

PyContainer
Python lightweight container
version 0.4

Rafal Sniezynski

September 5, 2004

Contents

1	Introduction	1
1.1	Lightweight containers and Dependency Injection	1
1.2	PyContainer overview	2
2	Download and installation	2
3	Usage	3
3.1	Basic usage example	3
3.2	Requirements for the components	4
3.3	The instantiation process	4
3.4	Container hierarchies	5
3.5	Lifecycle management	5
3.6	Interceptors	6
3.7	Custom object factories	8
3.8	Configuration file details	9
4	License	10
5	History	10
6	To do	10
7	Contact	11
8	Links	11

1 Introduction

1.1 Lightweight containers and Dependency Injection

Lightweight containers are non-invasive application frameworks that simplify the process of building component-based applications. They allow you to wire all the components of the application together without including any implementation-specific object creation code into the components. With a traditional approach, one instantiates a component in a component by explicitly specifying an implementation, or use Singletons. Lightweight containers use the Inversion of Control (IoC) approach, in which you let the container manage any dependencies between components of an application. This allows you to create highly modular applications with highly reusable components.

Lightweight containers are currently very popular among the Java developers as an alternative to other, heavyweight and complicated solutions. IoC containers, often combined with object-relational mappers, are a core of many modern Java web frameworks such as Spring or Webwork.

Inversion of Control is a general term used in many design patterns. In lightweight containers IoC means that the container is responsible for instantiating the components, not the components themselves. The more specific term is Dependency Injection. There are three common methods of injection:

- Constructor Injection
- Setter Injection
- Interface Injection

For more details I recommend reading this Martin Fowler's article.

1.2 PyContainer overview

PyContainer features are inspired by some of the features of different popular Java containers. PyContainer, although quite simple now, is under an active development. I appreciate any feedback from the users (bug reports, feature requests) that will help me make this product better. Most important PyContainer features:

- Dependency Injection through attribute setting – container binds a component instance to an instance of another component, so it's closest to Setter Injection. If you want to have more control over the injection, you have to define the attribute as a *property* (requires new-style class) or define `__setattr__`. The wiring is configured through an XML file.
- Container hierarchies – you can organize your containers in parent-child hierarchies, with children having full access to the parent's components; parents can also have limited access to children when performing lifecycle method calls.
- Lifecycle management – you can control the lifecycle of all the components by a single method call on the container. The container will take care of the correct order of invoking the method on all the components.
- Custom object factories – by default, PyContainer creates component instances from top-level classes of locally available modules. You can create a custom object factory to make the container instantiate components in a more sophisticated way, e.g. from remote data source.
- Interceptors – they are objects, organized in a stack, that sit between the implementation of the component and the client (another component), and can add some functionality to the core implementation such as logging, security, or transaction management.

2 Download and installation

PyContainer can be downloaded from:

<http://www.iem.pw.edu.pl/~sniezynr/pycontainer/PyContainer-0.4.zip>

Unpack the downloaded file into the temporary directory and type in the command line:

```
python setup.py install
```

If everything goes fine, PyContainer will be installed in the default location for third-party packages. PyContainer requires Python 2.2 or later.

3 Usage

3.1 Basic usage example

The idea is very simple – to inject an object into another object(s) as an attribute to let them cooperate as components, without directly instantiating components into other components or using patterns like Singleton or Service Locator. In a dynamic language like Python, where you can bind an arbitrary attribute to an already instantiated object, it doesn't seem to be such a big advantage and can be easily done programmatically. However, software like PyContainer provides a standardized way of doing this (the wiring is configured through a simple XML file) and also other useful features described in later sections.

The Martin Fowler's article mentioned before (please read it) provides a simple example of a piece of an application that lists the movies of a particular director implemented with a traditional approach and then with the IoC approach. Here's a Python-PyContainer version of this example.

