Programming Plone - The MySite Tutorial

A file-system based product development tutorial for Plone.

Raphael Ritz

Objective

Provide an introduction to product development for Plone.

Audience

Plone site managers that feel the need to turn into file-system based product development but have no clue where to start. It is assumed, however, that you know what the Zope Management Interface (ZMI) is and how to access it as well as how to install products for Zope/Plone in general. Some basic Python knowledge is definitively a plus but the tutorial should be helpful even without that.

Topics Covered

The tutorial demonstrates how to provide our own product with:

- your custom portal tool (MyTool - a collection of custom methods)
- customized plone tool (MembershipTool - overwrites `addUser` to also notify admin per email on join; adds `getMemberName(id)` to get a member's fullname)
- custom Archetypes-based content type (Deadline - to be addable to Events)
- customized Archetypes-based plone content type (Event - can have Deadlines as sub-objects)
- custom workflow (CompoundWorkflow - to change states of events including their deadlines)
- custom skin including:
  - custom portlet to display the next upcoming deadlines
  - custom search template plus scripts to restrict the portal types offered on the form
- default content added to the site on product install (an `event` topic with subtopics)
- inclusion of other 3rd-party products (FlexibleTopic - unreleased from one of my sites and ATExtensions)
- Internationalisation (i18n) support

Dependencies

Written for use with Plone 2.

Needs Archetypes (now included with Plone)

Installation

DON'T USE THIS PRODUCT ON A PRODUCTION SITE UNLESS YOU ARE SURE THAT YOU KNOW WHAT YOU ARE DOING. THIS IS EDUCATIONAL SOFTWARE WRITTEN TO HELP YOU EXPLORE PLONE. IT WILL IRREVERSIBLY CHANGE SOME OF YOUR SITE'S SETTINGS.

To install the product, copy, move, or link the three subfolders `MySite` , `FlexibleTopic` , and `ATExtensions` to your Zope's product folder and restart the server. Then use the quick installer to add `MySite` to a test-and-throw-away Plone instance. The product takes care to install all other ones it depends on (like Archetypes) if necessary and it also puts an `events` topic in your site as you will notice.

You can get the accompayning product from http://www.neuroinf.de/LabTools/MySite. The location of your `Products` folder depends on your local installation. In general, it is considered best practice to use the `Products` subfolder within `INSTANCE_HOME` but there is also one in the `SOFTWARE_HOME` directory. If you don't know where these folders are on your server, use the ZMI and go to the `Control_Panel` in the Zope root. There, you will find these and other settings specific to your local server. When you've installed Plone on Windows using the installer, the `Products` folder is: `<Plone 2 folder>/Data/Products` .

Enjoy,

Raphael Ritz (r.ritz(at)biologie.hu-berlin.de)

Building Blocks

Plone and **Archetypes** are an application of the **Content Management Framework (CMF)** for the web-application server **Zope** which is implemented in **Python.** In order to program Plone you need to have at least a basic understanding of these parts, what they provide and how they interact. So, lets consider them in turn:

Python

Python is a powerful interpreted scripting language yet it is easy to learn. Quoting Tim Berners-Lee: *I was happy to find that Python is a language you can get into on one battery! I have been happily hacking ever since.* (from http://www.w3.org/2000/10/swap/). You should make yourself at least familiar with Python's elementary data types and how to handle them (look in particular for the **string, list, tuple,** and **dictionary** types and their methods). Furthermore, you want to understand how Python supports object-oriented programming and how you define and use functions, classes and methods.

Maybe the most important single feature that Python provides for Plone is platform independence as Python runs on any major computing platform available today. There is a little twist to this statement as some parts of Zope and Zope-based, third-party products are written in C or are using C libraries which are not necessarily available for all platforms but Plone would not be available for most platforms if Python would not support this.

Zope

Zope is a flexible, multi-purpose, web-application server written in Python. Zope does a lot of things. From a more general perspective, the most important ones for us are:

- It establishes a web-interface to the power of Python.
- It comes with a hierarchical object database, the ZODB, and acquisition which uses the object hierarchy to extend qualified name spaces.
- It implements a powerful security mechanism based on roles and permissions.
- It provides template languages: DTML, ZPT (including TAL, TALES, and METAL)
- It comes with a variety of built-in object types ready to use.
- It provides a through-the-web management interface: the ZMI which you may think of being a browser and management tool for the ZODB. You in fact need a web browser to access it (almost any browser will do).

But nothing is perfect and so there are also some features missing that you would ideally expect such a tool to provide (and that will be provided by the next generation of Zope, namely Zope 3), e.g., an event or notification service and unique, persistent object identifiers. Especially the lack of the latter is hard to believe and you may say each object can be specified uniquely by its path in the ZODB and therefore its URL. This is true but objects can be renamed or moved and this changes their identifier, so there is no notion of a **persistent** identifier (URI). This is why it is not easy to have referable objects and robust relationship management within plain Zope.

When programming Plone, it is important to learn when to use Plone or CMF specific API versus Zope API. Maybe the most important aspect you want to look into is Zope's security mechanism and how to use it for your own developments (including proxy roles, local roles and executable ownership). For reference, read the respective chapters in the Zope book http://zope.org/Documentation/Books/ZopeBook/2_6Edition/Security.stx and the developer's guide

http://zope.org/Documentation/Books/ZDG/current/Security.stx.

While Zope is a great framework to use in general, it is not geared toward any specific kind of web-application, meaning that for complex applications like interactive community portals or document management systems, you still have to do a lot of programming yourself. But since content management in general is such a typical application there is an add-on Zope product available that is tailored to the needs of developers who need to provide such systems: the CMF.

CMF

Zope's content management framework, the CMF, adds infrastructure to Zope in order to support the development of content-management centric web-applications like community portals or intra-nets.

Following a component architecture, the extensions are split into a variety of different modules and classes, each being responsible for an individual task. Furthermore, the CMF distinguishes between functional components (the **tools** or services) and components for holding data to be displayed (the **content types**). The generic parts are grouped within one product called **CMFCore** . The more specific parts illustrating how to modify, extend and use the core components are found in **CMFDefault. CMFCalendar** and **CMFTopic** further illustrate how to extend the CMF using third-party products (actually, CMFCalendar was the first third-party contribution to the CMF; CMFTopic is another later extension).

Since the CMF is at the core of almost everything in Plone, we will briefly consider some of its main concepts now. You don't need to understand every detail mentioned in the following paragraphs when reading this for the first time but you should get an idea of the *CMF way* of doing things. Later on, you might want to return to this section and follow the links provided:

- **Member handling** is managed by the **membership tool** (`portal_membership`), except for registration which is delegated to the **registration tool** (`portal_registration`) and member data configuration and storage which is the responsibility of the **member data tool** (`portal_memberdata`). It is a rewarding exercise to browse through their source code to see how these tools interact:

When the `join_form` is submitted, the `register` (skin) script calls `addMember` from the registration tool. This method does some checking before it calls `addMember` from the membership tool and finally it calls the tool's `afterAdd` method - one of the methods for us to hook into the registration process. Then it returns the newly created member object.

The `addMember` method from the membership tool, in its turn, first creates a Zope user object in the `acl_users` folder before calling `getMemberById` , which in turn calls `wrapUser` from the membership tool. `wrapUser` from the membership tool does some checks and role mappings before calling `wrapUser` from the memberdata tool. The memberdata tool is used for wrapping in order to enable member data storage independent of the user source (think of authenticating against an external LDAP folder). The individual data of the members are stored within the tool's `_members` BTree. Next, `wrapUser` from the membership tool calls `createMemberarea` (if the TTW configurable flag is set). Note the clear distinction between the (Zope) user and the portal member object. Finally, `addMember` sets the member properties.

Further note that portal members are not considered content by the CMF and that they are therefore neither subject to workflow nor are member data fields as easily configurable as content types are when using archetypes. There is, however, a third-party product `CMFMember` from the collective that replaces the default memberdata tool to make membership subject to workflow and to enable memberdata configuration including the preferences form using the archetypes machinery.

- **Content handling** is managed using a number of tools. First and foremost there is the **types tool** (`portal_types`) which serves as a registry as well as a factory for the portal types (as called by

`invokeFactory` from `PortalFolder`). In addition, the types tool allows you to define new portal types using `scriptable types` or copy/past/rename plus property editing from existing types.

Other tools that enhance the portal types are the **discussion tool** (`portal_discussion`) which enables the users to add comments to content objects that way becoming the starting point of threaded discussions and therefore to behave folderish even without being necessarily object managers (it provides them with an *opaque item* called `talkback` which is a `DiscussionItemContainer`).

The **metadata tool** (`portal_metadata`) implements your metadata policy. Per default this is almost the full **Dublin Core** metadata standard element set (http://dublincore.org/) with the prominent exception of *Relations* (guess why).

The **catalog tool** (`portal_catalog` - a preconfigured ZCatalog) handles the indexing of content objects (by sub-classing `CMFCatalogAware` `PortalContent` "learns" how to (re)index itself). The most important method that the catalog tool provides is `searchResults` but, of course, it supports the full ZCatalog API .

The **workflow tool** (`portal_workflow`) extends Zope's security mechanism for portal content to become state dependent. This means that in addition to Zope's general security handling you can assign states to objects (like *private, pending,* or *published*) and make selected permissions dependent on the current state. This is a very powerful concept that you should make yourself familiar with (e.g. by reading http://www.zope.org/Members/hathawsh/DCWorkflow_docs). From a developer's perspective you should know that the workflow tool provides `getInfoFor` to get at workflow variables like the `review_state` and `doActionFor` to invoke workflow transitions.

There are some more tools in the CMF and, of course, the content types themselves but you should become able to figure out for yourself how to use them as we proceed. We now move on and turn to Plone itself.

Plone

Plone is an application of the CMF; i.e. it realizes an out-of-the-box portal application using Zope and its CMF. As Plone cares about user experience (user interface design, usability, accessibility, internationalization (i18n) support, multiple browser support, standards compliance, etc.) much of what actually constitutes Plone lives in skin folders because this is where most of the user interface is defined. Some people therefore even say Plone is just a skin for the CMF but I consider this a bit unfair given that this is a central and important contribution, which includes a great deal of additional machinery to enhance the user's and administrator's experience (like the form controller, the quick installer, the migration tool, and the error reporting tool to mention some examples).

That said, it should not be too surprising to learn that when *programming* Plone you are more confronted with Python, Zope, and the CMF than with Plone proper. Since Plone is the target of the following tutorial anyway and because Plone's major focus - templating - is not covered here we close this section with a brief note on archetypes.

Archetypes

Archetypes is another application of Zope and the CMF that adds two main features: easy content type generation based on schemata, and referencability. Easy content type generation is supported by highly flexible views and editing forms, including form processing scripts so that you can concentrate on data modeling while developing your application without having to worry about the necessary skin methods.

Finally, archetypes-based content types are equipped with a unique, persistent identifier (the UID) which allows for a reference engine to establish and maintain robust relationships between those objects.

In theory, Archetypes should be independent of Plone, meaning that you can use it in any CMF based site but I am not sure whether this is indeed always the case.

The Big Picture

If you aren't a programmer already but you got to know Plone from the *browser perspective* so far only then you should first understand that **everything** that you have done through the web up to now to customize your site can also be scripted or integrated into custom file-system based products. If this comes as a surprise to you then you should realize how web applications in general and web interfaces in particular work. Whenever you follow a link or press a button an HTTP request to the site's server is issued. In the case of Zope this request is analyzed by the ZPublisher to figure out which method to call on which object and which parameters to pass to the call. As Zope is (almost) entirely written in Python, those objects are usually instances of Python classes which provide the method to be invoked based on the way in which Python/Zope/CMF/Plone looks up things (see below for more on how this is done).

So every action that you can trigger through the web can also be invoked from your Python program as soon as you know how to access Zope's objects and how to call methods on them. This is basically what programming Zope is about. In general, this is quite easy due to the fact that Python's support for object-oriented programming is very straight-forward and intuitive. The hardest part (from my experience at least) is to figure out how to get at any particular object to begin with and, most importantly, how to figure out what you can call on which object (its Application Programmer interface; the API).

To illustrate this a bit further, consider the following situation: When you write or edit a page template you often include a statement like:

```
<tal:define="my_value python:here.some_object.some_method(argument)">

Anonymous User - Mar. 13, 2005 6:42 am:
 Not a python programming but I get the impression 'here' is a type of 'this' pointer?
Anonymous User - Apr. 13, 2005 10:19 am:
 'here' is equivalent to 'context' and is specific to python in Zope
```

What else could you use instead of `here` at the beginning of such an expression? What can `some_object` or `some_method` possibly be? Whereas the first question is quite straight-forward to answer (all variables defined globally plus any local variable from the current context; i.e., defined on an element containing the current element), the second question is harder to deal with. The short - but almost useless - answer is: "This depends on what `here` is referring to." While this is true, it typically doesn't help you that much as long as you don't know how Python/Zope/CMF/Plone finds things (the following section is quoted from my Beginners Guide at http://www.neuroinf.de/Miscellaneous/BeginnersGuide).

Object/Name/Method Look up (or how Zope finds out about things)

Apparently this is one of the most confusing issues beginners stumble accross as it involves a variety of advanced concepts.

Multiple Inheritance

As Zope is written in Python, Zope objects are instances of Python classes. Python supports multiple inheritance and Zope makes heavy use of this. This means, when you define a class as in:

```
class MyClass(BaseClass1,BaseClass2, ...)
```

you can derive it from as many other classes as you like, where each in turn can be derived from an arbitrary number of other classes, and so on. This can make it hard to figure out which methods are available to a particular class because you need to scan all base classes (and **their** base classes, etc.) to find out. This is what DocFinder does for you. With DocFinderTab you can even make DocFinder readily available within your ZMI. Note that the order in which the base classes are specified is important! Lookup proceeds from left to right. So in case of a conflict (e.g. two different base

classes provide a method with the same name but implement it differently) Python just takes the first it finds.

Example Say you want to define your own folderish content type for the CMF. You do this by deriving from `PortalContent` and `PortalFolder` . But depending on whether you use:

```
class myFolderishType(PortalContent,PortalFolder)

or

class myFolderishType(PortalFolder,PortalContent)

Anonymous User - Nov. 7, 2004 5:28 pm:
 Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes
 looks as follows:
 class DerivedClassName(Base1, Base2, Base3):
     <statement-1>
     .
     .
     .
     <statement-N>
 The only rule necessary to explain the semantics is the resolution rule used for class attribute references.
 This is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClassName, it is searched in
 Base1, then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in
 Base2, and so on.
```

you will end up having different behaviour, e.g., with respect to the `portal_catalog` because `PortalFolder` overwrites `indexObject()` and `reindexObject()` to do nothing. So in the first case above your content type will be covered by the catalog whereas in the second it won't be.

Acquisition

In the ZODB an object can only be created in a container which in turn has to "live" in another container (only `root` doesn't need a container) and so on. This imposes a hierarchical relationship among the objects and it is this structure that Acquisition makes use of. If a method is called on an object that the object itself does not provide (neither by itself nor through inheritance) then Zope starts looking for the method in the object's container. If it cannot be found there either, it looks into the container's container and so on. To make sensible use of this you need to carefully consider where to place which objects since this makes a difference depending on the container's properties. This concept for look up is called **acquisition by containment** .

To make acquisition even more powerful there is a further twist to the concept called **acquisition by context** : When you invoke a method on an object you can do so by specifing the object's path from the root and then appending the method. For example, look at: `/topfolder/somefolderA/subfolder/object/mymethod` (note the URL format of method invocation here; i.e., you can do this directly via a web browser or in TALES using a path expression). Now suppose none of the folders here nor the object provides the method `mymethod` but that `somefolderA` has a sibling `somefolderB` in the `topfolder` which provides `mymethod` . You can now extend your call to `/topfolder/somefolderB/somefolderA/subfolder/object/mymethod` and the method will be found. Through extending the path you have changed the **context** in which the method is called and acquisition makes use of this for the lookup.

Note: This works not only for method but also for object look up. *Example* :

```
<tal:define="results python:here.portal_catalog(<search parameters>)">
```

works from anywhere within a Plone site (no matter what `here` currently refers to) because the portal catalog is always found by acquisition in the portal's root folder. Furthermore, it is directly callable with `searchResults` being the default method so `here.portal_catalog.searchResults(<search parameters>)` and `here.portal_catalog(<search parameters>)` are equivalent.

Skinning

This is a concept to extend name/method look up that the CMF adds to Zope. I like to think of this as having multiple different search paths to choose from. Suppose you have a number of folders (or **layers** as they are called in the context of skinning) `f1, f2, ..., fn` holding different versions of the same set of methods (but there is no need to have each method in each folder). Now you define a skin by saying: for skin A first look in `f1` , then look in `f2` , then in `f3` whereas skin B might be defined as: first look in `f2` , then in `f3` , then in `f1` and so on. That way you can provide, e.g., different designs to choose from (for example by having different style sheets in the different folders).

But the concept is more powerful than to just provide different designs. In principle you can use it to provide *multiple object behaviour* , meaning that you can make objects behave differently in response to particular method calls if the method is implemented to do different things in the different skin folders.

Under the hood, the skins tool works by extending the acquisition context of all objects within a CMF site. But you cannot tell this from looking at the URL in your browser, because this is done "silently" when traversing a so-called SkinnableObjectManager , a class that the PloneSite class itself inherits from.

Another nice feature provided through this concept is that you can easily customize methods without interfering with the original source code (this is a huge maintenance plus). The CMF does this by making the default skin folders available as `file system directory views` to the ZODB and therefore to Zope's through-the-web (TTW) management interface (ZMI). If you now navigate to one of those folders and you select a method you can just click `customize` and Zope will place a copy of the method in your `custom` skin folder which you can edit now as you like. Since the `custom` skin folder is typically placed before the default skin folders in the look up pathes of the skins, your method will be found first whenever called. Note that you did not change the original method in any way. It is still there and as soon as you delete (or rename) your customized copy, it will be in place again (very useful if you screw things up).

This closes the general introductory part and we are now ready to dive into the details.

Getting Ready for the `MySite` Product

In order to be able to appreciate the changes that `MySite` does to a regular Plone instance, make yourself familiar with Plone's event and topic content types at least. For better comparison, it is recommended that you create two new Plone instances in your Zope root, one for reference and the other one where you install `MySite`.

This tutorial does not try to teach you programming from scratch. Instead, it contains a number of pointers, explanations, and exercises that provide a hands-on start with file-system based development for Zope, CMF, that way illustrating the Plone development process. So here is your first exercise:

Exercise (Fundamental background - Object orientation): Explain the different concepts of an **object,** a **class,** an **instance,** a **constructor** , an **attribute,** and a **method.** What are **properties, accessors,** and **mutators?** What are Python's `setattr, getattr, hasattr` used for? (Solution)

If you have no clue how to answer the questions above, you would definitively benefit from reading some background literature on object-oriented programming. But even in this case, you should just continue reading.

Now, we jump right into the middle of `MySite`. Go into Zope's `Products` folder and look what's in the `MySite` subfolder.

What are all these Different Files for?

File-system based products are composed of a number of files. Not all of the files in the MySite product directory are mandatory for all Plone products, however, the files do follow common Plone product naming and organisational conventions. Technically, only the `__init__.py` module and the `install` function from `Extensions/Install.py` are required to create a Plone product, but MySite is attempting to demonstrate best-practices for developers.

The files included in the MySite package:

- `CompoundWorkflow.py` defines a custom workflow,
- `config.py` declares some custom variables in one place in order to have other parts as generic as possible,
- `Deadline.py` and `Event.py` define custom, Archetype-based content types,
- `Extensions` contains the `Install.py` module with the `install` function (the latter is checked for and executed by the quick installer),
- `__init__.py` executes at Zope startup to initialise our classes to make them available to Zope (but not yet to an individual Plone instance; that's what the `install` is for),
- `MembershipTool.py` and `MyTool.py` define custom portal tools,
- `my_utils.py` provides local utility functions,
- `refresh.txt` allows for product refresh via the Zope Management Interface or auto-refresh (broken with the Archetypes 1.2 series but fixed for 1.3)
- `skins` provides the skin folders,

- `tool.gif` holds the icon for the custom tools, and
- `version.txt` differentiates between product versions.

The Gory Details

In this section you find comments on each of the files listed above in order to explain what they are for and how they do what they are supposed to do. We start with the config file.

config.py

The configuration file provides a central location for defining custom settings and variables which are to be used throughout the product. Centralizing these definitions is done primarily to make maintaining the definitions easier. Since each module in the product simply references the centralized definition, a single change will take effect throughout the product without tedious and error-prone manual editing of source files. Among the commonly defined values in a configuration file are the permissions (normally strings) defined and used by the product. As these permission strings must precisely match to be effective, the centralized configuration file prevents subtle errors from emerging with slightly misspelled permission strings. It is a common Python convention to use ALL-CAPS names for constant values, such as permission strings. In the context of a Plone product, seeing an ALL-CAPS identifier probably indicates that the value has been defined in config.py . In addition to constant values, the configuration file may include references to external resources (such as Python modules) from changes in which the author of the product would like to insulate the product. Providing "wrapper" code within `config.py` allows the developer to access the functionality throughout the product without directly accessing the external resource.

Exercise (Python background - collection types): Some of the variables in `config.py` are assigned a list-type value. Lists are one example of a **collection** , or more specific, of a **sequence** . What is the difference between a **sequence** type and a **mapping** type? What is **slicing** ? What are the differences between a `list` and a `tuple` ? (Solution)

The first piece of code that gets executed from our product on Zope's start up is `__init__.py` , so let's consider this one next.

