque : seuls les constructeurs (et les fonctions statiques) ne peuvent pas être virtuels. Toutes les autres fonctions le peuvent : que ce soit le destructeur, les opérateurs, ou des fonctions quelconques. Nous verrons par la suite ce que cela signifie et l'intérêt dans chaque cas.

Toutes (ou presque) les fonctions peuvent être virtuelles

```
struct my_type
{
//   virtual my_type(); // erreur : un
constructeur ne peut être virtuel
//   virtual static void s_function(); //
erreur : une fonction ne peut être ET statique ET
virtuelle
   virtual ~my_type();// OK
   virtual void function(); // OK
   virtual my_type& operator+(my_type const&); //
OK
};
```

La virtualité s'hérite : une fonction virtuelle dans la classe de base reste virtuelle dans la classe dérivée même si le mot clé virtual n'est pas accolé :

La virtualité s'hérite :

```
struct base
{
   void function_1();
   virtual void function_2();
   void function_3();
};

struct derived : public base
{
   void function_1();
   void function_2();
   virtual void function_3();
};
```

Nous avons avec cet exemple :

	base	derived
function_1	non virtuelle	non virtuelle
function_2	virtuelle	virtuelle
function_3	non virtuelle	virtuelle

Autant, comme le montre la troisième ligne, il est possible de masquer une fonction non virtuelle d'une classe de base par une fonction virtuelle dans une classe dérivée, autant il est impossible de s'en débarrasser. Une fonction est et sera virtuelle pour toutes les classes dérivant de la classe l'ayant définie comme telle. Cependant, afin d'éviter toute confusion, il est fortement recommandé d'utiliser le mot-clé virtual dans les classes dérivées :

Les classes dérivées devraient utiliser le mot-clé virtual pour les fonctions définies comme virtuelles dans la classe de base.

1.4. La surcharge de fonction

Introduisons un dernier point pour poser notre problématique : la surcharge de fonction. Il est aussi possible de surcharger une fonction dans une classe, c'est à dire définir plusieurs fonctions avec le même nom, à condition qu'elles diffèrent par :

- leur nombre d'arguments ;
- et/ou le type d'au moins un des arguments ;
- et/ou leur constance.

La signature d'une fonction en C++ désigne son nom, le nombre de ses arguments, leur type et la constance de la fonction. Surcharger une fonction F1 revient alors à proposer une nouvelle fonction F2, telle que la signature de F1 est différente de celle de F2 autrement que par le nom qu'elles partagent.
Par exemple, le code suivant présente différentes surcharges d'une fonction membre :

Surcharges d'une fonction membre :

```
struct my_type
{
   void function(double,double){}
   void function(double){} // le nombre
d'argument est différent
                           // de la précédente
définition
   void function(int){} // le type de l'argument
est différent
                            // de la précédente
définition
   void function(int) const {} // la constance
est différente
                                  // de la
précédente définition
   void function(char&){}
   void function(char const &){} // char& et char
const & sont deux types différents
};
```

La surcharge est un cas de polymorphisme en C++. Cela permet d'adapter la fonction à appeler selon les arguments en paramètre :

Appels des différentes surcharges d'une fonction membre :

```
#include <iostream>
struct my_type
{
   void function(double,double) // (1)
   {
      std::cout<<"(1)\n";
   }
   void function(double) // (2)
   {
      std::cout<<"(2)\n";
   }
   void function(int) // (3)
   {
      std::cout<<"(3)\n";
   }
   void function(int) const // (4)
   {
      std::cout<<"(4)\n";
   }
   void function(char&) // (5)
   {
      std::cout<<"(5)\n";
   }
   void function(char const &) // (6)
   {
      std::cout<<"(6)\n";
   }
};

int main()
{
   my_type var;
   var.function(1.,1.); // (1)
   var.function(1.); // (2)
```

```
   var.function(1); // (3)
   my_type const c_var=my_type();
   c_var.function(1); // (4)
   char c('a');
   var.function(c);  // (5)
   char const &rc = c;
   var.function(rc); // (6)

   return 0;
}
```

La surcharge a ses limites. En particulier, une même classe ne peut pas redéfinir une fonction avec le même nom d'une fonction existante si :

- elles ne diffèrent que par leur type retour ;
- elles ont les mêmes arguments, le même type retour mais l'une d'elles est statique ;
- elles ont les mêmes arguments, le même type retour, la même constance mais l'une d'elles est virtuelle (ou virtuelle pure) et l'autre normale ;
- elles ont les mêmes arguments, le même type retour, la même constance mais l'une d'elles est virtuelle et l'autre virtuelle pure ;
- un argument ne diffère qu'à cause d'un typedef ;
- elles ne diffèrent que par un const non significatif sur le type d'un argument.
- elles ne diffèrent que par les valeurs par défaut de leur(s) argument(s).

Ainsi les surcharges suivantes sont interdites dans une même classe :

Surcharges interdites :

```
struct my_type
{

   void function_1();
   int function_1(); // Erreur : seul le type
retour est différent

   static void function_2();
   void function_2(); // Erreur : function_2 est
déjà définie et statique
   void function_2()const; // const ne suffit pas
pour différencier une
                           // fonction non
statique et une fonction statique

   virtual void function_3();
   void function_3(); // Erreur : function_3 est
déjà définie et virtuelle

   virtual void function_3_bis();
   void function_3_bis()const; // OK : const
suffit pour différencier des
                          // fonctions membres
non statiques


   virtual void function_4();
   virtual void function_4()=0; // Erreur : la
fonction a déjà été déclarée
                                // virtuelle mais
non pure.
                                // L'erreur
aurait été la même si on avait
                                // d'abord
déclaré la fonction virtuelle pure
                                // puis la
fonction virtuelle (non pure)

   void function_5(int);
   typedef int t_int;
   void function_5(t_int); // Erreur : le typedef
n'est qu'un synonyme

   void function_6(char);
   void function_6(char const); // Erreur : le
const n'est pas significatif
                                 // pour
différencier les deux fonctions

   void function_6_bis(char *);
   void function_6_bis(char * const); // Erreur :
le const n'est pas significatif
                                  // pour
différencier les deux fonctions
   void function_6_ter(char *);
   void function_6_ter(char const *); // OK :
char const * et char * sont bien
                                  // deux types
différents

   void function_7(int );
   void function_7(int =42); // Erreur : les
définitions sont équivalentes
};
```

1.5. Quand l'héritage chamboule tout !

L'héritage importe dans la classe dérivée toutes les déclarations des classes de bases :

Héritage des membres de la classe parent :

```
struct base
{
   void function(){}
};

struct derived : base
{
};

int main()
{
   derived d;
   d.function(); // on récupère l'interface de
base

   return 0;
}
```

Cependant, une fonction déclarée dans une classe dérivée ayant le même nom qu'une fonction de la classe de base mais avec une signature différente masque la fonction de la classe de base dans la classe dérivée :

Masquage des fonctions de base dans la classe dérivée :

```
struct base
{
   void function(){}
};

struct derived : base
{
   void function(int){}
   void call_a_function()
```

```
    {
        function(); // Erreur base::function est
masquée par derived::function
        base::function(); // OK : on indique
explicitement la fonction à appeler
        function(1); // appel de
derived::function(int)
    }
};

int main()
{
    derived d;
    d.function(); // Erreur base::function est
masquée par derived::function
    d.base::function(); // OK : on indique
explicitement la fonction à appeler
    d.function(1); // appel de
derived::function(int)

    return 0;
}
```

La surcharge dans une classe dérivée d'une fonction définie dans une classe de base avec une signature différente masque la fonction de la classe de base dans et pour la classe dérivée.

Nous avons vu à la section précédente qu'il n'était pas possible dans une même classe de redéfinir une nouvelle fonction avec la même signature qu'une fonction existante (même nom, même paramètres, même constance). Et, nous en arrivons au point qui va nous intéresser, à savoir :

Une classe dérivée peut redéfinir une fonction d'une classe de base ayant la même signature !

Ainsi, en reprenant dans la section précédente l'exemple des surcharges interdites et en recopiant les surcharges interdites vers la classe dérivée, nous obtenons le code valide suivant :

Tout redevient possible avec l'héritage :

```
struct base
{

    void function_1();

    static void function_2();

    virtual void function_3();

    virtual void function_4();

    void function_5(int);

    void function_6(char);
    void function_7(int );
};

struct derived : public base
{
    int function_1(); // OK
    void function_2(); // OK
    void function_2()const; // OK
    void function_3(); // OK
    virtual void function_4()=0; // OK
    typedef int t_int; // OK
    void function_5(t_int); // OK
    void function_6(char const); // OK
    void function_7(int =42); // OK
};
```

L'objectif est maintenant de savoir quelles fonctions sont appelées lorsqu'une même signature est disponible dans une classe dérivée et une classe de base selon l'expression utilisée pour l'appel :

Quelle est la fonction appelée ?

```
struct base
{
    void function_1()
    {
    }
    virtual void function_2()
    {
    }
    void call_function_1()
    {
        function_1(); // Quelle est la fonction
appelée ?
    }
    void call_function_2()
    {
        function_2(); // Quelle est la fonction
appelée ?
    }
};

struct derived : public base
{
    void function_1()
    {
    }
    virtual void function_2()
    {
    }
    void call_function_1()
    {
        function_1(); // Quelle est la fonction
appelée ?
    }
    void call_function_2()
    {
        function_2(); // Quelle est la fonction
appelée ?
    }
};

int main()
{
    base b;
    b.function_1(); // Quelle est la fonction
appelée ?
    b.function_2(); // Quelle est la fonction
appelée ?

    derived d;
    d.function_1(); // Quelle est la fonction
appelée ?
    d.function_2(); // Quelle est la fonction
appelée ?

    base &rb = b;
    rb.function_1(); // Quelle est la fonction
appelée ?
    rb.function_2(); // Quelle est la fonction
```

```
appelée ?

   base &rd = d;
   rd.function_1(); // Quelle est la fonction
appelée ?
   rd.function_2(); // Quelle est la fonction
appelée ?

   return 0;
}
```

Pour arriver à répondre à toutes ces questions, il nous faut introduire une nouvelle notion : le type statique et le type dynamique d'une variable.

2. Type statique et type dynamique

Une variable possède deux types : un type statique et un type dynamique.
Le type **statique** d'une variable est celui **déterminé à la compilation**. Le type statique est le plus évident : c'est celui avec lequel vous avez déclaré votre variable. Il est sous votre nez lorsque vous regardez le code.
Le type **dynamique** d'une variable est celui **déterminé à l'exécution**. Le type dynamique quant à lui n'est pas immédiat en regardant le code. En effet, il va pouvoir varier à l'exécution selon ce que la variable va effectivement désigner pendant le déroulement du programme.
Le type dynamique et le type statique coïncident pour les variables utilisées par valeur :

Type statique et type dynamique de variables par valeur :
```
int a;
char c;
std::string s;
class a_class{/*[...]*/};
a_class an_object;
enum E_enumeration{/*[...]*/};
E_enumeration e;
```

Avec cet exemple, on a :

variable	type statique	type dynamique
a	int	int
c	char	char
s	std::string	std::string
an_object	a_class	a_class
e	E_enumeration	E_enumeration

Type statique et dynamique d'une variable par valeur

Encore une fois, l'héritage introduit une différence entre un type dynamique et un type statique. Cette différence apparaît avec les pointeurs et les références quand le type déclaré du pointeur ou de la référence n'est pas le type de l'objet effectivement pointé (resp. référencé) à l'exécution. Le type statique est celui défini dans le code, le type dynamique d'une référence ou d'un pointeur est celui de l'objet référencé (resp. pointé) :

Type statique, type dynamique, héritage, pointeurs et références :
```
struct base {};

struct derived : public base {};

int main()
{

   base b;
   derived d;
   base &rb = b;
   base &rd = d;
   base *pb = &b;
   base *pd = &d;

   return 0;
}
```

Avec l'exemple, ci-dessus, les types des variables sont :

variable	type statique	type dynamique
base b	base	base
derived d	derived	derived
base &rb = b	base	base
base &rd = d	base	derived
base *pb = &b	base	base
base *pd = &d	base	derived

Type statique et dynamique d'une variable par référence

Seul les pointeurs et les références vers des instances de classe ou de structure ont des types dynamiques et des types statiques pouvant diverger.

Le type statique d'un objet dans une fonction membre est celui où se déroule la fonction. Le type dynamique, this étant un pointeur, dépend de l'objet effectif sur lequel s'applique la fonction. Type statique et type dynamique peuvent alors être différents :

Type de this :
```
class base
{
public:
   void function()
   {
                // Quel est le type static de
this ?
                // Quel est le type dynamique de
this ?
   }
};
class derived : public base
{
public:
   void function_2()
   {
                // Quel est le type static de
this ?
                // Quel est le type dynamique de
this ?
   }
};
int main()
{
   base b;
   b.function();
   derived d;
   d.function();
```

```
   d.function_2();

   return 0;
}
```

fonction	type statique de this	type dynamique de this
Pour l'appel b.fonction, dans la fonction base::fonction	base	base
Pour l'appel de d.fonction, dans la fonction base::fonction	base	derived
Pour l'appel de d.fonction_2 dans la fonction derived::fonction_2	derived	derived

Type statique et dynamique dans une fonction membre

Le type statique d'une variable est soit le type dynamique de cette variable soit une classe de base directe ou indirecte du type dynamique.

Toute autre combinaison est une erreur pouvant aboutir à un plantage ou un comportement indéterminé.

