CHAPTER 1

TinyBlog : Extension du modèle et tests unitaires

1.1 Correction semaine précédente

Vous pouvez charger la correction de la semaine précédente en exécutant le code suivant :

```
Metacello new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    version: #week1solution;
    configuration: 'TinyBlog';
    load
```

Vous pouvez ouvrir un browser de code sur la classe `TBPost`. Maintenant que vous avez la correction, vous pouvez compléter le code de votre application TinyBlog si nécessaire.

1.2 Sauvegarder votre code

Alors que vous pouvez sauver l'image Pharo qui contient tous les objets du système et donc les classes elles-même, nous allons vous montrer comment les pharoers sauvent leur code: Vous allez sauver votre code sous forme de packages sur un serveur dédié à l'aide de Monticello le gestionnaire de versions de Pharo. Smalltalkhub http://smalltalkhub.com est une des forges pour sauver le code Pharo, vous pouvez aussi utiliser SS3 à http://ss3.gemstone.com.

Créer un dépôt de code

- Créer un compte sur le site http://smalltalkhub.com/.
- Se connecter au site.
- Créer un projet (il se peut que vous ayez des problèmes de connexion car le site est en beta version auquel cas essayez de changer de navigateur, si les problèmes persistent utiliser http://ss3.gemstone.com).
  - Nommez le "TinyBlog".

Sauver votre package

- Dans Pharo, ouvrez le Monticello Browser via le menu du monde.
- Ajouter un repository (de type soit SmalltalkHub soit HTTP pour http://ss3.gemstone.com).
- Sélectionner le repository et sélectionner l'item 'Add to package...' pour ajouter ce repository au package TinyBlog.
- Sélectionner votre package et pressez le bouton 'Save'.
- Editer l'information de description et sauver. Votre code vient d'être sauvé sur le serveur.

Le code de votre application TinyBlog est maintenant sauvegardé dans votre dépôt sur Smalltalkhub. Il est donc maintenant possible de charger votre code dans une nouvelle image Pharo. Par contre, cela va prendre un certain temps à charger car vous allez charger Seaside et plusieurs autres projets.

Dans le cadre de ce projet, nous vous suggérons de toujours utiliser l'image avec tous les packages web chargés que vous trouvez sur http://mooc.pharo.org. Cela vous permettra de pouvoir recharger votre package sans devoir vous soucier des dépendances sur d'autres packages.

1.3 A propos de dépendances

Les bonnes pratiques lors de développements en Pharo sont de spécifier clairement les dépendances sur les packages utilisés afin d'avoir une reproductibilité complète d'un projet. Une telle reproducibilité permet alors l'utilisation de serveur de construction tel Travis ou Jenkins. Pour cela, une configuration (une classe spéciale) définit d'une part l'architecture du projet (dependances et packages du projet) et les versions des packages versionnés.

Dans le cadre de ce projet, nous n'abordons pas ce point plus avancé. Un chapitre entier sur l'expression de configuration dans Deep Into Pharo est consacré à ce point.

1.4 La classe TBBlog

La classe `TBBlog` contient des posts. Nous allons développer `TBBlog` en écrivant des tests puis en les implémentant.

```
Object subclass: #TBBlog
   instanceVariableNames: 'posts'
   classVariableNames: ''
   package: 'TinyBlog'
```

Nous initialisons les postes sur une collection vide.

```
TBBlog >> initialize
   super initialize.
   posts := OrderedCollection new.
```

1.5 Un seul blog

Dans un premier temps nous supposons que nous allons gérer qu'un seul blog. Dans le futur, vous pourrez ajouter la possibilité de gérer plusieurs blogs comme un par utilisateur de notre application. Pour l'instant, nous utilisons donc un singleton pour la classe `TBBlog`.

