CHAPTER 1

TinyBlog: Extending and testing the model

1.1 Previous week solution

You can load the solution of the previous week using the following snippet:

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

Then, you can browse the code of the TBPost class. With this solution, you can complete the code of **your** TinyBlog application if needed before continuing.

1.2 Save your code

Even if you can save the Pharo image which contains all the objects of the system (and as such the classes you wrote), this section explains how Pharoers usually save their code as packages on dedicated servers with the Pharo versioning system named Monticello. Smalltalkhub http://smalltalkhub.com is one of these servers to host saved Pharo code. You can also use SS3 at http://ss3.gemstone.com.

Create a repository

- Create an account on http://smalltalkhub.com/.

- Log yourself on the web site.
- Create a project (it may happen that you get a connection problem because the web site is still in beta, in such a case retry with a different web browser. If the problems persist use http://ss3.gemstone.com).
  - Name it the way you want. For example "TinyBlog"

Save your package

- In Pharo, open the Monticello Browser available in the main menu.
- Add a repository (of type either SmalltalkHub or HTTP for SS3)
- Select the repository and the menu item 'Add to package...' to add this repository to the package TinyBlog.
- Select your package and press the 'Save' button.
- Enter a log describing the changes you did.

The code of your application is now in your SmalltalkHub repository. You should now be able to load your code in a new Pharo image. Nevertheless, if you use a stock Pharo image, you must also load Seaside and other libraries and frameworks.

In this project, we encourage you to always use the image we gave with the preloaded packages and that you can find at http://mooc.pharo.org. This will let you load new code without having to take care about package dependencies.

1.3 About dependencies

Good practices in Pharo developments are to clearly specify the dependencies of used packages. The idea is to ensure that the building of a project is fully reproducible. It is important because once we have such reproducible process we can take advantage about automatic build servers such as Jenkins or Travis.

To express dependencies between projects and packages within a project Pharo offers special classes called Configurations. Such configurations express the architecture (main repository, dependencies to other projects, structure of the projects) as well as versionned packages.

In the context of the TinyBlog project, we do not got any further on this. Note that a full chapter on dependency expression is available in the *Deep In Pharo* book.

1.4 TBBlog class

A TBBlog contains posts. We will now develop this TBBlog class and its unit tests.

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

We initialize the posts to hold an empty collection.

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

1.5 Only one blog object

In the rest of this project, we assume that we will manage only one blog. Later, you may add the possibility to manage multiple blogs such as one per user of the TinyBlog application. Currently, we use a Singleton design pattern on the TBBlog class.

Since all the management of a singleton is a class behavior, we define such methods at the class level.

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

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

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

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

1.6 Testing the Model

We now adopt a Test Driven Development approach i.e., we will write a unit test first and then develop the business functionality until the test is green. We will repeat this process for each functionality of the model.

We create unit tests in the TBBlogTest class that belongs to the TinyBlog-Tests tag. A tag is just a label to sort classes inside a package (See menu item 'Add Tag...'). We use a tag because using two packages will make this project more complex. However, while implementing a real application, it is recommended to have a separate package for the tests.

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

Before each test execution, the `setUp` method initializes the context of tests. For example, it erases the blog content, adds one post and creates another temporary post that is not saved.

```
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
```

As you may notice, we test different configurations. Posts do not belong to the same category, one is visible and the other is not visible.

At the end of each test, the `tearDown` method is executed and resets the blog.

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

Here we see the limits of using a Singleton. Indeed, if you deploy a blog and then execute the tests, you will lose all posts that have been created because we reset the Blog singleton.

We will now develop tests first and then implement all functionalities to make them green.

First test

The first test adds a post in the blog and verifies that this post is effectivly added.

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

If you try to execute it, you will notice that this test is not green because we did not define the methods `writeBlogPost:`, `removeAllPosts` and `size`. Let's add them:

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

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

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

The previous test should now pass (i.e. be green). We should also add tests to cover all functionalities that we introduced.

Test the number of blog posts

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

Remove all posts

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

1.7 Other functionalities

We now develop new functionalities as methods in the 'action' protocol of the `TBBlog`. While doing that, we regularly ensure that tests pass.

Retrieve all posts (visible and invisible)

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

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

Retrieve visible posts

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

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

Retrieve all posts of one category

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

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

Retrieve all visible posts of one category

```
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
    ] ]
```

Check unclassified posts

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

Retrieve all categories

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

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

1.8 Possible extensions

Many extensions can be made such as: retrieve the list of categories that contains at least one visible post, delete a category and all posts that it contains, rename a category, move a post from one category to another, make (in)visible one category and all its content, etc. We encourage you to develop some of them.

To help you testing the application, you can add the following method that creates multiple 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)
```

If you inspect the result of the following snippet, you will see that the current blog contains 5 posts:

```
TBBlog createDemoPosts ; current
```

1.9 Conclusion

You get now the full model of TinyBlog as well as some tests. You are now ready to implement more advanced functionality such as the database storage or a first HTTP server. Do not forget to save your code.