Attention, si nous avons dit que le pointeur a un type statique différent de son type dynamique, cela s'applique aussi bien au pointeur non déréférencé qu'au pointeur déréférencé :

Types statiques et dynamiques d'un pointeur :

```
struct base {};
struct derived : public base {};

int main()
{

   base b;
   derived d;
   base *pd = &d;

   return 0;
}
```

variable	type statique	type dynamique
pd	base*	derived*
*pd	base	derived

Type statique et dynamique d'une variable par référence

3. Types et appel de fonction

Lorsqu'une expression contient un appel d'une fonction sur un objet donné, la fonction effectivement appelée dépend de plusieurs paramètres :

- la fonction est-elle virtuelle ou non ?
- la fonction est-elle appelée sur un objet par valeur ou sur une référence/pointeur ?

Selon la réponse, la résolution de l'appel est faite à la compilation ou à l'exécution :

	fonction virtuelle	
appel sur un objet	COMPILATION	COMPILATION
appel sur une référence ou un pointeur	COMPILATION	EXÉCUTION

Quand est résolu l'appel ?

La résolution à la compilation utilise le type statique car c'est le seul connu à ce moment.
La résolution à l'exécution se base sur le type dynamique.
Ce qui donne :

	fonction non virtuelle	fonction virtuelle
appel sur un objet	type statique	type statique
appel sur une référence ou un pointeur	type statique	type dynamique

Type utilisé pour la résolution de l'appel

Un peu de code pour illustrer tout cela :

Résolution à la compilation ou à l'exécution des appels :

```
#include <iostream>

struct base
{
   void function_1()
   {
      std::cout<<"base::function_1\n";
   }

   virtual void function_2()
   {
      std::cout<<"base::function_2\n";
   }

};
struct derived :public base
{
   void function_1()
   {
      std::cout<<"derived::function_1\n";
   }

   virtual void function_2()
   {
      std::cout<<"derived::function_2\n";
   }
};

int main()
{
   base b;
   std::cout<<"\nappels pour base b; : \n";
   b.function_1();
   b.function_2();

   derived d;
   std::cout<<"\nappels pour derived d; : \n";
   d.function_1();
   d.function_2();

   base &rb = b;
   std::cout<<"\nappels pour base &rb = b; : \n";
   rb.function_1();
```

```
   rb.function_2();

   base &rd = d;
   std::cout<<"\nappels pour base &rd = d; : \n";
   rd.function_1();
   rd.function_2();

   return 0;
}
```

Ce code produit comme sortie :

Sortie :

```
appels pour base b; :
base::function_1
base::function_2

appels pour derived d; :
derived::function_1
derived::function_2

appels pour base &rb = b; :
base::function_1
base::function_2

appels pour base &rd = d; :
base::function_1
derived::function_2
```

Les types statiques et dynamiques sont :

variable	type statique	type dynamique
base b	base	base
derived d	derived	derived
base &rb = b	base	base
base &rd = d	base	derived

Type statique et dynamique des différentes variables

fonction_1 est une fonction non virtuelle et fonction_2 est une fonction virtuelle. Soit en reprenant le tableau précédent précisant la fonction appelée :

Appels sur b	fonction non virtuelle (fonction_1)	fonction virtuelle (fonction_2)
appel sur un objet	type statique : base::fonction_1	type statique : base::fonction_2
Appels sur d		
appel sur un objet	type statique : derived::fonction_1	type statique : derived::fonction_2
Appels sur rb		
appel sur une référence	type statique : base::fonction_1	type dynamique : base::fonction_2
Appels sur rd		
appel sur une référence	type statique : base::fonction_1	type dynamique : derived::fonction_2

Fonction appelée selon le moment de résolution de l'appel

Cette résolution dynamique présentée avec des références fonctionne de la même façon avec un pointeur qu'il soit utilisé en tant que tel ou déréférencé :

Références et pointeurs : même combat !

```
#include <iostream>

struct base
{
   virtual void function()
   {
      std::cout<<"base::function\n";
   }

};
struct derived :public base
{
   virtual void function()
   {
      std::cout<<"derived::function\n";
   }
};

int main()
{
   derived d;
   base &rd = d;
   base *pd = &d;

   // 3 appels équivalents :
   pd->function();
   (*pd).function();
   rd.function();

   return 0;
}
```

Le comportement est identique si l'expression utilise un pointeur de fonction :

Appel de fonction avec un pointeur de fonction membre :

```
#include <iostream>

struct base
{
   virtual ~base(){}
   virtual void function_1()
   {
      std::cout<<"base::function_1\n";
   }

   void function_2()
   {
      std::cout<<"base::function_2\n";
   }

};
struct derived : public base
{
   virtual void function_1()
   {
      std::cout<<"derived::function_1\n";
   }

   void function_2()
   {
      std::cout<<"base::function_2\n";
   }
```

```
};

void call_a_function(void (base::*pf_)())
{
   base b;
   (b.*pf_)();
   derived d;
   (d.*pf_)();
   base &rd = d;
   (rd.*pf_)();
}

int main()
{
   std::cout<<"function_1 :\n";
   call_a_function(&base::function_1);
   std::cout<<"function_2 :\n";
   call_a_function(&base::function_2);

   return 0;
}
```

La sortie produite est bien :

```
function_1 :
base::function_1
derived::function_1
derived::function_1
function_2 :
base::function_2
base::function_2
base::function_2
```

La liaison tardive prenant appui sur le type dynamique telle que nous la décrivons ici est valable presque tout le temps. Comme nous allons le voir par la suite, ce mécanisme présente quelque subtilité en particulier lors des phases sensibles que sont la construction ou la destruction d'un objet.

4. A quoi servent les fonctions virtuelles ?

L'utilisation du type dynamique pour résoudre l'appel d'une fonction virtuelle est une des grandes forces de la programmation orientée objet (POO). Elle permet d'adapter et de faire évoluer un comportement défini dans une classe de base en spécialisant les fonctions virtuelles dans les classes dérivées. La substitution d'un objet de type dynamique dérivant du type statique présent dans l'expression contenant l'appel vers une fonction virtuelle s'inscrit dans le cadre du polymorphisme d'inclusion (Lien45).

Ce polymorphisme d'inclusion permet l'abstraction dans un logiciel orienté objet. Ainsi, un objet peut être manipulé à partir d'un pointeur ou d'une référence vers la classe de base. Les membres publics de la classe de base déterminent les services proposés et la mise en oeuvre est déléguée aux classes dérivées apportant des points de variations ou spécialisant des comportements dans les fonctions virtuelles :

Le polymorphisme d'inclusion : un principe fondamental de l'objet !

```
#include <iostream>
struct shape
{
   virtual void draw() const
   {
      std::cout<<"une forme amorphe\n";
   }
};

void draw_a_shape(shape const &rs)
{ // ne connaît que shape. Pas ses classes
dérivées
   rs.draw();
}

struct square : public shape
{
   virtual void draw() const
   {
      std::cout<<"un carre\n";
   }
};
struct circle : public shape
{
   virtual void draw() const
   {
      std::cout<<"un cercle\n";
   }
};

int main()
{
   shape sh;
   draw_a_shape(sh);
   square sq;
   draw_a_shape(sq);
   circle c;
   draw_a_shape(c);

   return 0;
}
```

L'abstraction permet de mieux isoler les différents composants les uns des autres en ne les rendant dépendants que des services dont ils ont vraiment besoin. Elle favorise la modularité et l'évolutivité des architectures logicielles. Notre fonction draw_a_shape peut dessiner tout type d'objet non prévu lors de son écriture du moment que ces objets sont d'un type dynamique héritant de shape et spécialisant ses fonctions virtuelles. Il devient possible de rajouter de nouveaux types et de pouvoir les dessiner sans avoir à réécrire la fonction draw_a_shape.

Les fonctions virtuelles réduisent le couplage entre une classe ou une fonction cliente et une classe fournisseur en déléguant aux classes dérivées la réalisation d'une partie des services proposés par la classe de base.

Les fonctions virtuelles sont un mécanisme pour la mise en oeuvre du principe ouvert/fermé en permettant de faire évoluer une application par l'ajout de nouvelle classe dérivée (ouvert) sans avoir à toucher le code existant utilisant l'interface de la classe de base (fermé).

Les fonctions virtuelles favorisent la réutilisation. Toutes les fonctions ou les classes s'appuyant sur des références

ou des pointeurs de la classe de base peuvent être directement utilisées avec des objets d'un nouveau type dérivé.

Si le terme complet est polymorphisme *d'inclusion*, beaucoup de documents en C++ (articles - et celui-ci n'échappe pas à la règle - , cours, livres, etc.) omettent de préciser *d'inclusion*. *Polymorphe*, *polymorphique*, *polymorphisme*, *polymorphiquement* s'emploient souvent seuls dès qu'il s'agit de parler d'héritage et donc de la manipulation d'un objet d'une classe dérivée à partir d'une référence ou d'un pointeur d'une de ses classes de bases avec en arrière plan le mécanisme des fonctions virtuelles. C'est un raccourci qui peut faire oublier les autres formes de polymorphisme qui ne s'appuient pas sur les fonctions virtuelles et le mécanisme d'héritage. Il faut juste se souvenir que le polymorphisme ne se réduit pas au polymorphisme d'inclusion.

Retrouvez la suite de l'article de 3DArchi en ligne : Lien46

Appels de méthodes déterminés dynamiquement

Cet article présente une technique pour appeler des méthodes de certaines classes dans un système de type « script ». Dans une telle situation, les méthodes et leurs paramètres ne sont connus qu'à l'exécution, sous la forme de chaînes de caractères.

1. Les fonctions membres en C++

Dans le cadre de votre projet, vous avez peut être la nécessité d'appeler des méthodes de vos classes en ne connaissant leur nom et la valeur de leurs paramètres que sous forme de chaîne de caractères. Cela arrive quand la méthode à appeler est donnée par l'utilisateur ou lorsque des commandes sont lues dans un fichier.

Cet article décrit une technique pour « exporter » le nom des méthodes d'une classe. Le cœur du système tourne autour d'une classe de base pour toutes celles que l'on voudra exporter. Cette classe est paramétrée par le contexte d'exécution, qui pourra contenir des informations utiles dans la conversion des paramètres depuis une chaîne de caractères vers un type donné.

1.1. Exemple introductif

Imaginons un logiciel de traitement d'image par lot, nommé batch-img, pour lequel la classe d'image propose ces opérations classiques :

```
class image
{
public:
  /* Applique une rotation d'angle a à l'image,
autour du centre aux
     coordonnées (cx, cy). */
  void tourner( double a, int cx, int cy );

  /* Retourne l'image horizontalement. */
  void miroir();

  /* Copie l'image img sur l'image courante, à la
position (x, y). */
  void copier( image* img, int x, int y );

}; // class image
```

On aimerait utiliser ce logiciel en ligne de commande pour effectuer des traitements sur des images. En particulier, on aimerait que l'utilisateur puisse écrire des petits scripts comme celui ci-dessous, qu'il passerait au programme.

```
traitement.txt:
   image_1 miroir
   image_2 tourner 18.6 24 -16
   image_1 copier image_2 700 30
   image_1 miroir
```

L'utilisateur pourrait appliquer ce script à des images avec une commande comme celle-ci:

```
batch-img image_1=fred.jpg image_2=moscou.jpg
traitement.txt
```

1.2. Vue globale

Nous allons créer une classe base_exportable de laquelle devront hériter celles pour lesquelles nous voudrons exporter des méthodes. À côté de cela, nous écrirons une classe method_caller dont le rôle sera, à partir de chaînes de caractères représentant les valeurs des paramètres, d'appeler une méthode d'un base_exportable en lui passant les bonnes valeurs. Ainsi, la classe base_exportable contiendra une table statique qui associera à un nom de méthode une instance de method_caller. Lorsque nous voudrons appeler une méthode, il suffira de retrouver cette instance et de lui passer les valeurs des paramètres. method_caller sera redéfinie selon le nombre de paramètres à passer à la méthode. Cet exemple se limitera à trois paramètres, mais l'ajout d'un method_caller pour quatre paramètres ou plus est simple.

Toute la difficulté consiste à choisir automatiquement la classe dérivée de method_caller et les conversions de types en ne demandant que le nom de la méthode les types des paramètres à l'utilisateur de notre outil.

2. Les éléments génériques

Cette section présente les éléments au cœur du système.

2.1. Classe de base pour exporter les méthodes

base_exportable est la classe de base de toutes celles pour lesquelles nous voulons exporter des méthodes. Elle contient une méthode base_exportable::execute() qui permettra de retrouver une méthode à partir de son nom et de l'appeler.

Les paramètres de la méthode appelée sont passés sous

forme de chaînes de caractères et seront convertis vers des valeurs du type réel des paramètres. Par défaut, cette conversion utilise la lecture formatée d'un std::istringstream. Cependant, dans certaines situations, l'utilisateur peut vouloir gérer lui-même la conversion. Par exemple, dans le cas de batch-img, les images sont nommées. On aimerait donc que celles passées en paramètre aux méthodes soient retrouvées à partir de leur nom. Pour autoriser ce genre de situation, la classe base_exportable est paramétrée par un type <Context>, qui sera ensuite transmis à la classe string_to_arg ci-dessous. Le rôle de cette dernière est de convertir une chaîne de caractères représentant un paramètre en la valeur réelle du paramètre, via une méthode statique nommée convert. En spécialisant cette classe, nous pourrons ajuster la conversion en fonction du contexte.

```
/** Convertit un paramètre une valeur de type T.
*/
template<typename Context, typename T>
class string_to_arg
{
public:
  static T convert( const Context& context, const
std::string& arg )
  {
    T result;
    std::istringstream iss(arg);
    iss >> result;
    return result;
  }
}; // class string_to_arg
```

Le squelette de la classe base_exportable prenant en compte le contexte est présenté ci-dessous. Nous le complèterons dans la suite de l'article.

```
template<typename Context>
class base_exportable
{
public:
  // Le type du contexte.
  typedef Context script_context_type;

  /** Exécute une méthode de la classe. \a n est
le nom de la méthode, \a args
      ses paramètres. \a context est le contexte
d'exécution. */
  void execute( const std::string& n, const
std::vector<std::string>& args,
                const script_context_type&
context );

}; // class base_exportable
```

2.2. Exécuter une méthode d'une dérivée de base_exportable

La classe method_caller permet de faire la transition entre un appel en chaînes de caractères et l'appel effectif d'une méthode. Des instances seront stockées dans une table statique de base_exportable pour faire la correspondance.