__init__.py

Like most Python modules (pieces of code to be combined to form larger applications; in fact, our Plone product here is actually a Python package named `Products.MySite`) `__init__.py` starts with `import` statements to make other modules or functions or classes available here even though they are defined in other modules. The imports from `CMFCore` and `Archetypes` provide functions to be called for registering and initializing our various custom components (the tools, content types, and skins) with Zope. To have access to our custom components, they also need to be imported as well as our custom settings from the `config` file.

The individual imports are:

from Products.CMFCore import utils — This makes the utilities module from CMFCore available. We will later use its `ContentInit` and `ToolInit` classes.

from Products.CMFCore.DirectoryView import registerDirectory — A DirectoryView is used to make the contents of a file-system based folder available as if the content were placed in the ZODB. We will call its `registerDirectory` method to add our custom skin folder.

`from Products.Archetypes.public import process_types, listTypes` — These are convenience functions defined in the ArchetypesTool and explained below.

It should be obvious now that `registerDirectory(SKINS_DIR, GLOBALS)` makes our product's `skins` directory available to Zope (in order to make it accessible to `install_subskin` in the `install` method of the `Extensions/Install` module to make the folders in `skins` available as filesystem directory views) and that the following calls from `initialize` to `ContentInit` and `ToolInit` from CMFCore's `utils` are there to register our content types and tools respectively (this is what makes them available to the drop down `Add` menu in ZMI).

Content type initialization for CMF is quite complex, so there are some convenience methods provided by Archetypes to make it easier for us to put together all the information that needs to be passed to `ContentInit` . For Archetypes-based content types you have to do it this way because the constructors and the factory type information (fti) objects are not provided by us when we define the classes -- they are generated by the Archetypes machinery when calling `process_types` . That's what the:

```
content_types, constructors, ftis = process_types(
   listTypes(PROJECTNAME),PROJECTNAME)
```

call does. Note, that this only registers the content classes with our Zope instance, but not with any of our Plone instances. The way in which this is done is addressed in the next section.

Exercise (Fundamental background - name space): Explain the concept of a `name space` . What is the difference between the statement `from config import SKINS_DIR, GLOBALS, PROJECTNAME` and `from config import *` ? Would the latter also work here? (Solution)

`Extensions/Install.py`

To make our custom components available within a particular Plone instance, they have to be installed there as well. This is not done on Zope's start-up because you need to have a way to control which product's components should be made available to each CMF/Plone instance, as you can have more than one Plone instance in any given Zope instance.

To this end `install` from `Extensions/Install.py` has to be executed (note that `Install.py` is actually the biggest single file in this product, reflecting the fact that quite a number of things happen here).

When the Plone quick installer is asked to install a product in a Plone site, the installer looks for a callable object named `install` in the `Extensions/Install.py` module and applies it to the site root, so `self` refers to our Plone site here. The purpose of the install function is to install into the Plone instance all of the structures required to support the product's operation. For very simple products, the amount of work required is quite minimal, normally consisting of a call to `installTypes` and another to `install_subskin` .

For a more complex product such as `MySite` , the amount of configuration required to integrate the product into Plone is such that it is useful to split the process into multiple functions each of which is called from the install function. Each of the functions in the `Install.py` module have *self-explanatory names* , so it should be easy to figure out their purpose. The install function here is defined at the end of the module, and calls each of these defined functions in turn.

Each of these broken-out functions demonstrates how to accomplish an operation which, until now, you may have only known as a through-the-web operation. These functions demonstrate how to accomplish these tasks using Zope/CMF/Plone's API (Application Programming Interface).

Unfortunately, when you just start to program for Plone one of the hardest parts is to learn how to figure out this API (i.e., how to do what) but there are a number of tools that can help you here (e.g., DocFinder, Zpydoc) and reading source code of other products and modules (and there in particular the installs and tests) is always a good source of inspiration.

Exercise (Zope background - PropertyManager): Make yourself familiar with Zope's property manager (e.g., by reading its section in the Zope book's appendix on the API and its source code in `OFS/PropertyManager`). Where does `setupProperties` from the `Install` module use public API calls and where private? (Solution)

Exercise (Python Background - Monkey Patching): Read http://www.zope.org/Members/Caseman/Dynamic_HotFix_News/Dynamic_Hotfix to make yourself familiar with the concept of **Monkey Patching.** Provide a patch for the `PortalContent` class in `Products.CMFCore.PortalContent.py` to include the `properties` tab in ZMI for any portal content. Use this now to enable configurable suppression of the `document_byline` displayed for individual documents (e.g., for the portal's frontpage to avoid the wrong impression that nothing on your site changed since the last edit of the site's `index_html` document). (Solution)

For most functions provided by `Install.py` it should be straight-forward to figure out how they do their job. The trickiest part is maybe the installation of the custom membership tool (in `setupTools`), so let's consider this one a representative example:

- `actions_buffer = None` declares a variable and initializes it to the Python object None
- `if hasattr(self, "portal_membership"):` the Python way to check whether the current site (`self`) has an attribute called `portal_membership` . If so, then
- `actions_buffer = self.portal_membership._actions` stores the current action settings in the variable declared previously
- `self.manage_delObjects(["portal_membership"])` removes the current tool (Plone's membership tool)
- `out.write(...)` adds a statement to the install log
- `addTool = self.manage_addProduct["MySite"].manage_addTool` makes the referenced function available via the name `addTool` in order to get the next statement on one not too long line. The `manage_addProduct["MySite"]` demonstrates how to access any specific product from within application code. `manage_addTool` was made available when we initialized the product's tools via the call to `utils.ToolInit(...)` in the `__init__.py` module.
- `addTool("Membership tool", None)` now installs our custom membership tool
- `if actions_buffer:` checks whether we had saved any actions before and if so
- `self.portal_membership._actions = actions_buffer` restores them (otherwise we have CMFDefault action settings)
- 'if hasattr(self, `my_tool`):' (and the following lines) checks to see if we have already created a `my_tool` instance, and if so deletes the current instance. We then create the new `my_tool` instance in the same way as for the `portal_membership` tool before.

Note : Deleting and recreating the tool wouldn't be strictly necessary here as long as we can be sure that no other product provided a tool with the same name. But if so, ours wins (but we lose the other tool). If it would be more important to keep potential customizations to the tool, it would be better not to recreate it (changing `Install.py` respectively is left as an easy exercise).

The special treatment of the actions for the membership tool is necessary because Plone (for ease of migration) doesn't define the actions in its membership tool itself but rather changes the action settings from CMFDefault's membership tool on Plone's install. Without the action handling here we would lose those changes and fall back to CMFDefault's action setting.

Exercise (Zope background - ZCatalog): Make yourself familiar with Zope's catalog tool, the `ZCatalog` (see, e.g., the chapter on searching in the Zope book and the section on the ZCatalog in the API appendix). Explain the following concepts: **catalog index** , **catalog metadata** , **catalog brain** . Change `addCatalogFields` to also add a keyword index. (Solution)

One remark on `setupInitialContent` : Note, how content generation is typically a two step process. First, you instantiate the new content object (by calling `invokeFactory`), then you get a handle on it (`et = getattr(root, "events")`), and only then call `edit` on it to set its field values (plus some further manipulations like adding criteria to the topic).

Exercise (CMF background - content generation): Where is `invokeFactory` defined? What happens in the process triggered by calling `invokeFactory` ? Work your way through the source code to figure that out. What is Plone's factory (`portal_factory`) for? (Solution)

Now that we've made it through the `config` , `__init__` and `install` we can turn to those parts that are the real cause of all of these efforts: our custom tools and content types. We start with the easiest one, our custom tool declaration in `MyTool.py` .

MyTool.py

The first statement after the imports is also the most important one here:

`class MyTool(UniqueObject, Folder, ActionProviderBase):` This defines our class `MyTool` to be a subclass of `UniqueObject` , `Folder` , and `ActionProviderBase` . This demonstrates one of Python's key features called **multiple inheritance** . It is used here to equip our custom tool with all of the attributes and methods that are provided by those classes (and their base classes respectively).

- `UniqueObject` -- provide services for objects which should not be overwritten or shadowed, i.e. you cannot have another object with the same id in the entire CMF site (api-doc).

- `Folder` -- uses the common Zope folder as the base data type, with all of the features that we expect from a Zope folder, such as containment and creation of sub-objects (api-doc).

- `ActionProviderBase` -- isn't actually necessary for the current example because we don't define any actions here but it is included so that `MyTool.py` can serve as a template for you to provide your own custom tool which you may want to add actions to (this could also be done in the `install` if needed - api-doc).

Beyond the attributes and methods that it inherits from its superclasses, the only thing that MyTool provides is the `transform` method, which uses the `urlify` and `at` methods. There is nothing fancy here, and in principle you could have added those methods as through-the-web (TTW) Python scripts if we weren't using Python's regular expression module (`re`) in `urlify` which you are not allowed to use in TTW code. Providing such scripts as *External Methods* would be another option but when you must provide more than a handful of external methods it is usually more convenient to collect them in a custom tool.

The last statement (`InitializeClass(MyTool)`) does just what it says: it initializes the class for use with Zope, most critically by initializing the class' security settings. The security settings demonstrated here simply declare the entire tool (i.e., access to the tool itself as opposed to its individual methods - so-called *Object Security Assertions* as

opposed to *Method Security Assertions* as demonstrated by the next tool) to be protected by the `View` permission. That's what the two statements `security = ClassSecurityInfo()` and `security.declareObjectProtected(View)` are for. Read the chapter on security in the Zope Developer Guide for a more indepth coverage of security assertions (http://zope.org/Documentation/Books/ZDG/current/Security.stx).

The `manage_options` define the tabs that you see when you browse to the object via ZMI. Here they are changed from the Zope Folder ones by excluding the `view` tab (`Folder.manage_options[1]`) and optionally replacing it with the `actions` tab (if you uncomment the `+ ActionProviderBase.manage_options` . The `_properties = Folder._properties + ...` adds the `at_mask` to the TTW editable properties of our tool so that its value can be changed via ZMI.

Exercise (Fundamental background - Inheritance): Explain the concept of **inheritance** . What does **multiple inheritance** mean? What are **conflicts** in this context? How does Python deal with these conflicts? (Solution)

Next, we will consider our custom membership tool.

MembershipTool.py

The main difference between this and the previously discussed tool (`MyTool`) is that instead of providing an additional tool to Plone, we replace one of the existing tools with a subclass of that tool. This allows us to customize the behavior of the tool even for those objects which are not aware of our customization.

To this end, we import Plone's membership tool but we rename it to `BaseTool` on import (`from Products.CMFPlone.MembershipTool import MembershipTool as BaseTool`) to be able to call our own tool-defining class `MembershipTool` without creating a naming conflict. The statement `class MembershipTool(BaseTool)` defines our tool as a subclass of the original Plone tool, thereby inheriting all its functionality.

The method definitions that follow specify how our custom membership tool will differ from the original one. There are two ways we can change our subclassed tool: (i) by providing additional methods (as demonstrated by `notifyAdmin` and `getMemberName` , two methods that Plone's membership tool does not provide) and (ii) by changing the way in which methods provided by the original tool work (this is what's meant when you hear or read something like "just subclass ABC and override XYZ").

When overriding existing methods you again have two options: (ii a) provide the full method definition yourself or (ii b) call the original method from the parent class and do some custom stuff before or after. Following the latter approach is sometimes referred to as **loose coupling** and `addMember` demonstrates this by calling `BaseTool.addMember(self, id, password, roles, domains, properties)` (note the slightly different phrasing of the call) before calling `notifyAdmin` . The advantage of this is that our `addMember` method will automatically track changes to Plone's membership tool's `addMember` method as Plone evolves even though we override it.

In order to make this custom tool work, you (or your users) must be sure to configure your MailHost object (in the portal's root folder) and the `email_from_address` property of your site object appropriately or `notifyAdmin` will not work (throwing an error (exception), or just do nothing).

Note to the purists: I know that you don't have to subclass the MembershipTool to add such a notification because there are explicit hooks in the registration process for exactly that reason but I wanted to demonstrate the concept of loose coupling with something that is simple but meaningful. Suggestions for better examples are always welcome ;-)

Exercise (CMF/Plone background - createMemberArea): Use the `notifyMemberAreaCreated` hook that is called by `createMemberarea` to maintain a log file on disk about the generation of memberareas (when and for whom). (Solution)

One final remark on the security declarations: In contrast to the previous example (`MyTool`) this tool uses *Method Security Assertions* meaning that every method provided has an individual security assertion. Again, see the Zope book and Zope developer's guide for more on one of the key concepts of Zope.

Note: There is no explicit docstring on our subclassed membership tool. This is not an omission but done on purpose because later in the module we call:

```
MembershipTool.__doc__ = BaseTool.__doc__
```

to set the docstring to the one from the parent.

This concludes the comments on the tools and we now turn to the content types.

Deadline.py

The Deadline object is a straightforward Archetypes-based content type. Archetypes-based data types are defined primarily by their **schema** , which is a description of the fields which are available on the archetyped objects. The classes required to define a schema are generally available from `Public.Archetypes.public` , and include:

Fields — individual *slots* which store particular pieces of data, fields have an implicit data type (such as a String (of characters), an Integer (whole number), or a reference to an object (often seen as References tabs in Plone)). Each field has a number of properties which modify its behavior.

Widgets — display *controls* which are used to edit the values stored in the field. Widgets generally assume a particular data type, but there are often multiple widgets available for editing any particular field type.

BaseSchema — the core Archetypes-content-object schema, including such common fields as id and title. This object is actually the schema attribute of the `BaseContent` class, which in turn inherits most of its schema from the `BaseObject` class.

Marshallers — objects which can reduce the content object to a particular format, and then reconstitute the object from that format. In this case, the `RFC822Marshaller` is being used, this class provides encoding in the common format of emails, HTTP requests and WebDAV.

Each Field class has its own collection of properties which affect its behavior, here we are using:

name — the name/title of the field object, (the required first argument to the Field constructor)

searchable — whether the field should be included in full-text indexing, generally speaking you'll want any meaningful text to be indexed, but internal data (that which has no human-significant meaning) should not likely be indexed using the searchable property. You will notice that title is marked searchable, while deadline_type is not. [XXX why is title marked searchable when it's auto-generated? Normally you do want it searchable, but here it seems not-very-useful].

default — specifies a default value for cases where the field is not explicitly specified during object construction.

accessor — a method name which will be used to retrieve the value of the field. In the `title` field, this is used to calculate the title from the combination of the parent's id and the deadline_type field. Note that specifying the accessor in this way means that the user cannot alter the title of the Deadline. Depending on your needs, defining a default function rather than an accessing may be more appropriate.

widget — a Widget instance to be used for editing the associated value.

vocabulary — a method-name or function returning a DisplayList or a DisplayList instance which describes the available/common choices for the field. In the deadline_type field the method DeadlineTypes is specified as the source of the vocabulary.

Note, how the `modify_fti` hook is used to change some of the default settings on the archetypes-generated factory type information. In particular `global_allow` is set to 0 to remove it from the default list of addable types.

Detour: `FactoryTypeInformation` (FTI)

FTI and its variant `ScriptableTypeInformation` are the kind of objects used to register the content types with the types tool. They contain all the information necessary to deal with the individual types. That way it becomes possible to always use the types tool to get relevant information by just passing in an object without knowing its type. E.g., "Give me this items `view` method!" When passed in a document the types tool will return `document_view` but when passed in a news item, it will return `news_item_view` . Or the portal folder's `invokeFactory` just calls the types tool for an instance of a specific portal type and the types tool "knows" how to generate it - and whether the type asked for is allowed to be placed in the calling folder (and by the calling user). Note that you can edit (most of) the properties of the FTI objects for each type individually TTW (go to `portal_types` via ZMI and select the type you want to adjust the settings for).

When providing your own content type in a file-system based product you need to specify the FTI yourself but when using Archetypes a reasonable default will be created for you when the types are processed (by calling `processTypes` in `__init__.py`).

You can interfere with this processing to provide custom settings as demonstrated here: `global_allow` controls the impicitly addable flag. This flag controls whether the type can be added anywhere or only in folder types where it is explicitly allowed to add this one. To place restrictions on the types available to a folder `filter_content_types` needs to be set for this folderish type together with a list of `allowed_content_types` as demonstrated by our custom event type in the following. `content_icon` specifies which icon to use for this type. (End of detour -- XXX would this be the right place to put a more detailed description of sction settings???)

One method worth a note is maybe `parentURL` . This method demonstrates how to access an objects container by using `aq_parent` . In order to exclude acquisition in context to ensure that we always get the parent with respect to containment an `aq_inner` needs to be placed between `self` and `aq_parent` . So, `self.aq_inner.aq_parent` always refers to the deadline's parent folder (i.e., the event it is a deadline of) irrespective of the current acquisition context and `self.aq_inner.aq_parent.absolute_url()` is the Zope way of getting at its URL.

Another method worth a note is `getStart_date` . Events have a field called `start_date` defined in their schema. Therefore, `getStart_date` is the name of the auto-generated accessor for this field's value. In addition, `getStart_date` is indexed by the catalog. As a consequence thereof, `getStart_date` is called on deadlines whenever they get (re)indexed. Since deadlines on the other hand don't have a field called `start_date` themselves, `getStart_date` is found via acquisition at the deadline's parent, the corresponding event, and returns the event's start date which is not what we want. To avoid this, we define `getStart_date` as an alias for the deadline's `start` method.

`getDisplayList` will be explained further down below in the `my_utils.py` section.

Exercise (Archetypes background - basic concepts): Read the archetypes developer's guide at http://plone.org/documentation/archetypes/ArchetypesDeveloperGuide. Explain the concepts of a **schema** , **field** , and **widget** as used by archetypes. How can you group fields such that the edit becomes a multi-step process (split accross several forms)? (Solution)

A note in passing on `from Products.Archetypes.public import *` : Usually it is considered bad style to import everything from a module (that's what this statement does, or more precisely, everything that is listed in the

module's `__all__`) because you can easily provoke naming conflicts and import loops especially if you do this for several modules at the same time but here I consider this safe. Alternatively, we could have declared `from Products.Archetypes.public import BaseContent, BaseSchema, Schema, StringField, StringWidget, SelectionWidget, DateTimeField` but you see immediately that life is much easier if we just import everthing from `Archetypes.public` ; this type of situation is what the import * construct is intended to support.

Now we turn to a more interesting content type: our custom event.

`Event.py`

The custom archetypes-based event type demonstrates how to replace one of Plone's default content types, here the event type, to provide a more detailed schema and to make the event *folderish* (by subclassing from `BaseFolder`) to allow the addition of deadlines as child objects. To restrict the addable types to deadlines only, we need to do two things:

1. Set `filter_content_types = 1` to activate filtering for the allowed content types, and

2. Set `allowed_content_types = ["Deadline",]` to define the list of allowed content types to filter on.

The actual replacement of the default content type by the custom content types takes place in the `install` where the call to `installTypes` overwrites the factory type information for `Event` . Note, that for this to be effective, it is essential to install `MySite` after any other product that itself overwrites the factory type information for `Event` (like `ATContentTypes`) in this way because only the last change will be effective. So in general, these kinds of changes are error-prone and should be avoided unless you are sure that you know what you are doing.

Alternatively, we could add our custom event type as an additional content type and leave the original event untouched (or change its `global_allow` to false if we don't want it to be offered to the users).

Exercise (CMF Background - Type Registration): Change `MySite` to work as outlined in the previous paragraph. (Solution)

Note that making `Event` folderish also makes events appear in the navigation tree like regular folders. Since we don't want that, `setupProperties` from the `Install` module takes care to add them to the list of `metaTypesNotToList` in addition to providing the `event_types` and `deadline_types` properties to make their values customizable through-the-web via Zope's management interface.

Notice further how the `Deadline` sub-objects are included in the `Event` object's default property views using a `ComputedField` and how it is making use of more complex, structured fields as provided by the ATExtensions package. Furthermore, the `location` method demonstrats how to access individual subfields of a `RecordField` which the `LocationField` is a subclass of.

Exercise (Archetypes background - references): Change `Event.py` to manage deadlines using a `ReferenceField` . (Solution)

One rationale for implementing deadlines as separate content objects from their associated event is to make it easy to compile lists of upcoming deadlines using simple catalog queries. You can see how simple this process becomes by looking at the page template `skins/my_templates/portlet_deadlines.pt` and the `event` topic registered in the `Install.py` module.

However, while cataloging and searching deadlines becomes much easier with this approach, we have introduced a usability problem for our Event's workflow. Generally speaking, a `Deadline` should have the same workflow state as its parent `Event` . One can imagine some (rare) instances where a different state might be useful (for instance where

a deadline for a publicly visible project is internal), but this is not the norm. In the normal case, we would like the Deadline's workflow state to track that of its parent except for the `pending` state. We do not want users to have to submit and/or review each Deadline separately for every Event.

The easy solution to this problem would be to eliminate the explicit management of a Deadline's state, instead providing the Deadline with a method to return their parent Event's review state as their own. Obviously, this would make it difficult to implement that (admittedly rare) case where a Deadline requires a different state from the parent Event.

Instead of using this partial solution, we have provided a custom workflow for the Event object. The purpose of the custom workflow is to automatically update the Deadline children with exactly the same workflow transition as is experienced by their Event parent (except for the `submit` transition). You will notice that this approach still allows for manually altering a Deadline's publishing state for exceptional circumstances, but eliminates the tedious process of manually updating that state for the common case.