Comme la gestion du singleton est un comportement de classe, ces méthodes sont définies sur le coté class de la classe `TBBlog`.

```
TBBlog class
   instanceVariableNames: 'uniqueInstance'
```

```
TBBlog class >> reset
   uniqueInstance := nil
```

```
TBBlog class >> current
   "answer the instance of the TBRepository"
   ^ uniqueInstance ifNil: [ uniqueInstance := self new ]
```

Quand la classe est chargée, le singleton est réinitialisé.

```
TBBlog class >> initialize
   self reset
```

1.6 Tester les Règles Métiers

Nous allons écrire des tests pour les règles métiers et ceci en mode TDD (Test Driven Development) c'est-à-dire en développant les tests en premier puis en définissant les fonctionalités jusqu'à ce que les tests passent.

Les tests unitaires sont regroupés dans une étiquette (tag) `TinyBlog-Tests` qui contient la classe `TBBlogTest` (voir menu item "Add Tag..."). Un tag est juste une étiquette qui permet de trier de grouper les classes à l'intérieur

d'un package. Nous utilisons un tag pour ne pas avoir à gérer deux packages différents mais dans une réelle solution vous définirions une package séparé pour les tests.

```
TestCase subclass: #TBBlogTest
   instanceVariableNames: 'blog post first'
   classVariableNames: ''
   category: 'TinyBlog-Tests'
```

Avant le lancement des tests, la méthode `setUp` initialise le contexte des tests. Par exemple, elle efface son contenu, ajoute un post et en créé un autre qui provisoirement n'est pas enregistré.

```
TBBlogTest >> setUp
   blog := TBBlog current.
   blog removeAllPosts.

   first := TBPost title: 'A title' text: 'A text' category: 'First
    Category'.
   blog writeBlogPost: first.

   post := (TBPost title: 'Another title' text: 'Another text'
    category: 'Second Category') beVisible
```

On en profite pour tester différentes configurations. Les posts ne sont pas dans la même catégorie, l'un est visible, l'autre pas.

La méthode `tearDown` exécutée au terme des tests remet à zéro le blog.

```
TBBlogTest >> tearDown
   TBBlog reset
```

L'utilisation d'un Singleton montre ses limites. En effet, si vous déployez un blog puis exécutez les tests vous perdrez les posts que vous avez créés car nous les remettons à zéro.

Nous allons développer les tests d'abord puis ensuite passer à l'implémentation des fonctionnalités.

1.7 Un premier test

Commençons par écrire un premier test qui ajoute un post et vérifie qu'il est effectivement ajouté.

```
TBBlogTest >> testAddBlogPost
   blog writeBlogPost: post.
   self assert: blog size equals: 2
```

Ce test ne marche pas (n'est pas vert) car nous n'avons pas défini les méthodes: `writeBlogPost:` et `removeAllPosts`. Ajoutons-les.

```
TBBlog >> removeAllPosts
   posts := OrderedCollection new
```

```
TBBlog >> writeBlogPost: aPost
   "Write the blog post in database"
   posts add: aPost
```

```
TBBlog >> size
   ^ posts size
```

Le test précédent doit maintenant passer.

Ecrivons un test pour couvrir les fonctionalités que nous venons de développer.

Obtenir le nombre de posts dans le blog

```
TBBlogTest >> testSize
   self assert: blog size equals: 1
```

Effacer l'intégralité des posts

```
TBBlogTest >> testRemoveAllBlogPosts
   blog removeAllPosts.
   self assert: blog size equals: 0
```

1.8 Quelques autres fonctionnalités

Nous définissons les fonctionnalités et nous assurons que les tests passent. Les règles métiers sont regroupées dans le protocole 'action' de la classe `TBBlog`.