Comme nous ne savons pas encore, à ce niveau, de quel type est l'instance sur laquelle l'appel doit se faire, nous devrons nous contenter de la déclaration ci dessous.

```
template<typename Context>
class method_caller
{
public:
  typedef Context script_context_type;

public:
  /* Exécute une méthode de 'self' en lui passant
les paramètres
     de 'args' convertis selon le contexte
'context'. */
  virtual void execute
  ( base_exportable<script_context_type>* self,
    const std::vector<std::string>& args,
    const script_context_type& context ) const =
0;

}; // class method_caller
```

La méthode method_caller::execute est abstraite, ce qui signifie que nous laissons le soin d'effectuer l'appel à une classe dérivée, qui aura plus d'informations sur le type de la classe. Cette classe dérivée sera explicit_method_caller.

```
template<typename SelfClass>
class explicit_method_caller:
  public method_caller<typename
SelfClass::script_context_type>
{
public:
  typedef typename SelfClass::script_context_type
script_context_type;

public:
  /* explicit_execute va effectivement appeler la
méthode sur une instance de
     SelfClass. Cette méthode sera définie dans
des classes dérivées
     pour gérer la conversion des paramètres et
leur nombre. */
  virtual void explicit_execute
  ( SelfClass& self, const
std::vector<std::string>& args,
    const script_context_type& context )  const =
0;

private:
  virtual void execute
  ( base_exportable<script_context_type>* self,
    const std::vector<std::string>& args,
    const script_context_type& context ) const
  {
    SelfClass* s =
dynamic_cast<SelfClass*>(self);

    if ( s!=NULL )
      explicit_execute(*s, args, context);
  }
}; // explicit_method_caller
```

Là encore, comme nous ne connaissons pas la méthode à appeler, nous laissons le soin à une classe dérivée de faire l'appel. La classe explicit_method_caller ne sert donc qu'à convertir l'instance de base_exportable vers son type réel.

Il ne nous reste plus qu'à faire des dérivées de explicit_method_caller. Ces classes seront paramétrées par la méthode à appeler, définie par son type de retour, le type de ses paramètres et son adresse. Nous allons en faire une pour chaque nombre de paramètres. Nous ne pouvons pas

en faire une infinité mais il sera facile d'ajouter une nouvelle classe lorsque l'utilisateur de notre système l'utilisera avec un nombre de paramètres non prévu. Les cas d'une méthode sans paramètres et d'une méthode avec deux paramètres sont présentés dans la section suivante.

2.3. Implémentation de l'appel d'une méthode

Nous allons maintenant créer les patrons de classes qui appellent effectivement les méthodes. Les types paramétrés de cette classe entrent dans trois catégories : le type de l'objet sur lequel la méthode est appelée, la signature de ladite méthode, représentée par ses paramètres et sa valeur de retour, puis l'adresse de la méthode. La version la plus simple appelle une méthode sans paramètres et est présentée ci-dessous :

```
/** Cette classe définit un dérivé de
explicit_method_caller pour appeler une
    méthode donnée d'une classe donnée. Cette
dernière est définit par
    SelfClass. L'adresse de la méthode est Member
et elle ne doit prendre
    aucun paramètre et retourne une valeur de
type R. */
template< typename SelfClass, typename R,
          R (SelfClass::*Member)() >
class method_caller_args_0
{
public:
  // Le type de la méthode appelée
  typedef R (SelfClass::*mem_fun_type)();

  // Le type du contexte dans lequel la méthode
est appelée
  typedef typename SelfClass::script_context_type
script_context_type;

public:
  /** Cette classe appelle la méthode voulue sur
un objet passé en
      paramètre. */
  class caller_type:
    public explicit_method_caller<SelfClass>
  {
  private:
    /** L'appel de la méthode Member s'effectue
ici, sur l'objet self,
      en lui passant les paramètres du tableau
args. */
    void explicit_execute
    ( SelfClass& self, const
std::vector<std::string>& args,
      const script_context_type& context ) const
    {
      const mem_fun_type member(Member);
      (self.*member)();
    }
  }; // class caller_type;

public:
  /** Il suffit d'utiliser cet objet pour appeler
la méthode. */
  static const caller_type s_caller;

}; // class method_caller_args_0
```

L'appel de la méthode se fait enfin, dans caller_type::explicit_execute(), au niveau des lignes 29 et 30. Ici, la méthode ne prend pas de paramètres. Ainsi, "args" devrait être vide. Le contexte n'est pas utilisé ici mais il l'est dans la classe ci-dessous, à peine plus complexe, implémentant l'appel d'une méthode à deux paramètres :

```
/** Cette classe définit un dérivé de
explicit_method_caller pour appeler une
    méthode donnée d'une classe donnée. Cette
dernière est définit par
    SelfClasse. L'adresse de la méthode est
Member et doit prendre deux
    paramètres de types A1 et A2. */
template< typename SelfClass, typename R,
typename A0, typename A1,
          R (SelfClass::*Member)(A0, A1) >
class method_caller_args_2
{
public:
  // Le type de la méthode appelée
  typedef R (SelfClass::*mem_fun_type)(A0, A1);

  // Le type du contexte dans lequel la méthode
est appelée
  typedef typename SelfClass::script_context_type
script_context_type;

public:
  class caller_type:
    public explicit_method_caller<SelfClass>
  {
  private:
    /** L'appel de la méthode Member s'effectue
ici, sur l'objet self,
      en lui passant les paramètres du tableau
args. Les chaînes de
      caractères de ce dernier sont convertis
dans le type des
      paramètres en utilisant le contexte. */
    void explicit_execute
    ( SelfClass& self, const
std::vector<std::string>& args,
      const script_context_type& context ) const
    {
      const mem_fun_type member(Member);
      (self.*member)
        ( string_to_arg<script_context_type,
A0>::convert(context, args[0]),
          string_to_arg<script_context_type,
A1>::convert(context, args[1]) );
    }
  }; // class caller_type;

public:
  /** Une seule instance de caller_type est
suffisante dans la mesure où il
      n'y a pas de variables membres (s_caller ne
sert qu'à faire la
      transition vers la méthode réelle). On
pourra ainsi utiliser son adresse
      et éviter des allocations dynamiques. */
  static const caller_type s_caller;

}; // class method_caller_args_2
```

Cette classe est similaire à la précédente, sauf qu'elle appelle une méthode avec deux paramètres. Nous pouvons voir que les chaînes de caractères du tableau "args" sont converties dans le type des paramètres en utilisant le contexte. Cela nous permettra de faire la conversion vers un objet de type image à partir de son nom. En dehors de

cela, la difficulté est identique à la classe précédente.

Remarquons qu'il est nécessaire de faire une nouvelle classe de ce type pour chaque nombre de paramètres. Pour cela, nous pouvons utiliser la bibliothèque Boost.Preprocessor (Lien47). Celle-ci nous permettra de générer les classes method_caller_args_X automatiquement, pour tout X dans un intervalle donné.

2.4. Associer un method_caller à un nom de méthode

Pour résoudre le problème de l'export des méthodes, nous allons partir du résultat. Pour la méthode image::tourner(double, int, int), par exemple, nous aimerions ne rien avoir à dire d'autre que « je veux exporter la méthode tourner de la classe image, ses paramètres sont un réel, un entier et un autre entier ».

L'idéal, en effet, serait de n'avoir à indiquer que le nom de la méthode, le type des paramètres et la classe. À partir de ces indications, le système doit pouvoir associer une méthode à une instance adéquate de method_caller.

Pour que notre système puisse appeler une méthode à partir de son nom, il faut que l'association nom/méthode soit stockée quelque part. Un endroit correct semble être dans une table de la classe concernée. De plus, comme la méthode est indépendante de l'instance de la classe, il s'agit bien d'une table statique.

La première partie, ci-après, présente le type de cette table et les méthodes nécessaires pour bien l'utiliser. La seconde partie présente l'ajout de méthodes à cette table.

2.4.1. Types et méthodes nécessaires à l'export

Pour chaque méthode exportée, une instance du method_caller adéquat est associée au nom de la méthode dans un std::map. Nous stockons aussi une référence vers la table de la classe mère, pour pouvoir remonter dans la hiérarchie lors de la recherche d'une méthode. Toutes ces informations sont stockées dans la structure method_list ci-dessous :

```
/** La liste des méthode appelables. */
template<typename Context>
struct method_list
{
  /** La liste de la classe mère. Cela implique
qu'il n'y a pas d'héritage
      multiple pour l'instant. */
  method_list<Context> const* parent;

  /** La table des exécuteurs. */
  std::map<std::string, method_caller<Context>
const*> data;

}; // struct method_list
```

Enfin, pour simplifier la déclaration de la table des exécuteurs, nous définissons une macro pour l'effectuer. Cette macro est à utiliser dans la définition de toutes les classes pour lesquelles nous voulons exporter des méthodes. Elle suppose que le type script_context_type est un alias du type du contexte passé aux exécuteurs. Elle déclare trois éléments :

- une variable membre statique de type method_list<Context>, définie ci-dessus ;
- une méthode statique void self_methods_set(const std::string&, method_caller<Context> const*) permettant d'ajouter une méthode dans le method_list de cette classe ;
- une méthode virtuelle method_list<Context> const* get_method_list() const qui retourne la liste des méthodes exportées définie dans le premier point.

Du fait que cette dernière méthode soit virtuelle, nous pourrons récupérer depuis base_exportable::execute() la liste des méthodes exportées de la classe la plus basse dans la hiérarchie. Si la méthode demandée n'est pas dans cette liste, nous pourrons accéder à la liste de la classe mère en passant par method_list<Context>::parent.

La première partie de cette macro est définie ci-dessous :

```
/** Cette macro, déclare les variables membres
statiques utilisées pour
      stocker les exécuteurs. Elle est appelée
automatiquement par
      DECLARE_METHOD_LIST. */
#define DECLARE_METHOD_LIST_BASE                \
  protected:                                    \
  typedef method_list<script_context_type>
method_list_type;                               \
  typedef                                       \
  std::map<std::string,
method_caller<script_context_type> const*>      \
  method_list_data_type;                        \
                                                \
  static method_list_type s_method_list;        \
                                                \
  static void self_methods_set                  \
  ( const std::string& name,                    \
   method_caller<script_context_type> const* m )\
  { s_method_list.data[name] = m; }             \
                                                \
  private:                                      \
  virtual method_list_type const*
get_method_list() const                         \
  {                                             \
    init_method_list();                         \
    return &s_method_list;                      \
  }
```

Remarquons la présence d'un appel à une méthode init_method_list depuis get_method_list. Cette méthode a pour rôle d'initialiser la table s_method_list. Nous avons alors besoin de connaître la classe mère pour initialiser le membre method_list<Context>::parent. Or nous ne savons pas ici quelles sont les méthodes à exporter. Nous demanderons donc le nom d'une méthode statique à appeler, qui se chargera de faire des appels à self_methods_set pour ajouter les méthodes. Là encore, nous masquons tout cela dans une macro définie ci-dessous :

```
/** Cette macro, appelée dans la définition d'une
classe, déclare les
    variables membres statiques utilisées pour
stoker les exécuteurs.

    \param parent_class Le type de la classe mère
(héritant de ou étant
```

```
            base_exportable)
    \param export_func Une méthode statique à
appeler pour initialiser
            la liste des exécuteurs.
*/
#define DECLARE_METHOD_LIST(parent_class,
export_func)                                    \
  DECLARE_METHOD_LIST_BASE                      \
                                                \
  static void init_method_list()                \
  {                                             \
    if (s_method_list.parent == NULL)           \
      {                                         \
        parent_class::init_method_list();       \
        s_method_list.parent =
&parent_class::s_method_list;                   \
        export_func();                          \
      }                                         \
  }
```

Remarquons que cette macro appelle la précédente. Elle sera donc la seule à devoir être appelée dans la définition des classes de l'utilisateur.

Seule la macro DECLARE_METHOD_LIST_BASE sera appelée dans base_exportable. En effet, celle-ci va contenir une table de méthodes exportées et doit définir la méthode virtuelle get_method_list(). Cependant, elle n'a pas de classe mère à passer à DECLARE_METHOD_LIST. Nous devrons alors définir la méthode base_exportable<Context>::init_method_list directement dans la classe.

Enfin, la table s_method_list doit être implémentée quelque part, ce que l'utilisateur fera en appelant la macro ci-dessous dans le fichier d'implémentation de sa classe.

```
/** Cette macro est à utiliser dans le fichier
d'implémentation de votre
    classe. */
#define IMPLEMENT_METHOD_LIST(self_type)        \
  method_list<self_type::script_context_type>
self_type::s_method_list;
```

Nous pouvons maintenant compléter la classe base_exportable en déclarant une table d'exécuteurs et en écrivant la méthode execute :

```
/** Cette classe est la base de toutes les
classes pouvant exécuter des
    commandes. */
template<typename Context>
class base_exportable
{
public:
  // Le type du contexte. Déclaration
indispensable pour l'utilisation de
  // DECLARE_METHOD_LIST ci-après.
  typedef Context script_context_type;

  DECLARE_METHOD_LIST_BASE

public:
  /** Destructeur indispensable pour pouvoir
utiliser le dynamic_cast dans
      explicit_method_caller. */
  virtual ~base_exportable() {}
```

```
  /** Exécute une méthode de la classe. \a n est
le nom de la méthode, \a args
      ses paramètres. \a context est le contexte
d'exécution. */
  void execute( const std::string& n, const
std::vector<std::string>& args,
                const script_context_type&
context )
  {
    // on récupère la liste des exécuteurs dans
la classe la plus basse dans
    // la hiérarchie.
    const method_list_type*
m( get_method_list() );
    typename
method_list_data_type::const_iterator it(m-
>data.find(n));

    bool stop(false);

    while( !stop )
      if ( it == m->data.end() )
        {
          // On n'a pas trouvé la méthode, on
passe alors à la
          // liste de la classe mère, si elle
existe.
          if ( m->parent != NULL )
            {
              m = m->parent;
              it = m->data.find(n);
            }
          else
            {
              std::cout << "Method '" << n << "'
not found." << std::endl;
              stop = true;
            }
        }
      else
        {
          // On a trouvé la bonne méthode, on
l'exécute.
          stop = true;
          it->second->execute(this, args,
context);
        }
  }

}; // class base_exportable
```

base_exportable::execute commence par récupérer les méthodes exportées par la classe la plus basse dans la hiérarchie. Elle y cherche la méthode demandée puis l'exécute. Si la méthode demandée n'est pas exportée par cette classe, elle effectue la recherche dans la classe mère, jusqu'à la trouver ou arriver en haut de la hiérarchie.

