12

Réalisation de produits en Python

À quoi bon soulever des montagnes
quand il est si simple de passer par-dessus.
— Boris Vian

Nous avons vu, au chapitre précédent, comment réaliser un type particulier de produits, à savoir les ZClasses. Celles-ci, pourtant très simples à mettre en œuvre, n'en présentent pas moins de sérieuses limitations.

Dans ce chapitre, nous détaillerons la mise en œuvre de produits Zope, qui autorisent le degré d'extension maximal avec Zope. L'application en est donnée dans l'étude de cas présentée au chapitre 18, sur la création d'un site de news en Zope.

Le contenu très technique de ce chapitre semble davantage le destiner aux développeurs qu'aux concepteurs de sites web, mais sa lecture peut cependant éclairer tout un chacun sur les mécanismes internes de Zope.

Les produits Zope ou l'approche par composants

Dans son approche novatrice, Zope met en avant un concept qui a fait ses preuves dans le domaine du développement non spécifique au Web : le composant. Zope est ainsi le premier environnement web à proposer au développeur une programmation objet par composants.

Remarque

Alors que dans le chapitre précédent nous avons étudié la création de classes avec l'interface d'administration de Zope (ZClasses), nous allons ici détailler l'écriture de classes en Python. Le choix d'une technique plutôt que l'autre est fonction de la complexité du produit et des restrictions de sécurité imposées par les ZClasses.

Qu'est-ce qu'un produit ?

Un produit Zope est un ensemble de modules écrits en Python, qui interagissent étroitement avec Zope. Les produits sont installés sur le système de fichiers, avec le code de Zope, et sont soumis à nettement moins de restrictions que les objets éditables à travers l'interface d'administration de Zope (`DTML Documents`, scripts Python…).

Un produit est dans la plupart des cas composé de classes qu'il est possible d'instancier dans l'arborescence Zope comme les autres classes de Zope (`DTML Document`, `Folder`…).

Remarque

Lorsqu'un produit n'est composé que d'une seule classe, on utilise par extension le terme « produit », aussi bien pour le produit que pour la classe.

Un produit est un composant

Un composant est une brique logicielle fortement réutilisable. Les composants se répartissent en deux grandes familles : les composants techniques (par exemple, une case à cocher, un serveur HTTP, etc.) et les composants métiers (une facture, un client, etc.).

La programmation par composants se déroule généralement en deux temps et n'implique pas forcément les mêmes personnes :

- Conception, développement et test du composant – cette tâche est très technique.
- Assemblage des composants – cette tâche peut paraître plus simple : la complexité technique du composant est cachée ; l'utilisateur assembleur de composants se concentre uniquement sur la partie « fonctionnelle » de son assemblage.

Un produit est packagé

Un produit est un élément logiciel fourni clés en main. Une fois l'installation du produit terminée, l'utilisateur a très peu d'opérations à faire pour l'utiliser. Il n'a (quasiment) que des problèmes fonctionnels à gérer. Quant à la complexité technique, elle est masquée dans le code source du produit conçu par le développeur. Le produit est prêt à être utilisé.

Un produit est partagé

Autre énorme avantage de l'approche produit, le même code est utilisé ou partagé – au sens du partage par la communauté – par plusieurs sites à la fois. La qualité du code du composant est ainsi grandement éprouvée dans ces conditions, et la qualité intrinsèque du produit est potentiellement bien meilleure que si ce même bout de code n'était exécuté que sur un site.

Remarque

Plus un composant est distribué, plus la combinatoire des cas d'utilisation rencontrés augmente, plus les bogues potentiels sont soulevés puis corrigés, et meilleure est la garantie de qualité intrinsèque du logiciel.

De la même façon, un composant partagé par un grand nombre de sites et très utilisé par la communauté bénéficie en général d'une meilleure documentation, parfois récrite selon un autre point de vue par d'autres utilisateurs, étendue par des développeurs tiers, etc. Plus un produit est partagé, plus il se bonifie !

Remarque

Un produit peut très bien rester dans le giron de son créateur et ne jamais voir le jour au sein de la communauté. Une société peut très bien décider de développer des composants Zope, de les utiliser pour ses besoins propres et de ne pas les partager avec le reste de la communauté, par exemple lorsqu'il s'agit d'un fort avantage concurrentiel. Hormis le respect des licences, rien n'impose de partager son composant.

On voit ainsi éclore un modèle hybride : les composants sont conservés jalousement pendant un temps donné, celui de l'avantage concurrentiel (6 mois, 1 an, 2 ans), puis lâchés dans le domaine public (avec une licence GPL par exemple), pour un juste retour à la communauté ! La très grande majorité des produits de Zope (c'est son extraordinaire valeur ajoutée) sont des logiciels libres !

Un produit est sécurisé

Un produit, en tant que brique logicielle packagée, doit gérer sa propre sécurité. Il est responsable de la sécurité des opérations qu'il autorise. Un produit gère la sécurité d'opérations données (ajouter un produit X, modifier la propriété d'un produit X…), opérations que le concepteur de site doit ensuite pouvoir déléguer aux utilisateurs ou webmasters du site au travers des rôles – comme il le fait avec les autres produits standard de Zope.

Un produit est fortement réutilisable

Un produit est une brique logicielle. À ce titre, il est conçu pour rendre un service donné, avec pour objectif de pouvoir rendre ce service dans des contextes aussi divers que possible. Pour le développeur de produit, cela signifie faire le moins d'hypothèses possibles sur l'environnement technique dans lequel sera déployé son produit (plate-forme technique, présence de tel ou tel logiciel, présence de tel ou tel produit, nommé de telle ou telle façon, etc.).

On attend également d'un produit fortement réutilisable qu'il puisse s'étendre et qu'il soit à même de modifier quelques comportements par défaut, si possible d'une manière simple. Python, langage interprété orienté objet, est alors d'une grande aide.

Un produit est maintenable

Un produit, comme tout autre logiciel, est rarement parfait ou exempt de bogue lors de sa première utilisation. De ce fait, il est important que les utilisateurs de produits puissent mettre à jour leur version installée du produit. Avec Zope, ce processus de mise à jour est simple car les produits sont packagés. Dans la très grande majorité des cas, une simple réinstallation du produit et un redémarrage de Zope sont suffisants. Dans le cas où les données utilisées par le produit changent d'une version à l'autre, mieux vaut (c'est même recommandé) que le développeur fournisse un outil pour convertir les données d'un format à l'autre, ou au moins une documentation très détaillée sur la marche à suivre.

Démarche qualité du développement de produits

Le développement de produits impose quasiment que l'on adopte une démarche qualité, ce qui constitue son principal atout. Le logiciel y gagne en effet en qualité globale.

On verra que cette démarche induit un strict respect des phases d'analyse, conception, développement et test dans la réalisation d'un produit.

Bien que tout élément logiciel soit censé avoir subi l'épreuve de la conception, force est de constater que ce n'est pas toujours le cas ! Les composants logiciels, de par leur nature un peu plus abstraite, sont généralement issus d'une vraie phase de conception, au moins intellectuelle.

Ces phases (notamment, la conception et les tests), obligatoires pour obtenir un logiciel de qualité, sont un réel apport en matière de garantie de fonctionnement, ce qui est l'objet même d'un élément logiciel.

Quand développer un produit ?

Cette question est fondamentale car le produit n'est évidemment pas la solution miracle à tous les problèmes qui peuvent survenir dans la réalisation d'un site avec Zope.

Pour savoir dans quelles situations on peut développer un produit, il faut bien avoir à l'esprit les avantages, énoncés plus haut, qui s'y rapportent. *A priori*, il est utile de se lancer dans le développement d'un produit dans les cas suivants :

- Le produit amène une solution générale à un problème récurrent (réutilisabilité). Ex. : une interface avec un système externe (base XML par exemple), un outil web (barre de navigation de site web), etc.
- Le produit offre une solution à un problème technique particulier, assez complexe. Ex. : site multilingue, virtual hosting, etc.