CompoundWorkflow.py

The module `CompoundWorkflow.py` implements the custom workflow outlined in the previous section. The purpose of this workflow is to automatically alter the workflow state of `Deadline` objects when their `Event` object changes workflow state. To accomplish this, we duplicate the default Plone workflow then add a script to be executed for every state transition save the `submit` transition.

To create the `PythonScript` object, we pass the ID/name of the script to the constructor, then write the source code for the script to the resulting object. You will want to take particular note of the syntax for accessing the object being affected by the state transition (the Event, in this case).

Once we have created the `PythonScript` object, we need to make it available to the workflow object. The workflow object is *folderish* , which is to say, an object which can contain other objects (also known as an **ObjectManager**). Per default, a workflow has subfolders for `states` , `transitions` , and `scripts` . We store the `PythonScript` object using the identifier `plusChildren` within the workflow object's `scripts` subfolder using the `_setObject` method of `ObjectManagers` .

Having made the script available to the workflow, we can now set the script to be executed for each appropriate transition. To do this, we iterate over the sets of `transitions` , setting the `after_script_name` property to the identifier string `plusChildren` if the transition is not named `submit` . The `after_script_name` is looked up for every transition and if a non-empty value is found, a script with the corresponding id is looked up in the `scripts` folder and applied to the object (here the `plusChildren` script we have just created and registered).

Exercise (CMF background - workflow): Implement an alternative solution to this problem as outlined above (not having deadlines being subject to workflow themself). Hint: Make sure deadlines get reindexed on state changes of the parent event. To this end, figure out where `indexObject` and `reindexObject` are defined (e.g., using DocFinder or by searching through CMF's source code for `def index(` using a tool like `grep`). Now, override these methods in `Event` to also (re)index the deadlines. (Solution)

The approach demonstrated here is not fully generic because if you have a workflow where you can reach the `pending` state from states that do not provide the `publish` transition also, the subobjects can get stuck in there (think for a while why this is the case) but in combination with Plone's default workflow this can never happen.

Now we have covered the key elements of the `MySite` product. We have defined two content types, two portal tools, a product initialisation module, a product installation module, and a custom workflow. What remains are the supporting files and the `skins` folder.

my_utils.py

The `my_utils` module demonstrates the use of a simple Python module to encapsulate functionality for use throughout the product (and potentially for other products as well). You will notice that we have imported the `my_utils` module from both the `Event` and `Deadline` modules like so:

```
from my_utils import getDisplayList
```

This form of import relies on relative import semantics from Python. A more reliable form, and one which would work from another product, would be:

```
from Products.MySite.my_utils import getDisplayList
```

An even commonly-used idiom for this method of importing is to use:

```
from Products.MySite import my_utils
```

and then reference the function using `my_utils.getDisplayList` . This makes it clearer where methods defined by `my_utils` come from.

The `my_utils` module provides the `getDisplayList` function which is used by the content types to retrieve vocabularies from the portal properties setup by the Install script's `setupProperties` function. These properties can be edited by the site's administrator using the Zope Management Interface (ZMI) to customize the operation of the product.

Note the decomposition of the problem into three separate functions, one responsible for retrieving property values from the `portal_properties` tool, one for converting a list of values into a `DisplayList` , and one to provide the externally callable API for the module. This allows users of your module to reuse the functionality rather than needing to reimplement it in their own projects/customizations.

Exercise (Python Background - Signature): Make the `getDisplayList` function more useful by allowing for an optional argument to specify the property sheet from which to get the value. (Solution)

refresh.txt

During Zope product development (and Plone development is just a subset of Zope product development) it is often useful to have the product automatically update itself by being reimported. This process, called *refreshing* is enabled via the refresh tab in the ZMI. However, the refresh tab is only displayed if there is a `refresh.txt` file in the product directory (to prevent refreshing of products where this is not safe to do).

As far as Zope is concerned, the file can be empty (have nothing in it). However, certain archiving programs, particularly *WinZip* , tend to ignore files with 0 length. So, to support Windows users, it is good practice to never include such files in your products, instead add a few lines of text describing the purpose of the file.

For `refresh.txt` , leaving the file out has very little impact on the product. After all, it is simply enabling a feature intended for use by developers (not end-users/admins). Until a developer attempts to enable refreshing via the ZMI there is no difference in the operation of the product.

However, should you include a 0 length `__init__.py` , you may very well find that a sub-package of your product becomes unavailable.

Exercise (Optimization): Explain the diffuse statement made above about the problem that an empty `__init__.py` could cause. (Solution)

tool.gif

This file provides the icon for the tools that `MySite` provides. The icon is assigned to the tools in the `__init__.py` script during the call to `utils.ToolInit` . Icons are generally 16x16 pixels, in GIF or PNG format (with transparency).

Note that the icon must be available to Zope-level code to allow browsing via the ZMI, so it is not part of the skins used by the Plone-level code. [XXX Check this assertion, is there any way to register a skin-based icon for the tool? I think someone is working on making this possible.]

version.txt

Optional but recommended, even if you don't plan to distribute your product. Sooner or later you will have the need to destinguish between different versions of your product, so you better assign version numbers from the onset. Where to start and how to count is somewhat arbitrary if you are on your own but the bigger projects get the more formalized versioning tends to become (including dedicated rules for what to consider a major or minor release upgrade, when to call it `alpha` , `beta` , or a `release candidate (RC)` and what kind of changes are allowed (or more importantly: NOT allowed) in which phase).

Including a `version.txt` file in your product is optional, but generally recommended even if you are not intending to distribute your product. Sooner or later you'll needed to distinguish between multiple versions of your product, so assigning version numbers at the outset is generally a good idea.

Your choice of versioning scheme is largely a matter of choice. Larger projects will tend to have formalized versioning policies, including rules for what is considered a major or minor release, what is considered alpha, beta or release, and what kind of changes are allowed, or more importantly disallowed, in which phase of development.

The text you included in `version.txt` shows up as the `version` property of the product in the ZMI, and also under `plone setup -> Add/Remove products` -- it is therefore user-visible to some extent, and should try to be at least somewhat informative (don't make your user guess whether `zalphrod.venn` is an earlier version than `zalphrood.venn`).

Which leads us to the last element of the product, the directory holding the skin elements:

skins

This folder, which needs to be registered with Zope in MySite's `__init__.py` , holds three subfolders. These in turn get registered as `FilesystemDirectoryView` for the skins tool by the `install` function from the `Install` module calling `install_subskin` from `Archetypes.Extensions.utils` . They hold the skinnable elements which site administrators may customize.

Normally the five content items included here would not justify splitting into three separate subdirectories, but it does handily demonstrate the ability to have an arbitrarily nested skin folder structure. That demonstrated, try to keep your hierarchy flat, as nested subfolders (i.e. more than one level deep) can be difficult to work with when customizing skins through the web unless you are explicitly registering each further subfolder as its own `FilesystemDirectoryView` .

It is generally a good idea to group your content elements enough to keep your project manageable, but not so much as to make finding any given element more difficult. A good rule of thumb is to start by distinguishing between templates, images and scripts.

The example content provided by `MySite` should be immediately understandable. `my_images` contains the icon for the `Deadline` content type, `my_templates` provides two templates, one new (`portlet_deadlines.pt`) and one customized (`search_form.pt`), and `my_scripts` provides two little helper scripts that are needed by the custom search template.

`MySite` provides the following skinnable objects:

- `skins/my_images/deadline.png:` This is the default icon for the Deadline content type. It is specified as `deadline.png` in a class property of the `Deadline` class definition. Because it is a skinned element, it is loaded from the product context using the current skin resolution order.

- `skins/my_templates/search_form.pt:` Customized version of the default Plone search form. With `MySite` installed, you will find three different `search_form.pt` page templates in your Zope instance. One is in `CMFDefault/skins/zpt_generic` , another in `CMFPlone/skins/plone_forms` , and the last here in `MySite` .

- `skins/my_templates/portlet_deadlines.pt:` The portlet (*box*) displaying upcoming deadlines. Of particular note is the use of the portal_catalog.searchResults method to define the set of catalog brain objects to display. Also of note, defining a new portlet is technically speaking a process of defining a new `METAL` macro to be expanded in the main Plone page template, so the material surrounding the macro definition `DIV` doesn't appear in the final web pages.

- `skins/my_scripts/getSearchableTypes.py` and `getNotSearchableTypes.py:` Two `PythonScripts` used by the `search_form` page template. `getNotSearchableTypes` simply returns a static set of *non searchable* type names, i.e. those which should not be included on the search form. `getSearchableTypes` retrieves all types from the portal which are searchable (i.e. not *non searchable*).

Exercise (optimization): Figure out in which respect the custom search template differs from Plone's original one by (a) reading their respective source code and (b) invoking them through a browser. Change the way in which the non-searchable types are managed from a script to a portal property. (Solution)

The final remark is left for the last line of `getSearchableTypes.py` . Look at this list comprehension:

```
return [type for type in types if type not in types_to_exclude]
```

Pause for a moment to figure out what this does. If you see the beauty, then Python got you. ;-)

Exercise (List Comprehension): Rewrite this last statement in any other programming language of your choice and send it to me at `r.ritz@biologie.hu-berlin.de` . (Solution)

Customizing Archetypes

In the previous chapters you have seen how to **use** Archetypes including some of its extensions for generating custom content types. Here you will learn how to adapt Archetypes further to fit your needs. In order to be able to do so, however, you better have a basic understanding of Archetypes architecture first.

Archetype's Approach

Archetype's key feature is the support of **schema based** content type generation for CMF/Plone. That means that all you have to do (except for some minor add-ons if you don't use ArchGenXML) is to define a schema, assign it to a custom sub-class of `BaseContent` or `BaseFolder` , register, initialize, install and you are done.

Now what's a **schema** ? In a nutshell, a schema is just a list of fields, and that's exactly how you define it in your code. You import the `Schema` class and call it: `Schema(...)` . That way, you generate an instance. The constructor of the `Schema` needs to be passed at least one positional argument (the first) and that has to be a `tuple` containing the fields. This is why you have the somewhat funny looking two parentheses in your code.

Choosing the appropriate **fields** for your schema is usually the main part of your work when defining a custom content class. In more abstract terms, this is called **data modeling** , and it usually pays off to think first what you will need in the end because changing the data model later on generally doesn't make life easier. Archetypes provides a variety of generic fields that you can choose from and other products like ATExtensions provide some more specific ones of common interest. But if there is nothing that fits your needs, you might want to define your own field type as explained below. In your code, you instantiate fields the same way as your schema, i.e., you import the field defining classes you need and call them in the order that you want them to appear in your schema. Like the schema's constructor, fields need to be passed at least one positional argument defining the field's **id** . All other **properties** of the field (like `required` , `searchable` , `default` , etc) can be configured optionally. While most of these properties should be self-explaining there is one particular property that deserves more attention: the **widget** property.

Widgets are responsible for rendering the fields (or, more precisely, their value) in the different **views** of your content type like `base_view` or `base_edit` . Again, in the schema declaration you generate an instance of a widget by calling its class and assigning this instance to the field's `widget` property. You can optionally pass in named arguments to configure the widget if needed. Maybe the most important property to consider here is the `macro` property as this defines the page template where to look for the macros to use when rendering the widget. Like with fields, Archetypes provides a variety of generic widgets and further products like ATExtensions provide some more specific ones. But if there is nothing that fits your needs, you can define your own widgets or macros as outlined below.

In addition to the macros used by the individual widgets you can customize the macros used by different views like `base_view` , `table_view` or `base_edit` . When rendered, these views look for macros called `header` , `body` , `folderlisting` , and `footer` . To override the defaults that come with Archetypes for your specific types you need to provide a template with the id:

```
<portal_type>_view  or  <portal_type>_edit
```

Here you can provide your own macros now as needed. See below for an example.

Once `base_edit` gets submitted, it calls `validate_base` and if all values pass validation without error, `content_edit` is invoked to actually do the content update. Input **validation** is another key feature supported by Archetypes and there are a number of generic validators available for you to be used in your schema declaration. But maybe you have a need for a more specific validator. See below to learn how to define and add your own one.

That's it for the big picture. Now, let's turn to the details of *doing* the customizations. To this end, we need some content types making use of our customizations, so let's generate a little playground first.

Playground

Say we want to add an address book to our site. We want it to consist of a folderish type (`Address Book`) that is designed to hold and list contact informations (`Contact`). As of now, we don't want this to be a separate product (we will do that later) so we just add a module (`AddressBook.py`) to the `MySite` product folder containing the following code:

```
from Products.Archetypes.public import *

AddressBookSchema = BaseSchema

ContactSchema = BaseSchema + Schema((
    StringField('name'),
    StringField('email'),
    StringField('homepage'),
    ))

class AddressBook(BaseFolder):
    """specialized folder for managing contact information"""

    archetype_name = "Address Book"

    filter_content_types = 1
    allowed_content_types = ('Contact',)

    schema = AddressBookSchema

registerType(AddressBook)

class Contact(BaseContent):
    """entry in an address book"""

    global_allow = 0

    schema = ContactSchema

registerType(Contact)
```

Finally add the line `import AddressBook` to MySite's `__init__.py` , restart Zope and reinstall `MySite` . You will now have two new content types available in your site. Check them out! Add an address book with a few contacts and see how they look like.

Well, while this is a good starting point already, there are a number of things that could be improved. Email addresses and homepages could be links, for instance, `id` and `Title` could be used to hold the name or inferred from the it, the name could be structured into first name and last name in order to be able to sort on the last name in the address book's view which could also be more informative by listing the email address in addition, values for the email addresses and homepages could be validated on entry, maybe one or the other should be required, and so on. How to do all these kinds of changes will be the subject of the following sections.

Basic Customizations

Let's start with improving the contact's view. First, we want to switch from using `base_view` to using `table_view` . To achieve this, add the following action declaration to the `Contact` class:

```
actions = (
    {
    'id'          : 'view',
    'name'        : 'View',
    'action'      : 'string:${object_url}/table_view',
    'permissions' : (CMFCorePermissions.View,),
    'category'    : 'object',
    },
```

```
    )
```

and don't forget to import `CMFCorePermissions` before. Restart, reinstall, and see how the contact view has changed.

Note: Alternatively, you could do this change by editing the contact's type information registered with the types tool in ZMI. Go to `portal_types > Contact > Actions` and change the action property of the `view` action. Note, however, that you will have to repeat this change after every re-installation of `MySite` , so I suggest you do the changes in the class definition.

BaseSchema (part 1)

It seems natural to use the title field here to be used for the name, so let's do this now. Remove the `StringField("name")` declaration from the schema and customize `BaseSchema` instead as follows:

```
ContactBase = BaseSchema.copy()
ContactBase['title'].widget.label='Name'
ContactBase['title'].widget.description='The name of the person'

ContactSchema = ContactBase + Schema((
    StringField('email',
    ...
```

Don't forget to copy `BaseSchema` before customizing it, otherwise all content types using `BaseSchema` would be affected. Note further, that individual fields can be accessed by *subscription* (that means using `a[b]` like syntax) whereas field and widget properties can simply be qualified like attributes (`a.b` like syntax, also called *attribute-like* access) even though we declare them as values in the `_properties` dictionary. The fact that this works in one of the consequences of registering the properties as demonstrated below.

Macros

Next, we want to embed the email address and the homepage URL in anchor (`<a>`) tags. As it is not considered best practice these days to offer email links in anonymous views, we also want to restrict the email link to non-anonymous visitors of our site.

As outlined above, we need to provide page templates with the definitions of the view macros to be used by our email and homepage field respectively. So, add a page template called `email_widget.pt` to the `my_templates` skin folder (alternatively, you can do this TTW using the `custom` skin folder) containing the following code:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      i18n:domain="plone">
  <head><title></title></head>
  <body>
    <!-- Email Widget -->
    <metal:view_macro define-macro="view"
                      tal:define="email accessor">
    <div
     tal:condition="isAnon"
     tal:content="python:email.replace('@','(at)')">
       email
    </div>
    <a href="#"
     tal:condition="not:isAnon"
     tal:attributes="href string:mailto:$email"
     tal:content="email">
       email
    </a>
    </metal:view_macro>
```

```
  </body>
</html>
```

The important part here is the macro definition. Everything else is stereotypic and needed to turn the template into proper XHTML.

Now we need to inform our contact's email field that it should use this macro instead of the StringWidget's one by specifying the widget's `macro_view` property in the field declaration:

```
StringField('email',
            widget=StringWidget(macro_view="email_widget"),
            ),
```

Restart Zope (or refresh `MySite`) and see how the contact's view changes. Note, that you neither need to reinstall the product nor that you have to provide any migration script nor do a schema update for this change to become effective even for already existing contact instances. This is because the schema is a so-called class attribute where the value is a schema instance which is instantiated on Zope startup and inherited by all contact instances by virtue of being generated from this class (XXX is this the right way of phrasing this???)

Exercise (widget macros): Repeat the analogous steps to embed the homepage URL in an anchor tag.

Widgets

In the email widget's view macro above we mask the `@` sign to make it more difficult for robots to harvest the email address. Hard-coding the value replacing the `@` in the template is not the most brilliant way of doing this as it cannot easily be changed or configured in the schema declaration.

There are various ways now in which we could enable this, one being to define a custom widget where we add a property to the widget as follows: Generate a new module (called, e.g., `MyATE.py` for MyArcheTypesExtensions) and put this code in there:

```
from AccessControl import ClassSecurityInfo
from Products.Archetypes.public import StringWidget
from Products.Archetypes.Registry import registerWidget, registerPropertyType

class EmailWidget(StringWidget):
    _properties = StringWidget._properties.copy()
    _properties.update({
        'macro_view' : "email_widget",
        'at_mask' : '(at)',
        })
    security = ClassSecurityInfo()

registerWidget(EmailWidget,
               title='Email',
               description="Renders an Email address in an anchor tag.",
               used_for=('Products.Archetypes.Field.StringField',)
               )
registerPropertyType('at_mask', 'string', EmailWidget)
```

Note, how we define the email widget to be a subclass of the string widget and how we extend its properties by first copying them from the string widget before updating them. And while we are at it, we also put the view macro declaration here. Next, the widget gets a security declaration and finally we register the widget and its new property in order to make them available to the entire framework.

In the macro defining the view above we can now access the value of this property like any other widget property, i.e. `widget/at_mask` in path expressions or `widget.at_mask` in Python expressions. Now, change the contact's schema declaration: import the email widget and make the email field using it. Finally, edit the view macro in the `email_widget` template to use the new property and convince yourself that it works.

Exercise (widgets): Repeat the steps above to generate an URL widget where the value of the anchor's `target` attribute can be configured in the schema declaration.

Fields

Defining custom fields works pretty much the same way as generating custom widgets. You subclass an existing one, copy its properties and update and extend them as needed. In addition, you can add new methods to the field or override inherited ones. Here, we will illustrate this by defining a `UrlField` where we add a property called `default_protocol` in order to provide a configurable way to set a - well - default protocol. We then override the StringField's `set` method by fixing the value to be stored if no protocol (`http` , `https` , `ftp` ...) was entered on the contact's edit form. To do all this add the following code to MyATE.py:

```
class UrlField(StringField):
    _properties = StringField._properties.copy()
    _properties.update({
        'type' : 'url',
        'widget' : UrlWidget,
        'default_protocol' : 'http',
        })
    security = ClassSecurityInfo()

    security.declarePrivate('fixProtocol')
    def fixProtocol(self, value):
        if not value or '://' in value:
            return value
        return '://'.join([self.default_protocol, value])

    security.declarePrivate('set')
    def set(self, instance, value, **kwargs):
        value = self.fixProtocol(value)
        StringField.set(self, instance, value, **kwargs)
```

where the `UrlWidget` has to be defined before like in:

```
class UrlWidget(StringWidget):
    _properties = StringWidget._properties.copy()
    _properties.update({
        'macro_view' : "url_widget",
        'target' : '',
        })
    security = ClassSecurityInfo()
```

The last thing needed is to register the new field, widget and properties:

```
registerWidget(UrlWidget,
               title='Url',
               description="Renders a Url in an anchor tag.",
               used_for=('Products.MySite.MyATE.UrlField',)
               )

registerField(UrlField,
              title='Url',
              description='Used for storing urls.')

registerPropertyType('target', 'string', UrlWidget)
registerPropertyType('default_protocol', 'string', UrlField)
```

That's it. Now, our new field is ready to be used in the contact's schema declaration.

Exercise (fields): Define a custom email field to incorporate the widget settings made so far. Note, how the schema declaration for the contact type gets apparently simpler.

Views

From Archetypes-1.3.x onwards, `base_view` includes a folder listing for folderish types like our address book. If you look at the source code of `base_view` you see that there are altogether four macros (`header` , `body` , `folderlisting` and `footer`) that define the view. What we want to do now is to selectively override the `folderlisting` macro for the `AddressBook` type.

Furthermore, we want to be able to configure the fields of the contacts in the address book to be shown in the listing. So, we first add a `LinesField` called `listed` to the address book's schema that will only be visible in the address book's edit view:

```
AddressBookSchema = BaseSchema + Schema((
    LinesField('listed',
               default=('email','homepage'),
               vocabulary='contactFields',
               widget=PicklistWidget(description="Here, you can specify the fields from the contacts to be included in
                                     visible = {'view' : 'invisible'},
                                     ),
               ),
    ))
```

Note the `visible` declaration on the widget. To provide the vocabulary for this field, add:

```
def contactFields(self):
    """returns a list with allowed field ids from the contact's
    schema for inclusion in the listing"""
    return [f.getName() for f in ContactSchema.filterFields(schemata='default')\
            if f.getName() not in ['id','title','name']]
```

to the AddressBook's class definition. See how we obtain the field names from the `ContactSchema` restricted to the `default` schemata set. Otherwise all meta data fields would be included here as well.

Next, we create a page template (e.g. in the `custom` skin layer) with the id `addressbook_view` making use of this field:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      lang="en"
      metal:use-macro="here/main_template/macros/master"
      i18n:domain="plone">

<body>
    <div metal:fill-slot="main">
        <metal:listingmacro define-macro="folderlisting">
        <tal:foldercontents define="folderContents python:here.contentValues()">
        <tal:listing condition="folderContents">
            <table class="listing">
            <thead><tr>
            <th>name</th>
            <div tal:repeat="field_name here/getListed">
                 <th tal:content="field_name"> name </th>
            </div>
            </tr></thead>
            <metal:block tal:repeat="obj folderContents">
                  <tr tal:define="oddrow repeat/obj/odd"
                      tal:attributes="class python:test(oddrow, 'even', 'odd')">
                <td metal:define-macro="listitem"
                    tal:define="url obj/absolute_url;
                                icon python: obj.getIcon(1)">
                <a href="#" tal:attributes="href url">
                        <img src="" alt=""
                             tal:attributes="src string:${utool}/${icon};alt here/title_or_id" />
                </a>
                <a href="#" tal:attributes="href url"
                            tal:content="obj/title_or_id" />

                </td>
                <td tal:repeat="field_name here/getListed">
                    <metal:data use-macro="python:obj.widget(field_name)" />
                </td>
```

```
                </tr>
                </metal:block>
            </table>
        </tal:listing>
        <p class="discreet"
           tal:condition="not: folderContents"
           i18n:translate="description_no_contacts_in_address_book">
         There are currently no entries in this address book.
        </p>
        </tal:foldercontents>
        </metal:listingmacro>
    </div>
</body>
</html>
```

This is only a slightly modified version of Plone's default folder listing macro (but with a different id to match Archetypes naming) where we make use of the configured fields to be included in the listing and where most of the type specific declarations have been removed because we don't need them here. Note specifically, how we use the widget specific macros to render the field values in this listing as well (the line that reads `<metal:data use-macro="python:obj.widget(field_name)" />`).