La dernière étape est d'ajouter des méthodes dans la table des méthodes exportées. Cela est le sujet de la section suivante.

En pratique, la gestion des méthodes est similaire au système des vtables et vpointers (Lien48) créés par le compilateur pour les méthodes virtuelles. Ici, la table s_method_list correspondrait à la vtable tandis que la méthode get_method_list serait le vpointer de l'instance. Par conséquent, il est possible ici de rendre virtuelles des méthodes qui ne le sont pas. Par exemple, supposons une

classe base ayant une méthode f non virtuelle et une classe dérivée héritant de base et redéfinissant la méthode f. Si ces deux classes exportent cette méthode sous le même nom, alors celle de dérivée sera appelée sur toute instance de dérivée, même en passant par le type base. En pratique, il est tout à fait possible de masquer n'importe quelle méthode d'une classe mère en exportant une autre méthode avec le même nom.

2.4.2. Ajout de méthodes à la table

L'ajout d'une méthode dans la table se fait en appelant base_exportable<Context>::self_methods_set() lors de l'appel de la méthode donnée en second paramètre à la macro DECLARE_METHOD_LIST. Il suffit de lui donner le nom et le method_caller adéquat, obtenu à partir des classes method_caller_args_0 et similaires.

Par exemple, dans le cadre de la classe image, le résultat pourrait ressembler au code ci-dessous :

```
class image:
 public base_exportable<my_context>
{
  DECLARE_METHOD_LIST(base_exportable<my_context>
, set_exported)

private:
  void miroir();
  void tourner( double a, int cx, int cy );
  void copier( const image& img, int x, int y );

private:
  static void set_exported()
  {
    self_methods_set
    ( "miroir",
      &method_caller_args_0<image, void,
&image::miroir>::s_caller );
  }
}; // class image
```

Écrit de cette façon, l'ajout de la méthode n'est pas très élégant. D'une part, le nom de la méthode est à indiquer plusieurs fois, d'autre part, l'utilisation de s_caller n'est peut-être pas intuitive. Nous allons donc à nouveau utiliser des macros pour simplifier la syntaxe.

Le code ci-dessous est celui de la macro déclarant un method_caller_args_0, pour appeler une méthode sans paramètres. Le premier paramètre est le type de la classe exportant la méthode, le second est le nom de la méthode et le troisième est son type de retour.

```
/** Cette macro est utilisée dans
l'implémentation d'une classe dérivant de
      base_exportable pour ajouter un exécuteur
d'une méthode sans paramètres.
  */
#define CONNECT_METHOD_0( self_type, method_name,
R )                                              \
  self_methods_set                               \
  ( #method_name,                                \
    &method_caller_args_0                        \
    <                                            \
    self_type,                                   \
    R,                                           \
    &self_type::method_name>::s_caller )
```

Pour reprendre l'exemple précédent, il suffit d'utiliser cette macro dans la méthode image::set_exported pour exporter la méthode :

```
class image:
 public base_exportable<my_context>
{
  DECLARE_METHOD_LIST(base_exportable<my_context>
, set_exported)

private:
  void miroir();
  void tourner( double a, int cx, int cy );
  void copier( const image& img, int x, int y );

private:
  static void set_exported()
  {
    CONNECT_METHOD_0( image, miroir, void );
  }
}; // class image
```

La macro ci-dessous effectue la même chose pour un method_caller_args_2. Nous pouvons remarquer deux nouveaux paramètres, qui sont les types des paramètres de la méthode à appeler.

```
/** Cette macro est utilisée dans
l'implémentation d'une classe dérivant de
      base_exportable pour ajouter un exécuteur
d'une méthode ayant deux
      paramètres, de type T1 et T2. */
#define CONNECT_METHOD_2( self_type, method_name,
R, T1, T2 )                                      \
  self_methods_set                               \
  ( #method_name,                                \
    &method_caller_args_2                        \
    <                                            \
    self_type,                                   \
    R, T1, T2,                                   \
    &self_type::method_name >::s_caller)
```

Comme pour la définition de method_caller, il faudra déclarer une macro pour chaque nombre de paramètres des méthodes à exporter.

3. Utilisation du système

Tous les éléments du système d'export ont maintenant été déclarés, il ne reste plus qu'à les utiliser. Nous utiliserons un contexte nommé my_context pour les classes exportables de notre programme, du type my_exportable déclaré comme suit :

```
/** my_context est le contexte d'exécution de nos
scripts, transportant les
    informations dont nous avons besoin pour leur
exécution (voir
    my_context.hpp). */
class my_context;

/** La classe de base des classes que nous
voulons exporter. */
typedef script::base_exportable<my_context>
my_exportable;
```

Dans l'exemple situé au début de l'article, nous pouvons

remarquer que des images sont passées en paramètre à des méthodes et identifiées par leur nom. Pour pouvoir faire la conversion de ce nom en une image en mémoire, nous allons devoir faire l'association dans notre contexte et spécialiser la classe string_to_arg.

Le contexte se présente alors comme suit :

```
/* prédéclaration de la classe des images pour
pouvoir l'utiliser dans
   my_context. */
class image;

/** Un contexte spécifique à notre programme de
traitement d'images. */
class my_context
{
public:
  /** Associe une image à un nom. */
  void add( const std::string& n, image* img )
  {
    m_images[n] = img;
  }

  /** Récupère l'image associée à un nom. */
  image* get_image( const std::string& n ) const
  {
    m_images.find(n)->second;
  }

private:
  /** Les images manipulées. */
  std::map<std::string, image*> m_images;

}; // class my_context
```

Enfin, pour que les paramètres de type image soient retrouvés automatiquement à partir de leurs noms, nous effectuons la spécialisation ci-dessous de string_to_arg.

```
/** Spécialisation pour retourner une image à
partir de son nom. On fait une
    prédéclaration ici pour que le compilateur
sache qu'elle existe. */
template<>
class string_to_arg<my_context, const image&>
{
public:
  static const image&
  convert( const my_context& context, const
std::string& arg )
  {
    return *context.get_image(arg);
  }
}; // string_to_arg
```

Le contexte étant défini, il ne reste plus qu'à exporter les méthodes de la classe image. Dans le fichier d'entête :

```
class image:
 public my_exportable
{
  DECLARE_METHOD_LIST(my_exportable,
set_exported)

private:
  void miroir();
  void tourner( double a, int cx, int cy );
  void copier( const image& img, int x, int y );

  static void set_exported()
  {
    CONNECT_METHOD_0( image, miroir, void );
    CONNECT_METHOD_3( image, tourner, void,
double, int, int );
    CONNECT_METHOD_3( image, copier, void, const
image&, int, int );
  }
}; // class image
```

Dans le fichier source, nous devons aussi ajouter l'appel de macro ci-dessous pour implémenter la liste des méthodes exportées :

```
IMPLEMENT_METHOD_LIST(image)
```

Et voilà, les méthodes de la classe image peuvent être aisément appelées sous forme de texte. Pour nous en convaincre, essayons ce petit programme :

```
int main()
{
  image img1, img2;
  my_context context;

  context.add("image_1", &img1);
  context.add("image_2", &img2);

  std::vector<std::string> args;
  args.push_back("image_2");
  args.push_back("24");
  args.push_back("18");

  img1.execute("copier", args, context);

  return 0;
}
```

4. Conclusion

Nous avons mis en place un système permettant d'appeler des méthodes de certaines classes en donnant sous forme de chaînes de caractères leur nom et leurs paramètres. Ce système remplit correctement sa tâche, son utilisation reste très simple et permet d'avoir des résultats satisfaisants rapidement, pour peu que les classes d'exécuteurs de méthodes soient correctement déclarées, de même que les macros permettant d'exporter les méthodes. En particulier, nous avons utilisé ce système pour ajouter des scripts dans notre jeu Plee the Bear (Lien49).

Cependant, ce système a plusieurs inconvénients. Tout d'abord, nous pouvons remarquer que la valeur de retour des fonctions exportées est totalement ignorée et perdue dans les appels. De plus, le système est intrusif : il est nécessaire que les classes exportées héritent de base_exportable. Ainsi, il n'est pas possible d'exporter des classes que l'utilisateur ne peut modifier (objets d'une bibliothèque annexe, par exemple). Le lecteur souhaitant un système moins intrusif pourra s'intéresser à CAMP (Lien50). Si l'objectif est un système de script, l'utilisateur pourra aussi s'orienter vers LUA (Lien51), pour lequel un tutoriel est présent sur developpez.com (Lien52).

Retrouvez l'article de Julien Jorge en ligne : Lien53

Programmation iPhone 3. Conception, développement et publication

La réussite d'une application iPhone repose sur sa conception et sa réalisation : elle exige un savoir-faire en ergonomie mobile et la maîtrise de l'ensemble des contraintes spécifiques à la plate-forme. La référence du développeur iPhone professionnel : de la conception à la publication sur l'App Store

De la conception de l'application - encadrée par de strictes règles d'ergonomie - jusqu'à son déploiement, cet ouvrage détaille les bonnes pratiques garantissant la qualité de vos développements iPhone : gestion de projet et architecture MVC, ergonomie mobile et design patterns d'interface. Les fondamentaux du développement iPhone sont détaillés, de l'Objective-C et sa gestion spécifique de la mémoire aux contrôleurs de vue, en passant par la mise en place des vues et des TableView.

Écrit par le directeur technique de l'une des premières agences spécialisées dans le développement sur plate-forme mobile et iPhone, l'ouvrage traite en profondeur d'aspects avancés tels que l'accès aux services web iJSON, XMLI, la gestion de flux audio et vidéo, la persistance avec le framework CoreData et l'utilisation du service de notifications Apple. Enfin, il fournit de précieux conseils pour publier sur l'App Store et y gagner en notoriété.

Couvre les nouveautés de la version 3 de l'iPhone OS.

A qui s'adresse cet ouvrage ?

- Aux professionnels de la conception web et mobile qui souhaitent être présents sur le marché des services portés sur iPhone ;
- A tous les particuliers et fans d'iPhone qui souhaitent concevoir, publier ou vendre une application sur l'App Store.

Critique du livre par Jean-Marie Macé

Programmation iPhone 0S 3, est le livre idéal pour toute personne souhaitant s'investir sur la plateforme iPhone. En effet, Thomas Sarlandie, est un développeur français ayant déjà beaucoup d'expérience sur le sujet et nous la fait partager tout au long de son ouvrage d'une façon simple et concise.

On apprécie tout au long du livre, les analogies avec le Java qui permettent de se rattacher à quelque chose de connu. La progression dans l'ouvrage est bien sentie, et aborde tous les points essentiels lors d'un développement applicatif pour iPhone. À terme, on peut laisser le bouquin en guise de pense bête, et s'appuyer sur le centre de développement Apple qui se suffit presque à lui même, comme l'indique l'auteur.
L'ouvrage est, pour cela, une excellente introduction au centre de développement Apple.

Seul petit oubli, selon moi, les exercices pratiques.
Certes, le centre Apple possède suffisamment d'exemple, mais un essai grandeur nature avec le livre aurait été parfait.

Enfin, on apprécie le dernier chapitre sur l'aspect plus "commercial" de l'après développement, les petites idées sur le marketing, ...
Bref, c'est bien vu!

Retrouvez cette critique de livre sur la page livres Mac : Lien54

Le guide de survie : AppleScript

L'essentiel du langage et de ses applications : Finder, Os X, iApps, iWork, Adobe CS, Office, XPress

Ce Guide de survie est l'outil indispensable pour contrôler votre Mac et les applications qu'il exécute avec AppleScript.
Vous pourrez ainsi automatiser tout travail répétitif, simple ou complexe, et créer des séquences de tâches personnalisées.

Concis et Maniable.

Facile à transporter, facile à utiliser -- finis les livres encombrants !

Pratique et Fonctionnel

Une bibliothèque d'extraits de code, classées par applications, à exploiter en l'état ou à insérer dans vos propres scripts !

Jean-Philippe Moreux, ingénieur INSa Toulouse (informatique), est éditeur scientifiqie. Il a coécrit aux mêmes éditions Automatisez vos tâches sous Mac OS X.

Aurérien Gaymay est technicien informatique et utilise AppleScript dans son entreprise. Il est coresponsable de la rubrique Mac du site www.developpez.com et rédacteur pour le site www.logicielmac.com.

Critique du livre par Vincent Brabant

Prendre le temps de lire ce livre consacré à l'AppleScript ne fut en aucun cas une perte de temps.

Après nous avoir fait découvrir, lors de la première partie, les bases du langage AppleScript, sa syntaxe quelque peu familière, et l'outil de développement qu'Apple met à notre disposition, nos deux auteurs nous explique comment, à l'aide d'AppleScript, nous pourrons faire obéir au doigt et à l'oeil non seulement notre système d'exploitation, mais aussi toute une série d'applications, que ce soit iTunes, iPhoto, Quicktime, iCal, iMail, mais aussi iWork, MS Office, Adobe CS, XPress.