- Le produit propose une version packagée d'un type de site web particulier, réutilisable. Dans ce cas, on a affaire à de « gros » produits, dont l'exemple le plus connu est Squishdot. Ex. : produit de gestion de magasin online, de news avec commentaires, de dialogue, etc.

En dehors de ces situations, le développement d'un produit est souvent une mauvaise approche. Il est très rare en effet qu'il faille recourir à un produit pour répondre à un problème fonctionnel isolé. Par exemple, écrire les règles de gestion applicative d'un site web à l'intérieur des méthodes d'un produit est rarement une solution idéale, sauf s'il s'agit de résoudre des problèmes génériques (cas du panier d'achats, par exemple).

Quelques rappels

Avant d'aller plus loin dans la réalisation de produits, nous allons rappeler quelques notions fondamentales de la théorie objet et approfondir le concept d'acquisition, que nous avons utilisé auparavant.

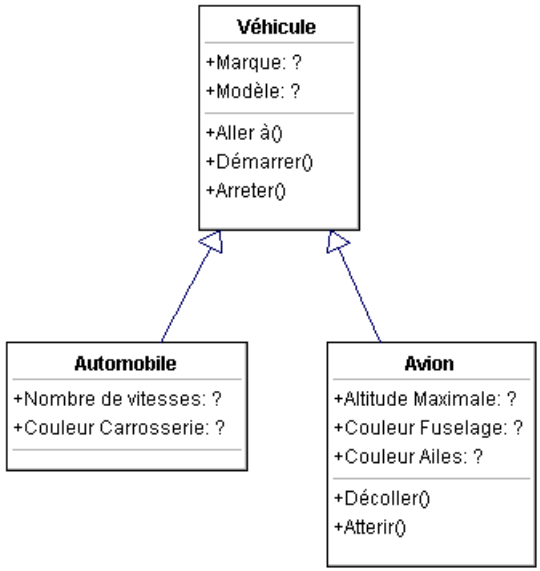
Rappels sur l'héritage

Sans reprendre toutes les théories relatives à l'héritage, nous nous contenterons ici d'éclairer certains concepts fondamentaux directement liés à Zope.

L'héritage est un mécanisme fondamental de la théorie objet. Il s'applique entre des classes. On dit qu'une classe A « hérite » d'une classe B si la structure de la classe A reprend au moins dans tous ses éléments la structure de la classe B. Ainsi, toutes les instances de la classe A seront caractérisées par au moins toutes les caractéristiques des instances de la classe B.

L'héritage doit traduire la relation « est une sorte de », comme le montre le modèle de classe suivant (voir figure 12-1), qui utilise la notation UML. Dans la notation UML, les relations d'héritage sont symbolisées par des flèches.

Figure 12-1
Relation d'héritage en notation UML



Dans ce modèle simpliste, on note les deux relations d'héritage : l'automobile hérite du véhicule et l'avion hérite du véhicule. En lisant la relation dans l'autre sens, on dit que la classe des véhicules englobe celle des automobiles et des avions.

Remarques sur l'acquisition

Comme nous l'avons vu dans les chapitres précédents, l'acquisition permet à un dossier d'utiliser (ou d'acquérir) les propriétés d'un de ses dossiers conteneurs. Mais cet exemple ne montre que la partie la plus visible de l'acquisition. L'acquisition est un mécanisme à la fois simple et complexe. Simple, précisément, parce qu'il s'énonce simplement : « Mécanisme qui permet à un composant d'hériter les propriétés de l'objet qui le contient, récursivement. » Complexe, car il faut un certain temps avant de maîtriser ses finesses et ses effets de bord.

En effet, contrairement à bon nombre de mécanismes objets tels que l'héritage, l'acquisition est un phénomène qui s'applique à des *instances* et non à des classes. C'est là un point fondamental : l'acquisition est la faculté qu'a une instance de s'approprier les attributs et méthodes d'une autre instance, dynamiquement. En poussant ce raisonnement, on peut rapprocher l'acquisition de la notion d'« héritage d'instance ».

Réalisation d'un produit minimal

La partie fonctionnelle

Le premier produit que nous allons réaliser est purement démonstratif, mais il va nous permettre d'exploiter tout le potentiel d'un produit. Il a simplement pour objet d'afficher un « Hello World ! », en gras, au sein d'une page web rendue par Zope (voir figure 12-2).

Nous allons partir du minimum fonctionnel de notre produit et y ajouter successivement les différentes couches logicielles qui le finaliseront et le rendront conforme à Zope. Une fois réalisé, il apparaîtra alors dans la liste des produits disponibles, au même titre que `DTML Document` ou `Image`. Il suffira ensuite de l'instancier, puis d'écrire `<dtml-var monProduit>` pour afficher « Hello World ! » en gras dans le corps de la page web qui sera rendue.

Le minimum fonctionnel du produit est réalisé par le code suivant :

```
class ZhelloWorld:

    def render(self) :
        return "<B>Hello World !</B>"
```

On reconnaît là la définition d'une classe nommée `ZhelloWorld` (le nom de notre produit), et d'une méthode chargée du rendu, `render`. Cette méthode renvoie juste la chaîne demandée, formatée en HTML.

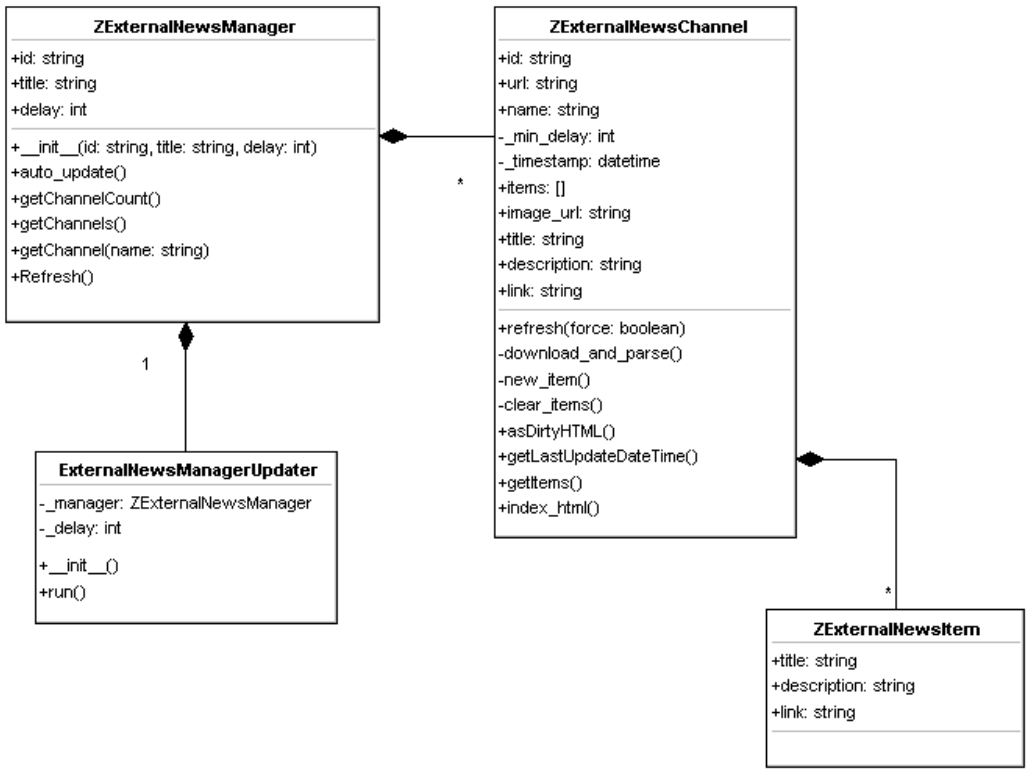


Figure 12-2
Le produit ZhelloWorld en action

Emplacement du code source dans le système de fichiers

Il convient tout d'abord de savoir où placer les fichiers du produit. Contrairement à la grande majorité des éléments de Zope, les produits se stockent sur le système de fichiers. Il doivent être placés dans le répertoire `$ZOPE/lib/python/Products/`*`MyProduct`*, où *`MyProduct`* est le nom du produit (le nom du répertoire n'a aucune incidence technique). Ici, nous parlerons du produit `ZHelloWord` (que nous décrirons ultérieurement).