Validators

Another key feature of Archetypes is its capability to validate the user input. For every field in the schema you can configure whether a value is required or whether its value has to be in accordance with certain rules. To this end there are various different validators provided by Archetypes but it is also easy to define custom ones.

Since valid `email` and `url` values do require a certain structure it makes of course sense to check for it. Add the `validators` declaration to the `EmailField` if you did the exercise above already or define the field now in 'MyATE':

```
class EmailField(StringField):
    _properties = StringField._properties.copy()
    _properties.update({
        'type' : 'email',
        'validators' : ('isEmail',),
        'widget' : EmailWidget,
        })
    security = ClassSecurityInfo()
```

Add an analogous declaration to the `UrlField` (`validators` :(`isURL` ,)). Note that `validators` (plural!) is actually expecting a sequence meaning you can add an arbitrary number of different validators to be called one after the other to do their respective check (the concept of a `validationChain` as this is called has been introduced after AT-1.3.0-b2).

Did you notice that we have a problem now? Our `default_protocol` feature of the `UrlField` is not working anymore because the URL validator complains before the value gets changed. As the validators are alway called before `content_edit` which in turn calls the fields' mutators there is no other way out then skipping validation here or, better yet, providing our own validator. To do so, add the following code to 'MyATE':

```
import re
from Products.validation.interfaces import ivalidator

class PartialUrlValidator:
    __implements__ = (ivalidator,)
    def __init__(self, name):
        self.name = name
    def __call__(self, value, *args, **kwargs):
        field = kwargs.get('field')
        if hasattr(field, 'fixProtocol'):
            value = field.fixProtocol(value)
        pattern = re.compile(r'(ht|f)tps?://[^\s\r\n]+')
        m = pattern.match(value)
        if not m:
```

```
            return ("Validation failed(%s): %s is not a url."%(self.name,
                repr(value)))
        return 1
```

Note, how we access the field here that gets passed to the validator via the keyword arguments (`**kwargs`) as does the REQUEST and the contact instance. The `hasattr` check is done to catch cases where this validator is applied to fields that don't provide the `fixProtocol` method and all the rest is basically copied from the original url validator (admittedly not the most robust one).

In order to make our new validator available to field definitions or schema declarations we still have to register it with the validation machinery. This is usually done in your product's `__init__.py` but as we don't provide our own product here (yet), add the following lines to MySite's '__init__.py':

```
# the custom validator
from Products.validation import validation
from MyATE import PartialUrlValidator
validation.register(PartialUrlValidator('isPartialUrl'))
```

Now, we can use our new `isPartialUrl` validator after restarting Zope (don't forget to change the `UrlField` declaration to also use it). Note that this is the second time that we change code from a product that we should better leave alone in order to be able to update it independently of our new additions. Whenever changes like these have to be made it is worthwhile considering whether we would not be better of if the new stuff were organized as a product of its own. This is what we will do now.

Refactoring

Rewriting or reorganizing (parts of) a product's code in order to make the software more generic and robust is generally referred to as **refactoring.** While there are various things to be considered when refactoring a Plone/Archetypes based product, I will only illustrate two particular situations here: splitting products and changing a schema. The main issue to worry about when doing those changes is to be backwards compatible in order not to loose any data.

Splitting Products

If products develop over time they tend to grow bigger and bigger in terms of their code base and functionality provided. It then often deems appropriate to split a product into several ones but there is at least one issue to be aware of when doing this with any Python/Zope based software module as we will see now.

Say we want to factor out our address book. That means we move all address book related code out of the `MySite` product into a new product that we will call `MyAddressBook` . The basic stuff is easy: create a new folder called `MyAddressBook` in your Zope's product folder (the same where the `MySite` product folder is located). Copy the `AddressBook.py` and `MyATE.py` modules over and create a `skins` folder containing an `addressbook` sub-folder where you place the `addressbook_view` , `email_widget` , and `url_widget` templates. Revert the changes that you did to MySite's `__init__.py` (remove the registration of our custom validator and the import of the `AddressBook` module) and add an initialization module to `MyAddressBook` instead. This `__init__.py` should contain the following code:

```
from Products.CMFCore import utils
from Products.CMFCore.DirectoryView import registerDirectory
from Products.Archetypes.public import process_types, listTypes
from config import *

# the custom validator
from Products.validation import validation
from MyATE import PartialUrlValidator
validation.register(PartialUrlValidator('isPartialUrl'))

# the module containing the content types
import AddressBook
```

```
registerDirectory(SKINS_DIR, GLOBALS)

def initialize(context):
    content_types, constructors, ftis = process_types(
        listTypes(PROJECTNAME),
        PROJECTNAME)
    utils.ContentInit(
        PROJECTNAME + ' Content',
        content_types      = content_types,
        permission         = ADD_CONTENT_PERMISSION,
        extra_constructors = constructors,
        fti                = ftis,
        ).initialize(context)
```

provided that we also add an appropriate `config.py` file:

```
from Products.CMFCore.CMFCorePermissions import AddPortalContent

ADD_CONTENT_PERMISSION = AddPortalContent
PROJECTNAME = "MyAddressBook"
SKINS_DIR = 'skins'
GLOBALS = globals()
```

Finally, we have to add the `Extensions` folder containing our `Install.py` with the following code:

```
from StringIO import StringIO
from Products.Archetypes.public import listTypes
from Products.Archetypes.Extensions.utils import installTypes, install_subskin
from Products.MyAddressBook.config import PROJECTNAME, GLOBALS

def install(self):
    out = StringIO()
    installTypes(self, out,
                 listTypes(PROJECTNAME),
                 PROJECTNAME)
    print >> out, "Installed %s's content types." % PROJECTNAME
    install_subskin(self, out, GLOBALS)
    print >> out, "Installed %s's skin." % PROJECTNAME
    print >> out, "Successfully installed %s." % PROJECTNAME
    return out.getvalue()
```

You may want to add further files like `version.txt` , `refresh.txt` or a `README` but from a functional perspective you are done now. Note that these steps are quite stereotypic, so you can basically cut and paste those files from other products and adjust them as needed.

Restart Zope and check out your new product. Do this first in a *new* Plone site and convince yourself that it can be installed using the quick-installer.

Return to the site that you have been using previously and to be on the safe side, go to its types tool in ZMI and remove the factory type informations for the `AddressBook` and `Contact` types. Now install `MyAddressBook` here as well. Convince yourself, that you can add address books to the site including contacts. Fine, so everything works as expected? Well, not quiet yet. As you may have noticed by now, all address books that you had generated previously seem to be gone! That's usually not what you want. But done worry too much. The data are not gone. It is just that Python cannot unpickle them (or with other words: Zope/Plone cannot get them out of the ZODB) because their *class path* changed.

It appears to Python that the classes from which the old address books and contacts had been generated have disappeared and it therefore does not know how to re-instantiate them. In order to get your old address book and contact instances back, you have several options: either re-introduce them to `MySite` or add a so-called **module alias.** For the first option, go into the the `MySite` product folder and create a file called `AddressBook.py` (more generally, you use the name of the module that used to contain the class definitions for the content types before the refactoring) containing one single line of code:

```
from Products.MyAddressBook.AddressBook import AddressBook, Contact
```

Even better is the creation of a **module alias** because you can do this without even touching `MySite` at all. To this end put the following lines into the `__init__.py` of `MyAddressBook` after the `import AddressBook` statement:

```
import sys
sys.modules['Products.MySite.AddressBook'] = AddressBook
```

This has the effect that our class definitions will be available again using their old path even though their actual position moved to a different location. Restart Zope and confirm that your old address books are back again including their content.

Note: Of course we could and should have done this immediately when factoring out the address book from `MySite` . It was left out on purpose to demonstrate the problem that results from omitting this.

Schema Evolution

Purists in the field say schemata don't evolve but they are fixed to begin with. While this helps of course a lot, practice tells that this is simply unrealistic. Fortunately for us, changing a schema isn't that hard with Archetypes.

Adding a field

Adding a field to a schema declaration is really straight-forward. Just add the appropriate field declaration to the schema in question, restart Zope (or refresh the product) and you are done. Confirm this by adding two fields to the contact schema to hold a street address and phone numbers. There are specific field types tailored to this end available from ATExtensions, so all you need to do is to add the following lines to 'AddressBook.py':

```
from Products.ATExtensions.at_extensions import AddressField, PhoneNumbersField
ContactSchema = ...
    ...
    UrlField('homepage'),
    AddressField('address'),
    PhoneNumbersField('phone'),
    ))
...
```

Deleting a field

Deleting a field is as easy as adding one: just remove it from the schema declaration, restart and you are done. There is only one thing to be aware of. Removing a field from the schema does not remove its value from the objects affected. For fields holding short strings or numbers this is usually not a problem - the old instances are just a little bit *heavier* than necessary - but if you are dealing with large text or image values this might be a problem. So if you want to remove the field's value from its storage you usually write a little script (a so-called *visitor*) that iterates over all objects in question and removes the value explicitly (XXX: check whether `updateSchema` from the archetypes tool actually does this). Note, that we actually did remove a field already when we were moving the name handling to the title field above and therefore removed the name field from the schema. So to remove the name values as well we could apply the following script (to be declared as an external method on portal root):

```
def removeNameValuesFromContact(self):
    """removes the 'name' attribute from contacts"""
    contacts = self.portal_catalog(portal_type='Contact')
    for contact in contacts:
        obj = contact.getObject()
        if hasattr(obj.aq_base, 'name'):
            delattr(obj.aq_base, 'name')
```

Note, how we use the catalog to get at all `Contact` instances and how we make sure not to delete any acquired attributes by restricting the operation to the objects **acquisition base** (i.e., the object itself). If the field would not use

`AttributeStorage` (which is the default storage in Archetypes) we would of course have to change the way in which we access and clear the value according to the storage used.

Renaming a field

Renaming a field consists of a combination of the two previous steps. For the schema declaration it is really just a renaming of the field's id but to transfer the value also we need to apply a script similar to the one above where we not only delete the old attribute but also copy the value to an attribute with the new id like in:

```
def restoreValues(self,type,old_id,new_id):
    """visits all instances of 'type' and moves the value
       from the 'old_id' attribute to a 'new_id' attribute
    """
    brains = self.portal_catalog(portal_type=type)
    for brain in brains:
        obj = brain.getObject().aq_base
        if hasattr(obj, old_id):
            setattr(obj, new_id, getattr(obj, old_id))
            delattr(obj, old_id)
```

As you can see, this method is a little bit more generic than the previous one and maybe some time in the future the archetypes tool will contain methods like this one for our convenience (XXX: check whether this isn't already there). Updating the catalog once you are done with those changes is usually also a good idea.

Changing the Type of a Field

When changing the type of a field there are two situations to be distinguished:

1. The value's type stays the same: This is the easy case, like when turning a `StringField` into an `EmailField` as we did above, because here, the value itself is just a string in both cases. So we only need to change the field declaration in the schema definition, restart Zope and we are done.

2. The value's type changes as well: This is the harder case, as we now have to take care ourselves to cast the value from the old type to the new one. This is really application specific and there is usually no way out for you other than writing a special purpose script to do the migration. See the next subsection for how to do something similar to this.

Customizing BaseSchema (part 2)

Wouldn't it be nice if our contacts *knew* about first or given names versus last or family names? To enable this, we can add a `SimpleNameField` from `ATExtensions` which is a subclass of the `RecordField` with two subfields for the `first_name` and the `last_name` , or, from the Python perspective, a field that holds a dictionary with these two keys.

If we now introduce this new field, we are confronted with the case 2 above if we want to transfer the names from the existing contacts to the new field. As of now, the name is put as plain string into the title field, so we need to find a way to infer first and last names from there. It is this kind of change (moving from a less structured to a more structured data type) that is usually hard (if not impossible) to perform in general. Let's assume here, that we had always entered first names first and that last names are usually one word. Then we can split the title's value on whitespace and take the last element as last name all the others being first names.

So add:

```
from Products.ATExtensions.at_extensions import SimpleNameField, RecordWidget
...
ContactSchema = ContactBase + Schema((
    SimpleNameField('name',
                    mutator ='setName',
                    widget = RecordWidget(visible={'view':'invisible'})
```

```
                    ),
    ...
```

to `AddressBook.py` and provide a migration script that looks like:

```
def migrateNames(self):
    """visit all contacts and populate the new name field
       from the title value
    """
    brains = self.portal_catalog(portal_type='Contact')
    for brain in brains:
        obj = brain.getObject().aq_base
        name = obj.Title()
        if name:
            parts = name.split()
            if len(parts) == 1:
                first_name = ''
                last_name = parts[0]
            else:
                first_name = ' '.join(parts[:-1])
                last_name = parts[-1]
            obj.name = {'first_name':first_name,
                        'last_name':last_name}
```

Look over all contacts after the migration and see whether you have to fix some entries where the heuristics used here didn't apply.

The last thing we want to do is to hide the `id` and `title` fields from the edit form and to infer and set their values from the name field instead. To do so, change the customization of `BaseSchema` to look like:

```
ContactBase = BaseSchema.copy()
ContactBase['id'].widget.visible = {'view':'invisible',
                                    'edit':'invisible'}
ContactBase['title'].required = 0
ContactBase['title'].widget.visible = {'view':'invisible',
                                       'edit':'invisible'}
```

Now, the only thing left is to provide the custom mutator for the name field in order to also set the values for the `title` and `id` fields. This can be done by extending the `Contact` class definition to include the custom mutator:

```
def setName(self, value, **kwargs):
    """custom mutator to change the id and title as well"""
    # set the id from the name
    id = self.getId()
    if self.isIDAutoGenerated(id) or id.startswith('copy_of'):
        if value:
            id = self._generateId(value)
            self.setId(id)
    # now infer and set the title
    name_string = ' '.join([value.get('first_name', ' '),
                            value.get('last_name', ' ')]).strip()
    self.setTitle(name_string)
    # finally call the original mutator
    SimpleNameField.set(self.schema['name'], self, value, **kwargs)

# helper methods for the id generation
def _generateId(self, value):
    # use last name if possible
    candidate_id = self._normalize(value.get('last_name', 'name'))
    first_name = self._normalize(value.get('first_name', 'my'))
    while candidate_id in self.aq_inner.aq_parent.objectIds():
        candidate_id = self._nextId(candidate_id, first_name)
    return candidate_id

def _normalize(self, text):
    """lowers and removes all non-ascii characters"""
    text = text.lower()
    return ''.join([c for c in text if c in string.letters])
def _nextId(self, current, extension):
```

```
    if not '_' in current:
        return current + '_' + extension[0]
    else:
        base, ext = current.split('_',1)
        if len(ext) < len(extension):
            return current + extension[len(ext)]
    if current.endswith(extension):
        return current + '_1'
    else:
        try:
            number = int(current.split('_')[-1]) + 1
        except ValueError:
            number = 123   # XXX fix me
        return '_'.join(current.split('_')[:-1].append(str(number)))
```

Note, how we extend the mutator to also call `setId` and `setTitle` after inferring appropriate values from the `name` dictionary (which is a bit involved for the id).

To finally make use of our structured name, change the `folderlisting` macro to sort the entries on last name and first name. Change the definition of `folderContents` using the `sort` function from the `sequence` module:

```
<tal:foldercontents define="items here/contentValues;
                            sort_on python:(('getLastName','nocase','asc'),
                                            ('getFirstName','nocase','asc'));
                            folderContents python:sequence.sort(items,sort_on)">
```

The methods `getLastName` and `getFirstName` still need to be added to the `Contact` class:

```
def getFirstName(self):
    """accessor for the name's 'first_name' subfield"""
    return self.getName().get('first_name','')

def getLastName(self):
    """accessor for the name's 'last_name' subfield"""
    return self.getName().get('last_name','')
```

and that's it. Apply the changes outlined above, restart Zope and see what a nice address book we have gotten.

Conclusions

Pause for while to realize what we have achieved now. Starting from a straight-forward Archetypes based product we have developed a substantially customized product with only very little reconfiguring and additional code. This is the big plus of using the Archetypes framework. Admittedly it is not always easy to figure out where and how to best interfere with the framework as this needs quite some understanding of the internals but I hope I was able to demonstrate this at least to the extend that you can get started doing your own customizations.

Probably the most advanced customization demonstrated here concerns the name handling. Note that on the edit form we ask for first and last name separately and the values entered will be hold in the `name` field where the value is a dictionary with these keys. From there, we construct a plain text version of the name and put it in the title field. Furthermore, we also infer the contact's id by using the (normalized) last name if possible and variants thereof if not. First and foremost this should demonstrate how you can make non-trivial customizations within the Archetypes framework without specifying everything a new. Note in particular that we are still using Archetype's generic views like `base_view` , `base_edit` and `table_view` and that we only provide one custom mutator for the `name` field. All custom templating was done by selectively overriding specific macros. Everything else we use as provided or generated by the framework.

Internationalisation (i18n)

(contributed by Jean Jordan, Upfront Systems, http://www.upfrontsystems.co.za).

Definitions

Before we address this interesting topic we need to define our terms.

The two central concepts are **internationalisation** and **localisation** (`i18n` and `l10n` for short .. because 18 and 10 are how many letters were left out to make the abbreviation). Perhaps surprisingly, the concepts are somewhat independent: a site can be internationalised without being localised, and vice versa, although internationalisation supports localisation. A helpful dictionary definition is: Internationalisation abstracts out local details, localisation specifies those details for a particular locale. To sum up:

Internationalisation — Creating an infrastructure and guidelines that supports translation of the user interface and content.

Localisation — Providing translations and locale specific content, making use of the i18n infrastructure. Note that localised content need not necessarily imply translations.

Can you think of occasions where it doesn't? innerlink 1

Plone today is thoroughly internationalised, and products exist that help site authors to localise their sites.

Plone and Internationalisation

Internationalisation in Zope has a long history and many products, which I won't summarize here. Plone itself has used a number of products and approaches (such as `Localizer` and `TranslationService`), depending on what the state of the art was at the time ... you can find traces of this on the web. Currently, the standard approach is to use `PlacelessTranslationService` , `I18NLayer` , and `I18NFolder` .

Future plans for Plone include the unification of `I18NLayer` and `I18NFolder` , and `LinguaPlone` , which will provide a less intrusive method of associating content with translations than the current containment-based approach, and will probably replace the `I18N*` products.

For the purposes of this tutorial, we'll look at three topics, namely:

- Translating Plone
- MySite and translation
- Translating Content

Plone is Internationalised

A lot of work has already gone into internationalising Plone. To experience this, go to the preferences page of your browser and change the selected language to another of Plone's supported languages.

(To be added: picture of Language preferences in Mozilla)

(To be added: picture of Language preferences in Internet Explorer)

Check `plone.org` for the list of supported languages.

When you now visit your local Plone (or indeed most public Plone sites), you should see most of the user interface elements (all the text supplied by the system) in your language of choice. In our case, the exception would be any Events and their Deadlines that we added in the course of the MySite tutorial. For them, we would still see texts such as "Event Type" and "Upcoming Deadlines". Third party products are responsible for integrating with Plone i18n infrastructure, and providing their own translations.

If you want to translate Plone to a new language, you need to work with the Plone i18n Team. This team is responsible for assigning message IDs, quality assurance and translation conventions. Read about them on the web.