Cet ouvrage est abondamment illustré de bout de code AppleScript qu'il nous sera aisé de combiner ensemble pour créer nos propres script AppleScript répondant à nos besoins.

De plus, la mise en page est vraiment très agréable, rendant la lecture de ce livre plus agréable.

Et une fois qu'on a pu prendre en main AppleScript, et qu'on commence à écrire ses scripts, la table des matières fortement détaillée (bien plus que celle reprise ci-dessous qui ne donnent que les grandes lignes) et la richesse de son index font qu'on retrouvera facilement le script dont on a besoin pour effectuer une action bien précise.

Vraiment, si vous n'avez encore jamais utilisé AppleScript, c'est le moment de s'y mettre. Vous ne regretterez pas d'avoir perdu du temps à lire ce livre. Car ce temps "perdu" sera bien vite récupéré une fois que vous aurez écrit vos scripts répondant à vos besoins journaliers.

Mac OS X 10.6 Snow Leopard Efficace

Si Mac OS X brille par son confort d'utilisation et son interface intuitive aux nombreux effets graphiques, il demeure un système d'exploitation de la famille Unix, puissant et complet.

Cet ouvrage présente les bonnes pratiques qui feront de vous un expert de Snow Leopard.

Exploitez toutes les dimensions de votre Mac

- Installez et personnalisez le système
- Gagnez en efficacité avec le Finder, le Dock, Spotlight et Exposé
- Bénéficiez de toutes les applications connectées, en Wi-Fi, 3G, ou en réseau avec des PC
- Synchronisez vos e-mails, calendriers et carnets d'adresses grâce à MobileMe et à MS-Exchange
- Protégez et sauvegardez vos données
- Sécurisez l'accès aux données en les chiffrant avec Filevault et en créant des comptes utilisateurs
- Dépannez vos applications et votre système
- Réussissez la délicate gestion des polices et des imprimantes
- Apprenez à automatiser les tâches avec Automator et AppleScript

À qui s'adresse cet ouvrage ?

- Aux passionnés de Mac qui souhaitent découvrir Snow Leopard
- Aux utilisateurs de PC qui désirent passer à Mac OS X
- À tous ceux qui, dans un cadre professionnel, doivent retrouver leurs marques dans l'univers Apple

Avec un préface de Philippe Nieuwbourg.

Critique du livre par Aurélien Gaymay

Ce livre est pour moi une référence dans les livres dédiés à Mac OS X.
C'est le livre qui devrait être livré avec tous les Macs.
C'est vraiment le mode d'emploi ultime.

Ce livre est conçu pour les nouveaux utilisateurs Mac, les habitués, mais aussi les plus expérimentés.

Dans ce livre, vous découvrirez des tonnes d'astuces et d'informations qui faciliteront l'utilisation de votre Mac.
Vous apprendrez à optimiser votre temps de travail avec une meilleure gestion des différents outils qu'Apple propose, comme la gestion du Carnet d'Adresse, d'iCal (calendrier), Mail, etc...

Je vous conseille vivement la lecture de ce livre. Il est destiné aussi bien aux utilisateurs de Mac qu'aux utilisateurs Windows, qui voudraient passer au Mac.

Retrouvez cette critique de livre sur la page livres Mac : Lien54

Tester du code intestable avec JMockit

JMockit est un framework de mocks pour les tests unitaires. En plus de proposer les fonctionnalités habituelles de mocking, il permet de poser des tests sur du code dit intestable. Absolument tout est mockable : les méthodes statiques, les initialiseurs statiques, les constructeurs et même les méthodes privées.

1. Qu'est-ce que JMockit ?

JMockit est un framework de mocking pour les tests unitaires : il permet de créer des simulacres (Lien55).

Encore un ? Il y en a déjà et ils font bien leur travail : JMock, Easymock et Mockito par exemple. C'est exact, ces frameworks sont très bien... *quand le code est testable*. Les applications sur lesquelles nous travaillons sont malheureusement envahies de composants intestables. Cela constitue autant de failles si nous nous résignons à ne pas les tester. Comme le montre Misko Hevery dans son guide (Lien56), un code testable comporte les caractéristiques suivantes :

- le constructeur ne fait aucun travail : il se contente d'assigner les paramètres aux attributs de la classe ;
- il n'y a pas d'état global : pas de singleton, de variable statique, de méthode statique ;
- les méthodes ne "creusent" pas les collaborateurs pour obtenir d'autres objets. Les collaborateurs sont directement *utilisés*, ils ne servent pas d'intermédiaire pour atteindre l'objet dont on a réellement besoin ;
- une classe a peu, voire une seule responsabilité

Les deux derniers critères complexifient l'écriture de tests mais ne la rendent pas impossible. Le fait qu'il n'y ait pas de référence sur les collaborateurs du fait d'instanciation explicite d'objets ou qu'il y ait des états globaux empêche réellement toute tentative de test : nous ne pouvons simplement *pas* injecter de mock.

```
public class MyClass
{
  public MyClass()
  {
    MyService service = new HttpService();
    // whatever
  }
}
```

Nous ne voulons certainement pas nous traîner un véritable *HttpService* dans nos tests, avec ce que sa construction implique (un client HTTP qui tourne par exemple). Pour pouvoir injecter un faux *MyService*, nous transformons la variable *service* en attribut de classe et ajoutons un paramètre dans le constructeur.

```
public class MyClass
{
  private MyService service;

  public MyClass(MyService service)
  {
    this.service = service;
    //whatever
  }
}
```

Extrait du test

```
MockService mockService = new MockService();
MyClass myClass = new MyClass(mockService);
```

Quand nous rencontrons ce genre de code, il faut donc respirer un bon coup et **refactorer**. Mais souvent, *le client préfère que l'on travaille sur de nouvelles features plutôt que du refactoring sans valeur visuelle*, *le temps presse*, *l'équipe est déjà en retard*, *cette fonctionnalité n'existera plus dans un an*, *une refonte technique est déjà prévue*, *personne ne sait ce que ce code fait mais il marche alors pas touche*, *le code spaghetti me donne des nausées*... Autant de raisons font que le refactoring n'est pas toujours possible.

C'est ici que JMockit intervient : il permet de définir des mocks pour des classes, même si nous n'avons **pas de référence dessus**. Il ne s'agit plus de créer *une* instance de mock mais de mocker *toutes les instances* d'une classe. Ainsi, les états globaux ne sont plus un problème. Le framework permet également de **mocker des constructeurs, des méthodes statiques, des blocs statiques**. Les instanciations directes ne nous gênent plus non plus. JMockit peut même mocker des **méthodes privées** !

Le framework est constitué de plusieurs API : JMockit Core, JMockit Annotations, JMockit Expectations et depuis peu JMockit Verifications. Ce tutoriel n'évoquera pas tous les aspects du framework, loin de là, mais suffisamment pour vous donner une idée de sa puissance et l'envie d'aller plus loin.

Les exemples ont été testés avec JMockit 0.993 sous une JDK6.

Le site officiel : Lien57

Documentation d'installation officielle : Lien58

Le tutoriel officiel : Lien59

2. Installation de JMockit

Première étape : télécharger le JAR sur le site de JMockit : Lien60.

Parce qu'il s'appuie sur l'instrumentation, **JMockit ne fonctionne que depuis Java 5**.

2.1. Classiquement

Le jar de JMockit contenant le nécessaire, il suffit de l'ajouter au classpath.

2.2. Avec Maven

JMockit est hébergé sur java.net : Lien61.

Pour l'utiliser, ajouter l'artefact dans le *pom.xml* :

Artefact jmockit
```
<dependency>
        <groupId>mockit</groupId>
        <artifactId>jmockit</artifactId>
        <version>0.993</version>
</dependency>
```

3. Lancer les tests

3.1. Sous Java 5

Pour lancer les tests avec JMockit sous Java 5, la JVM a besoin de savoir où est le JAR via l'option *javaagent*.

3.1.1. Avec Eclipse

Au moment de lancer les tests, dans le runner, il faut ajouter l'argument à la JVM :

```
-javaagent:C:\.m2\repository\mockit\jmockit\0.993
\jmockit-0.993.jar
```

Le recours à une variable de substitution facilite les montées de version :

```
-javaagent:${JMOCKIT_JAR}
```

3.1.2. Avec Maven

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <argLine>-javaagent:"${settings.LocalReposito
ry}/mockit/jmockit/0.993/jmockit0.993.jar"</argLi
ne>
    <useSystemClassLoader>true</useSystemClassLoa
der>
  </configuration>
</plugin>
```

3.2. Sous Java 6

Il n'y a rien à faire : ni d'annotation *@RunWith* à mettre sur la classe, ni de *javaagent* à spécifier.

Si vous développez sous Mac OS X, ajoutez *<jdkDir>/lib/tools.jar* à votre classpath, où *<jdkDir>* est la racine de votre JDK 6 locale.

Si JMockit ne fonctionne pas sous votre environnement en dépit de votre JDK 6, ajoutez le paramètre *javaagent* comme pour une JDK 5. Cela arrive notamment avec la version IBM J9 JDK 1.6.

Si vous rencontrez d'autres problèmes, la documentation d'install officielle (Lien62) peut vous aider.

Pour information, les anciennes versions de JMockit nécessitaient de restaurer manuellement les définitions originales. Autrement, les mocks définis dans les tests précédents perturbaient le test courant.

```
@After
public void tearDown() {
    // to avoid pertubation on other junit test
    Mockit.restoreAllOriginalDefinitions();
}
```

Cette précaution n'est plus nécessaire.

4. JMockit Core et JMockit Annotations

JMockit Core permet notamment de substituer tous les appels à une classe A par une classe B, B étant une classe de mock. Cette API a le mérite d'être simple à comprendre, mais rend le test un peu moins lisible dans la mesure où il faut *déclarer une classe de mock*.

JMockit Annotations présente les mêmes possibilités que Core, avec en plus des vérifications sur le nombre de fois où une méthode est invoquée. Comme les appels sont vérifiés, il est possible d'aussi vérifier les *paramètres passés*. L'API a cependant le même problème de lisibilité que JMockit Core, pour la même raison.

Dans les deux cas, si une méthode de la classe A est appelée et que B ne la définit pas, alors c'est la méthode réelle qui est appelée. Aussi, il est plus juste de parler de *redéfinition de méthodes* que de *redéfinition de classe*, que l'on peut voir dans les autres frameworks de mocks.

4.1. Redéfinir une méthode

L'idée est de dire qu'à chaque fois qu'une classe sera sollicitée, ce sera notre mock qui sera utilisé à sa place. Cela est pratique quand la classe que l'on souhaite mocker n'est pas accessible. Nous souhaitons tester la méthode *doOperation* :

```
package fr.fchabanois.tutorial.jmockit;

public class MyService
{
  public void doOperation(String value)
  {
    DatabaseLogger logger = new DatabaseLogger();
    logger.info("starting operation : " + value);

    // Execution de l'opération
    // ....
    // ....
    // ....
```

```
  }
}
```

Nous ne souhaitons pas que notre test écrive quoi que ce soit en base à chaque fois qu'il est lancé. Or le *databaseLogger* n'est pas une instance passée en paramètre, ce qui fait que nous ne pouvons pas le mocker tel que le code est écrit. L'idéal serait de refactorer le code, mais nous ne pouvons parfois pas nous le permettre.

4.1.1. Avec JMockit Core

JMockit Core permet de résoudre ce genre de problématique avec la méthode redefineMethods.

```
package fr.fchabanois.tutorial.jmockit;

import mockit.Mockit;

import org.junit.Test;

public class MyServiceTest
{
  @Test
  public void testDoOperation_nominalCase()
  {
    MyService service = new MyService();
    // Definition du mock :
    //  la classe MockDatabaseLogger sera utilisé
à la place de DatabaseLogger
    //  à chaque fois que cette dernière sera
appelée
    Mockit.redefineMethods(DatabaseLogger.class,
MockDatabaseLogger.class);

    service.doOperation("anything");
  }

  public static class MockDatabaseLogger
  {
    public Object info(String sentence) {
      System.out.println("MOCK
DatabaseLogger.info()");
      return null;
    }
  }
}
```

En lançant le test, nous constatons que c'est bien la méthode *info* de ***MockDatabaseLogger*** qui est appelée grâce à la console :

```
MOCK DatabaseLogger.info()
```

Techniquement, vous pouvez valider les paramètres passés dans le mock et retourner de faux objets. Notez que ce n'est pas forcément pertinent car JMockit Core ne vérifie pas que la méthode est appelée. Si vous avez besoin de vérifier l'appel, orientez-vous plutôt vers JMockit Annotations ou JMockit Expectations. Voici un exemple de vérification de paramètre :

```
public static class MockDatabaseLogger
{
  public Object info(String sentence) {
    System.out.println("MOCK
DatabaseLogger.info()");
    assertThat(sentence, is("starting operation :
" + sentence));
    return null;
  }
}
```

et un autre encore plus précis :

```
package fr.fchabanois.tutorial.jmockit;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import mockit.Mockit;

import org.junit.Test;

public class MyServiceTest
{
  @Test
  public void testDoOperation_nominalCase()
  {
    MyService service = new MyService();
    // Definition du mock
    Mockit.redefineMethods(DatabaseLogger.class,
MockDatabaseLogger.class);
    final String param = "anything";
    MockDatabaseLogger.expectedParam = "starting
operation : " + param;

    service.doOperation(param);
  }

  public static class MockDatabaseLogger
  {
    public static String expectedParam;

    public Object info(String sentence) {
      System.out.println("MOCK
DatabaseLogger.info()");
      assertThat(sentence, is(expectedParam));
      return null;
    }
  }
}
```

Maintenant, que se passe-t-il si une méthode de *DatabaseLogger* est appelée mais que *MockDatabaseLogger* ne la définit pas ? Testons.

DatabaseLogger comporte aussi une méthode intitulée *trace()* :

```
public void trace(String sentence) {
  System.out.println("DatabaseLogger.trace - " +
sentence);
}
```

Modifions la classe de service pour invoquer *trace*.