> **Remarque**
> *Dans certains cas, il est possible de créer un répertoire* `$ZOPE/Products` *pour simplifier l'administration. Nous n'en faisons pas mention ici.*

On trouvera donc dans ce répertoire, dans notre cas, au moins les fichiers suivants :

```
ZHelloWorld.py
addZHelloWorldForm.dtml
```

Le choix du stockage des produits sur le système de fichiers est pour l'essentiel lié à la sécurité. En effet, un produit Zope étant un code source Python sans restrictions, il est donc possible, depuis ce code source, d'accéder à toutes sortes de ressources, par exemple sur la machine serveur ou sur le réseau local du serveur. Les auteurs de Zope ont donc décidé de confier la responsabilité de la sécurité des produits Zope à l'administrateur du système qui héberge Zope. Ainsi la politique de sécurité des produits Zope est-elle réduite à un problème de sécurité de fichiers sur un système de fichiers, problème classique auquel les administrateurs système sont rompus. Cette stratégie interdit l'installation d'un produit par un utilisateur qui n'a accès qu'à l'interface web du serveur Zope, ce qui renforce la sécurité en empêchant l'installation de produits sur un système « public ».

Une deuxième explication tient à ce qu'en tant qu'extension de Zope, les produits Zope trouvent leur place naturellement à côté des sources de Zope lui-même, sur le système de fichiers (le chemin l'atteste).

Enfin, le développement de produits Zope étant une activité d'extension de Zope, et donc de développement, il est avant tout confié aux développeurs, qui ont leurs habitudes et leurs outils dans ce genre d'environnement (notamment avec CVS, pour y stocker et gérer leurs fichiers source).

Évolution vers un produit

Il y a plusieurs règles à respecter pour créer un produit. Elles sont quasiment toutes techniques et ont notamment pour objet de renseigner Zope sur l'existence et le comportement du produit.

Nous allons maintenant modifier et faire évoluer le code source présenté plus haut. Les lignes ajoutées ou modifiées apparaissent en gras. Commencez par écrire le code précédent dans le fichier `$ZOPE/lib/python/Products/ZhelloWorld/ZhelloWorld.py`.

Les méthodes doivent êtres publiques

La première modification est simple : tout code qui interagit avec le navigateur du client doit être « public ». Cette notion, qui n'existe pas en tant que telle dans Python, est ici formalisée par une convention simple de rédaction du code source : une méthode est dite publique si un commentaire lui est associé. Cette astuce présente l'avantage de contraindre les développeurs à documenter leur code, de la même façon que l'indentation de Python les force à rédiger ce même code d'une manière lisible.

Rappel

Pour documenter une méthode ou une classe en Python, on ajoute une ligne de chaîne dans la ligne qui suit la déclaration. Ce commentaire est appelé `docstring` *ou chaîne de documentation.*

```
class ZhelloWorld :
  """Produit pour apprendre"""

    def render(self) :
      """render(self) -> string. Retourne la chaine formatée en HTML"""
        return "<B>Hello World !</B>"
```

Les produits doivent fournir un constructeur

Comme on a pu le signaler dans les chapitres précédents, tous les objets de la base objet de Zope ont un nom, unique dans le contexte du dossier dans lequel ils résident : l'`id`. Cet identifiant unique intervient très tôt dans le cycle de vie du produit : dès son instanciation.

La méthode qui est appelée lors de l'instanciation d'une classe est appelée constructeur. En Python, il s'agit de la méthode dédiée `__init__`. Dans le cadre d'un produit, cette méthode est appelée lorsque l'utilisateur a choisi de créer une nouvelle instance du produit, en lui donnant un `id`. Zope transmet alors à la nouvelle instance son `id`.

```
class ZhelloWorld
  """Produit pour apprendre"""

    def __init__(self, id):
      """Constructeur, méthode d'initialisation"""
      self.id=id

    def render(self) :
      """render(self) -> string. Retourne la chaine formatée en
      HTML"""
       return "<B>Hello World !</B>"
```

On se contente de stocker l'`id` qui nous est transmis. On reviendra sur le mécanisme exact qui se cache derrière la simple notation `self.id`.

Rendu d'un produit

Notre produit a désormais la faculté d'exister dans une base ZODB. Voyons comment l'utiliser dans une page web en rappelant que l'objectif est de pouvoir l'appeler dynamiquement avec `<dtml-var` *`monInstance`*`>`.

Mais qu'entend-on exactement par « l'appeler », et quelle est l'action qui est déclenchée lorsqu'on utilise `<dtml-var monInstance>` ? Comme on l'a vu dans les chapitres précédents, cette

instruction est en fait un raccourci pour `<dtml-var "monInstance(_.None, _)">` ; on retrouve ici l'invocation d'une méthode en langage Python. Cette méthode est un peu particulière car elle est implicite : `monInstance` est en effet une instance d'objet, pas une méthode. Il faut ici se souvenir que, lorsqu'on invoque directement une instance, c'est la méthode spéciale `__call__` qui est appelée. Les lecteurs avertis auront reconnu ici un mécanisme général de Python.

C'est ce raccourci dont se sert Zope pour permettre une notation simple et élégante.

Pour remplir notre objectif, nous pouvons utiliser l'implémentation suivante pour la méthode `__call__` :

```
def __call__(self, client=None, REQUEST={}, RESPONSE=None, **kw):
 """__call__() -> string. Méthode appelée lors de l'invocation
 ➥par <dtml-var >. Chargé de renvoyer la chaîne à afficher.
   return self.render()
```

Les paramètres de cette méthode sont particuliers : ils sont contextuels et automatiquement renseignés par Zope. L'utilisateur du produit ne les utilise jamais explicitement. Ici, les paramètres sont les suivants :

- `client` : client vaut généralement `None` au sein de Zope ; cette variable est utilisée pour pouvoir exploiter la classe indépendamment de Zope.
- `REQUEST` (facultatif) : variable de contexte de Zope.
- `RESPONSE` (facultatif) : objet HTTP que le serveur Zope s'apprête à renvoyer. L'accès à cet objet est très utile, par exemple pour faire des « redirect ».
- `**kw` : paramètres nommés annexes (transmis sous la forme d'un dictionnaire, voir chapitre 9, « Le langage Python »).

Cela étant dit, on voit bien que cette méthode se contente de déléguer à la méthode `render` le rendu et renvoie simplement le résultat, qui sera inséré en lieu et place de `<dtml-var monInstance>` dans le DTML appelant, pour être rendu sur le navigateur du client.

Un produit a un nom

Dans l'interface d'administration, un produit est identifié par un nom tel que `DTML Document`, `DTML Method`, `Image`, `Folder`… Ce nom est également appelé le métatype du produit. Il apparaît dans dans la liste déroulante Add d'une page de `Folder` et permet d'ajouter une instance de produit dans le dossier courant. Il est obligatoire de spécifier ce nom, sous forme de simple chaîne portée par la variable de classe `meta_type` :

```
class ZhelloWorld
  """Produit pour apprendre"""

    meta_type = 'Hello, World'

    def __init__(self, id):
```

```
        """Constructeur, méthode d'initialisation"""
        self.id=id
...
```

Attention

Il ne faut pas confondre le `meta_type` *d'un produit et l'*`id` *des instances des produits !*

Un produit est une sous-classe d'une classe de Zope

Pour fonctionner correctement, notre produit doit se conformer à l'API de Zope.

