

Le module objet de Xoops

Catégorie : Fiches techniques

Publié par [Christian](#) le 17/04/2005

Le module objet de Xoops est basé sur 2 classes XoopsObject et XoopsObjectHandler toutes les deux codées dans le fichier /kernel/object.php. Ces 2 classes forment la couche d'accès aux données persistantes telle qu'elle est décrite dans le DAO pattern. C'est en les spécialisant que l'on implémente le module pour une entité particulière (souvent associée à une seule table de BD) comme le font toutes les classes du répertoire kernel qui constitue le noyau de xoops. J'encourage fortement les programmeurs de nouveaux modules xoops s'appuyant sur des tables de BD à utiliser ce module. Les raisons d'utiliser ces 2 classes sont multiples : -> uniformiser la manière de programmer et être près des standards de codage de xoops -> augmenter la lisibilité du code et ainsi faciliter la maintenance -> fiabiliser votre code en s'appuyant sur du code xoops maintes fois validé -> mais aussi bénéficier d'un certain nombre de mécanismes génériques intégrés dans Xoops XoopsObject possède tous les services relatifs à la gestion d'un objet (une instance de la classe) et de ses attributs (getter et setter) alors que XoopsObjectHandler sert de manipulateur ou de contrôleur des instances (insertion, modification ou sélection d'objets).

Dans une première partie, je décris en détail ces 2 classes en prenant exemple sur une classe du noyau xoops XoopsPrivmessage qui s'appuie sur la table priv_msgs puis dans une seconde partie je présente un petit générateur de code de ce module. XoopsObjectHandler et XoopsObject nous allons utiliser comme exemple la table priv_msgs qui permet d'envoyer des messages entre membres du portail.

Figure 1 : Description de la table priv_msgs
XoopsObject est la classe mère permettant de manipuler les attributs d'un objet données. Il s'appuie sur le tableau \$vars pour manipuler les attributs de l'objet. \$vars contient les attributs de l'objet sous la forme clé, valeur. La clé désigne le nom de l'attribut c'est à dire souvent le nom d'une colonne de la table de BD. XoopsObject offre des mécanismes génériques d'accès aux données ; ses principales méthodes sont :->initVar (\$key, \$data_type) permettant d'initialiser la définition d'un attribut ->setVar(\$key, \$value) qui met à jour l'attribut ->getVar(\$key) qui restitue la valeur de l'attribut ->cleanVars () nettoie les attributs des caractères spéciaux pour les stocker dans la BD La classe associée au Privmessage ne doit ainsi finir que les attributs qu'elle veut gérer class XoopsPrivmessage extends XoopsObject

```
{  
  
/**  
 * Constructor  
 **/  
 * * * * * fonction XoopsPrivmessage()  
 * * * * * {  
 * * * * * $this->XoopsObject();  
 * * * * * $this->initVar('msg_id', XOBJ_DTYPE_INT, null, false);  
 * * * * * $this->initVar('msg_image', XOBJ_DTYPE_OTHER, 'icon1.gif', false, 100);  
 * * * * * $this->initVar('subject', XOBJ_DTYPE_TXTBOX, null, true, 255);  
 * * * * * $this->initVar('from_userid', XOBJ_DTYPE_INT, null, true);
```

```

    $this->initVar('to_userid', XOBJ_DTYPE_INT, null, true);
    $this->initVar('msg_time', XOBJ_DTYPE_OTHER, null, false);
    $this->initVar('msg_text', XOBJ_DTYPE_TXTAREA, null, true);
    $this->initVar('read_msg', XOBJ_DTYPE_INT, 0, false);
}

```

Il existe plusieurs manières d'utiliser un tableau plutôt qu'autant d'attribut de classe que de variables (couramment fait en java). Il est plus efficace en terme de temps d'exécution de manipuler de tels tableaux car ce modèle est plus proche de la structure de la requête http postée. En travaillant sur un tableau on économise le chargement du tableau en variable. La requête (issu par exemple d'un formulaire de mise à jour d'un objet) s'utilise directement pour charger un objet. Voici un exemple ou par convention les champs HTML vont être nommés comme le nom des attributs. \$pm = &\$pm_handler->get(\$idPm);

```

// Occupation de l'objet
$pm->setVars($_post); // Mise à jour de l'objet à partir de la requête
$pm_handler->insert($pm); // mise à jour physique dans la BD C'est un peu lourd
// puisque l'on passe toute la requête mais le setVars fera le "mapping" en vérifiant si la clé est
// un attribut géré par la class. Une autre solution est de parcourir les vars de la class pour aller
// chercher que ceux dont on a besoin :
foreach($pm->getVars() as $key => $value) {
    if(isset($_POST[$key])) {
        $pm->setVar($key, $_POST[$key]);
    }
}