```
public class MyService
{
  public void doOperation(String value)
  {
    DatabaseLogger logger = new DatabaseLogger();
    logger.info("starting operation : " + value);

    // Execution de l'opération
    // ....
    // ....
```

```
    // ....
    logger.trace("coucou");
  }
}
```

La console affiche...

```
MOCK DatabaseLogger.info()
DatabaseLogger.trace - coucou
```

En conclusion, si le mock ne redéfinit pas la méthode, alors **c'est la méthode de l'objet original qui est appelée**.

4.1.2. Avec JMockit Annotations

Nous venons de voir comment redéfinir une classe (*DatabaseLogger*) qu'une autre classe (*MyService*) utilisait.

Or, le fait de pouvoir redéfinir des *méthodes* nous permet d'aller plus loin : de tester l'invocation d'une **méthode de la classe que nous sommes en train de tester**. Il s'agit en quelque sorte d'un *mock partiel*.

Testons la méthode *doSport()* de *HumanBeing*. Nous voulons que cette dernière appelle la méthode *dress()*.

```
public class HumanBeing extends
AbstractLivingBeing {
  protected void doSport(){
  }

  protected void dress(){
  }
}
```

Autrement dit, nous avons besoin de solliciter la véritable méthode *doSport()* mais de mocker *dress()*. Une tactique pour contourner ce problème est d'étendre la classe testée et de surcharger la méthode que l'on souhaite mocker.

```
public static class MockHumanBeingDress extends
HumanBeing {
  static boolean dressHasBeenCalled = false;

  protected void dress(){
    dressHasBeenCalled = true;
  }
}

@Test
public void testDoSport_mockMaison(){
  HumanBeing human = new MockHumanBeingDress();

  human.doSport();

  Assert.assertTrue("dress has not been
called!",MockHumanBeingDress.dressHasBeenCalled);
}
```

JMockit permet de le faire de façon plus élégante avec les annotations, sans avoir besoin de déclarer une variable temporaire pour flaguer l'appel de la méthode *dress* :

```
@Test
public void testDoSport(){
  HumanBeing human = new HumanBeing();
  Mockit.setUpMocks(MockHumanBeing.class);

  human.doSport();
}

@MockClass(realClass = HumanBeing.class)
public static class MockHumanBeing {
  @Mock(invocations = 1)
  protected void dress(){
  }
}
```

Mockit.setUpMocks (attention au *s* à la fin !) permet de déclarer le mock.

@MockClass précise quelle classe le mock remplacera.

@Mock(invocations = 1) vérifie que la méthode annotée n'est appelée qu'une seule fois.

Admettons qu'en fonction du type de sport, nous souhaitions passer un certain type de vêtement :

```
protected void doSport(String typeOfSport){
}

protected void dress(String typeOfClothes){

}
```

Le test deviendrait :

```
@Test
public void testDoSportWithString(){
        HumanBeing human = new HumanBeing();
        Mockit.setUpMocks(MockHumanBeing_dressWit
hStringCalledOnce.class);
        MockHumanBeing_dressWithStringCalledOnce.
expectedCloth = "short";

        human.doSport("basket");
}

@MockClass(realClass = HumanBeing.class)
public static class
MockHumanBeing_dressWithStringCalledOnce {
        static String expectedCloth;

        @Mock(invocations = 1)
        protected void dress(String typeOfCloth){
                Assert.assertEquals("type of
cloth", expectedCloth, typeOfCloth);
        }
}
```

L'implémentation suivante satisfait le test :

```
protected void doSport(String typeOfSport){
  if ("basket".equals(typeOfSport)){
    dress("short");
  }
}

protected void dress(String typeOfClothes){

}
```

4.2. Redéfinir un constructeur

Vraiment, ce n'est pas assez dit : ne faites rien d'autre qu'assigner des attributs dans un constructeur... Malheureusement, les constructeurs que l'on rencontre font souvent bien plus : ils initialisent des objets, invoquent des factory, font des appels statiques. C'est mal parce que pour poser un test sur un objet, nous sommes obligés de l'instancier... et donc de faire avec toute cette mécanique et gérer les effets de bord à *chacun* de nos tests.

JMockit permet de s'affranchir de ces constructeurs maudits grâce à la méthode $init.

4.2.1. Avec JMockit Annotations

Nous avons un travailleur, et souhaitons implémenter sa façon de faire le dîner le lundi :

```
public class WorkingHuman extends HumanBeing {
   public WorkingHuman() {
   }

   protected void makeDinner(String dayOfWeek) {
   }
}
```

Le lundi, c'est *dur*... Alors le travailleur fait chauffer une pizza. Une basique.

```
public class Pizza extends Food {
        public final static String[]
BASIC_INGREDIENTS = new String[]
{"cheese","tomato"};

        public Pizza(String...ingredients) {
        }
}
```

Autrement dit, nous souhaitons vérifier que lorsqu'un travailleur fait le dîner (*makeDinner*) :

- le constructeur de *Pizza* soit appelé ;
- les paramètres du constructeur soient les ingrédients les plus basiques.

En test, cela donne :

```
@Test
public void testMakeDinner() {
  WorkingHuman human = new WorkingHuman();

  Mockit.setUpMocks(MockPizza_ConstructorCalledOn
ce.class);
  MockPizza_ConstructorCalledOnce.expectedIngredi
ents = Pizza.BASIC_INGREDIENTS;

  // Method tested
  human.makeDinner("monday");
}

@MockClass(realClass = Pizza.class)
public static class
MockPizza_ConstructorCalledOnce {
  static String[] expectedIngredients;
  // Can't be static !!!
  Pizza it;

  @Mock(invocations = 1)
  public void $init(String ... ingredients) {
    assertThat(ingredients,
is(equalTo(expectedIngredients)));
  }
}
```

Mockit.setUpMocks : active les mocks passés en paramètres. Attention au *-s* à la fin de *SetUpMocks* (rien à voir avec *SetUpMock*).

@MockClass(realClass = Pizza.class) : définit un mock et la classe réelle qu'il remplace.

Variable d'instance **it** : permet d'avoir une référence sur l'objet construit, comme le constructeur ne peut rien renvoyer. C'est utile pour vérifier que l'objet est bien passé à une autre méthode par exemple. Attention à ne pas rendre la variable statique car cela ne fonctionnerait pas.

@Mock(invocations = 1) : spécifie qu'une méthode doit être appelée une et une seule fois.

$init : méthode appelée à la place du constructeur, pour le mocker. N'oubliez pas le **$** au début d'*$init*.

4.2.2. Avec JMockit Core

C'est exactement la même méthode *$init* à implémenter. Si une référence sur l'objet instancié est nécessaire, une variable d'instance nommée *it* la contiendra de la même façon.

Par rapport à JMockit Annotations, il n'est pas possible de vérifier les invocations de méthodes. Si le constructeur n'est pas appelé, il n'y aura pas d'erreur... Par contre, si le constructeur est appelé avec les mauvais paramètres, ce sera détecté.

Version avec JMockit Core

```
@Test
public void testMakeDinner_core() {
        WorkingHuman human = new WorkingHuman();

        Mockit.redefineMethods(Pizza.class,
MockPizza_ConstructorCore.class);
        MockPizza_ConstructorCore.expectedIngredi
ents = Pizza.BASIC_INGREDIENTS;

        // Method tested
        human.makeDinner("monday");

}

public static class MockPizza_ConstructorCore {
        static String[] expectedIngredients;
        // Can't be static !!!
        Pizza it;

        public void $init(String ... ingredients)
{
            assertThat(ingredients,
is(equalTo(expectedIngredients)));
        }
}
```

En conséquence, JMockit Core est plus utile pour faire une fausse Pizza, pour avoir un bouchon (Lien63), que pour effectuer des assertions.

L'auteur de JMockit signale sur son site (Lien64) qu'il y a trois avantages à utiliser *$init* pour mocker le constructeur, plutôt que d'avoir un constructeur de mock directement (*new MockPizza(){}*) :

- l'attribut *it* est accessible dans *$init* alors que dans un vrai constructeur, il ne serait pas initialisé avant la fin de la construction ;
- la méthode *$init* sera appelée sur l'instance du mock, même s'il a été défini dans le test alors qu'avec *MockPizza*, ce ne serait pas possible car il est impossible d'appeler le constructeur d'une instance qui existe déjà ;
- étant une méthode, *$init* peut être statique, contrairement à un constructeur.

4.3. Redéfinir un bloc statique

Les blocs statiques peuvent devenir un véritable cauchemar lorsque nous essayons d'introduire des tests sur du code existant. Le bloc étant statique, nous avons beau mocker, il est tout de même appelé. Avec JMockit, il devient possible de mocker aussi ces blocs et de les rendre caduques.

La classe *Candidate* contient un bloc d'initialisation statique.

```
public class Candidate
{
  protected static List<String>
minimumQualifications;

  static
  {
    minimumQualifications = new
ArrayList<String>();
    minimumQualifications.add("typing");
    minimumQualifications.add("english
speaking");
  }
}
```

Nous allons écrire un test pour vérifier que le bloc statique est bien redéfini, en vérifiant que *minimumQualifications* est *null*. La méthode magique pour mocker l'initialiseur est la même pour Core et Annotations : public void $clinit.

4.3.1. Avec JMockit Core

JMockit Core est plus approprié lorsque nous ne souhaitons rien vérifier, juste faire en sorte que l'initialisation statique ne vienne pas perturber notre mock.

```
@Test
public void testPizza_staticInitWithCore() throws
Exception {
        Mockit.redefineMethods(Candidate.class,
MockCandidate_staticCore.class);

        Candidate candidate = new Candidate();

        Assert.assertEquals("qualifications
mocked", null, Candidate.minimumQualifications);
}

static class MockCandidate_staticCore {
        void $clinit(){
                // no initialisation of
qualifications
        }
}
```

Mockit.redefineMethods définit la classe par laquelle il faut remplacer *Candidate.class*.

$clinit est appelée à la place du bloc statique de *Candidate*.

Le test passe : le bloc statique n'a effectivement pas été pris en compte.

4.3.2. Avec JMockit Annotations

JMockit Annotation permet d'avoir des attentes particulières en plus : par exemple que le bloc statique est bien appelé.

```
@Test
public void testPizza_staticInitWithAnnotations()
throws Exception {

        Mockit.setUpMocks(MockCandidate_staticAnn
otations.class);

        Candidate candidate = new Candidate();

        // To assert that the static initializer
has been called
        Assert.assertEquals("qualifications
mocked", null, Candidate.minimumQualifications);

}

@MockClass(realClass = Candidate.class)
public static class
MockCandidate_staticAnnotations {

        @Mock(invocations=1)
        void $clinit(){
                // no initialisation of
qualifications
        }
}
```

Mockit.setUpMocks déclare le mock. C'est l'annotation sur la classe de mock qui fait le lien avec la classe mockée : @MockClass(realClass = XXX.class).

@Mock(invocations=1) vérifie que le bloc statique n'est appelé qu'une seule fois.

void $clinit() est appelée à la place du bloc statique de la classe mockée.

Le test passe lui aussi. Et il échoue si nous commentons le bloc statique dans la classe *Candidate*.

Que se passe-t-il s'il y a plusieurs blocs statiques ? Pas grand chose en fait, car le compilateur Java fusionne les séquences des blocs statiques en un seul bloc "<*clinit*>" interne. Tous vos blocs seront donc mockés pareillement dans le *$clinit* que vous aurez redéfini.

4.4. Créer une implémentation vide d'une interface

Il y a les objets que l'on souhaite mocker parce que l'on en attend des choses (que telle méthode soit appelée, avec tel paramètre) et les objets que l'on souhaite simplement bouchonner, pour affranchir notre test unitaire des dépendances de l'objet testé.

La méthode newEmptyProxy de JMockit comblera les besoins de bouchon simple sur une **interface**.

Reprenons le tout premier exemple, en un peu plus enrichi :

```
  public void doOperation(String value)
  {
    DatabaseLogger logger = new DatabaseLogger();
    FullMessage info = logger.info("starting
operation : " + value);

    // Execution de l'opération
    // ....
    // ....
    // ....
    logger.trace("coucou");
  }
```

Nous retournions *null* dans la méthode *logger.info* de notre mock, pour garder les choses simples. Néanmoins, nous pouvons avoir besoin de retourner un objet réel, si nous voulons tester que l'objet retourné est passé en paramètre d'une autre méthode par exemple.

FullMessage est une interface, que *FullMessageImpl* implémente. Mais son instanciation coûte cher (le constructeur a des accès en base ou lit un fichier) et nous n'avons pas de vérification particulière à faire dessus. Nous aimerions retourner un objet plus simple que *FullMessageImpl*, une espèce de coquille vide. C'est possible manuellement, en créant un sous-type de l'interface.