Pour simplifier le développement et faciliter l'intégration entre les différents composants, ce Framework propose des classes de base pour tous les produits. Ces classes fournissent un ensemble de services de base, comme l'enregistrement du produit auprès de Zope, la persistance des attributs/propriétés, l'accès à l'acquisition, l'insertion des instances de produits dans ZODB, etc.

Il y a deux grandes classes de base : `Item` et `ObjectManager`. Il faut retenir que `ObjectManager` est la classe de base des objets conteneurs (dont relève le produit standard `Folder` lui-même), et `Item` celle des produits atomiques (tel `DTML Document`). Dans notre cas, c'est évidemment la classe `Item` qui correspond à notre problème.

La classe `Item` se trouve dans le module `SimpleItem`, du package `OFS` (pour Object File System, package de base de ZODB).

On obtient :

```
from OFS import SimpleItem

class ZhelloWorld(SimpleItem.Item)
  """Produit pour apprendre"""

    meta_type = 'Hello, World'

    def __init__(self, id):

      """Constructeur, méthode d'initialisation"""
      self.id=id
..
```

Nous reviendrons plus en détail sur les différentes classes de base dans la suite de ce chapitre. Il est intéressant de noter que nous avons déjà évoqué l'héritage basé sur `ObjectManager` dans le chapitre précédent.

Un produit utilise un formulaire HTML pour l'instanciation

Lorsqu'on ajoute un objet à un `Folder`, Zope commence par présenter une page où les informations d'instanciation peuvent être spécifiées, telles que l'`id` de la nouvelle instance.

C'est au produit de fournir cette page à Zope.

La partie *utile* de cette page peut se présenter comme suit :

```
<form action="manage_addZHelloWorld" method="post">
  Id: <input type="text" name="id:string" size="25" value=""><br>
  <input type="submit" name="Add">
</form>
```

On note la présence d'un formulaire, qui demande la saisie d'une variable `id` et la transmet à la page `manage_addZHelloWorld`. Cette méthode est détaillée juste après : elle se charge d'instancier le produit et de le stocker dans la ZODB.

La page d'instanciation est rarement écrite aussi sommairement ; on la complète plus souvent de la façon suivante :

```
<dtml-let form_title="'Add ZHelloWorld'">
<dtml-if manage_page_header>
  <dtml-var manage_page_header>
  <dtml-var manage_form_title>
<dtml-else>
  <html><head><title>&dtml-form_title;</title></head>
  <body bgcolor="#ffffff">
  <h2>&dtml-form_title;</h2>
</dtml-if>
</dtml-let>

<p class="form-help"> Nous allons créer une nouvelle instance de ZHelloWorld.
Il faut lui trouver un nom : </p>

  <form action="manage_addZHelloWorld" method="post">
    <table cellspacing="2">
      <tr>
        <th class="form-label">id</th>
        <td><input type="text" name="id:string" size="25" value=""></td>
      </tr>
      <tr>
        <th></th><td class="form-element"><input class="form-element"
        type="submit" name="Add"></td>
      </tr>
    </table>
  </form>
```

```
<dtml-if manage_page_footer>
  <dtml-var manage_page_footer>
<dtml-else>
  </body></html>
</dtml-if>
```

Fonctionnellement, cette version est identique à la précédente, à ceci près qu'elle est graphiquement plus élégante. On y retrouve en italique la partie *utile*. Les parties du début et de la fin se chargent de rendre cette page homogène avec les pages de produits standard de Zope (on notera l'utilisation d'une feuille de styles). Il est conseillé de prendre cette habitude, même si l'ajout des éléments de charte graphique n'est en rien obligatoire, techniquement.

À présent, où doit-on stocker cette page ? Ou plutôt, dans un premier temps, comment Zope va-t-il la réclamer ?

Par un processus de déclaration (ou d'enregistrement) sur lequel nous reviendrons, on fournit à Zope le nom d'une fonction dans un module, qui est chargée de retourner cette page (son source HTML/DTML en l'occurrence). Habituellement, cette fonction est nommée `add`*`MyProduct`*`Form` (où *`MyProduct`* est le nom de la classe du produit). Ici, cette fonction s'appelle `addZHelloWorldForm`.

On pourrait donc mettre le texte de cette page dans une chaîne, directement dans la méthode `addZHelloWorldForm`. On préfère à cette solution, peu confortable et peu maintenable, mélangeant le code et l'apparence, celle qui va consister à stocker cette page dans un fichier externe.

Pour ce faire, nous allons créer un fichier `add`*`MyProduct`*`Form.dtml`, où *`MyProduct`* est le nom de la classe du produit, ce qui donne ici `addZHelloWorldForm.dtml`. Nous verrons ultérieurement dans quel répertoire stocker ce fichier.

La fonction `add`*`MyProduct`*`Form` fait un simple appel à une fonction fournie par Zope, qui se charge de lire le fichier et de le renvoyer. Cette fonction s'appelle `GetDTML` et se trouve dans le module `Globals`, fourni par Zope.

```
from Globals import DTMLFile
...

class ZhelloWorld(SimpleItem.Item)
 """Produit pour apprendre"""


...
    def addZHelloWorldForm(self):
        return DTMLFile('addZHelloWorldForm', globals())
...
```

Le deuxième paramètre, `globals()`, sera familier aux utilisateurs assidus de Python. Cette fonction renvoie l'ensemble de l'espace de noms courant (souvent appelé contexte). Elle permet à la fonction `DTMLFile` d'avoir à sa disposition l'ensemble des variables nécessaires à la résolution des variables contenues dans le fichier DMTL fourni.

Une instance de produit s'insère dans un `Folder`

Lors de son instanciation, un produit doit fournir les mécanismes qui lui permettent de s'insérer dans un `Folder`. Pour ce faire, les classes de base, dont le produit hérite, fournissent une méthode `_setObject()` qu'il suffit d'invoquer.

Cette fonction doit elle-même être appelée dans une fonction usuellement désignée `manage_add`*`MyProduct`*, où *`MyProduct`* est le nom de la classe du produit (à ne pas confondre avec son métatype). C'est en fait *cette* fonction qui est appelée par l'interface d'administration de Zope après qu'elle a sélectionné un nom de produit à instancier, et après que l'utilisateur a donné l'`id` de l'objet à insérer.

```
class ZhelloWorld(SimpleItem.Item)
"""Produit pour apprendre"""
...
    def manage_addZHelloWorld(self, id, RESPONSE=None):
    """manage_addZHelloWorld() -> nothing
       Insère l'objet dans son conteneur et retourne la page
       index_html"""
         self._setObject(id, ZHelloWorld(id))
         RESPONSE.redirect('index_html')
...
```

L'implémentation demande quelques explications, notamment la ligne `self._setObject(id, ZHelloWorld(id))` :

- `ZHelloWorld(id)` renvoie une nouvelle instance de la classe `ZHelloWorld` : notre instance de produit. La valeur `id` est transmise au constructeur : elle est fournie par Zope et transmise après saisie par l'utilisateur. On remarque que l'appel au constructeur correspond bien à la déclaration qu'on en a fait plus haut.
- `self._setObject()` prévient ZODB que l'on ajoute un nouvel objet, dont on précise l'`id` et dont on fournit la valeur.
- Le `RESPONSE.redirect` permet d'afficher une page par défaut après l'insertion. C'est la page `"index_html"` qui est choisie ici.

Au final, on peut dire que cette opération équivaut à insérer une nouvelle instance de `ZHelloWorld`, avec l'id `id`, et à afficher la page par défaut du dossier courant.

Distribution d'un produit

Un package Python

Un produit Zope est distribué et il interagit avec le système en tant que package Python. On aura gardé à l'esprit qu'un package Python dispose d'un fichier un peu particulier, `__init__.py`, qui est exécuté au moment du chargement du produit, au démarrage du serveur Zope.