Visitors to your Plone site may or may not have their browser's Language preferences configured, depending on their computer supplier or IT staff, and they may not know what is determining their language choice. We'll see later how the browser negotiates which language to show.

Using the I18N infrastructure

Zope i18n uses gettext and PlacelessTranslationService.

gettext

Underlying all the Zope i18n work is the Gnu project's gettext system.

PlacelessTranslationService builds on gettext. Gettext supplies conventions for programs to support message catalogs and tools for parsing the catalogs. The gettext message catalogs in your Plone installation may be found in `Products/<ProductName>/i18n` directories on the filesystem.

PlacelessTranslationService (PTS)

PlacelessTranslationService forms part of the Plone 2.0 distribution. If it's present in your Zope instance's `Products` directory, it shows up in the ZMI `Control_Panel` at `/Control_Panel/TranslationService/manage_main` after a restart. If you go there, you'll see the list of all the message catalogs found during Zope startup, and whether they were successfully parsed. PTS looks for message catalogs in directories called `i18n` . It searches for these directories in your Zope's INSTANCE_HOME, as well as in each product directory in $INSTANCE_HOME/Products . If you watch the Zope logfile when restarting, you'll see a lot of messages similar to this one:

```
------
2004-06-18T20:10:21 INFO(0) PlacelessTranslationService
Initialized: ['portaltransport-cs.po', 'portaltransport-ru.po']
from /<path to zope>/Products/PortalTransport/i18n
```

Translating Page Templates

The content of this section owes a lot (sometimes verbatim) to a good overview called i18n For Developers.

We'll begin with a theoretical digression on the translation of Page Templates, and then start translating MySite.

Let's take a simple page template (the one from the overview linked above):

```
<html>
<body>
  <p>Welcome to Plone.</p>
```

```
  <img src="plone.gif" alt="Plone Icon" />
  <p>There have been over
    <span tal:content="here/download_count">100,000</span>
    downloads of Plone.</p>
  <p>Please visit <a href="about">About Plone</a>
    for more information.</p>
</body>
</html>
```

That template needs *four different kinds* of translation. They are:

1. The text "Welcome to Plone.".

2. The `alt` attribute "Plone Icon".

3. The phrase "There have been over *100,000* downloads of Plone." The number *100,000* should be calculated dynamically, and in another language it may appear in a different position in the sentence.

4. The phrase "Please visit *About Plone* for more information.". In this case, *About Plone* should remain a link, it should be translatable separately, and the position of the link name in the sentence may vary in other languages.

Apart from the translation work, we need to alert the PlacelessTranslationService that it needs to process this template. We do this by specifying a domain on the HTML element. For our product, we'll use a product-specific domain:

```
<html i18n:domain="mysite">
```

The value `mysite` specifies which message catalogs should be consulted during translation. If we were working on Plone templates, the value would be `plone` .

Case One: Welcome to Plone

We need to associate the text to translate with a *message id* that will be used to look up translations. If you leave the message id blank, the text to be translated will itself be used as the id (here: "Welcome to Plone.").

If this text is a part of Plone proper (in the `plone` domain), for which the i18n team takes responsibility, you need to give them a chance to assign a message id to each string to translate. Do this by using `XXX` for the id. Now, when the translation team use their tools to extract all the msgids from the page templates, it's obvious which ones need fixing. If this is a product of your own, you'll need to choose an id yourself. There are some guidelines for the naming of ids (from the Guidelines for translators innerlink 4):

- 'heading_': for `<h>` elements
- 'description_': Explanatory text directly below
- 'legend_': Used in `<legend>` elements
- 'label_': For field labels, input labels, i.e. `<label>` , and for `<a>` elements
- 'help_': Any text that provides help for form input.
- 'box_': Content inside portlets.
- 'listingheader_': For headers in tables (normally of class "listing").
- 'date_': For date/time-related stuff. E.g. "Yesterday", "Last week".

- 'text_': Messages that do not fit any other category, normally inside
- 'batch_': for batch-related things - like "Displaying X to Y of Z total documents"

Exercise: When will you not choose an id yourself? innerlink 2

Here, we're translating a simple phrase. To do this, we add an `i18n:translate` attribute to the `P` tag containing the translation. Let's pretend we're translating Plone, and use "XXX" as value for 'i18n:translate':

```
<p i18n:translate="XXX">Welcome to Plone.</p>
```

Ultimately, the translation team will choose a message identifer (msgid) to uniquely identify this phrase, replacing "XXX" with their id, and that will become the new value of the i18n:translate attribute.

Case Two: "Plone Icon"

To translate attributes, we use `i18n:attributes` instead of `i18n:translate` . The value of `i18n:attributes` is a list of all the attributes that require translation. This list may include the message ids; if it doesn't, the value of the attribute (here: "Plone Icon") will be used as the id.

For our example, the simplest possibility is:

```
<img src="plone.gif" alt="Plone Icon" i18n:attributes="alt" />
```

If we wanted to translate multiple attributes, we'd have:

```
<img src="plone.gif" alt="Plone Icon" title="Plone Icon Title"
    i18n:attributes="alt title">
```

If we also wanted to specify message ids, we'd have something like:

```
<img src="plone.gif" alt="Plone Icon" title="Plone Icon Title"
    i18n:attributes="alt mysite_plone_icon;
                     title mysite_plone_icon_title">
```

Note that the simple list of attributes is **space** -seperated, while the attribute-and-msgid list is **semicolon** -seperated.

Case Three: "There have been over 100,000 downloads of Plone."

As before, we use the `i18n:translate` attribute on the outer containing element (here, `P`) to mark the phrase for translation. **In addition** , we add `i18n:name` attributes on dynamic child elements that may change position. This gives us:

```
<p i18n:translate="XXX">There have been over
  <span tal:content="here/download_count"
    i18n:name="count">100,000</span>
  downloads of Plone. </p>
```

When this is processed, the resulting message for translation will look like this:

```
msgid "XXX"
msgstr "There have been ${count} downloads of Plone."
```

While msgids need to be unique within their i18n:domain (that's one reason for using naming conventions), the name only needs to be unique within the context of the enclosing tag.

Case Four: "Please visit About Plone for more information."

Here we are again translating a phrase that contains a subelement that may appear in different positions in other languanges. Furthermore, the subelement should **itself** be translated. To manage this, we use `i18n:translate` together with `i18n:name` . Let's pretend that this is part of our product, and assign msgids:

```
<p i18n:translate="mysite_more_plone_info">Please visit
  <span i18n:name="about-plone">
    <a href="about"
       i18n:translate="mysite_about_plone">About Plone</a>
  </span> for more information.
</p>
```

This results in two messages in the catalog, that may be translated individually:

```
msgid "mysite_more_plone_info"
msgstr "Please visit ${about-plone} for more information."

msgid "mysite_about_plone"
msgstr "About Plone"
```

The final remaining possibility is that such a sub-element may also include an attribute to be translated. In this case, handle it as follows:

```
<p i18n:translate="mysite_more_plone_info">Please visit
  <span i18n:name="about-plone">
    <a href="about" i18n:translate="mysite_about_plone"
       i18n:attributes="title" title="Go to About Page">
    About Plone</a>
  </span> for more information.
</p>
```

This results in three messages in the catalog: the above two, and additionally:

```
msgid "Go to About Page"
msgstr "Go to About Page"
```

Exercise: supply a msgid for the attribute. innerlink 3

Translating MySite

OK, enough theory. Let's translate MySite and see what problems we encounter on the way!

Preparing the templates

Take a look at the MySite tempates in `skins/my_templates` . You'll notice that the preparatory work has already been done: the `HTML` elements have `i18n:domain` attributes, and there are `i18n:translate` attributes elsewhere in the templates. (You'll notice that MySite has very little content to translate.)

portlet_deadlines

The default domain for translations in this template is specified on the `HTML` element:

```
<html xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      i18n:domain="mysite">
```

Since standard Plone doesn't mention deadlines anywhere, the first string for translation belongs in this domain:

```
<h5 i18n:translate="box_deadlines">Upcoming Deadlines</h5>
```

The next (and last!) string in this template is one that is common to most of the boxes in Plone. It already has a translation in the `plone` domain, so let's reference it there:

```
<a href=""
    class="portletMore"
    tal:attributes="href string:${portal_url}/events/deadlines"
    i18n:domain="plone"
    i18n:translate="box_morelink"> More...
</a>
```

And that's it!

search_form

As mentioned in the solution to an earlier exercise ("Figure out in which respect the custom search template differs from Plone's original one"), the only change to MySite's `search_form` is where the list of types to search comes from. None of the text has changed. Therefore, the `i18n:domain` can be left at Plone:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      lang="en"
      metal:use-macro="here/main_template/macros/master"
      i18n:domain="plone">
```

(Note to self: what's the role of the `lang` attributes here?)

Preparing the translation directory

PlacelessTranslationService needs an `i18n` directory, so let's create it:

```
MySite $ mkdir i18n
```

Now we need to put a message catalog in there. To get the strings for translation from our templates and into the catalog, we'll use `i18ndude`, a tool which is part of the plone-i18n project at SourceForge -- download it from there. (It requires Python 2.3, so check that you're up to date.) The tool includes help, but there's an i18ndude HowTo if you're looking for a shortcut. The only slightly tricky things are choosing the right domain for processing, and to tidy the resultant catalog. In our case, this little dance works:

```
MySite $ TEMPLATES=`find . -iregex '.*\..?pt'`
MySite $ i18ndude rebuild-pot --pot i18n/mysite-tmp.pot \
         --create mysite $TEMPLATES 2> i18n/report.txt
MySite $ i18ndude filter i18n/mysite-tmp.pot \
         ~/.../CMFPlone/i18n/plone.pot > i18n/mysite.pot
```

The first two lines are the same as the HowTo, but the last one is new. This is because the HowTo doesn't deal with templates that contain more than one `i18n:domain`, and `i18ndude` isn't clever enough to parse one domain at a time. That is, it generates a 'pot'-file putting **all** the messages in the `mysite` domain. We filter out the ones that are in the `plone` domain by comparing the generated 'pot'-file with the Plone master file (that already contains these msgids). This leaves us with `i18n/mysite.pot` containing a grand total of one (1) message (hey, it's the idea that counts!).

Translating code in Python files

In a well-designed Zope Product, the user interface should ideally be restricted to page templates in a skin layer. At the moment, however, there are some corners where text lingers in Python classes. One of these places is the labels and descriptions of widgets. One way of quickly finding widgets to translate is:

```
MySite $ grep -r  "Widget(" `find -name "*.py"`
```

In our case this yields Event.py and Deadline.py. Here are the widgets in 'Event.py':

```
StringField('event_type', vocabulary='EventTypes',
  widget=SelectionWidget(
      label='Event Type',
      description='The type of the event'),),
...
UrlField("event_url",
  widget=UrlWidget(
      label="Homepage",
      description="Webpage of the event"),),
```

And here they are after internationalisation:

```
StringField('event_type', vocabulary='EventTypes',
  widget=SelectionWidget(
      label='Event Type',
      label_msgid='label_event_type',
      description='The type of the event',
      description_msgid='help_event_type',
      i18n_domain='mysite'),),
...
UrlField("event_url",
  widget=UrlWidget(
      label="Homepage",
      label_msgid="label_homepage",
      description="Webpage of the event",
      description_msgid="help_homepage",
      i18n_domain='mysite'),),
```

What changed? And what motivated the domain choice? innerlink 5

Here are the widgets in 'Deadline.py':

```
StringField('title',
  searchable=1,
  default='',
  accessor='Title',
  widget=StringWidget(visible={'edit':'hidden'}),
  ),
StringField("deadline_type",
  vocabulary='DeadlineTypes',
  widget=SelectionWidget()
  ),
```

Take a close look at those. Are those all the fields you see when adding a deadline? Nooo .. we're still lacking the Date and the Comment fields. In the schema definition they're present as:

```
DateTimeField("date"),
StringField("comment"),
```

Since no widget is specified, they get the default widgets for their fields. These are `CalendarWidget` and `StringWidget` (you need to climb a Jacob's ladder of imports and inheritance to find this out).

In order to get all these fields translated, add the i18n parameters:

```
StringField('title',
  searchable=1,
  default='',
  accessor='Title',
  widget=StringWidget(
      visible={'edit':'hidden'},
      label='Title',
      label_msgid='label_title',
      i18n_domain='plone',
      ),
  ),
StringField("deadline_type",
```

```
  vocabulary='DeadlineTypes',
  widget=SelectionWidget(
      label='Deadline Type',
      label_msgid='label_deadline_type',
      description='Choose the type of deadline',
      description_msgid='help_deadline_type',
      i18n_domain='mysite',
      )
  ),
DateTimeField("date",
  widget=CalendarWidget(
      label="Date",
      label_msgid="label_date",
      i18n_domain='plone',
      )),
StringField("comment",
  widget=StringWidget(
      label="Comment",
      label_msgid="label_deadline_comment",
      description="Any further detail regarding the deadline",
      description_msgid="help_deadline_comment",
      i18n_domain='mysite',
      )),
```

Here's a quick question: why provide for the translation of the Deadline's Title, if it isn't visible? innerlink 6 Another question: why translate the Comment label? Surely Plone has already translated it? innerlink 7

Note: this is fine, as far as it goes, but it does not take translation of vocabularies (e.g. `vocabulary="DeadlineTypes",`) into account.

Updating the i18n directory

We now have some more msgids to take care of. `i18ndude` won't find them in the `*.py` files, so we have to handle them manually. Let's follow Plone's lead, and put them into a `manual.pot` that we keep up to date ourselves. When you've done that, append contents of `manual.pot` to `mysite.pot` , to get a single complete master catalog.

Now, we need catalogs for our translations. I'm going to translate into Afrikaans, since that's what I know. First, ensure that the file exists, and then bring it into sync using 'i18ndude':

```
MySite $ touch i18n/mysite-af.po
MySite $ i18ndude sync --pot i18n/mysite.pot i18n/mysite-af.po
i18n/mysite-af.po: 10 added, 0 removed
```

Remember to edit all the required the metadata (such as `Language-Code` !) in the opening stanza of this file, and also in `mysite.pot` . Go ahead and translate the rest of the file as well. You'll find `mysite-af.po` in the MySite distribution, but create a new language that you know.

Note: Don't use single quotes in your message catalogs, as `i18ndude` ignores them. Wrong:

```
msgid 'help_deadline_type'
msgstr 'Kies die soort keerdatum'
```

Right:

msgid "help_deadline_type" msgstr "Kies die soort keerdatum"

You need to restart Zope or refresh the catalog from the Zope `Control_Panel` to see the new translations.

Set your browser to the language of your translation, and you should see Events and Deadlines in your language of choice. innerlink 8

Are we done yet?

Well, yes and no. Plone and Archetypes aren't going to do any more automatically, but we haven't translated everything. What have we missed? innerlink 9

The Plone templates look up actions in the `plone` domain. Look at `main_template` and follow the white rabbit from there to e.g. `global_contentviews` , where you'll see:

```
<span tal:omit-tag="" i18n:translate="">
    <span tal:replace="action/name">dummy</span>
</span>
```

As long as our actions are named the same as existing Plone actions, this code will translate them. If we have other actions that need to be translated, we'd have to add them to the plone domain. How do we do this? innerlink 10

Although a product ideally would keep all user-visible text in Page Templates, and out of Python classes, this isn't always possible. And sure enough, our little MySite has a string squirreled away in Deadline.

Translating text in Python code

Here is the untranslated method:

```
class Deadline(BaseContent):
  [... snip ...]
  # provide a mainingful title
  def Title(self):
      """overwrite the default title field"""
      return "%s for '%s'" % (getattr(self, 'deadline_type', 'deadline'),
                              self.aq_parent.getId(),
                              )
```

Here it is translated:

```
def Title(self):
    """overwrite the default title field"""
    m = {'deadline_type': getattr(self, 'deadline_type', 'deadline'),
         'parent_id': self.aq_parent.getId(), }
    return self.translate('deadline_title',
                          mapping=m, domain='mysite')
```

This corresponds to the third use case under "Translating Page Templates" above. The corresponding Page Template snippet would be:

```
<span tal:omit-tag="" i18n:translate="deadline_title">
  <span i18n:name="deadline_type">Deadline</span> for
  <span i18n:name="parent_id">Event</span>
</span>
```

What would the entry in the message catalog look like? innerlink 11

Translating vocabularies

Both our Event and Deadline classes have fields with associated vocabularies that get their initial values from `config.py` . (They are `event_type` and `deadline_type` .) Archetypes does try to translate these vocabularies when rendering the widgets, but it always uses the `plone` domain (see the macros in `Products/Archetypes/skins/archetypes/widgets/`). Therefore, if we need to translate these vocabularies, we need to provide message catalog files with more msgids for the `plone` domain.

This is one way to do it:

- Create Products/MySite/i18n/plone.pot and prepare it just as with manual.pot.
- Create and generate plone-XX.pot (where XX is your language) using i18ndude.
- Go ahead and translate.

Localising content

That's it for the translation of Products. The next step is to provide for the delivery of different content depending on the locale preferences of a user. To do this, we need current versions of the `I18NLayer` and `I18NFolder` products. The first is part of the Plone Collective project; get it at the I18NLayer download area The other one is provided by Ingeniweb. Get it from their I18NFolder homepage To install them, just unpack the tarballs in your Zope instance's Products folder and restart. Then, login to your Plone instance as a Manager user, visit the Add/Remove Products section of the Plone Control Panel, and install these products.

Although `I18NLayer` and `I18NFolder` have their limitations, you can build a fine internationalised site with them. Let's see, what are the limitations?

Folders and documents are treated differently — Most documents (i.e. content types that don't contain anything) can be translated using I18NLayer, but I18NFolder provides only a translatable version of Folder. Other containers (such as the MySite Event, which can contain Deadlines) cannot be translated.

Not all content types can be translated — You won't be able to translate your Member details, or issues in an issue tracker. How about bibliography entries? I'm not sure.

They aren't integrated into all aspects of Plone — In particular, the templates for searching, the news page, the news portlet, and other summary views will contain duplicates unless they're customised.

That said, let's create a translated area of our site.

- Add an I18NFolder, and browse to it. Do you see a language dropdown just below the content view tabs? Err, right, you don't. To fix this, go have a look at the ReadMe for I18NFolder in the Plone Control Panel, fix it, and come back here. innerlink 12
- Now you should see a language dropdown. Unfortunately this dropdown contains all the languages known to Plone, which makes it unusable. To narrow it down, we need another product.

Fetch it from the collective PloneLanguageTool download area and install as usual. Configure the `portal_languages` tool it provides by visiting it in the Zope Management Interface. In the *Allowed Languages* field, choose only the ones that you'll want to provide. Visit the new folder's language dropdown again, and you should see only the chosen languages.

In the translated folder, add some content. Content types that work well include Document, News Item, Event, Image and Photo. Topics can just about be accommodated, as long as they have no subtopics. (Don't add I18NLayer instances.)

When you've added a Document, and before trying to translate it, visit the folder in the ZMI (append `manage_main` to the URL). Just take note of the icon and satisfy yourself that you're browsing to a `Document` instance here.

That done, browse to it via the Plone UI and select a new language from the dropdown. Translate everything except the `id` field, which will have changed to the ISO language code of your target language. When you've saved the

translation, go back to the folder in the ZMI, where you'll find that the icon has changed to the green `I18NLayer` icon. Browse to this item, and inside it you'll see not one but two Document instances, each named for the language code of the translation they provide.

This is how `I18NLayer` works its magic: it automatically wraps translated items in an `I18NLayer` container, and renders the correct one depending on the user's preferences.

Solutions and Examples

1] Here's a couple of examples:

- Some people use a flag to indicate the locale a document is meant for (providing alternative images, not translations);
- A business might display different information depending on the locale of a website visitor, e.g. listing German forums before forums in other languages (providing alternate information, not only translations).

2] When you're re-using an existing Plone translation. Even though it requires quite intimate familiarity with Plone's message catalog, it can be a big win to use Plone's message ids within your product. There are many generic phrases (e.g. "Next ${number} items", msgid `batch_next_x_items`) that your product could use, and that are already maintained in more languages than you could manage on your own.

3] Then you'll have:

```
<p i18n:translate="mysite_more_plone_info">Please visit
  <span i18n:name="about-plone">
    <a href="about" i18n:translate="mysite_about_plone"
       i18n:attributes="title mysite_about_plone_title"
       title="Go to About Page">
    About Plone</a>
  </span> for more information.
</p>
```

4] This document recommends using product-prefixes: "There are also Product-specific prefixes, eg. the Product ZWiki has a heading, then the prefix would be `zwiki_heading_edit_wiki_page` . This prevents collision between Message IDs." When the product is in its own i18n domain, I don't believe it's necessary to use product-specific prefixes.

5] We passed the parameters `label_msgid` , `description_msgid` and `i18n_domain` in both cases. Although standard Plone also provides an `Event` type as part of CMFCalendar, that type isn't an archetype, and therefore doesn't accommodate this kind of internationalisation. MySite's `Event` type belongs to its domain.

6] It's only hidden on the edit form. It shows up on the view template.

7] We *could* find "Comment" in the `plone` i18n domain, but we won't find a translation for "Any further detail regarding the deadline". Because we can only specify one `i18n_domain` for this field, we need to translate both.

8] If you choose a language that isn't included among the 30-some provided with Plone, only Events and Deadlines will be translated and everything else will be in a fallback language, probably English.

9] We've missed the action names (the `actions = (...)` parts), and the configuration text (`EVENT_TYPES` and `DEADLINE_TYPES` in `config.py`). We've also missed any text that might be returned from our Python code, if we're evil enough to have such text.

10] Err, search me! Either hack the CMFPlone message catalogs, or supply another catalog for the `plone` domain. Probably the second one of these.

11] For Afrikaans:

```
msgid "deadline_title"
msgstr "${deadline_type} vir '${parent_id}'"
```

12] At `portal_skins/manage_propertiesForm` , move up the `I18NFolder` layer so it looks like this:

```
i18n_layer
I18NFolder
i18n_layer_plone2
```

Testing

Why testing?

(explain the rationale and give reference to general introductions on testing) - Put in less bugs -- *a bug is test not yet written*

When writing tests first you get feedback on your code as early as possible so you can evaluate your code while writing it. This almost inevitably results in **better code.**

- Increase confidence in your code -- from the onset you know whether your code works as expected or where it fails.
- Easier maintenance -- It allows refactoring code at a later time and make sure the module still works correctly. This provides the benefit of encouraging programmers to make changes to the code since it is easy to check if the piece is still working properly.
- Enable collaboration -- When working on code originally written by others you will notice immediately should you break something. This is an almost *must have* feature in the distributed open source development projects.
- Key ingredient in modern development paradgims like **Test Driven Development** (TTD), **Extreme Programming** (XP) and other **Agile** approaches.

Digression: Test-Driven Development (TDD)

Quoting Kent Beck innerlink 1 : The "rhythm" of TDD can be summed up as follows:

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

The surprises are likely to include:

- How each test can cover a small increment of functionality
- How small and ugly the changes can be to make the new tests run
- How often the tests are run
- How many teensy-weensy steps make up the refactorings

Testing what?

Unit tests

- Test individual classes.
- Test everything that can possibly break.
- Ideally, they are written in parallel with (or even before) the class by the programmer, i.e, they are **programmer tests.**
- They are kept and shipped with the code.

Integration tests

- Assure that individual units work together.
- They are slightly *above* unit tests because they test interactions between objects.
- They should be written by the author of a newly integrated class
- They often use the same framework as unit tests for implementation.
- They are kept and shipped with the code.

Regression tests

- Make sure a software program still does what it used to do.
- Today's unit tests are tomorrow's regression tests (as for integration and functional tests)
- They are often written according to the "feature list".
- If a software has no unit tests yet, writing regression tests is a way to raise confidence in the code.
- They are also kept and shipped with the code.

Functional tests

- Test the software from a users point of view.
- Sometimes even written by users or customers and therefore also referred to as **customer tests.**
- Also considered **acceptance tests.**
- Take a *black box* approach and may therefore use even a different platform.
- May also be shipped with the code.

How to do it in general

XUnit

Originally designed for `SmallTalk` by Kent Beck and later generalized. Key concepts are:

Test Case — tests a single scenario

Test Fixture — preparations needed to run a test

Test Suite — aggregation of multiple test cases

Test Runner — runs a test suite and presents the results

Implementations

There are implementations of XUnit for all major plattforms and languages like `JUnit` for Java, `PyUnit` for Python, `CppUnit` and `Unit++` for C++, `PerlUnit` for Perl etc.

DocTesting

Originally introduced by Jim Fulton into the Zope 3 development process it is now creaping into the Python world.

Taking the documentation aspects of testing more serious by incorporating the tests into the **doc strings** of the methods and classes right away (exploiting a specific Python feature) and allowing them to be human-readable.

Running Zope's unit tests

(quoting Lennart Regebro's contribution at http://zopewiki.org/HowToRunZopeUnitTests)

Use Zope 2.7.3 or later

The hardest thing about Zope unit tests was, and sometimes still is, running them. It can be a terrifying black art.

In Zope 2.6 and earlier, you had a test runner for products in `lib/python/Products` , another for CMF products and you needed a third-party patch for those in the standard Products-directory. Zope 2.7.0 introduced a new test runner, test.py but retained the old without deprecation warnings, and introduced more obscure bugs, making this even more confusing and terrifying. But slowly during the life of Zope 2.7 things have become better. See the troubleshooting section below if you get stuck.

So, use Zope 2.7.3 or even better, 2.7.4. You can use the test.py from latest Zope with earlier versions of Zope 2.7 but you will need to know what you are doing. Earlier versions of test.py aren't useful at all. Testing a product

Usually you want to run unit tests for one or more products. The basic command is:

```
zopectl test
```

This will run all unit tests in your Products directory. `zopectl` is found in 'INSTANCE_HOME/bin'; it runs the `bin/test.py` testrunner, but adds some useful defaults:

```
-v increases verbosity. You can add -vv for even more output.
--config-file etc/zope.conf reads in the Zope configuration file,so that important paths are set up.
--libdir Products tells test.py to include Products as a module path and to include the tests there.
```

NB if you have several zope instances, you must use the `zopectl` from the instance where your product is.

`zopectl` doesn't run on windows, so there you need to spell it out:

```
c:\path\to\python.exe bin\test.py -vC etc\zope.conf --libdir Products
```

Perhaps it's simpler after all to make sure `SOFTWARE_HOME/bin/test.py` is in your path and just do:

```
cd myproductdir; test.py --libdir . -v
```

To see the rest of the command options you can run it with '--help':

```
zopectl test --help
```

Quite often you don't want to run all the tests in Products, it takes too long time. You can limit the search for tests to a specific directory, with the --dir parameter:

```
zopectl test --dir Products/CMFCore/tests/
```

You can also add different filter statements to the command, to specify it even further. This command will run only test in the `testCookieCrumbler.py` file:

```
zopectl test --dir Products/CMFCore/tests/ testCookieCrumbler.py
```

And this will run only the test called `testCookieLogin` , in the `testCookieCrumbler.py` file:

```
zopectl test --dir Products/CMFCore/tests/ testCookieCrumbler.py testCookieLogin
```

The test output should look something like this:

```
Running unit tests at level 1
Running unit tests from /home/regebro/Zopes/Zope-trunk/Products/CMFCore/tests
Parsing /home/regebro/Zopes/Zope-trunk/etc/zope.conf
.....................
----------------------------------------------------------------------
Ran 21 tests in 0.130s

OK
```

Testing Zope

If you want to run the unit tests for Zope itself, then libdir should be set to 'SOFTWARE_HOME/lib/python':

```
zopectl test --libdir SOFTWARE_HOME/lib/python
```

or:

```
python test.py -vC etc/zope.conf
```

They do the same thing.

ZopeTestCase

(explain where to obtain and how to install `ZopeTestCase` and illlustrate the different fixtures: `PloneTestCase` , `ArchetypesTestCase` , etc.)

Providing unit tests for your product

(develop tests for MySite)

Writing functional tests

(if only I knew how. Probably better to ask someone else to write this)

Writing doc tests

(same as with functional tests)

References

A lot of material presented here was - in part literally - taken from Stefan Holek's ZopeTestCase Wiki. Thanks Stefan!

In addition to the links already included, I can recommend:

1] Kent Beck: Test-Driven Development by Example (Addison-Wesley, 2003)

Some Python specific links:

- Python's unittest module
- The PyUnit website
- Mark Pilgrim: Dive into Python

Debugging Plone

When developing for Plone you often need to know more about that is going on when your code gets executed, why errors arise, what values specific variables have, whether certain conditions are met, etc.

To this end, there are various ways in which you can make Zope and Plone be more informative about what is going on. Some of these techniques will be explained here, starting with the easiest but least powerful (printing to the browser) and working through more sophisticated methods all the way to interactive Python debugger sessions.

Printing to the browser

If all you need is some knowledge about, e.g., the result of a specific function call or the value of a certain variable it is often sufficient to write a (TTW) Python script which prints the information you need and returns it to the browser. Example: Make a Script (Python) object called `myContentItems` with the following body:

```
id = context.getId()
items = context.contentItems()

print "%s's content items:"% id
for k,v in items:
    print "%s/%s (%s)" % (id, k, v)

try:
    id = context.aq_parent.getId()
    items = context.aq_parent.contentItems()
    print "\n%s's content items:"% id
    for k,v in items:
        print "%s/%s (%s)" % (id, k, v)
except AttributeError:  # Zope root has no 'contentItems'
    pass

return printed
```

Convince yourself that you can now call this anywhere within your site by adding it to the URL of an object in your browser. So anything like `http://<your_zope_server>/MySite/Events/myContentItems` should work (replace `MySite` with your site's id of course). Note, how the result differs depending on whether you call it on a folderish or non-folderish item. For non-folderish items `contentItems` gets acquired from the parent folder yielding somewhat unexpected results. In essence, all you need to realize is how to actually use print in TTW code.

Printing to the console

Printing to the console is even easier as any `print` statement that you include in your product's code (or Plone's or CMF's or Zope's for that matter) prints to the console as soon as you run Zope in non-daemon mode, i.e., without detaching from the shell. To obtain the latter, start your Zope by using `runzope` instead of `zopectl [re]start` and you are done. But don't do this on a production server. Use a development site as using `runzope` also puts it in debug mode which will slow down Zope considerably (XXX: check whether that's actually true).

Logging

If you don't want to run in debug mode but still need to output status messages or any other kind of information you can use Zope's logging facilities. There are actually several ones (e.g., Plone provides its own) but the most convenient one is usually `zLOG` .

To illustrate its usage, here an example from the membership tool's `wrapUser` method. This particular example only works for file-system code as importing `ERROR` is not allowed in TTW code (but see below for your TTW logging

options)::

```
from zLOG import LOG, ERROR
import sys
LOG('CMFCore.MembershipTool',
    ERROR,
    'Error during wrapUser',
    error=sys.exc_info(),
    )
```

The logger takes three positional arguments: the subsystem issuing the statement, the severity level (here `ERROR` but this could alternatively be `TRACE` , `DEBUG` , `BLATHER` , `INFO` , `PROBLEM` , `WARNING` or `PANIC`), and a short description. Optional arguments are `detail` to provide a more detailed description or `error` to pass in a Python error handling object. See `ZOPE_HOME/lib/python/zLOG/__init__.py` for more.

The result of the logging statement will be added to the `event.log` file located in the `INSTANCE_HOME/log` directory. Whatch it using, e.g.:

```
tail -n 20 -f event.log
```

on a Unix-like system (show the last 20 lines of event.log and follow up on changes).

In TTW code you are not allowed to import the more severe security levels as raising them might trigger further actions but the purely informative ones are fine.

If you just want to provide some logging information in form of a plain string, Plone provides for convenience a `plone_log` method. It can be acquired from anywhere within the site and it takes one string as an argument. Use it like:

```
'context.plone_log(<some message>)'
```

from within a TTW Python script. It passes the message along to zLOG's logger with the subsystem set to `Plone Debug` and the severity to `INFO` .

Don't confuse the above with **transaction notes.** Transaction notes are comments on individual transactions in the ZODB. In Zope, you define a note on a transaction by calling:

```
get_transaction.note(<message>)
```

Plone wraps this call to make it available to TTW code. You can use it in Python scripts like:

```
from Products.CMFPlone import transaction_note
transaction_note(<message>)
```

Transaction notes do not end up in the `event.log` file but they are written to the ZODB. They may show up in the user interface to the `undo` action or provide guidance when debugging the ZODB.

Up to now we have dealt with getting at very specific, predefined information at runtime which is usually not really considered debugging your code though it may help in doing so. Now we will see how to get into real debugging sessions.

Using `zopectl debug`

Starting with the Zope 2.7 series, there is `zopectl` - at least for Linux/Unix users - which also supports the command line argument `debug` . It will launch an interactive Python session with your Zope root bound to `app` . Make sure your Zope site is shut down first (unless you are using ZEO; more on that below) or `zopectl` will complain with an `IOError` because it will be unable to lock the database file. A transcript of a debugging session follows to illustrate some of your options. I (ab)use this here also to illustrate some of the internals of Archetypes. For the session

presented here I used Archetypes-1.3.1-RC3. Do not expect the later parts to be identical on your site if you repeat this session using an older Archetypes version. Comments are added inbetween to explain what is going on.

Note to Windows users: As far as I can tell (AFAICT), `zopectl` does not work under Windows not even using cygwin. While part of its functionality is implemented by the `Plone Controller`, the `debug` option is not. Or as Duncan Booth explained this on the plone-devel mailing list (`zdctl` is the Zope daemon controller which is called by `zopectl`):

```
zdctl communicates with the running Zope instance through
a unix socket and unfortunately the AF_UNIX protocol is not
supported on Windows. I think that to get any zopectl
functionality on Windows you would have to modify zdctl to
support AT_INET as well as AF_UNIX. There are programs such
as 'Plone Controller' which provide similar functions to parts
of zopectl, but not the debugging.
```

The only way I know of to obtain this functionality under Windows is to buy an add-on like the Enterprise Server from Enfold Systems where this feature is included among others. (**Note to the note:** It is the absolute exception that I mention a particular product and company here as Enfold Systems is not just some company but led by Alan Runyan and Andy McKay - two of the godfathers of Plone.)

But let's get started now. First, you see the directory where `zopectl` lives on my development server (in general this is `$INSTANCE_HOME/bin`):

```
[ritz@pitts bin]$ pwd
/extra2/Zope-2.7.3-0/bin
```

Start `zopectl debug` and you will see the standard startup message telling you the name of your Zope root object:

```
[ritz@pitts bin]$ zopectl debug
Starting debugger (the name "app" is bound to the top-level Zope object)
```

Note: In case you wonder whether it is save to do this with an existing site that you still need for other purposes: You cannot do any damage to your site (it's a development site anyway, isn't it) as long as you don't commit your changes explicitly. More on that below.

Let's see what Zope objects we find in `app`. Compare these with what you see in the ZMI when you are at Zope root level. In addition to a fresh Zope site, I have two Plone instances here: one called `Plone`, a native Plone site, and one called `MySite`, a Plone site where the MySite product has been installed and where the sample content has been uploaded:

```
>>> for id, ref in app.objectItems():
...     print "%-25s: %s"% (id, repr(ref))
...
acl_users                 : <UserFolder instance at b6cf1860>
Control_Panel             : <ApplicationManager instance at b6cf90e0>
temp_folder               : <SimpleTemporaryContainer instance at b6cf9620>
session_data_manager      : <SessionDataManager instance at b6cf13e0>
browser_id_manager        : <BrowserIdManager instance at b6cdf8f0>
error_log                 : <SiteErrorLog at /error_log>
standard_error_message    : <DTMLMethod instance at b6cf9230>
standard_template.pt      : <ZopePageTemplate at /standard_template.pt>
standard_html_header      : <DTMLMethod instance at b6cf92c0>
standard_html_footer      : <DTMLMethod instance at b6cd14d0>
index_html                : <DTMLMethod instance at b6cf9050>
Plone                     : <PloneSite instance at b6cf9260>
MySite                    : <PloneSite instance at b6cdf830>
```

And here you see the ids of all Zope objects in 'MySite':

```
>>> app.MySite.objectIds()
['portal_actions', 'portal_catalog', 'portal_memberdata', 'portal_skins',
'portal_types', 'portal_undo', 'portal_url', 'portal_workflow',
```

```
'portal_discussion', 'portal_registration', 'portal_properties',
'portal_metadata', 'portal_syndication', 'plone_utils',
'portal_navigation', 'portal_factory', 'portal_form', 'portal_migration',
'portal_actionicons', 'portal_calendar', 'portal_quickinstaller',
'portal_groups', 'portal_groupdata', 'MailHost', 'cookie_authentication',
'content_type_registry', 'Members', 'index_html', 'acl_users',
'HTTPCache', 'RAMCache', 'caching_policy_manager', 'portal_interface',
'portal_controlpanel', 'portal_form_controller', 'error_log',
'mimetypes_registry', 'portal_transforms', 'archetype_tool',
'uid_catalog', 'reference_catalog', 'portal_membership', 'my_tool',
'events', 'Events']
```

To show only objects that have a `portal_type` assigned, we can use `contentItems()`:

```
>>> for k,v in app.MySite.contentItems():
...     print "%-11s: %s"%(k,repr(v))
...
Members    : <LargePloneFolder instance at b6800260>
index_html : <Document at /MySite/index_html>
events     : <FlexibleTopic instance at b6800da0>
Events     : <PloneFolder instance at b6800aa0>

>>> app.MySite.Events.contentIds()
['plone_conf', 'euro_python']
```

Note how `contentItems()` and `contentIds()` filters out all *non-portal-typish* objects. But you can do much more than just looking at ids and object references. First, let's define a local object reference for easier access:

```
>>> plone_conf = app.MySite.Events.plone_conf
```

Next, you will see the local namespace of the specific event instance. These are all attributes and methods intrinsic to the particular object instance (but not the inherited or acquired ones - some of the formating below I did by hand when editing the text source).:

```
>>> for k,v in plone_conf.__dict__.items():
...     print "%-20s: %s"% (k,repr(v))
...
event_url           : 'http://plone.org/events/conferences/2'
event_type          : u'Conference'
_signature          : 'W\xd1\x96\x0b\x01\xbd\xd2\xb6\x98\xf99\x92\x0b\x8c\x14\x0e'
creation_date       : DateTime('2004/11/09 13:05:43.803 GMT+1')
marshall_hook       : None
registration        : <Deadline at registration>
_Access_contents_information_Permission: ['Anonymous', 'Manager']
event_location      : {'city': 'Vienna', 'country': 'Austria'}
id                  : 'plone_conf'
modification_date   : DateTime('2004/11/09 13:05:43.856 GMT+1')
title               : u'Plone Conference'
demarshall_hook     : None
workflow_history    : {'compound_workflow': (
                          {'action': None,
                           'review_state': 'visible',
                           'actor': 'ritz',
                           'comments': '',
                           'time': DateTime('2004/11/09 13:05:43.819 GMT+1')},
                          {'action': 'publish',
                           'review_state': 'published',
                           'actor': 'ritz',
                           'comments': '',
                           'time': DateTime('2004/11/09 13:05:43.954 GMT+1')})}
_Change_portal_events_Permission: ('Manager',)
_objects            : ({'meta_type': 'Deadline', 'id': 'registration'},)
start_date          : DateTime('2004/11/19 13:05:43.738 GMT+1')
end_date            : DateTime('2004/11/21 13:05:43.738 GMT+1')
portal_type         : 'Event'
_at_uid             : 'f21a9f93784f71f437aad76306a12ac9'
at_references       : <Folder instance at b680c050>
_Modify_portal_content_Permission: ('Manager',)
_View_Permission    : ['Anonymous', 'Manager']
```

```
talkback            : <DiscussionItemContainer instance at b680c0b0>
contact             : {'phone': '+43 2741 7054',
                       'fax': '+43 2741 7054 30',
                       'address': ['BlueDynamics KEG', 'Goldegg 2', 'A-3110 Neidling'],
                       'email': 'ploneconf@bluedynamics.com',
                       'name': 'BlueDynamics'}
_md                 : {'contributors': (),
                       'language': u'en',
                       'rights': <BaseUnit instance at b67f5980>,
                       'subject': (),
                       'effectiveDate': None,
                       'expirationDate': None,
                       'allowDiscussion': None,
                       'creators': (),
                       'description': <BaseUnit instance at b67f5aa0>}
__ac_local_roles__  : {'ritz': ['Owner']}
```

This is by the way also the information that is saved in the database for this event. Most of these entries should be familiar to you but some may not. You can learn a lot about Zope/CMF/Plone/Archetypes by just inspecting objects in this way.

Also, by paying attention to the detailed data structure, you can learn more about what to expect when calling an accessor for a particular attribute and how to handle its result (see in particular the `workflow_history` for a more complex example).

But this is not the end of the game yet. If you really want to see more of the object's namespace you can use Python's `dir(...)` function to add all attribute and method names that the object obtained via its parent class and through inheritance. I will not display the full result here because it is way to long but just illustrate it:

```
>>> len(dir(plone_conf))
800
>>> dir_list = dir(plone_conf)
>>> for i in range(0,len(dir_list)-2, 3):
...     print "%-20s\t%-20s\t%-20s" % \
...     (dir_list[i],dir_list[i+1],dir_list[i+2])
...
COPY                    COPY__roles__           Contributors
Contributors__roles__   CreationDate            CreationDate__roles__
Creator                 Creator__roles__        Creators
Creators__roles__       DELETE                  DELETE__roles__
Date                    Date__roles__           Description
Description__roles__    EffectiveDate           EffectiveDate__roles__
Epoz                    EpozGetRelativeUrl      EpozTidy
EventTypes              ExpirationDate          ExpirationDate__roles__
Format                  Format__roles__         HEAD
HEAD__roles__           Identifer               Identifier
Identifier__roles__     LOCK                    LOCK__roles__
...
```

It is highly recommended to take a look at the full list though.

Let's dig deeper now into the internals of Archetypes and look at the schema next (some of the formatting below I did by hand):

```
>>> for k,v in plone_conf.Schema().__dict__.items():
...   print "%-10s: %s\n" % (k,v)
...
__name__  : default

_fields   : {'event_url': <Field event_url(url:rw)>,
             'end_date': <Field end_date(datetime:rw)>,
             'description': <Field description(text:rw)>,
             'contributors': <Field contributors(lines:rw)>,
             'title': <Field title(string:rw)>,
             'language': <Field language(string:rw)>,
             'start_date': <Field start_date(datetime:rw)>,
```