```
        @Test
  public void
testDoOperation_nominalCase_withManualProxy()
  {
    MyService service = new MyService();

    // Definition du mock
    Mockit.redefineMethods(DatabaseLogger.class,
MockDatabaseLoggerWithObjectToReturn.class);
    final String param = "anything";
    MockDatabaseLoggerWithObjectToReturn.expected
Param = "starting operation : " + param;
    MockDatabaseLoggerWithObjectToReturn.objectTo
Return = new FullMessage() {

      @Override
      public void helloTata()
      {
        // TODO Auto-generated method stub

      }

      @Override
      public void helloTiti()
      {
        // TODO Auto-generated method stub
      }

    };

    service.doOperation(param);
  }

  public static class
MockDatabaseLoggerWithObjectToReturn
  {
    public static String expectedParam;
    public static Object objectToReturn;

    public Object info(String sentence)
    {
      System.out.println("MOCK
DatabaseLogger.info()");
      assertThat(sentence, is(expectedParam));
      return objectToReturn;
    }
  }
```

Le gros désavantage est l'encombrement de la solution : le code est brouillé. Il faut implémenter toutes les interfaces et corriger tous les tests si un jour une méthode est ajoutée. JMockit propose une syntaxe plus élégante :

```
  @Test
  public void
testDoOperation_nominalCase_withEmptProxy()
  {
    MyService service = new MyService();

    // Definition du mock
    Mockit.redefineMethods(DatabaseLogger.class,
MockDatabaseLoggerWithEmptyProxy.class);
    final String param = "anything";
    MockDatabaseLoggerWithEmptyProxy.expectedPara
m = "starting operation : " + param;
    MockDatabaseLoggerWithEmptyProxy.objectToRetu
rn = Mockit.newEmptyProxy(FullMessage.class);

    service.doOperation(param);
  }
```

Nous avons maintenant une référence sur l'objet retourné et pouvons faire des vérifications dessus.

5. JMockit Expectations

JMockit Annotations et JMockit Core imposent la définition d'une classe de mock supplémentaire. JMockit Expectations permet d'effectuer des expectations sans redéfinir de classes de mock. En contrepartie, l'API a d'autres limitations comme l'impossibilité de mocker un constructeur ou un bloc statique. Elle est aussi plus stricte : l'ordre des appels de méthodes compte.

5.1. Mocker une méthode "simple"

Nous souhaitons vérifier que la méthode *live* de *HumanBeing* invoque la méthode *sleep()*.

```
public class HumanBeing extends
AbstractLivingBeing
{
  public void live()
  {
```

```
  }
}
```

Nous écrivons le test correspondant :

```
@Test
public void testLive_nominalCase() {
  HumanBeing being = new HumanBeing();

  new Expectations(true)
  {

    // "Premier bloc" :
    //   Mettre ici les classes que l'on souhaite
mocker,
    //   ainsi que les méthodes dont on veut
vérifier l'appel dans l'annotation
    @Mocked(methods={"sleep"})
    HumanBeing mock;

    {
      // "Deuxieme bloc" :
      // Mettre ici les expectations c'est à dire
les appels attendus. Les méthodes attendues
doivent
      //   aussi apparaitre dans l'annotation
pour être vérifiées.
      mock.sleep();
    }

  };

  // La methode testee
  being.live();
}
```

Le test échoue pour l'instant, comme nous n'avons pas encore implémenté le code. Notez les *doubles accolades* après new Expectations :

Le premier bloc permet de déclarer les objets devant être mockés. L'annotation @Mocked nous informe que c'est le type *HumanBeing* que nous mockons. Le paramètre methods={"sleep"} signifie que nous avons des attentes sur la méthode *sleep*.

Le deuxième bloc définit les expectations : quels paramètres sont passés, dans quel ordre les méthodes sont appelées ?

Après implémentation du code, le test passe :

```
public void live()
{
  sleep();
}
```

Remarquez que le test passe toujours avec l'appel de *eat()* en plus, car nous n'avons pas défini d'attentes particulières sur cette méthode.

```
public void live()
{
  sleep();
  eat();
}
```

5.2. Mocker une méthode statique

Le principe reste le même pour mocker une méthode statique. Même si elle n'est jamais utilisée, il faut déclarer une instance dans le premier bloc pour signaler que la classe est mockée.

Après *sleep*, c'est *Entertainer.displayTvShow* qui doit être appelé. Commençons déjà par déclarer le mock sur la méthode statique :

```
@Test
public void testLive_demoStaticMethod() {
  HumanBeing being = new HumanBeing();

  new Expectations(true)
  {

    // "Premier bloc" :
    // Mettre ici les classes que l'on souhaite
mocker,
    //   ainsi que les méthodes dont on veut
vérifier l'appel dans l'annotation
    @Mocked(methods={"sleep"})
    HumanBeing mock;

    @Mocked(methods={"displayTvShow"})
    Entertainer entertainer;

    {
      // "Deuxieme bloc" :
      // Mettre ici les expectations, ie les
appels attendus. Les méthodes attendues doivent
      //   aussi apparaitre dans l'annotation
pour être vérifiées.
      mock.sleep();
    }

  };

  // La methode testee
  being.live();
}
```

Le test passe toujours. Maintenant, nous voulons vérifier l'appel de *displayTvShow* après l'appel de *sleep* :

```
new Expectations(true)
{
  // "Premier bloc" :
  // Mettre ici les classes que l'on souhaite
mocker,
  //   ainsi que les méthodes dont on veut
vérifier l'appel dans l'annotation
  @Mocked(methods={"sleep"})
  HumanBeing mock;

  @Mocked(methods={"displayTvShow"})
  Entertainer entertainer;

  {
    // "Deuxieme bloc" :
    // Mettre ici les expectations, ie les appels
attendus. Les methodes attendues doivent
    //   aussi apparaitre dans l'annotation pour
être vérifiées.
    mock.sleep();
    Entertainer.displayTvShow();
  }
```

```
};
```

Le test échoue bien. Nous pouvons modifier l'implémentation pour le faire passer :

```
public void live()
{
  sleep();
  Entertainer.displayTvShow();
}
```

5.3. Mocker toutes les méthodes d'une classe

Si l'annotation *Mocked* n'a pas de valeur, alors toutes les méthodes de la classe *HumanBeing* sont mockées.

```
new Expectations(true)
{
  @Mocked
  HumanBeing mock;
  {
```

Nous avons le même comportement sans mettre d'annotation du tout, tant que nous sommes dans le premier bloc des *Expectations*.

```
new Expectations(true)
{
  HumanBeing mock;
  {
```

5.4. Mocker toutes les méthodes d'une classe sauf certaines

Or nous ne voulons pas tout mocker de la classe *HumanBeing*, surtout pas la méthode *live* puisque c'est elle que nous testons... L'attribut inverse de @Mocked permet de dire "je veux mocker toutes les méthodes de la classe, SAUF celle que je spécifie". Prenons l'implémentation de la méthode *liveNoTv* :

```
public void liveNoTv()
{
  sleep();
  chat();
  goToWork();
  eat();
}
```

Plutôt que d'énumérer toutes les méthodes dans l'annotation, nous utilisons l'attribut *inverse* :

```
@Test
public void testLiveNoTv_allMockedExcept() {
  HumanBeing being = new HumanBeing();

  new Expectations(true)
  {
  @Mocked(methods={"liveNoTv"},inverse=true)
    HumanBeing mock;
    {
      mock.sleep();
      mock.goToWork();
      mock.eat();
    }
   };

  // La methode testee
  being.liveNoTv();
}
```

5.5. Définir l'expectation d'une méthode privée

L'être humain peut parler : ne nous privons pas de bavarder. Par contre, pas sûr que nos enfants sachent le faire dès la naissance. Cette méthode sera donc privée.

```
public void liveAndChat()
  {
    eat();
    chat();
  }

  private void chat()
  {
    // let's talk
  }
```

invoke permet de mocker des méthodes privées avec l'introspection :

```
@Test
public void testLiveAndChat_demoPrivateMethod() {
  HumanBeing being = new HumanBeing();

  new Expectations(true)
  {
    @Mocked(methods={"eat","chat"})
    HumanBeing mock;
    {
      mock.eat();
      invoke(mock, "chat");

    }
  };

  // La methode testee
  being.liveAndChat();
}
```

Le gros désavantage est que le nom de la méthode est passé en paramètre : si son nom change, le refactoring ne le prendra pas en compte. Le fait de lancer les tests régulièrement limite le problème mais cela reste assez contraignant.

Certains préféreront simplement changer la portée de la méthode en *protected*. D'autres argueront que la méthode privée est indirectement testée par la méthode publique d'une part et surtout qu'il s'agit d'un détail d'implémentation qui ne devrait pas casser des tests.

Personnellement, je n'aime pas le code redondant et si ma méthode privée est privée justement pour éviter des redondances, je préfère tester qu'elle soit appelée plutôt que d'avoir du code dupliqué dans mon test. Si par contre ma méthode est privée pour avoir un nom plus lisible par exemple, alors effectivement tester uniquement la méthode publique me semble plus pertinent.

5.6. Vérifier qu'une méthode est appelée n fois

Et si j'ai envie de bavarder également avant de manger ?

```
  public void liveAndChatMore()
  {
    chat();
    eat();
    chat();
  }
```

Il y a deux façons de formaliser le test. L'expectation sur *chat* est spécifiée deux fois :

```
@Test
public void testLiveAndChat_demoRepeat() {
  HumanBeing being = new HumanBeing();

  new Expectations(true)
  {
    @Mocked(methods={"eat","chat"})
    HumanBeing mock;
    {
      invoke(mock, "chat");
      mock.eat();
      invoke(mock, "chat");

    }
  };

  // La methode testee
  being.liveAndChatMore();
}
```

Si les deux appels se suivaient, nous aurions pu utiliser la méthode repeats, pour dire combien de fois une méthode doit être appelée.

```
invoke(mock, "chat");repeats(2);
```

Comme l'ordre des invocations compte dans JMockit Expectations, *repeats* ne convient pas ici.

5.7. Vérifier les arguments passés à une méthode

Travailler, bavarder, c'est bien, il faut aussi se bouger un peu ! L'humain peut-il courir ? Testons.

```
@Test
public void testLiveAndChat_demoCheckParam() {
  HumanBeing being = new HumanBeing();

  new Expectations(true)
  {
    @Mocked(methods={"sleep","doSport"})
    HumanBeing mock;
    {
      mock.sleep();
      mock.doSport("running");
    }
  };

  // La methode testee
  being.liveAndMove();
}
```

Le test affiche l'erreur suivante :

```
java.lang.AssertionError: Parameter 0 of void
fr.fchabanois.tutorial.jmockit.HumanBeing#doSport
(String) expected "running", got "foot"
        at
fr.fchabanois.tutorial.jmockit.HumanBeing.doSport
(HumanBeing.java)
        at
fr.fchabanois.tutorial.jmockit.HumanBeing.liveAnd
Move(HumanBeing.java:22)
        at
fr.fchabanois.tutorial.jmockit.HumanBeingTest.tes
tLiveAndChat_demoCheckParam(HumanBeingTest.java:1
64)
        at
sun.reflect.NativeMethodAccessorImpl.invoke0(Nati
ve Method)
        at
java.lang.reflect.Method.invoke(Method.java:597)
        at
org.eclipse.jdt.internal.junit4.runner.JUnit4Test
Reference.run(JUnit4TestReference.java:45)
        at
org.eclipse.jdt.internal.junit.runner.TestExecuti
on.run(TestExecution.java:38)
        at
org.eclipse.jdt.internal.junit.runner.RemoteTestR
unner.runTests(RemoteTestRunner.java:460)
        at
org.eclipse.jdt.internal.junit.runner.RemoteTestR
unner.runTests(RemoteTestRunner.java:673)
        at
org.eclipse.jdt.internal.junit.runner.RemoteTestR
unner.run(RemoteTestRunner.java:386)
        at
org.eclipse.jdt.internal.junit.runner.RemoteTestR
unner.main(RemoteTestRunner.java:196)
```

Apparemment, l'humain préfère le foot. Après tout, tant que c'est du sport, c'est bien. Nous assouplissons notre test : le sport que tu voudras faire, tu pourras grâce à withAny:

```
@Test
public void testLiveAndChat_demoCheckParam() {
  HumanBeing being = new HumanBeing();

  new Expectations(true)
  {
    @Mocked(methods={"sleep","doSport"})
    HumanBeing mock;
    {
      mock.sleep();
      mock.doSport(withAny(new String()));
    }
  };

  // La methode testee
  being.liveAndMove();
}
```

5.8. Mocker l'objet de retour d'une méthode

La méthode returns permet de mocker l'objet de retour d'une méthode :

```
protected String give(String something)
{
  return something;
}
```

Typiquement, l'intérêt est de vérifier que cet objet est passé en paramètre d'une autre méthode. Par exemple, la chaîne retournée par *give* doit être passée en paramètre à *say*.