Pour un produit Zope, c'est ce fichier qui est chargé du référencement du produit auprès de Zope : on va y déclarer les noms de méta-classes, leur icône, le nom des méthodes à invoquer pour la création des instances des classes, etc.

Il faut noter qu'un même package ne peut contenir qu'un seul produit.

Pour notre produit, le fichier `__init__.py` est le suivant :

```
import ZHelloWorld

  meta_types=(
    {
      'name' : 'ZHelloWorld',
      'action' : 'manage_addZHelloWorldForm' },
    )

  methods={
    'manage_addZHelloWorldForm' : ZHelloWorld.addZHelloWorldForm,
    'manage_addZHelloWorld' : ZHelloWorld.addZHelloWorld,
    }

__ac_permissions__=(
    ('Add ZHelloWorld', ('manage_addZHelloWorldForm', 'manage_addZHelloWorld')),
  )

  misc_={
    'ZHelloWorldIcon' : ImageFile('ZHelloWorldIcon.gif', globals()),
  }
```

On y retrouve les éléments suivants :

- l'importation du module qui contient le produit ;
- les noms de métatypes concernés, ainsi que le nom de la méthode à invoquer quand un utilisateur appuie sur le bouton Add (variable `meta_types`) ;
- `ac_permissions` définit la politique de sécurité, sur laquelle on reviendra ;
- `misc_` contient des éléments complémentaires. Ici on n'a ajouté qu'une icône, sous la forme d'un fichier `gif`. C'est cette icône qui est utilisée pour identifier visuellement les instances du produit dans les dossiers Zope.

Il faut noter qu'on a introduit un nouveau fichier externe : `ZHelloWorldIcon.gif`, qu'il faut penser à ajouter dans le répertoire du produit.

Notre produit est donc composé à ce stade des fichiers suivants :

- `ZHelloWorld.py`
- `addZHelloWorldForm.dtml`
- `__init__.py`
- `ZhelloWorldIcon.gif`

De la même façon, le fichier `__init__.py` peut contenir une méthode `Initialize`, appelée lors du (re)démarrage de Zope. On sera amené à utiliser cette fonctionnalité.

Cosmétique

Les dernières modifications sur notre produit sont d'ordre cosmétique. On a remarqué que, dans l'interface d'administration d'un quelconque élément de Zope, il y a une barre d'onglet en haut de la fenêtre : les onglets de cette barre diffèrent (en nombre et en contenu) suivant le produit concerné.

On peut ajouter ces onglets au travers de la variable de type tuple nommée `manage_options`, qui est une variable de classe du produit. Ce tuple contient autant d'entrées que d'onglets à ajouter, chaque entrée étant un dictionnaire qui décrit l'onglet.

Ce dictionnaire doit contenir les clés suivantes :

- label : c'est la chaîne qui sera utilisée pour le nom de l'onglet à l'écran ;
- action : nom de la méthode à invoquer lorsqu'on clique sur l'onglet.

Dans notre cas, il faut ajouter :

```
class ZhelloWorld(SimpleItem.Item):
  """Produit pour apprendre"""
...
   manage_options = (
    {'label' : 'View', 'action' : 'render'},
  )
...
```

Remarque

Les classes de base apportent les onglets minimaux tels que Content pour les `Folder`.

Mise en boîte

Notre produit minimal est maintenant terminé. On peut le distribuer sur d'autres systèmes en préparant une archive `.tar.gz` (dite « tarball ») ou `.zip` qui contient simplement tous les fichiers que l'on a préparés.

Pour déployer le produit sur une autre machine, il suffira de décompacter l'archive en question de telle sorte que les fichiers apparaissent à nouveau dans `$ZOPE/lib/python/Products/`*`MyProduct`*, où *`MyProduct`* est le nom du produit.

Quant à la protection du code source, il faut savoir que l'on peut distribuer un produit Zope sans fournir le code source : pour ce faire, il suffit de ne copier que les fichiers `.pyc` dans l'archive. Ces fichiers sont les versions compilées par Python des fichiers `.py`.

Méthodologie de développement

On trouvera sous ce paragraphe de simples recommandations, qui sont le fruit de l'expérience de l'auteur dans le développement de produits et de composants en général. Ces éléments relèvent plutôt d'une approche globale que technique.

Avant de suivre ces quelques conseils, il faut se demander si le problème que l'on se propose de résoudre peut bien l'être avec l'écriture d'un produit.

Cas d'utilisation

Dans le développement de composants, on doit en premier « coucher » sur le papier les cas d'utilisation, issus de la méthodologie UML. Ces cas d'utilisation doivent mettre en lumière les différentes fonctions du produit que l'on veut développer mais aussi, et surtout, ses interactions avec les utilisateurs.

Cette première étape, familière aux utilisateurs connaissant UML, permet de ne rien omettre dans l'expression des besoins. On dispose ainsi, en début de projet, d'une référence fonctionnelle que l'on peut consulter à tout moment lors de l'implémentation, et qui peut servir de base à un plan de tests, puis à une documentation utilisateur.

On identifie dans ces cas d'utilisation une liste de fonctionnalités du produit. Pour chacune de ces fonctionnalités, on décrit l'ensemble des étapes qui permettent d'utiliser la fonctionnalité, pour un observateur situé en dehors du système. Il faut bien comprendre que le cas d'utilisation ne peut en rien préjuger de l'implémentation : en effet, l'implémentation est une phase postérieure, issue de l'analyse des cas d'utilisation !

De l'ensemble de ces cas d'utilisation, on déduit toutes les classes d'utilisateurs qui interagissent avec le système : ce sont les acteurs. Ces acteurs seront les rôles du produit Zope. En général, pour un produit Zope, on identifie au moins deux rôles : l'utilisateur (authentifié ou non) et l'administrateur du produit (souvent, le manager du site Zope).

Une fois que la liste de ces rôles est définie, on est à même de construire la politique de sécurité de Zope : telle fonctionnalité requiert tels droits, etc.

À l'issue de la rédaction des cas d'utilisation, on dispose donc de trois éléments fondamentaux :

- une description externe du produit attendu, et de ses fonctionnalités, qui fait office de référence ;
- une liste exhaustive des rôles qui interagissent avec le produit ;
- une ébauche de la politique de sécurité à mettre en œuvre dans le produit.

Diagramme de classe

De l'analyse des cas d'utilisation, on peut déduire un grand sous-ensemble des classes qui constituent le produit. Pour un cas simple, on a généralement une ou deux classes ; mais, pour un cas plus compliqué (un produit de gestion de planning, par exemple), on peut compter des dizaines de classes.

Une fois la liste des classes faite, on peut identifier leurs attributs et leurs méthodes.

On a pour habitude de mettre ces descriptions en place au travers d'un diagramme de classes, qui permet de voir d'un coup d'œil l'ensemble des classes, leurs attributs et méthodes, mais surtout les relations qui les lient (héritage, association, agrégation, composition…). Pour représenter ce diagramme de classes, on peut choisir la notation UML, qui est la plus répandue.

La modélisation est une phase stratégique du développement d'un composant. Lors de cette étape, il est difficile de savoir avec quel degré de granularité on va décrire le système que l'on développe :

- soit on décide de ne pas adopter une approche trop technique dans la rédaction du diagramme : on écrit alors un diagramme de classes assez fonctionnel qui identifie bien toutes les classes métier, mais qui n'exprime que leurs attributs et méthodes strictement fonctionnels. On retrouvera bien là les classes issues des cas d'utilisation ;
- soit on décide d'opter pour une rédaction beaucoup plus technique, dans laquelle on voit apparaître tous les détails techniques des objets décrits. On y retrouve non seulement toutes les méthodes et attributs fonctionnels, mais aussi les éléments très techniques, intimement liés au langage cible (ici, Python) et à l'environnement de développement (ici, le Framework de Zope). L'expression retenue sera très proche du code source, et ce sera une façon de le formaliser autrement. Parmi ces éléments très techniques, on note par exemple la définition de métatype (attribut de classe), la définition de la méthode `__call__` et l'apparition de variables comme `REQUEST`, propres à Zope. Dans ce type de schéma, un grand nombre de classes ne seront pas directement issues des cas d'utilisation, mais seront déduites.

