Clean code, better testability and less mixing of concern

Clean code, better testability, less mixing of concern was always suggested good development practice. Which I have found in Wicket too, and thought to share. Just want to mention one thing before we move on, that, of course it depends person to person how he want to implement his code, but I felt this has some advantage then the other.

Lets look to a code sample that we find on most of the wicket examples. Which as other newbie developer gave me the idea that, perhaps that's the best way to do it. But more I moved on, more I disagreed with that concept. (this code was taken from the post DropDownExample

```
class MyPage extends WebPage {
   .....
   public MyPage() {
      .......
      DropDownChoice dropDownChoice=new DropDownChoice("phoneVendor", phoneVendors) {
         protected boolean wantOnSelectionChangedNotifications() {
            return true;
         }
         protected void onSelectionChanged(final Object newSelection) {
            String phoneVendor = (String) newSelection;
            myModel.setPhoneModel(null);
            phoneModelDDC.setChoices(getTerminalsByVendor(phoneVendor));
         }
      };
      add(dropDownChoice);
      ......
   }
   .....
}
```

In the constructor we are instantiating the drop down choice and adding the on change behaviour. Now consider that inside the constructor there twenty more components. So it will be much more code then we see above. And using that anonymous drop down choice class we are putting the on change behaviour. And this can happen for most of component of wicket (form, link, dropdownchoice,....). Also if I want to test only the onSelectionChange behaviour then I have to bring in the whole page into the scenario. I can not test the onSelectionChanged behaviour isolated. I don't want to care application, page and components. So what I would suggest is -

```
public class DropDownChoiceEx<T> extends DropDownChoice<T>{
   private DropDownChoiceStateChangeListener<T> listener;

   public DropDownChoiceEx(String id, IModel<T> model, List<? extends T> choices,
DropDownChoiceStateChangeListener<T> listener) {
      super(id, model, choices);
      this.listener=listener;
   }
   //add more constructors if you want.

   @Override
   protected void onSelectionChanged(T newSelection) {
      if(wantOnSelectionChangedNotifications()) {
         T modelObject = getModelObject();
         listener.onStateChange(modelObject);
      }
   }

   @Override
   protected boolean wantOnSelectionChangedNotifications() {
      if(listener==null) {
         return false;
      }
         return true;
   }
}
```

And rewrite above code with new DropDownChoiceEx-

```
class MyPage extends WebPage {
   .....
   public MyPage() {
      .......
      DropDownChoiceStateChangeListener<PhoneVendor> listener=new MyListener<PhoneVendor>();
      DropDownChoiceEx<PhoneVendor> dropDownChoice=new DropDownChoiceEx<PhoneVendor("phoneVendor",
phoneVendors, listener);
      add(dropDownChoice);
      ......
   }
   .....
}

class MyListener implements DropDownChoiceStateChangeListener<T> {
   //write constructor and bring in other objects if you need other then model object.
   //because model object is there as parameter on state change method

   public void onStateChange(T modelObject) {
      // write on change code.
   }
}
```

Now look the constructor of MyPage, it is much cleaner, more readable and separating the behaviour from initialization. Also another good part of this way of development is, we can test the OnStateChange behaviour totally isolated. To test the onStateChange behaviour I don't have care about Wicket Application, WebPage and the owning component. Because We could have huge amount of code which is related to domain model or other network communication or something else.

Now some people might say, how we set a response page and get hold of the page that was having the drop down choice. The suggestion is, still we can use the earlier way to get hold of everything if we want. But if you liked my suggested approach, then you can get lots of things out from factory method of Application (Application.get()), and using the application you can do lots of things.

--Jahid Shohel