```
  public void liveWithLove() {
    sleep();
    String string = give("a kiss");
    say(string);
  }
```

Voici le test au complet :

```
@Test
public void testLiveWithLove_demoReturn() {
  HumanBeing being = new HumanBeing();


  final String aHug = "a hug";
  new Expectations(true)
  {
    @Mocked(methods={"sleep","give","say"})
    HumanBeing mock;
    {
      mock.sleep();
      mock.give("a kiss");returns(aHug);
      mock.say(aHug);
    }
  };

  // La methode testee
  being.liveWithLove();
}
```

6. Conclusion

JMockit est un framework de test prometteur, puissant, simple et permet de sécuriser même du code "sale". Je n'ai à mon souvenir jamais rencontré de code que je ne pouvais pas tester grâce à JMockit.

Cette puissance est un peu son défaut. Le fait que JMockit permette de tout mocker peut assez ironiquement nous encourager à écrire du code "intestable", sans nous en rendre compte... On en rajoute encore et encore et c'est **l'application qui devient "intestable"**. Est-ce que c'est grave, puisqu'elle est testée ?

Oui. C'est un peu comme si l'on mangeait des frites au Nutella à longueur de journée (bon appétit !), parce qu'on sait qu'au pire, la liposuccion existe. JMockit est un remède et il vaut mieux prévenir que guérir. De plus :

- le fait que le remède soit quasiment "unique" fait qu'**on en est dépendant**. Impossible de changer de framework de test. C'est JMockit et point barre, même si le framework devient une usine à gaz incompréhensible. Tous les développeurs sont *obligés* d'apprendre cette API, sinon il n'y pas de tests ;
- écrire du code testable s'apprend et prend du temps. C'est une bonne pratique qui ne devrait pas être spécifique à un langage. En Java, poser des tests sur de code sale est possible mais c'est vraiment une chance, un cas particulier. Une chance que vous n'aurez peut-être pas dans votre prochain poste. Si vous voulez être en mesure de poser des tests dans n'importe quel langage ou ne serait-ce que sur du Java 1.4, il faut **savoir écrire du code testable et pas compter sur un framework magique**.

Le code de l'application devrait rester testable avec n'importe quoi, même des mocks maison. Pour que ce soit le cas, j'utilise par défaut un framework de mocks traditionnel et programme en Test Driven Development (Lien65). Le TDD nous incite naturellement à écrire du code testable. Ce n'est que **sur du code existant, quand je dois poser des tests après coup, que je n'y arrive pas avec les moyens "normaux" et que je ne peux pas refactorer que j'ai recours à JMockit**. Pour des raisons de lisibilité, j'utilise principalement les *Expectations* de JMockit.

Pour finir, je rappelle que ce tutoriel n'est qu'un petit aperçu du framework. Il rassemble les besoins que j'ai eus pendant mes développements, mais vous en avez peut-être d'autres. Rogerio Liesenfeld travaillant très activement sur ce framework, n'hésitez pas à jeter un oeil sur les évolutions qu'il y apporte jour après jour (Lien66).

Retrouvez l'article de Florence Chabanois en ligne : Lien67

Les livres Conception

SCRUM

le guide pratique de la méthode agile la plus populaire

Cet ouvrage s'adresse aux développeurs qui souhaitent s'initier aux méthodes de développement agile, et aux chefs de projet qui veulent améliorer l'organisation et l'efficacité de leurs équipes. Il sera également très utile à ceux qui ont déjà une première expérience de Scrum et qui souhaitent maîtriser la méthode en profondeur. Le but affiché d'une méthode agile est d'améliorer la qualité du produit, tout en réduisant les délais et les coûts. Aujourd'hui Scrum s'est imposée comme la plus répandue des méthodes agiles. Quand on applique Scrum les fonctionnalités sont collectées dans un backlog et classées par priorité. Une version (release) est produite à l'issue d'une série d'itérations, appelées sprints. Pendant un sprint des points de contrôle sont effectués par l'équipe au cours de " mêlées " quotidiennes appelées scrums. Cet ouvrage rassemble non seulement une présentation détaillée des pratiques, mais aussi de nombreux conseils concrets issus de la vraie vie. Il commence par une présentation des méthodes agiles et de Scrum. Il explique ensuite les rôles des deux personnages-clé que sont le ScrumMaster et le Product Owner. Les chapitres suivants expliquent comment organiser et mener à son terme un projet en appliquant Scrum.

Critique du livre par Pierre Chauvin

Hasard ou non, cette publication de Claude Aubry coïncidait la semaine passée avec ma participation à une formation ScrumMaster, sanctionnée par un examen (certes discutable) pour une certification de même nom. J'étais donc déjà plongé dans Scrum depuis plusieurs semaines lorsque l'on m'a proposé cette lecture. Lors de

cette formation j'avais rencontré des consultants de Akka Technologies (Lien68), vecteurs d'une bonne dynamique Scrum dans le bassin toulousain, avec qui j'ai eu l'occasion de parler des projets de Claude Aubry et de son investissement sur le sujet. Entre les guides officiels, la littérature de Schwaber, Sutherland et Cohn, il ne me manquait donc plus qu'un livre en français. C'est chose faite, et plutôt bien faite suis-je tenté de souligner.

Bien sincèrement, le livre de Claude Aubry aborde Scrum d'une manière moins dogmatique, mais assurément plus pragmatique. En fait je n'avais pour le moment pas encore lu d'ouvrage structuré qui dispense de manière aussi complète des conseils sur la pratique de Scrum. Le retour d'expérience de l'auteur sur chaque *précepte* Scrum (les artefacts, les rôles, meetings) est très intéressant, et chaque ScrumMaster ou membre d'une équipe Scrum y trouvera des éléments de réflexion de qualité (comme la vie du backlog, la phase de conception, la planification de release, etc.). Je pense que je serais amené à relire certains chapitres ou paragraphes sur des sujets précis si je suis pris en défaut, car les explications et exemples sont clairs et invitent à l'essai.

Vous retrouverez ainsi les rappels des fondamentaux de ce processus *empirique*, une description des différents rôles (Product Owner, ScrumMaster, l'équipe), du Product Backlog, des différents timeboxes (planification de release, planification de sprint, daily scrum, rétrospective de sprint), mais aussi des définitions pratiques comme la notion de travail *fini*, le contexte du projet, les mesures et indicateurs. On notera également le chapitre sur les outils pour Scrum, ou IceScrum est présenté, ainsi qu'un chapitre utile pour les décideurs : "La transition à Scrum".

J'ai apprécié ce livre en français de Claude Aubry, qui apporte une contribution de qualité à la compréhension, à une application pragmatique, et à l'adoption de Scrum en France. Je pense ne pas me tromper en précisant que se livre est relativement accessible à tout acteur de l'IT, sensible aux méthodes agiles et à l'optimisation des développements logiciels, qu'il soit décideur, chef de projet, développeur, ou futur "Product Owner".

Critique du livre par Bruno Orsier

Ce livre très attendu est écrit par Claude Aubry, un représentant particulièrement actif de la communauté agile française, et préfacé par Francois Beauregard, un "agiliste" francophone de la première heure.

J'ai découvert avec plaisir un livre clair et passionnant sur un processus de développement, ce qui est rare : la simplicité et l'élégance de SCRUM y contribuent certainement, mais l'auteur a fait preuve de beaucoup de pédagogie. En particulier tous les concepts, entités, rôles, responsabilités sont clairement expliqués et bien reliés les uns aux autres, et leurs aspects temporels et dynamiques sont bien couverts. L'ouvrage montre donc bien la grande cohérence de SCRUM.

Et l'auteur a fait des choix appréciables : les chapitres courts et faciles à lire, les nombreux schémas qui facilitent la compréhension du texte, les dessins humoristiques... Il a également tiré parti de son expérience considérable pour mettre en garde le lecteur contre divers risques et pièges qui l'attendent dans la mise en pratique : bien que simple, Scrum n'est pas facile à pratiquer.

Les 11 premiers chapitres, soit environ 150 pages, décrivent complètement Scrum. Suivent ensuite 7 chapitres qui donnent des conseils pratiques sur de nombreux aspects, comme l'adaptation au contexte de votre entreprise, l'élaboration de la vision d'un produit (avant donc les sprints), la relation features-stories-tests, la traçabilité, l'ingénierie du logiciel, les indicateurs. Le dernier chapitre est un panorama de Scrum en France.

Un livre assez orienté "processus" tout de même, et qui passe peut-être un peu vite sur les aspects "équipe" et "individuel", auxquels on est forcément conduit à s'intéresser si l'on veut tirer complètement parti de Scrum : peut-on faire l'économie d'approfondir ce que veut dire être un développeur professionnel dans ce nouveau contexte ? Ken Schwaber a d'ailleurs mis en garde contre le fait qu'appliquer Scrum avec des gens qui faisaient du sale boulot avant permettait juste de produire du sale boulot à intervalles réguliers !

Un regret au niveau de la bibliographie : elle est un peu légère. Je trouve dommage de ne pas voir cités des auteurs essentiels comme les Poppendieck (Lean Software Development), Esther Derby (les rétrospectives), Larman et Vodde (Scaling Scrum), Lencioni (les dysfonctionnements d'une équipe), Robert C. Martin (sur Clean Code), Lisa Crispin, Janet Gregory (Agile Testing). Une meilleure bibliographie aurait contribué au caractère de référence que va prendre cet ouvrage.

En conclusion, ce livre est un excellent guide pour démarrer avec Scrum, ou pour revisiter les bases si vous avez déjà pratiqué. Il contient suffisamment de matière pour vous accompagner pendant de nombreux sprints. Même si vous vous intéressez déjà à des questions qui relèvent plus de la bibliographie ci-dessus, vous achèterez certainement le livre pour vos collègues et partenaires !

Retrouvez ces critiques de livre sur la page livres Conception : Lien69

Liens

Lien1 : http://www.objis.com/formation-java/spip.php?article43
Lien2 : http://objis.developpez.com/tutoriels/aspectj/log/
Lien3 : http://www.programmationgwt2.com/web/guest
Lien4 : http://linsolas.developpez.com/critiques/apache-maven/
Lien5 : http://java.developpez.com/livres/
Lien6 : http://www.developpez.net/forums/d871290/php/langage/facebook-devoile-re-ecriture-php-hiphop-traduit-php-cpp-puis-compile-gpp/
Lien7 : http://www.developpez.net/forums/d864002/php/outils/zend/zend-framework/php-sortie-version-zend-framework-1-10-0-beta1/#post4954722
Lien8 : http://framework.zend.com/download/latest
Lien9 : http://framework.zend.com/issues
Lien10 : http://www.developpez.net/forums/d864002/php/outils/zend/zend-framework/php-sortie-version-zend-framework-1-10-0-beta1/
Lien11 : http://www.phpbuilder.com/columns/marc_plotz10012009.php3?page=1
Lien12 : http://www.faqs.org/rfcs/rfc1738.html
Lien13 : http://jcrozier.developpez.com/tutoriels/web/php/filtres-securite/
Lien14 : http://www.ibm.com/developerworks/opensource/library/os-php-command/index.html
Lien15 : http://g-rossolini.developpez.com/tutoriels/php/cours/?page=introduction#LI-G-2
Lien16 : http://php.net/manual/fr/function.escapeshellarg.php
Lien17 : http://www.php.net/manual/fr/function.escapeshellcmd.php
Lien18 : http://y-komotir.developpez.com/tutoriels/php/utiliser-outilsen-ligne-commande-php/
Lien19 : http://www.css3.info/making-an-image-gallery-with-target/
Lien20 : http://css.developpez.com/tutoriels/creer-galerie-images-avec-target/fichiers/
Lien21 : http://daniel.glazman.free.fr/weblog/targetExample.html#general
Lien22 : http://css.developpez.com/tutoriels/creer-galerie-images-avec-target/
Lien23 : http://www.admixweb.com/2009/05/20/how-to-easily-create-a-javascript-framework-part-1/
Lien24 : http://dico.developpez.com/html/1710-Internet-Ajax-Javascript-Asynchrone-et-XML.php
Lien25 : http://www.prototypejs.org/
Lien26 : http://jquery.com/
Lien27 : http://developer.yahoo.com/yui/
Lien28 : http://dojotoolkit.org/
Lien29 : http://www.crockford.com/
Lien30 : http://kalyparker.developpez.com/articles/js/VOZ-partie-1/fichiers/vozpart1_fr.html
Lien31 : http://kalyparker.developpez.com/articles/js/VOZ-partie-1/
Lien32 : http://www.admixweb.com/2009/06/05/how-to-easily-create-a-javascript-framework-part-2/
Lien33 : http://kalyparker.developpez.com/articles/js/VOZ-partie-1/
Lien34 : http://kalyparker.developpez.com/articles/js/VOZ-partie-2/fichiers/vozpart2_fr.html
Lien35 : http://kalyparker.developpez.com/articles/js/VOZ-partie-2/
Lien36 : http://www.quirksmode.org/js/cookies.html
Lien37 : http://dico.developpez.com/html/219-Internet-HTTP-HyperText-Transmission-Protocol.php
Lien38 : http://ppk.developpez.com/tutoriels/javascript/gestion-cookies-javascript/fichiers/
Lien39 : http://ppk.developpez.com/tutoriels/javascript/gestion-cookies-javascript/
Lien40 : http://msdn.microsoft.com/en-us/library/aa374224%28VS.85%29.aspx
Lien41 : http://www.codeproject.com/KB/COM/regsvr42.aspx?msg=3179532
Lien42 : http://www.mazecomputer.com/
Lien43 : http://habrahabr.ru/blogs/javascript/52027/
Lien44 : http://omen999.developpez.com/tutoriels/vbs/RegFreeSxS/
Lien45 : http://cpp.developpez.com/faq/cpp/?page=OO#DEFINITION_polymorphisme_inclusion
Lien46 : http://apais.developpez.com/tutoriels/c++/fonctions-virtuelles-en-cpp/
Lien47 : http://www.boost.org/doc/libs/1_41_0/libs/preprocessor/doc/index.html
Lien48 : http://apais.developpez.com/tutoriels/c++/fonctions-virtuelles-en-cpp/?page=page_6
Lien49 : http://plee-the-bear.sf.net/index-fr.html
Lien50 : http://www.tegesoft.com/products/camp
Lien51 : http://www.lua.org/
Lien52 : http://mdeverdelhan.developpez.com/tutoriel/lua/tutoriel1/
Lien53 : http://j-jorge.developpez.com/cpp/appels-dynamiques/
Lien54 : http://mac.developpez.com/livres/
Lien55 : http://bruno-orsier.developpez.com/mocks-arent-stubs/
Lien56 : http://misko.hevery.com/code-reviewers-guide/
Lien57 : http://code.google.com/p/jmockit/
Lien58 : http://jmockit.googlecode.com/svn/trunk/www/installation.html
Lien59 : http://jmockit.googlecode.com/svn/trunk/www/tutorial.html
Lien60 : http://code.google.com/p/jmockit/downloads/list
Lien61 : http://download.java.net/maven/2/mockit/jmockit/
Lien62 : http://jmockit.googlecode.com/svn/trunk/www/installation.html
Lien63 : http://bruno-orsier.developpez.com/mocks-arent-stubs/
Lien64 : https://jmockit.dev.java.net/javadoc/mockit/Mock.html
Lien65 : http://fr.wikipedia.org/wiki/Test_Driven_Development
Lien66 : http://jmockit.googlecode.com/svn/trunk/www/changes.html
Lien67 : http://fchabanois.developpez.com/tutorial/java/jmockit/
Lien68 : http://www.akka.eu/
Lin69 : http://conception.developpez.com/livres/