Le choix entre ces deux stratégies n'est pas simple et c'est le projet concerné qui permettra de trancher. Si le projet de produit est très technique, destiné à résoudre un problème local, il conviendra généralement d'opter pour un diagramme assez technique. Inversement, si le projet se veut de grande envergure, qu'il doit être implémenté dans des langages différents (une partie en Java ou en C++ par exemple), ou que la solution du problème que l'on apporte doit être fortement réutilisable (composants métiers), il est plus sage d'axer sa réflexion sur une modélisation plus fonctionnelle.

Rien n'empêche de réaliser dans un premier temps un diagramme très fonctionnel, d'en garder une copie, puis de le faire évoluer vers un modèle technique. Procéder dans ce sens (fonctionnel puis technique) est en général plus avisé, bien que l'inverse soit aussi possible : nettoyer un modèle technique pour en faire un modèle fonctionnel doit permettre de se poser les questions fonctionnelles avant celles qui sont techniques.

Il n'est pas conseillé de choisir une solution intermédiaire, dont l'utilité ne se ferait sentir ni pour l'implémentation du composant ni pour la réutilisation des concepts élaborés.

À l'issue de la rédaction de ce diagramme, dont on fait l'hypothèse ici qu'il est technique, on a identifié l'ensemble des classes, leurs attributs, les méthodes et les relations structurantes qui les unissent.

Code squelette

Fort de ce diagramme de classe technique, on peut facilement écrire le code squelette du produit à réaliser. Cela revient à traduire le diagramme de classe en langage Python. C'est une étape qui est parfois fastidieuse, mais rarement difficile, et où il faut faire montre de rigueur.

On pourrait se faire assister dans cette étape d'un générateur de code source. Il est fortement recommandé d'en user sur de très gros projets, pour assurer la qualité du code.

À l'issue de cette étape, on dispose d'un code correct d'un point de vue syntaxique, mais qui ne fait rien (la plupart des méthodes ne sont pas implémentées et sont vides).

Code fonctionnel

Cette étape consiste à « remplir les blancs ».

On procède à l'implémentation des méthodes laissées vides dans le code squelette. Les cas d'utilisation peuvent servir ici de référence pour l'implémentation des méthodes.

Il est fortement recommandé de procéder à des tests du code à ce stade (par exemple, avec une suite logicielle de tests, telle que celle fournie avec Zope : `ZUnit`).

Code technique

On dispose maintenant d'un code fonctionnel, testé. Il faut le mettre en conformité avec les contraintes techniques de Zope.

Pour ce faire, le travail que l'on va exécuter est très proche de celui qui nous a permis de réaliser notre « produit minimal ». On va transformer un composant, dont on sait qu'il est conforme avec les éléments décrits dans les cas d'utilisation, en un produit Zope, qui va interagir avec l'utilisateur sur le Web.

Tests

C'est une étape fondamentale.

Les tests d'un composant s'effectuent en général en suivant un plan de tests. Ce plan énumère un ensemble d'interactions qu'il y a lieu de tester avec le produit, dans un ordre donné. À chaque étape, le plan décrit comment on vérifie que les réactions du composant sont bien celles qui sont attendues.

Un produit fonctionne si aucune des étapes du plan de tests n'échoue.

Pour concevoir un plan de tests, il est habituel de se baser sur les cas d'utilisation. Pour des projets simples, on peut même se contenter de ne suivre que les cas d'utilisation, un par un, et de vérifier que le composant réagit bien comme cela est décrit.

Packaging et distribution

Enfin, une fois le produit terminé et testé, il suffit de le mettre en boîte, comme nous l'avons expliqué dans une section précédente.

Sécurité des produits

Définition des rôles et permissions

Les produits Zope intègrent pleinement la gestion de la sécurité, avec toutes les finesses et tous les niveaux de granularité que permet le Framework.

Gérer la sécurité, c'est prévoir les scénarios d'utilisation du produit que l'on réalise. Ces scénarios sont issus d'une phase de réflexion qui permet d'identifier au moins les éléments suivants : les acteurs, les rôles et les permissions.

Les acteurs

La première phase de réflexion doit permettre d'identifier les acteurs d'un produit. En général, ils émanent de trois grandes familles : les utilisateurs-concepteurs, les utilisateurs-auteurs et les utilisateurs-visiteurs.

En principe, les utilisateurs-concepteurs sont les personnes qui conçoivent le site, qui instancient les produits au travers de l'interface d'administration de Zope. Ils sont souvent dotés du rôle Manager. Les utilisateurs-auteurs sont quant à eux responsables d'une partie du contenu du site et, en général, modifient les propriétés de produits qui sont instanciés par les concepteurs.

Enfin, les utilisateurs courants sont les simples visiteurs du site web (authentifiés ou non). Ils utilisent le produit sans pour autant avoir conscience de son existence. Pour eux, le produit n'existe pas, seule sa représentation HTML dans la page rendue a un sens.

Ce modèle est une proposition générale, qui dépend *toujours* du problème précis sur lequel on travaille. Il n'y a pas de réponse générale à cette phase de réflexion.

Les rôles

Une fois les acteurs identifiés, il est très simple d'en déduire les rôles. Ces derniers permettent d'identifier des catégories d'utilisateurs qui interagissent avec le produit Zope, et leurs permissions respectives. Si on reprend le schéma général incluant des utilisateurs-concep-

teurs, utilisateurs-auteurs et utilisateurs courants, on retrouve trois grands rôles : Manager, Author et User.

En général, on rencontre autant de rôles que l'on a précédemment identifié d'acteurs. Parfois, on peut avancer que deux acteurs différents partagent le même rôle, et compter moins de rôles que d'acteurs, mais rarement l'inverse.

Les permissions

Les permissions dépendent des opérations qu'un produit permet d'effectuer *et* de la granularité que l'on veut donner à la gestion de la sécurité pour son produit. Une permission est un droit que l'on laisse, ou qu'on ne laisse pas, à tel ou tel rôle, pour une action donnée, à savoir l'invocation d'une méthode donnée sur l'instance du produit. À titre d'exemple, créer une instance du produit, en afficher ou modifier les propriétés, affecter une valeur à une propriété sont autant d'actions qui peuvent être autorisées ou interdites pour un rôle donné. On le voit bien avec ces quelques propositions, la granularité dépend du besoin que l'on a de contrôler plus ou moins finement l'utilisation qui sera faite d'un produit.

Les notions que recouvrent les permissions sont elles aussi les mêmes que celles qui ont été développées dans le cadre du chapitre 3.

D'expérience, il apparaît que plus le nombre d'acteurs est grand – c'est-à-dire qu'*a priori* plus le nombre de rôles est grand – et plus le besoin en granularité s'accroît.

Sécurité au niveau de l'implémentation

Il ne faut pas perdre de vue que l'implémentation de la sécurité dans Zope est déclarative : c'est le produit qui déclare ses jeux de permissions.

Suivant le niveau de granularité souhaité, l'expression de la sécurité n'est pas la même dans l'implémentation d'un produit Zope. Par défaut, si le développeur ne fait rien, un produit est très peu permissif. Il ne permet même pas à l'utilisateur de lire ses propriétés ni d'invoquer ses méthodes.

À ne pas utiliser en production : `__allow_access_to_unprotected_subojects__`

La méthode la plus simple pour gérer la sécurité dans un produit Zope est aussi la plus dangereuse. Elle consiste à donner le droit de lecture à tout un chacun au travers d'une variable de classe : `__allow_access_to_unprotected_subojects__`.