```
             'rights': <Field rights(text:rw)>,
             'modification_date': <Field modification_date(datetime:rw)>,
             'event_type': <Field event_type(string:rw)>,
             'creation_date': <Field creation_date(datetime:rw)>,
             'deadlines': <Field deadlines(computed:r)>,
             'effectiveDate': <Field effectiveDate(datetime:rw)>,
             'contact': <Field contact(contact:rw)>,
             'expirationDate': <Field expirationDate(datetime:rw)>,
             'allowDiscussion': <Field allowDiscussion(string:rw)>,
             'creators': <Field creators(lines:rw)>,
             'event_location': <Field event_location(location:rw)>,
             'id': <Field id(string:rw)>,
             'subject': <Field subject(lines:rw)>}

_names    : ['id', 'title', 'allowDiscussion', 'subject', 'description',
             'contributors', 'creators', 'effectiveDate', 'expirationDate',
             'language', 'rights', 'creation_date', 'modification_date',
             'event_type', 'start_date', 'end_date', 'event_location',
             'event_url', 'contact', 'deadlines']

_props    : {'marshall': <Products.Archetypes.Marshall.RFC822Marshaller
                          instance at 0xb72b086c>}

_layers   : {'marshall': <Products.Archetypes.Marshall.RFC822Marshaller
                          instance at 0xb72b086c>}
```

In the `_fields` dictionary above you see references to the individual field objects. Let's look at the title field in more detail:

```
>>> plone_conf.Schema()['title']
<Field title(string:rw)>
>>> for k,v in plone_conf.Schema()['title'].__dict__.items():
...   print "%-15s: %s" % (k,v)
...
widget         : <StringWidget instance at b77a99b0>
generateMode   : veVc
read_permission: View
searchable     : 1
enforceVocabulary: False
mutator        : setTitle
validators     : ()
__name__       : title
accessor       : Title
edit_accessor  : getRawTitle
index          : None
default_content_type: text/plain
isMetadata     : False
default        :
_layers        : {'storage': <Storage AttributeStorage>}
required       : 1
storage        : <Storage AttributeStorage>
write_permission: Modify portal content
index_method   : _at_accessor
languageIndependent: False
schemata       : default
multiValued    : False
default_method : None
vocabulary     : ()
force          :
type           : string
mode           : rw
```

Note that nowhere in the schema or in the field itself we find the actual value of the title whereas in the event instance's dictionary we can clearly see it. The latter is simply due to the fact that the title field uses `AttributeStorage` which means that the value itself is stored as an attribute on the instance. This allows for a clean separation of the instance, the schema, the fields, and the actual values. This can be further illustrated by the way in which you can get the actual value back given the field. Just look at this:

```
>>> plone_conf.Schema()['title'].getAccessor(plone_conf)
```

```
<bound method Event.Title of <Event instance at b67f3d40>>

>>> plone_conf.Schema()['title'].getAccessor(plone_conf)()
'Plone Conference'
```

Note that `getAccessor` needs the event instance (`plone_conf`) to be passed in to re-establish the connection between the field and the instance. Calling this accessor then finally returns the actual value.

So much for introspection (and a somewhat lenthy digression into the internals of Archetypes). But what is really cool is that you can also manipulate objects from the Python prompt directly:

```
>>> plone_conf.Title()
'Plone Conference'

>>> plone_conf.setTitle('Second Plone Conference')

>>> plone_conf.Title()
'Second Plone Conference'
```

You can even do the *forbidden thing* and change attributes directly (it's just Python after all) without calling their accessors. This should, of course, only be done if you need to change attributes for which there is no accessor available to avoid unnecessary side effects:

```
>>> plone_conf.title = 'Next Plone Conference'

>>> plone_conf.Title()
'Next Plone Conference'
```

Don't be shy in checking things out this way. As stated above, you cannot do much damage to your site as long as you don't commit your changes explicitly. That means, you can leave a debugging session at any time using `Ctrl-D` (the Python way to quit an interactive session) and your site will be as before no matter what you have changed interactively. <!-- % Anonymous User - Nov. 17, 2004 9:36 pm: You may want to state this as a NOTE: up in the begining when you start the zopectrl debug session. -->

If, on the other hand, you want your changes to be made persistent (e.g., because you fixed a broken instance) you need to tell the ZODB to keep your changes by calling:

```
get_transaction().commit()
```

Be aware that this writes ALL changes that you did up to this call into the ZODB. If you called this in error you can still use the `undo` facilities of the ZODB. If the site can be restarted, this can even be done later TTW via the `undo` tab in ZMI.

Using Python's debugger (`pdb`) with `emacs`

Probably the most convenient and advanced way to develop and debug Plone is to use an IDE (an Integrated Development Environment). While there are a few IDEs around (commercial as well as freely available ones), I personally never felt the need to get one. As of now, I never used more than `emacs` (including its Python mode) and Python's debugger `pdb` .

To get started, you need to know how to set a **break point** , i.e., how to tell Python to stop at a certain point in the processing chain. To do so, edit the function or method that you want to debug to include the following line at the point where you want Python to take a break:

```
import pdb; pdb.set_trace()
```

Now comes the trick: Restart the server from a shell within `emacs` using `runzope` . Use a browser to go to a page or action that evokes the method containing the break point. Return to your `emacs` session and observe that there are now two windows: the shell where you started Zope became an interactive `pdb` session and the second window

shows you the source code with the cursor at the point where the process is currently at.

Now use whatever pdb command you like, e.g., **n** (ext): execute the next command, **s** (tep): step into the next command called, **u** (p): move up the execution stack, **d** (own): move down the stack, **p** (rint), etc. Use **h** (elp) to learn more about your options and see http://docs.python.org/lib/module-pdb.html for an overview of `pdb` and http://docs.python.org/lib/debugger-commands.html for the commands available in `pdb` .

Note how the content in the source code window changes as you step through the code. That way you can always see immediately where and how the current method is actually defined.

Last but not least, note that everything you type in at a `pdb` prompt that cannot be interpreted by `pdb` itself is treated as a regular Python expression.

To terminate a `pdb` session type **q** (uit). Don't worry about the `BdbQuit` error that comes up in the browser. Your site is fully functional until you hit the breakpoint again.

You can get `emacs` even closer to a real IDE by combining it with **Bicycle Repair Man** (see http://c2.com/cgi-bin/wiki?BicycleRepairMan for more).

(Thanks to Jim Fulton for sharing this knowledge with Thomas Förster who in turn passed this on to me.)

Using ZEO

Zope Enterprise Objects (or ZEO for short) provides access to one ZODB for several clients. Its main usage is to improve site performance by enabling several computers to act as webservers in parallel for one Zope site - all sharing the same data.

But ZEO is also useful for debugging a live site because it allows you to connect a debugging session as outlined above to a running site. No need to shut it down anymore. All you need is a ZEO setting with at least two Zope instances as clients. All but one are serving your site as usual while you can use the remaining client for debugging.

See also:

- Setting up and using ZEO in general: http://www.plope.com/Books/2_7Edition/ZEO.stx
- Ken Manheimer's wiki on Zope debugging at zope.org: http://zope.org/Members/klm/ZopeDebugging/FrontPage
- Ken's original paper on the matter: http://zope.org/Members/klm/ZopeDebugging/ConversingWithZope
- A more advanced coverage of the matter at zopewiki.org: http://zopewiki.org/DebuggingZopeWithPythonDebugger2

Conclusions

(still to be written)

What did we cover

What didn't we cover (templating, Zope 3, Five)

Where to go from here

Take-home message

Solution Proposals for the Exercises

1] Fundamental Background - Object Orientation

See any modern textbook on programming or read the respective chapter from the Zope book (http://zope.org/Documentation/Books/ZopeBook/2_6Edition/ObjectOrientation.stx)

In a nutshell, an **object** is a programmatic representation of some real-world or conceptual entity. A **class** then is a template that formalizes and implements the object's definition in order to serve as a *machine* to generate **instances** of those objects by calling its **constructor** (in Python, this is the class's `__init__` method).

Classes are typically equipped with **attributes** (Variables) and **methods** (functions applicable to instances of this class) and so are the instances generated from the class. When adding or changing attributes or methods you can either do that for the entire class (by changing its definition) or for an individual instance. The latter is what you typically do when working TTW with Plone, the former when writing your custom tools and types.

Properties are in the abstract sense managed attributes. This means that there are dedicated methods for accessing (hence the term **accessor**) and for changing the value of an attribute (the **mutator**).

Quoting from http://docs.python.org/lib/built-in-funcs.html:

getattr(object, name[, default]) Return the value of the named attribute of object. name must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, getattr(x, `foobar`) is equivalent to x.foobar. If the named attribute does not exist, default is returned if provided, otherwise AttributeError is raised.

hasattr(object, name) The arguments are an object and a string. The result is 1 if the string is the name of one of the object's attributes, 0 if not. (This is implemented by calling getattr(object, name) and seeing whether it raises an exception or not.)

setattr(object, name, value) This is the counterpart of getattr(). The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, setattr(x, `foobar` , 123) is equivalent to x.foobar = 123.

2] Python Background - Collection Types

In abstract terms, any number of instances of some type is sometimes referred to as a **collection** . A collection may be ordered or not. If so, it is called a **sequence** . Otherwise it is just a **set** . If the elements from the collection can be uniquely accessed via some **key** , it is called a **mapping** .

In Python there are currently six sequence types: strings, Unicode strings, lists, tuples, buffers, and xrange objects but there is only one built-in mapping type: the dictionary.

For an illustration of **slicing** , see examples 3.8 and 3.9 from http://diveintopython.org/native_data_types/lists.html

The major difference between a `list` and a `tuple` is that lists are mutable whereas tuples are not. With other words, a tuple is an immutable list. A tuple can not be changed in any way once it is created. This is why you should generally use tuples in a multi-threaded application like Zope whenever possible and why you should NEVER iterate over a list that you want to change within the loop. If you need to do this, make a copy of this list beforehand (`copy_of_mylist = mylist[:]`).

For a complete overview over the types available in Python see http://docs.python.org/lib/types.html

3] Fundamental Background - Name Space

In a nutshell, a name space is just a collection of terms understood by a programm at runtime. It is useful to distinguish between unqualified names (e.g., `x = 1` makes `x` a local, unqualified name) and qualified names (e.g., `object.x = 1` puts `x` in the name space of `object`, so to access `x` later again, you need to *qualify* `x` via `object`, i.e., use `object.x` for access).

Unqualified names follow the local-global-builtin, or LGB, rule meaning a name is first looked up locally, then globally and finally among the (Python) built-ins.

Qualified names are looked up by Python in the qualified object's name space. First, the object instance itself is checked for the name, next, the class the object is an instance of is searched and finally the superclasses are looked up (if there are any).

Zope extends the name space for objects that "live" in the ZODB through acquisition and CMF via skinning (which is also based on acquisition).

See http://www.python.org/dev/doc/devel/ref/naming.html for more on Python's naming and binding or http://www.neuroinf.de/PloneDevTutorial/big_picture#1-4 for a slightly more detailed account on Zope/CMF/Plone's name space handling.

4] Zope Background - PropertyManager

Quoting from '.../yourZope/lib/python/OFS/PropertyManager.py':

The `PropertyManager` mixin class provides an object with transparent property management. An object which wants to have properties should inherit from PropertyManager.

An object may specify that it has one or more predefined properties, by specifying an _properties structure in its class:

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

The _properties structure is a sequence of dictionaries, where each dictionary represents a predefined property. Note that if a predefined property is defined in the _properties structure, you must provide an attribute with that name in your class or instance that contains the default value of the predefined property.

Each entry in the _properties structure must have at least an `id` and a `type` key. The `id` key contains the name of the property, and the `type` key contains a string representing the object's type. The `type` string must be one of the values: `float`, `int`, `long`, `string`, `lines`, `text`, `date`, `tokens`, `selection`, or `multiple section`.

For `selection` and `multiple selection` properties, there is an addition item in the property dictionay, `select_variable` which provides the name of a property or method which returns a list of strings from which the selection(s) can be chosen.

Each entry in the _properties structure may *optionally* provide a `mode` key, which specifies the mutability of the property. The `mode` string, if present, must contain 0 or more characters from the set `w`, `d`.

A `w` present in the mode string indicates that the value of the property may be changed by the user. A `d` indicates that the user can delete the property. An empty mode string indicates that the property and its value may be shown in

property listings, but that it is read-only and may not be deleted.

Entries in the _properties structure which do not have a `mode` key are assumed to have the mode `wd` (writeable and deleteable).

To fully support property management, including the system-provided tabs and user interfaces for working with properties, an object which inherits from PropertyManager should include the following entry in its manage_options structure:

```
{'label':'Properties', 'action':'manage_propertiesForm',}
```

to ensure that a `Properties` tab is displayed in its management interface. Objects that inherit from PropertyManager should also include the following entry in its __ac_permissions__ structure:

```
('Manage properties', ('manage_addProperty',
                       'manage_editProperties',
                       'manage_delProperties',
                       'manage_changeProperties',)),
```

So far for the quote. Note that all portal content inherits from `PropertyManager` . Further note that defining security the way in which it is described here is deprecated. Today, we should use declarative security assertions instead as described in the main section for MyTool . Again, see http://zope.org/Documentation/Books/ZDG/current/Security.stx for more on this central issue.

Now, we turn to `setupProperties` from `Extensions/Install.py` as defined here:

```
def setupProperties(self, out):
    # install my properties
    ptool = getToolByName(self, 'portal_properties')
    psheet = getattr(ptool, 'my_properties', None)
    if not psheet:
        ptool.addPropertySheet('my_properties', 'Properties of my custom site')
        ps = getattr(ptool, 'my_properties')
        ps._properties = ps._properties + (
            {'id':'event_types', 'type':'lines', 'mode':'w'},
            {'id':'deadline_types', 'type':'lines', 'mode':'w'},
            )
        ps._updateProperty('event_types', EVENT_TYPES)
        ps._updateProperty('deadline_types', DEADLINE_TYPES)
```

Here, the private (as indicated by the underscore) `_properties` attribute and private `_updateProperty` method are used as opposed to the correponding public API methods:

```
manage_addProperty(self, id, value, type, REQUEST=None)
```

and

```
manage_changeProperties(self, REQUEST=None, **kw)
```

so an alternative way to write `setupProperties` is:

```
def setupProperties(self, out):
    # install my properties
    ptool = getToolByName(self, 'portal_properties')
    psheet = getattr(ptool, 'my_properties', None)
    if not psheet:
        ptool.addPropertySheet('my_properties', 'Properties of my custom site')
        ps = getattr(ptool, 'my_properties')
        ps.manage_addProperty('event_types', EVENT_TYPES, 'lines')
        ps.manage_addProperty('deadline_types', DEADLINE_TYPES, 'lines')
```

5] Python Background - Monkey Patching

For the patch, add the following lines to the `__init__.py` of `MySite` :

```
# adding the 'Properties' tab to ZMI for PortalContent
from Products.CMFCore.PortalContent import PortalContent
from OFS.PropertyManager import PropertyManager

manage_options = PortalContent.manage_options +\
                 PropertyManager.manage_options

PortalContent.manage_options = manage_options
# end of monkey patch
```

and restart your server. Note, that the `properties` tab is now also available for non-folderish content objects (per default, only folderish portal content provides this option).

Next, via ZMI, add a boolean property called `displayByline` with a value of 1 (True) to the plone root object of your site. Repeat this for the site's `index_html` document but with a value of 0 (False).

Finally, customize the `document_byline` template to include a test on this property to decide whether to display anything or not. To this end, place:

```
tal:condition="python:here.getProperty('displayByline')"
```

into the opening `div` tag of the template.

Now, whenever the `document_byline` gets called, the property `displayByline` will be acquired from the portal root, unless the item to be displayed or one of its parent folders overrides its value.

6] Zope Background - ZCatalog

Quoting from http://www.neuroinf.de/Miscellaneous/BeginnersGuide:

This is what you need in order to search the database (effectively). For plain Zope you need to *build* one yourself whereas CMF/Plone come with a pre-configured `portal_catalog` . There are some issues that aren't obvious at the beginning, like indexes, brains, and meta data so I sketch them here.

- **Index** : You can query the catalog only on an index and there are several types of indexes supporting different functionality, e.g., on `field indexes` you can sort (and do range searches) whereas on `text indexes` you can't sort but enable *globbing* (using wildcards).

- **brains** : The catalog does not hold (copies of) the objects indexed themself but rather a reference object to them called `brain` or `catalog brain` in Zope terminology. Brains provide methods like `getObject()` or `getURL()` to get at the respective object or its URL. Furthermore brains can be equiped with

- **catalog meta data** to have some attribute values or results of method calls to the object availabel right away (without having to access the object). It's up to you how many meta data you want (or can afford) to put on your brains. You basically trade disc space for performance.

For a real intro see http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/SearchingZCatalog.stx

7] CMF Background - Content Generation

CMF's basic method for content generation is `invokeFactory` . It is originally defined in the `PortalFolder` class as defined in the `Products/CMFCore/PortalFolder.py` module. It gets the type information for the current object (the one where the new item should go into) from the types tool (`portal_types`), checks whether the requested

new content type is allowed to be added to this type of object (note that you can configure this TTW) and if so, the types tool's `constructContent` is called. The types tool now checks whether the current user is allowed to add the new portal content to the current folder and if so, `constructInstance` from the new content type's type info object is called. This looks up the registered constructor for the new type, calls it and now the new object is in place.

But there are still a few things left to do: After constructing the new instance `constructInstance` calls `_finishConstruction` (defined in the `TypeInformation` class of the `Products/CMFCore/TypesTool.py` module) to take care to set the portal type (by calling `_setPortalTypeName`) and to notify the workflow - in case the new object is subject to one - by calling `notifyWorkflowCreated` .

Note, that adding portal content via ZMI skips the last part (`_finishConstruction`) because via ZMI only the object's constructor is called. That's why you should usually refrain from using the ZMI for the creation of portal content.

Plone's folder (more precisely, `BasePloneFolder` in `Products/CMFPlone/PloneFolder`) overrides `invokeFactory` to return the id of the newly created object.

Furthermore, Plone wraps the call to `invokeFactory` by a skin method `createObject` to provide an auto-generated id and to optionally buffer object creation by Plone's factory.

`portal_factory` , an instance of the `FactoryTool` class as defined in `Products/CMFPlone/FactoryTool.py` , provides `doCreate` . This can be used to buffer the creation of the new object in a temporary folder. This is needed to offer the user a proper cancelling option while (s)he is editing the newly created object. Without using portal factory you will end up having empty content objects whenever one of your users adds a new content item without submitting an edit form for it as well. Note, that you have to turn on this functionality explicitly by indicating the types for which portal_factory should be used for object creation. To do so, go to `portal_factory/Factory Types` via ZMI and select the types you want to enable this functionality for.

Digression - Object Instantiation: As Plone is a high-level application, there are almost always several ways to do the same thing. This is illustrated here using object instantiation as an example:

1. The Python way: `myobbject = MyClass()` - calling the class generates an object created by executing `__init__.py`

2. The private Zope way - `parent` is an `ObjectManager` :

```
myid = 'foo'
myobject = MyClass()
parent._setObject(myid, myobject)
```

3. The public Zope way: `parent.addMyClass("foo")` - calling a Zope-specific constructor

4. An internal CMF way - `parent` is a `PortalFolder` (this is useful if you need to bypass the allowed types checking):

```
portal_types.constructContent(type_name='MyType', container = parent, id = 'foo')
```

5. The "official" CMF way: `parent.invokeFactory(type_name="MyType", id = "foo")`

6. The Plone way: `parent.createObject(type_name="MyType", id = "foo")` - this is a skin method that is worth taking a look at!

In general, it is recommended to use the highest possible API level (in terms of abstraction) unless you know that you don't want some of the services included there or if you need to bypass restrictions. Probably the hardest task when learning to programm Plone is become aware of the different options offered at the different levels of the framework and to make correct and consistent use of its API. In order to be able to do so, you should have at least a good

overview over the different concepts provided by Python, Zope, CMF, Plone, and Archetypes. That is why, in the end, it is not so easy as it may seem at first glance to become a good Plone programmer. Most importantly, you need to get practice and - alas- read the source!

8] Fundamental Background - Inheritance

The solution to this problem is given in the look-up section of the Big Picture chapter of this book. (This exercise is here to bring your attention to this chapter in case you started reading with the main part.)

9] Plone Background - createMemberArea

(still to be written)

10] Archetypes Background - Basic Concepts

(still to be written)

11] CMF Background - Type Registration

Probably the easiest way to achive that our custom event type does not replace the default one but is made available additionally is to give it a name other than `Event` , e.g., `EventWithDeadlines` .

Thanks to the magic of archetypes there is not much you need to do in order to achieve this. First of all, change the name of the class in `Event.py` (the module may keep its name) to `EventWithDeadlines` . Then make sure that the call to `registerType(Event)` is changed accordingly to `registerType(EventWithDeadlines)` .

Finally, there are a number of places in `Install.py` that you want to adjust:

- Add `EventWithDeadlines` to the meta types not to list in the navigation tree in `setupProperties` .
- Add it to the content types to be included in the calendar in `editCalendarTool` .
- `setupCompoundWorkflow` should now, of course, set the default chain for the renamed type.
- The `event` and `no_deadline` topics created in `setupInitialContent` should include the new one in their criterions on `portal_type`

12] Archetypes Background - References

(still to be written)

13] CMF Background - Workflow

To make the handling of a deadline's review state not subject to the workflow machinery itself but to leave this for the parent event we will make three major changes:

1. Remove the deadline types from the types subject to workflow

This can either be done though the web via ZMI under `portal_workflow` by erasing the `(Default)` entry for deadlines under *workflow by type* or by adding something like::