So let's assume that we have this naive `MovieLister` class:

```
class MovieLister (object):
    def __init__ (self):
        self.finder = ColonMovieFinder("movies1.txt")
    def moviesDirectedBy (self, arg):
        allMovies = self.finder.findAll()
        theMovies = []
        for movie in allMovies:
            if movie.getDirector() == arg:
                theMovies.append(movie)
        return theMovies
```

and the `MovieFinder` implementation – `ColonMovieFinder`:

```
class ColonMovieFinder (object):
    def findAll (self):
        # some code here
        return allTheMovies
```

Now what we want to have is the `MovieLister` class independent of the `MovieFinder` implementation – we want `ColonMovieFinder` to be interchangeable with any other implementation of `MovieFinder`. That's why we remove initialization of `ColonMovieFinder` from `MovieLister`:

```
class MovieLister (object):
    def __init__ (self):
        pass
    ...
```

and create the following PyContainer config file (let's call it `components.xml`):

```
<components>
  <component id="MovieLister" class="somemodule.MovieLister">
    <property name="finder" local="MovieFinder" />
  </component>
  <component id="MovieFinder" class="anothermodule.ColonMovieFinder">
    <property name="filename">"movies1.txt"</property>
  </component>
</components>
```

The *class* attribute of the *component* elements contains "classpath" – a Python module path followed by dot followed by a top-level class name. Now the most important part – creation of the container:

```
from pycontainer import PyContainer

movieAppContainer = PyContainer()
movieAppContainer.configXml("components.xml")
lister = movieAppContainer.getInstance("MovieLister")
```

or:

```
movieAppContainer = PyContainer(config="components.xml")
lister = movieAppContainer.getInstance("MovieLister")
```

You can also access the container like a read-only dictionary:

```
lister = movieAppContainer["MovieLister"]
```

What we have in the container now is the `MovieLister` instance with attribute `finder` bound to `ColonMovieFinder` instance, which is the second component in the container. `ColonMovieFinder` object has its `filename` attribute set to "movies1.txt". `lister` is bound to the `MovieLister` component, so you can call `moviesDirectedBy()` method on it. You will probably use `getInstance()` or dictionary access to get the entry point to the application.

The container can contain any number of components wired in an arbitrary way. The details of the configuration file's syntax are described in section 3.8.

3.2 Requirements for the components

Component classes are regular classes, without any additional code required. There is just a couple of limitations:

- component's `__init__` method must not reference any other component from the container
- component's class must be callable without arguments (`__init__` must have only `self` or all other optional)

The first limitation is obvious. The second is to preserve simplicity – you can always pass required parameters in the configuration file's *property* element.

Note: These requirements concern the default object factory, a custom factory (see section 3.7) can be written to somehow avoid them.

3.3 The instantiation process

`PyContainer` performs lazy instantiation. That means that the components are instantiated only when they are required, either if they are referenced as properties of other components that are being instantiated or are accessed through `getInstance()`.

Component types

`PyContainer` allows two component types that differ in number of instances:

- Singleton (default) – the container holds only one instance which is shared among all the components that depend on this component.

- Prototype – every component that depends on this component gets its own instance; also instantiation through `getInstance()` creates new instance every time. Calling lifecycle methods (see section 3.5) causes calling them on every instance of a prototype, one after another (in order of instantiation).

See section 3.8 for details of how to define the component type. Singleton is the usual case and the default type, so usually no additional configuration options are necessary.

Circular dependencies

PyContainer allows circular dependencies between components without any complications, because it performs component wiring recursively. However, circular dependencies are often discouraged as they are considered as a sign of a design problem.

3.4 Container hierarchies

You can create parent-child hierarchies with PyContainer:

```
father = PyContainer(config="parent.xml")
...
son = PyContainer(config="child.xml", parent=father)
```

Child accesses parent whenever it cannot find appropriate component in itself when it is referenced (through `getInstance()` or by another component). Parent container usually doesn't have access to its children (see also section 3.5).