Quand cette variable est positionnée à `1` dans un produit, tous les utilisateurs peuvent manipuler les variables d'instance et invoquer les méthodes.

D'une manière générale, ce n'est *pas* la bonne façon de procéder pour sécuriser un produit, excepté dans certains cas (nous y reviendrons). C'est à n'en pas douter une façon de reporter le problème à plus tard, lorsqu'on est en train de développer un produit.

Ex :

```
class ZhelloWorld(SimpleItem.Item)
"""Produit pour apprendre"""
...
    __allow_access_to_unprotected_subojects__=1
...
```

Définition des permissions : variable de classe `__ac_permissions__`

Pour définir le jeu de permissions des méthodes, il faut utiliser la variable de classe `__ac_permissions__`. Cette variable (un tuple de tuples) permet notamment de définir le nom des permissions pour toutes les méthodes et propriétés de la classe du produit que l'on développe.

La syntaxe de cette variable est la suivante :

```
(
  ('Nom permission',( 'methode1', 'methode2', 'method3', …), ('Role par défaut1',
  'Role par défaut1', … ),
  (…),
)
```

`Nom permission` est le nom de la permission.

`MethodeX` sont les noms des méthodes que l'on protège.

`Role par défaut X` sont les rôles affectés à cette permission par défaut, lors de l'instanciation du produit. Cette partie est optionnelle. En cas d'omission, aucun rôle ne possède implicitement cette permission.

À ce stade, il faut s'appuyer sur le travail qui a été fourni lors de la phase de réflexion et de conception.

En voici un exemple :

```
(
  ('View',( 'index_html', 'preview_html'), ('Anonymous', 'Manager')),
  ('Change properties', ('change_properties',)),
)
```

Remarque

La syntaxe de cette variable devrait être modifiée dans les versions ultérieures de Zope (probablement, au profit d'une table de dictionnaires). C'est la raison pour laquelle les produits Python et les méthodes externes sont dits « non restrictifs » : ils n'ont pas cette restriction de sécurité.

On notera que l'on ne peut pas gérer la permission qui autorise l'ajout d'instance du produit. C'est logique : ce ne sont pas les méthodes du produit qui gèrent l'instanciation !

> **Remarque**
> *Les règles de sécurité ne s'appliquent qu'aux interactions via le Web. Une méthode d'un produit peut toujours accéder aux variables d'instance de l'objet...*

Onglet d'administration

Modifier le comportement des onglets d'administration d'un produit Zope, ou leur ajouter du code, est assez simple.

Déclaration

Parmi les attributs de classe que le développeur fournit au produit, on distinguera l'attribut `manage_options`, qui est un tuple de dictionnaires.

Dans ce tuple, on déclare une entrée par onglet. Un onglet est représenté par un dictionnaire à deux clés : `label` et `action`. La clé `label` contient le texte du nom de l'onglet tel qu'il apparaît dans l'interface d'administration et la clé `action` est simplement la méthode qui sera invoquée quand l'utilisateur cliquera sur l'onglet.

```
manage_options = (
    {'label':'Calculer', 'action':'manage_compute'},
    )
```

Dans ce cas, le produit ne possède qu'un onglet : `Calculer`. Le choix de cet onglet par l'utilisateur invoquera la méthode `manage_compute`.

Héritage

Il faut noter que la déclaration d'une variable `manage_options` masque l'arbre d'héritage ; elle exige en effet que l'on rappelle la déclaration qu'avaient faite les ancêtres de cette classe.

Pour retrouver les onglets proposés par les ancêtres du produit, il faut les déclarer explicitement :

```
manage_options = (
    {'label':'Calculer', 'action':'manage_compute'},
    )+MaClassAncetre.manage_options
```

Dans ce cas, l'ancêtre est `MaClasseAncetre` et on additionne les deux tuples pour former le nouveau.

Implémentation

Une façon courante de procéder avec la méthode désignée par la clé `action` du dictionnaire consiste à déclencher une action silencieuse (par exemple, calculer la valeur d'une propriété, ou mettre à jour un élément du produit) et d'afficher un message de confirmation.

Par exemple :

```
def manage_compute(self, REQUEST):
    """Méthode de calcul de delta"""
    self.delta = ( self.b * self.b ) - ( 4 * self.a * self.c )
    return MessageDialog(
       title='Calcul',
       message="La valeur de Delta a été mise à jour en fonction de a, b et c",
       action = "./manage_main"
    )
```

`MessageDialog` est une fonction proposée par Zope pour écrire des boîtes de dialogue simples sur le modèle des pages d'administration.

Couplage d'un produit avec une ZClasse

Pour une meilleure réutilisabilité

Pour concevoir un produit réutilisable, on peut par exemple bien séparer le fonctionnel de la représentation (autrement dit, bien séparer la logique de l'affichage).

Comme on a pu le noter au chapitre précédent, Zope propose un concept bien pratique de développement rapide de composants, graphiques pour l'essentiel : les ZClasses.

Pour bien respecter la séparation logique/affichage qui garantit la réutilisabilité du composant, le mariage des produits et des ZClasses est idéal : on utilise les produits pour la logique, et les ZClasses pour la production des pages HTML.

Dans la pratique

Dans la pratique, le couplage se fait par un lien d'héritage. On conçoit, on développe et on teste le produit comme cela a été précisé dans ce chapitre, en se gardant de peaufiner l'interface graphique (la génération de HTML).

On crée ensuite une nouvelle ZClasse qui va hériter entre autres la classe du produit que l'on vient de créer. On va ensuite doter cette ZClasse de méthodes qui vont être chargées du « rendu » du produit, et qui pourront s'appuyer sans peine sur les propriétés et les méthodes exposées par le produit dont on hérite.

Inconvénient

Le principal inconvénient de cette pratique est qu'elle distribue le code à deux endroits différents : une partie dans la ZODB (les ZClasses) et une partie sur le système de fichiers (le produit).

En raison de cette ubiquité, il est donc plus délicat de procéder à des modifications dans le code.

En outre, les méthodes de déploiement et de distribution n'étant pas les mêmes, l'opération de packaging de ces produits est plus difficile.

ZODB et la persistance dans Zope

La base de données objet transactionnelle embarquée – ZODB – est un des composants techniques fondamentaux de Zope.

La persistance

Lorsqu'on pratique Python, les objets que l'on crée n'existent que dans le cadre de notre session de travail. Dès que l'on referme la console Python, les objets sont définitivement détruits de la mémoire. En voici la démonstration :

```
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> exit
```

Puis, dans une nouvelle session :

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: There is no variable named 'a'
>>> exit
```

Les objets persistants, quant à eux, ont un cycle de vie qui dure au-delà de la session (ou du programme). Si les objets Python standard étaient persistants, notre exemple donnerait ceci :

```
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> exit
```

Puis, dans une nouvelle session :