```
wf_tool.setChainForPortalTypes( ('Deadline',), '')
```

in a function similar to `setupCompoundWorkflow` (which does not need to be called any longer) in the `install` .

2. Add a method to the `Deadline` class called `review_state` to return the parent's state (except for pending):

```
def review_state(self, default = 'private'):
    """ return the parent's review state except when it is pendig
        return None in this case"""
    my_parent = self.aq_inner.aq_parent
    from Products.CMFCore.utils import getToolByName
    wt =  getToolByName(self, 'portal_workflow')
    parent_state = wt.getInfoFor(my_parent, 'review_state')
    if parent_state == 'pending':
        parent_state = default
    return parent_state
```

3. Override `indexObject` and `reindexObject` in the `Event` class to also (re)index the deadlines:

```
def indexObject(self):
    """override indexObject from CatalogMultiplex to also update deadlines"""
    # first the default
    BaseFolder.indexObject(self)
    # now the deadlines
    for deadline in self.contentValues('Deadline'):
        deadline.indexObject()

def reindexObject(self, idxs=[]):
    """override reindexObject from CatalogMultiplex to also update deadlines"""
    # first the default
    BaseFolder.reindexObject(self, idxs)
    # now the deadlines
    for deadline in self.contentValues('Deadline'):
        deadline.reindexObject(idxs)

<seqreset id="number">
```

See http://www.neuroinf.de/epydoc/at-api/public/Archetypes.CatalogMultiplex.CatalogMultiplex-class.html and http://www.neuroinf.de/epydoc/at-api/public/Products.CMFCore.CMFCatalogAware.CMFCatalogAware-class.html for the original `(re)indexObject` methods.

Now, whenever an event is updated and (re)indexed, the event's deadlines will also get (re)indexed. While compiling the catalog metadata `review_state` will be called on the deadlines to return the parent's review state except when the event is pending review.

14] Python Background - Signature

To enable the specification of an optional argument to `getDisplayList` from the `my_utils.py` module to specify the porperty sheet to be used for the look-up of the property value change it to:

```
def getDisplayList(self, prop_name=None, prop_sheet=None):
    return makeDisplayList(getValues(self, prop_name, prop_sheet))
```

For this to work, `getValues` has to be extended to something like:

```
def getValues(self, prop_name, prop_sheet):
    if not prop_sheet: prop_sheet = 'my_properties'  # hard coded default
    ptool = getToolByName(self, 'portal_properties', None)
    if ptool and hasattr(ptool, prop_sheet):
        return getattr(ptool, prop_sheet).getProperty(prop_name, None)
    else:
        return None
```

(Note: add a digression on named arguments and what `**kw` is about)

15] Python Background - Submodules

It may well happen that your product becomes so complex that you would like to structure the package by grouping modules into subfolders (like moving MySite's `Deadline.py` and `Event.py` into a subfolder `MyContentTypes`).

Now, in order to support statements like:

```
from Products.MySite.MyContentTypes.Deadline import Deadline
```

`MyContentTypes` has to be a Python product or package itself. But a folder in the file system is not per default considered a package by Python. On start-up, Python looks through every folder in a package for a file called `__init__.py` and if it findes one the folder is considered a package (and the content of `__init__.py` gets executed if there is any). So to tell Python that your folder should be recognized as a package it is sufficient to place an `__init__.py` file in there, even if it is empty.

Now consider what happens if such an empty `__init__.py` file gets lost on distributing your product. Your subfolder will not be treated as a package anymore, statements like the `import` above will fail (raise an `ImportError`) and your product will be broken. So to be on the safe side, place at least some dummy content in there.

16] Optimization - Search Template

The custom (advanced) search template differs from Plone's original template in one single expression: Where the original template gets the types to offer in the form from the types tool directly by calling:

```
<tal:contenttypes repeat="type python:portal_types.listContentTypes()">
```

MySite's custom template gets this list from calling:

```
<tal:contenttypes repeat="type here/getSearchableTypes">
```

a custom script from the `my_scripts` skin folder which in turn starts out with the same list:

```
types = context.portal_types.listContentTypes()
```

but then only returns it after removing all entries listed in:

```
types_to_exclude = context.getNotSearchableTypes()
```

This demonstrates a combined white list/black list approach, meaning there is a *positive* or *white* list of all types that should possibly be considered here and a *negative* or *black* list containing all types that should definitively not be considered.

At first glance this may seem redundant but realize that the white list is generated dynamically (all types known to the types tool at runtime) whereas the black list is static (but customizable). Anytime you register a new content type for your site it will appear in the list returned by the types tool and therefore among the content types offered for searching. If this is not what you want you have to manually include it in the blacklist but the default behavior is probably what you want in most cases.

Note, that the list of content types offered for addition to a folder are determined in the same way by calling `getAddableTypes` which filders then on `getNotAddableTypes` .

Using a script like `getNotSearchableTypes` to just return a static list may seem ackward. A static list should possibly be managed using a property instead. To this end, add a property to `portal_properties.my_properties` . You can do this either through-the-web or from MySite's `install` by adding:

```
ps.manage_addProperty('not_searchable_types', NOT_SEARCHABLE_TYPES, 'lines')
```

to `setupProperties` after defining `NOT_SEARCHABLE_TYPES` in the `config` file. Then change the second line in `getSearchableTypes` to:

```
types_to_exclude = context.portal_properties.my_properties.getProperty('not_searchable_types', [])
```

and you are done. Alternatively, the property could be added to the portal root object itself, then access would be easier thanks to acquisition:

```
types_to_exclude = context.getProperty('not_searchable_types', [])
```

and we gain the possiblity to override the list's value from within any subfolder of the site. For the definition of searchable types this may seem like YAGNI (you ain't gonna need it) but for defining the not-addable types this should be the way to go (and maybe Plone will do it that way in the future).

17] List Comprehension

No implementations in other languages have been submitted up to now. Send me one and I will include it here together with your name (or not; as you like).

Major Python/Zope/CMF/Plone/Archetypes resources

Python: http://www.python.org

Zope: http://www.zope.org (the community) and http://www.zope.com (Zope Corporation)

Zope's Content Management Framework (CMF) http://cmf.zope.org

Plone, an advanced front end to the CMF at http://plone.org

Archetypes, which enables easy creation of custom content types for CMF and Plone (http://sourceforge.net/projects/archetypes)

Documentation

Python

- the definitive reference: http://www.python.org/doc/
- to get started (for otherwise experienced programmers): http://diveintopython.org/
- to get started from scratch: How to think like a computer scientist http://www.ibiblio.org/obp/thinkCSpy/

Zope

- The Zope Book (2.6 edition): http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition
- The Zope Book (2.7 edition; forthcoming) http://www.plope.com/Books/2_7Edition
- Zope Developers Guide: http://www.zope.org/Documentation/Books/ZDG
- Zope Cookbook at http://www.zopelabs.com
- Dieter Maurer's upcoming(?) *Building Dynamic Web Sites with Zope* : http://www.dieter.handshake.de/pyprojects/zope/

CMF

Unfortunately there is no comprehensive book on the CMF available up to now but there are bits and pieces on various topics. And there is some documentation that ships with the distribution and that's available through the Help section of the management interface.

Plone

- The documentation section on plone.org (http://plone.org/documentation) and there in particular the book and the how-tos:
- The book of Plone (this is for Plone 1): http://plone.org/documentation/book/

- Andy McKay: The Definitive Guide to Plone (this is for Plone 2): http://plone.org/books/definitive_guide

Some chapters from the book are available in electronic form from
http://cvs.sourceforge.net/viewcvs.py/plone-docs/PloneBook/

- Plone HowTos: http://plone.org/documentation/howto
- The `plone-docs` project at sourceforge in general: http://cvs.sourceforge.net/viewcvs.py/plone-docs/

Archetypes

- Entry page to various documents: http://plone.org/documentation/archetypes

More Specific Tools for Finding or Generating Documentation

or how to effectively search for something related to your problem:

Full-text Searchable Mailing List Archives

- at gmane as linked from plone.org: http://plone.org/documentation/lists
- at active state: http://aspn.activestate.com/ASPN/Mail
- at New Information Paradigms Ltd.: http://zope.nipltd.com/public/lists.html

Documentation Generating Tools

DocFinder

http://www.handshake.de/~dieter/pyprojects/zope/DocFinder.html in conjunction with DocFinderTab:
http://www.zope.org/Members/shh/DocFinderTab

DocFinder can be of great help when discovering object APIs and debugging security problems. It furthermore treats docstrings as structured text which is a very powerful feature as it allows to write nicely-formatted documentation right into the source, while having it instantly available TTW.

HappyDoc

http://happydoc.sourceforge.net/ HappyDoc is a tool for extracting documentation from Python source code. It differs from other such applications by the fact that it uses the parse tree for a module to derive the information used in its output, rather that importing the module directly. This allows the user to generate documentation for modules which need special context to be imported.

epydoc

http://epydoc.sourceforge.net/ Epydoc is a tool for generating API documentation for Python modules, based on their docstrings. For an example of epydoc's output, see the API documentation for epydoc itself (html - http://epydoc.sourceforge.net/api/ , pdf - http://epydoc.sourceforge.net/epydoc.pdf). A lightweight markup language

called epytext can be used to format docstrings, and to add information about specific fields, such as parameters and instance variables. Epydoc also understands docstrings written in ReStructuredText, Javadoc, and plaintext.

Examples illustrating the kind of documentation generated by epydoc are available online (as long as we can affort the bandwidth) for

- Archetypes at http://www.neuroinf.de/epydoc/at-api
- Plone at http://www.neuroinf.de/epydoc/plone-api

Zpydoc

http://www.last-bastion.net/Zpydoc Zpydoc is an initiative to apply the standard documentation formalisms utilised within Python to Zope - itself a Python application. Its basically a wrapper to make `pydoc` (but also much more) available from within Zope directly.

Other Development-Supporting Tools

VerboseSecurity

http://hathawaymix.org/Software/VerboseSecurity VerboseSecurity is an add-on product for Zope that helps explain the reason for denied security access.

It attempts to explain the complete reasoning for failed access. It shows what object was being accessed, what permission is required to access it, what roles map to that permission in that context, the executable object and its owner, the effective proxy roles, and other pertinent information. All of this information appears in the exception message when access is denied.

TCPWatch

http://hathawaymix.org/Software/TCPWatch TCPWatch is a utility written in Python that lets you monitor forwarded TCP connections or HTTP proxy connections. It displays the sessions in a window with a history of past connections. It is useful for developing and debugging protocol implementations and web services.

BoaConstructor

http://boa-constructor.sourceforge.net/ is a cross platform Python IDE (integrated development environment) with Zope support.

ZopeProfiler

http://www.dieter.handshake.de/pyprojects/zope/#bct_sec_2.4 ZopeProfiler provides profiling support for the development of Zope applications. It can derive both high and low level timing statistics (Zope object call and Python function call level, respectively).

Unlike with the standard Zope profiling support, Zope runs normally in multi-threaded mode. Statistics gathering can be enabled/disabled dynamically via the ZopeProfiler object in the Control_Panel. This object supports most features of Python's pstats.Stats timing analysis class and is much smarter than the standard Zope profiler. Should the features be insufficient, then statistics data can be saved to files and later analysed with Python's Stats class.

Installing Zope, Plone and Products

There are several ways to make a Plone installation. Some depend on the operating system you are using. For a development set-up, I recommend to do an installation from the source files because this is usually the only way to work right out of current subversion (or CVS) check outs. If you are unfamiliar with versioning systems as they are used for software development, see here for an introduction to CVS which is used for the collective projects hosted at SourceForge or for SubVersion which is used by Zope and Plone as well as an increasing amount of collective projects now hosted on svn.plone.org/collective.

Source Installations in General

Source installations usually always work the same way: You download some source code - most often packed as a **tar** archive (also called *tar-ball*) so that you can get an arbitrarily complex directory tree as one single file. The first thing to do then is to *unpack* the tar-ball with "tar -xzvf <something.tgz> " into some directory of your choice. Alternatively, you make a CVS or SubVersion check-out. Then look for instructions. Files called "README" or "INSTALL" are your best bet. Most of the time, however, it boils down to the following three steps: `configure; make; make install` . Calling `configure` checks your computer for available software that may be needed and most importantly, figures out *where is what* in order to set variables needed for the following compilation correctly. Then `make` does the main work: it compiles the source code that you've just downloaded so that it can be executed later on your machine. Finally, `make install` puts the compiled files where they belong on your system or where you have defined them to go. This last step usually requires root or administrator priviledges, so if you don't have them make sure to change the configuration such that the compiled files go to a place where you have `write` permissions.

Zope and Plone

Zope

Download the current tar-ball from zope.org and unpack it into a directory of your choice. Change into the just created `Zope-2.x.y-something` folder and call `./configure` followed by `make` and `make install` . If you want an installation that can easily be removed later, you can put everything in this folder by answering `.` (the current directory) when asked where to put Zope. For a standard installation this is it. You can now start your Zope by calling `bin/zopectl start` or `bin/runzope` and then point your browser to `localhost:8080` to contact it.

Plone

Plone installs like any other Zope 2 extension by putting the unpacked or checked out source code into Zope's `Products` folder. The only thing to note here is that Plone (like `MySite`) consists of several so-called *Products* each of which needs to go into Zope's `Products` directory individually. Restart your Zope server (`bin/zopectl restart` if you have a running Zope started with `bin/zopectl start` or by pressing `Crl^c` followed by `bin/runzope` otherwise) and `Plone Site` (among other things) should be available now in your Zope's management interface (ZMI) from the drop-down add list.

Products

Products for Zope

As just mentioned, most of the time installing products for Zope is as easy as putting its source code into the right directory and restarting the Zope server. Some products, however, may deviate from this pattern, as they might need to

go into a different location (like `ZopeTestCase` needs to go into Zope's `lib/python/Testing` folder) or require some further action to be taken. For instance performance-critical applications like `TextIndexNG` require an extra compilation to be performed as they are not entirely written in Python but also contain components written in C. Other products may require additional software to be installed on your server which may or may not be available for your platform.

Products for Plone

Making the features provided by a product available in your Plone site most often requires one additional step: installing it via the quickinstaller. This is not done automatically on server start-up because you need to have a way to control which products to make available in which site, given that you can have arbitrarily many Plone sites on one Zope server. Accessing the quickinstaller TTW can either be done from within Plone via `configuring Plone -> add products` or from the ZMI by selecting the `portal_quickinstaller.` Check the product(s) you want to install and click the `install` button - that should be it.

But what if things go wrong? This is the subject of the next section.

Troubleshooting

There are various ways and levels at which product installation can fail: Zope itself might have problems. If you've put the code in the wrong place it may not be found. There might be problems with access rights at file system level, required additional software might be missing and so on. A more comprehensive checklist follows below.

But even if products install just fine at the level of Zope, Plone might have its problems when trying to install them using the quickinstaller. The new product might even not be offered for installation at all. So here is what you should check:

4. Is Zope starting? Use `bin/runzope` or check the log file for error messages. You may be missing additional software or may have forgotten to compile something for instance. Or the product's `__init__.py` may just be buggy and cannot be run (in which case I would not have much confidence in the product at all).

5. Is the new product known to Zope? Look in ZMI whether it is listed in `Control_Panel/Products` . If not, check whether you have really put the source files in the right place and whether the user Zope is running under has the access rights necessary.

6. Does the quickinstaller offer the new product for installation? Not all products actually require this but if this should be the case and you cannot see it, this usually hints at a problem with the product's `install` function in its `Extensions/Install.py` module. Try to invoke it *by hand,* that is, go to your portal's root folder in ZMI and add an `external method.` Give it any id and title you like but the module's value needs to be `<product name>.Install` and the method name `install` . Saving this will result in an error should the install function contain invalid Python code for instance. The traceback tells you exactly where it failed and why.

7. Does the install get through? See, whether the install log contains any errors and if so, work your way through from there.

Zeo

Zeo is a tool provided by Zope to allow several clients accessing one ZODB simultaneously. This is useful for high-traffic sites where you want to deploy several machines and therefore several Zope servers in parallel. It is also useful for development set-ups where you want to be able to access the ZODB of a running Zope in a debug session as outlined in the debugging chapter.

Installing Zeo doesn't require any action to be taken as Zeo comes included with Zope. All you need to do is to call `bin/mkzeoinstance` and then have your Zope clients configured to contact ZEO instead of creating their own ZODB. How this is done in detail is best illustrated by way of an example.

My Development Setup

At the time of this writing (April 2005) I have a particular development set-up made to test my products (like `MySite`) in Plone 2.0.5 and the forthcoming Plone 2.1. So I want two sites to run simultaneously using different product versions for some products (here most notably: Plone and Archetypes) but not all (`MySite` should be the same in both sites). I also want both sites to be accessible to live debugging sessions while being manipulated through a browser. In the end I need two ZEO instances - one for each site - and four Zope clients - two for each site - where the Zeos and one Zope client per site are continously running while the additional Zope clients can be used as needed for debugging or running the unit tests.

Now wait you may say: 2 ZEO and 4 Zope servers, that adds up to 6 (in words six) servers you have to set up just to be able to develop and test a product? Isn't that overkill? Isn't that a hell of a lot of work to do before you can get started? Wouldn't you need a high-end computer for this to run on, as there are four servers supposed to be running continously? The short answer is: Don't worry. It's really straight-forward and it only takes a few minutes to set this up. You don't believe me? Well, here we go:

On the hard disk of my desktop, I have an extra partition (called `extra2`) for development purposes in order not to interfere with other stuff while messing around but any folder where you have write access would do for the following. On this partition, I have an `in-place` Zope installation, currently Zope 2.7.5, so `/extra2/Zope-2.7.5` is my `ZOPEHOME` , and `/extra2/Zope-2.7.57/lib/python` is `SOFTWAREHOME` . On this partition I also have a folder called `Sites` and in there I now create subfolders called `Plone205` , `Plone210` and `SharedProducts` . `Plone205` will contain the Zeo instance and the Zope clients implementing a - you guessed it - Plone 2.0.5 site, `Plone210` is the same for Plone 2.1 and `SharedProducts` will contain those products that I'm going to test in both setups.

Before starting the installation there is one thing left to consider: If we want to have six (HTTP) servers running on one machine, we need to bind them to different ports. As some of the Zope servers should also be accessed using a browser, we better have port numbers that we can associate with our sites, so I have chosen port 9205 for the ZEO serving the Plone 2.0.5 site's ZODB, 9210 for the other site, 8205 and 8206 for the Plone 2.0.5 site clients (and 8221 and 8222 for their FTP servers), and 8210 and 8211 for the Plone 2.1.0 clients (FTP: 8231 and 8232). Now we are finally ready to go:

```
/extra2/Zope-2.7.5-final/bin/mkzeoinstance.py /extra2/Sites/Plone205/zeo 9205
/extra2/Zope-2.7.5-final/bin/mkzeoinstance.py /extra2/Sites/Plone210/zeo 9210
/extra2/Zope-2.7.5-final/bin/mkzopeinstance.py -d /extra2/Sites/Plone205/client1 -u admin:secret
/extra2/Zope-2.7.5-final/bin/mkzopeinstance.py -d /extra2/Sites/Plone205/client2 -u admin:secret
/extra2/Zope-2.7.5-final/bin/mkzopeinstance.py -d /extra2/Sites/Plone210/client1 -u admin:secret
/extra2/Zope-2.7.5-final/bin/mkzopeinstance.py -d /extra2/Sites/Plone210/client2 -u admin:secret
```

(to be continued)

Acknowledgement

Thanks to all the individual people and communities who contribute to Plone. Particular thanks for the *big pieces* go to

- Guido van Rossum, the PythonLabs, and the Python community for **Python**
- Jim Fulton, Zope Corporation, and the Zope community for **ZODB** and **Zope**
- Tres Seavers, Yvo Schubbe, and the CMF community for **CMF**
- Alan Runyan, Alex Limi, and the Plone community for **Plone**
- Benjamin Saller, Sidnei da Silva, and the Archetypes community for **Archetypes**

Further special thanks for individual contributions go to

- Shane Hathaway for **DCWorkflow, VerboseSecurity, TCPWatch, Ape** (to mention just a few)
- Martijn Faassen for **Formulator, Silva,** and **Five**
- Dieter Maurer for **DocFinder, ZopeProfiler,** and exceptionel community support
- Andreas Jung for **TextIndexNG, PloneCollectorNG,** and continuous community support
- Stefan Holek for **DocFinderTab, ZopeTestCase**
- Christian (Tiran) Heimes for **PlacelessTranslationService, CMFPhoto(Album),** and **ATContentTypes**
- Jean Jordaan for the chapter on i18n
- Mike Fletcher and Jean Jordaan for extensive and constructive comments on this book

and from my personal hall of fame

- Linus Torvalds and the Linux community for **Linux**
- Tim Berners-Lee, CERN, and the W3C for the **Web**

Please add a comment if you would like to add more people here.

About this Document

This document was written by **Raphael Ritz** (r.ritz@biologie.hu-berlin.de). It is covered under a Creative Commons license and under the GNU Free Documentation License, with one invariant section: this section, `About this document` , must remain unchanged. Otherwise, you can distribute it, make changes to it, etc. as long as you give credit.

If you have any comments, corrections, or changes, please use the comment hooks provided. They are there for a reason. Thanks!

BackTalk

The on-line version of this document is done using BackTalk (http://backtalk.sourceforge.net/) and the pdf generation is done by the ReportLab Toolkit from http://www.reportlab.org.