Obtenir l'ensemble des posts (visibles et invisibles)

```
TBBlogTest >> testAllBlogPosts
   blog writeBlogPost: post.
   self assert: blog allBlogPosts size equals: 2
```

```
TBBlog >> allBlogPosts
   ^ posts
```

Obtenir tous les posts visibles

```
TBBlogTest >> testAllVisibleBlogPosts
   blog writeBlogPost: post.
   self assert: blog allVisibleBlogPosts size equals: 1
```

```
TBBlog >> allVisibleBlogPosts
   ^ posts select: [ :p | p isVisible ]
```

Obtenir tous les posts d'une catégorie

```
TBBlogTest >> testAllBlogPostsFromCategory
   self assert: (blog allBlogPostsFromCategory: 'First Category')
    size equals: 1
```

```
TBBlog >> allBlogPostsFromCategory: aCategory
   ^ posts select: [ :p | p category = aCategory ]
```

Obtenir tous les posts visibles d'une catégorie

```
TBBlogTest >> testAllVisibleBlogPostsFromCategory
   blog writeBlogPost: post.
   self assert: (blog allVisibleBlogPostsFromCategory: 'First
    Category') size equals: 0.
   self assert: (blog allVisibleBlogPostsFromCategory: 'Second
    Category') size equals: 1
```

```
TBBlog >> allVisibleBlogPostsFromCategory: aCategory
   ^ posts select: [ :p | p category = aCategory and: [ p isVisible
    ] ]
```

Vérifier la gestion des posts non classés

```
TBBlogTest >> testUnclassifiedBlogPosts
   self assert: (blog allBlogPosts select: [ :p | p isUnclassified
    ]) size equals: 0
```

Obtenir la liste des catégories

```
TBBlogTest >> testAllCategories
   blog writeBlogPost: post.
   self assert: blog allCategories size equals: 2
```

```
TBBlog >> allCategories
   ^ (self allBlogPosts collect: [ :p | p category ]) asSet
```

1.9 Futures évolutions

Plusieurs évolutions peuvent être apportées telles que: obtenir uniquement la liste des catégories contenant au moins un post visible, effacer une catégorie et les posts contenus, renommer une catégorie, déplacer un post d'une catégorie à une autre, rendre visible ou invisible une catégorie et son contenu, etc. Nous vous encourageons à les développer.

Afin de nous aider à tester l'application nous définissons une méthode permettant de créer quelques posts.

```
TBBlog class >> createDemoPosts
   "TBBlog createDemoPosts"
   self current
      writeBlogPost: ((TBPost title: 'Welcome in TinyBlog' text:
     'TinyBlog is a small blog engine made with Pharo.' category:
     'TinyBlog') visible: true);
      writeBlogPost: ((TBPost title: 'Report Pharo Sprint' text:
     'Friday, June 12 there was a Pharo sprint / Moose dojo. It was a
     nice event with more than 15 motivated sprinters. With the help
     of candies, cakes and chocolate, huge work has been done'
     category: 'Pharo') visible: true);
      writeBlogPost: ((TBPost title: 'Brick on top of Bloc -
     Preview' text: 'We are happy to announce the first preview
     version of Brick, a new widget set created from scratch on top
     of Bloc. Brick is being developed primarily by Alex Syrel
     (together with Alain Plantec, Andrei Chis and myself), and the
     work is sponsored by ESUG.
      Brick is part of the Glamorous Toolkit effort and will provide
     the basis for the new versions of the development tools.'
     category: 'Pharo') visible: true);
      writeBlogPost: ((TBPost title: 'The sad story of unclassified
     blog posts' text: 'So sad that I can read this.') visible: true);
      writeBlogPost: ((TBPost title: 'Working with Pharo on the
     Raspberry Pi' text: 'Hardware is getting cheaper and many new
     small devices like the famous Raspberry Pi provide new
     computation power that was one once only available on regular
     desktop computers.' category: 'Pharo') visible: true)
```

Vous pouvez ensuite inspecter le résultat de l'évaluation du code suivant :

```
TBBlog createDemoPosts ; current
```

1.10 Conclusion

Vous devez avoir le modèle complet de TinyBlog ainsi que des tests et maintenant vous êtes prêt pour des fonctionalités plus avancées comme le stockage ou un premier serveur HTTP. C'est aussi un bon moment pour sauver votre code.