```
>>> a
[1, 2, 3]
>>> exit
```

Un objet est dit « persistant » si son état perdure au-delà de la durée d'exécution du programme du processus qui l'utilise.

> **Rappel**
> *On appelle état d'un objet l'ensemble des valeurs de ses attributs. Les méthodes n'interviennent pas dans l'état d'un objet.*

Avantages de la persistance

Cet état de persistance des objets apporte d'énormes avantages.

La transparence

Lorsqu'il travaille avec des objets persistants, le programmeur n'a pas à se soucier du cycle de vie de ses objets. Ceux-ci sont chargés en mémoire au fur et à mesure qu'ils sont accédés, d'une manière complètement transparente. Et, lorsque leur état est modifié, ces modifications sont répercutées dans la base de données d'une manière tout aussi transparente (on verra comment dans la suite de ce chapitre).

L'indépendance par rapport au stockage

Les états des objets sont stockés au travers des sessions dans une base de données. Ni le format de cette base, ni son type, ou quelque autre considération la concernant, n'influent (en théorie !) sur le travail du développeur.

On trouve des systèmes de persistance objet pour toutes sortes de supports : fichiers plats, base de type DBM, base relationnelle, etc.

Dans le cas de la ZODB telle qu'elle est fournie avec Zope, les objets sont stockés dans un seul fichier (`$ZOPE/var/Data.fs`).

Persistance automatique

Quel que soit le type d'attributs que porte l'objet, son état est stocké dans la base (dans le cas de la ZODB, il faut respecter quelques règles, comme nous allons le voir). Tous les objets du programme peuvent être rendus persistants, d'une façon automatique.

Le programmeur n'a plus à se soucier du stockage de ses objets ; il peut se concentrer sur ses problèmes fonctionnels.

ZODB

Au cœur des concepts novateurs de Zope, on trouve la base objet ZODB (pour Zope Object Data Base). La ZODB est indépendante du mode de stockage. Par défaut, Zope livre la ZODB configurée de façon que le stockage se fasse dans une base au format `dbm`, dans le fichier `$ZOPE/var/Data.fs`. Mais on peut faire fonctionner Zope en configurant une ZODB qui permette de stocker l'état des objets dans une base relationnelle Oracle, par exemple (que nous ne détaillerons pas ici).

Caractéristiques de la ZODB

La ZODB propose un certain nombre de caractéristiques très avancées.

La ZODB est transactionnelle

On retrouve ici le concept de transaction qui ne surprendra pas les lecteurs familiers des bases de données relationnelles.

La ZODB permet de travailler sur la base de données objet dans un contexte transactionnel.

> **Rappel**
>
> *Le concept de transaction implique l'atomicité d'un ensemble d'actions. Dans le cas d'un processus d'achat en trois étapes, par exemple, on est amené à enchaîner les opérations suivantes : 1) L'utilisateur valide son panier ; 2) L'utilisateur saisit son numéro de carte bancaire ; 3) Le serveur met à jour le fichier des stocks. En effet, si, pour une raison quelconque, l'étape 3 se passe mal, il faut pouvoir annuler l'étape 2...*

La ZODB est « multi-threadée »

Sous réserve d'utiliser correctement le Framework, la ZODB garantit que les objets qu'elle contient sont « thread-safe ». Nous ne pouvons dans le cadre de cet ouvrage détailler les mécanismes sous-jacents que cela implique, toutefois, dans ses opérations courantes, le développeur doit savoir que la ZODB lui apporte un minimum de garantie sur ce point.

Arbre d'héritage et classes de base de l'API de Zope

Pour créer un produit, Zope et son API (Framework) fournissent un grand nombre de classes de base, qu'il faut bien comprendre, en raison de la possibilité d'héritage multiple, dès lors que l'on veut programmer un produit Zope.

On a vu qu'il existait deux grandes familles de classes :

- `Item`, pour les éléments simples
- `ObjectManager`, pour les éléments qui contiennent d'autres éléments.

Acquisition.Implicit

Il s'agit de la classe de base des classes qui supportent l'acquisition. Elle est qualifiée d'implicite car la résolution des noms d'attributs et de méthodes au travers de l'acquisition est implicite : il n'est pas nécessaire d'écrire de code particulier pour l'activer (notation `self.toto`).

Globals.Persistent

Cette classe de base a pour fonction de faire persister les attributs de la classe du produit. Nous détaillerons ce mécanisme dans le paragraphe consacré à la ZODB et à la persistance.

OFS.SimpleItem.Item

C'est la classe de base standard des classes de produits de type `item`. Elle fournit un ensemble de services de base : interface d'administration, exposition par FTP et WebDAV, fonctionnalités d'annulation (Undo), de gestion de la propriété des objets. Sont également fournies les méthodes `manage_main`(), `title_or_id`(), `getId`(), `title_and_id`() et `this`() que l'on connaît. Cette classe impose de définir les attributs de classe suivants : `meta_type`, `id`, `title`, déjà étudiés dans la définition du produit minimal.

AccessControl.Role.RoleManager

Cette classe permet de gérer la sécurité du produit en s'appuyant sur le cadre standard fourni par Zope (voir plus haut la section « Sécurité des produits »).

OFS.ObjectManager

Cette classe de base a pour caractéristique principale de pouvoir contenir des instances de `OFS.SimpleItem.Item`. C'est la classe qui sert de base technique à `OFS.Folder.Folder`. En tant que conteneur, elle apporte les fonctionnalités d'import/export d'objets Zope et les méthodes `objectIds`(), `objectValues`() et `objectItems`() que l'on connaît. On notera également l'apport de l'attribut de classe `meta_types` qui permet d'affiner les types d'objets que l'instance d'`ObjectManager` peut recevoir. Nous retrouverons cette notion dans l'étude de cas sur le produit `ZExternalNews`, au chapitre 18.

OFS.PropertyManager

Cette classe apporte une possibilité d'utilisation simplifiée des propriétés. Elle exploite l'attribut de classe `_properties` :

On retrouve :

```
_properties = ( {'id' : 'title', 'type' : 'string', 'mode', 'w'},
                {'id' : 'delay', 'type' : 'int'},
              )
```

OFS.SimpleItem.SimpleItem

En y recourant, le développeur peut procéder à la simple préparation d'une classe qui agrège les classes auxquelles il recourt usuellement. Sa déclaration est parlante :

```
class SimpleItem(Item, Globals.Persistent,
                 Acquisition.Implicit,
                 AccessControl.Role.RoleManager,
                 ):
    manage_options=Item.manage_options+(
        {'label':'Security',   'action':'manage_access'},
        )
    __ac_permissions__=(('View', ()),)
```

OFS.Folder.Folder

On retrouve le même raisonnement qu'avec `SimpleItem` :

```
class Folder(
    ObjectManager.ObjectManager,
    PropertyManager.PropertyManager,
    AccessControl.Role.RoleManager,
    webdav.Collection.Collection,
    SimpleItem.Item,
    FindSupport.FindSupport,
    ):
    meta_type='Folder'
    _properties=({'id':'title', 'type': 'string'},)
    manage_options=(
        (ObjectManager.ObjectManager.manage_options[0],)+
        (
        {'label':'View', 'action':'index_html',
         'help':('OFSP','Folder_View.stx')},
        )+
        PropertyManager.PropertyManager.manage_options+
        AccessControl.Role.RoleManager.manage_options+
        SimpleItem.Item.manage_options+
        FindSupport.FindSupport.manage_options
        )
    __ac_permissions__=()
```

Remarque

Dans la plupart des cas, seules `OFS.SimpleItem.SimpleItem` *et* `OFS.Folder.Folder` *sont utilisées.*

En résumé…

Nous avons vu, dans ce chapitre, les notions et les mécanismes impliqués dans la création de produits, à travers l'exemple d'un produit minimaliste – mais parfaitement fonctionnel. Nous avons vu que la création de produits permettait de repousser les limites de l'automatisation de Zope.

Nous avons souligné qu'il existait deux types de produits : les produits simples et les produits de type ObjectManager, pour lesquels les objets peuvent eux-mêmes contenir d'autres objets. L'étude de cas présentée au chapitre 18 offre un exemple d'un tel produit – un outil de fédération de news – dont les objets contiennent eux-mêmes d'autres objets.

Nous en avons donc terminé avec l'apprentissage purement théorique de la programmation sous Zope. Il reste maintenant à approfondir ces notions, à en faire une synthèse que nous exploiterons dans les études de cas. Mais, avant cela, il convient de revenir sur l'intégration de Zope dans son environnement: nous avons vu, au chapitre premier, les bases de l'installation de Zope. Nous allons maintenant étudier plus en détail son interfaçage avec les différents systèmes d'exploitation, et avec les autres serveurs web tels qu'Apache.