```

Quelque soit l'option choisie, en passant l'objet au handler, la mise à jour est stockée dans la BD : \$pm_handler->insert(\$pm); **XoopsObjectHandler** est une classe abstraite qui une fois implémentée permet de manipuler les objets d'une class particulière (ma classe) étendant la class xoopsObject. Ce Handler (ou manipulateur) peut se occuper à l'aide de la méthode xoops_gethandler du fichier /include/function.php en passant le nom de la classe ma classe que l'on veut manipuler. Attention toutefois car cette méthode se base sur une convention de nom, il faut avoir nommé le handler xoopsmaclasseHandler. On peut aussi simplement instancier la class.

```

$pm_handler = &xoops_gethandler('privmessage');
// Ou bien
$pm_handler = new XoopsPrivmessageHandler($xoopsDB)

```

Voici les principales méthodes proposées par ce handler qu'il s'agit d'implémenter :
 - create : création d'un nouvel objet à l'aide de la méthode (référence au pattern fabrique)
 - insert : modification d'un objet
 - delete : suppression d'un objet
 - get(\$id) : accès direct à un objet à l'aide de son identifiant

Les différentes implémentations de XoopsObjectHandler des classes du répertoire /kernel introduisent 2 autres méthodes bien utiles non présentes dans la classe mère :
 - getObjects renvoie sous forme de tableau de xoopsobjet une sélection d'objets à partir d'un ensemble de critères (class criteria du répertoire /class). Ceci est bien utile pour construire un formulaire de recherche
 - getCount renvoie le nombre d'occurrences d'une sélection d'objets à partir de critères

Typiquement voici un exemple de sélection de l'ensemble de privateMsg d'un utilisateur puis l'affichage du sujet et de la date du message :
 \$criteria = new Criteria('to_userid', \$xoopsUser->getVar('uid'));

```

$pms = &$pm_handler->getObjects($criteria);
foreach($pms as $pm) {
    echo $pm->getVar('subject').'. $pm->getVar('msg_time');
}

```

L'utilisation avec un template smarty sera aussi simple en affectant l'ensemble de l'objet :
 \$pms = &\$pm_handler->getObjects(new Criteria('to_userid', \$xoopsUser->getVar('uid')));
 \$xoopsTpl->assign('pms', \$pms); Associe au template smarty : section name=i loop=\$pms}

```
$pms[i]->vars.À subject.value}>
```

```
$pms[i]->vars.À msg_time.value}>
```

section}> Class_generator : un g n rateur tr s simple. Pour  viter le fastidieux codage de l'impl mentation de ces 2 classes, nous avons d velopp  un petit g n rateur qui nous a fait gagn  du temps et a fiabilis  et homog nis  le code de type DAO. Il s'appuie sur un seul template smarty d crivant le fichier des 2 classes impl mentent respectivement XoopsObject et XoopsObjectHandler. Le template peut bien entendu  tre modifi  pour prendre en compte les besoins sp cifiques. Ce g n rateur utilise uniquement les informations issues de la base de donn es ce qui impose d'avoir d j une table de base de donn e sur laquelle s'appuyer. Class_generator g n re un fichier par table de Base de donn es. Il se base aujourd'hui sur la convention que la table source ne doit poss der qu'une seule cl  primaire. Ceci pour g n rer la m thode du handler getld(\$maClePrimaire). Une fois le module install , l'acc s au g n rateur s'effectue dans la partie administration. Dans le formulaire " G n rer une class " apr s avoir s lectionn  le module sur lequel on travaille et la table de BD existante, le clic sur le bouton " g n rer " g n re un fichier nomm  du nom de la table de BD s lectionn e dans le r pertoire /class du module. Le module est t l charg  sur le site dev.oryxvet.com OryxVet