3.5 Lifecycle management

PyContainer supports simple lifecycle management. Calling `start()`, `stop()` or `dispose()` on a container instance will call those methods on all components that have them implemented. If the container is a parent of another container, it can also call those methods on its children under one condition – a child must be initialized with additional parameter `register=True` (or `register=1`):

```
mother = PyContainer(config="parent.xml")
...
daughter = PyContainer(config="child.xml", parent=mother, register=True)
```

The `start()` method is called on current container first, then on its children (to the deepest descendant); `stop()` and `dispose()` are called in reverse order. The order of calls inside the container:

- for `start()` method:
 1. components that don't depend on other local components
 2. components that depend on other local components which are already started
 3. components with circular dependency, specific order of calls is then undetermined
- `stop()` and `dispose()` are called in exactly reverse order to `start()`

Custom lifecycle methods

Methods described above are just examples. They are inspired by similar functionality of `PicoContainer` for Java. In `PyContainer`, `start()`, `stop()` and `dispose()` are only convenience shortcuts for an universal method invoker. Every container has a method called `method` that accepts four arguments, three of them optional. They can be used either as positional or keyword arguments, but the latter is recommended, so I provide the names here:

1. `name` – name of the method to be called on components
2. `args` – list of positional arguments of the method call (optional, default is empty)
3. `kwargs` – dictionary of keyword arguments of the method call (optional, default is empty)
4. `order` – a boolean value (optional, default is `True` or `1`):
 - `True` – the calls will be performed in order analogical to the `start()` method described above
 - `False` – the calls will be performed in order analogical to the `stop()` and `dispose()` methods described above

Example:

```
pyco = PyContainer(...)
...
pyco.method(name="pay", args=["Rafal", 1000000], order=True)
```

It would cause calling `pay("Rafal", 1000000)` on every component that has `pay` method implemented, in order analogical to the `start()` method.

3.6 Interceptors

Interceptors are objects that are placed between the component instance and code (usually of another component) that accesses it through the container. Interceptors can be very useful as they can provide additional functionality with code that is executed before every call of the component's method. Interceptors are organized in a stack, which defines the order of processing.

Interceptors are also components and must be defined in the components configuration file. To use interceptors, the component definition in the configuration file must contain one or more *interceptor-ref* elements.

```
<components>
  <component id="component1" class="mycomponents.ComponentClass">
    <property ... />
    ...
    <interceptor-ref name="interceptor1" />
    <interceptor-ref name="interceptor2" />
    ...
  </component>
  <component id="interceptor1" class="myinterceptors.Interceptor1">
    ...
  </component>
  <component id="interceptor2" class="myinterceptors.Interceptor2">
    ...
  </component>
</components>
```

The order of *interceptor-ref* elements defines the order of interceptors in the interceptor stack. If a component definition contains at least one *interceptor-ref* element, the component is instantiated differently. It is wrapped in a special object that puts the interceptor stack between the client code and the actual implementation:

- Access to the component's non-callable attributes is fully transparent, the wrapper object returns them directly from the component.
- Access to the component's callable attributes (methods) is performed through the interceptors organized in a stack.

Interceptor components should subclass the `Interceptor` class from module `pycontainer.interceptors` and override the `intercept()` method:

```
def intercept (self, invocation):  
    ...
```

This method should accept one argument (`invocation`) – an object that has two attributes used in further processing:

- `invoke` – a method that should be called without arguments to pass the control to the next interceptor on the stack, or, eventually, to execute the method that was called by the client code
- `call` – a tuple of four elements:
 - 0 – reference to the actual component instance
 - 1 – name of method that was called (a string)
 - 2 – list of positional arguments of the call, if any
 - 3 – dictionary of keyword arguments of the call, if any

A typical `intercept()` method should be built like this:

```
def intercept (self, invocation):  
    # code executed before the method  
    result = invocation.invoke()  
    # code executed after the method  
    return result
```

In order to return the return value of the real implementation of the method that was called on the component, every interceptor should return the result of `invocation.invoke()` (of course, this won't be the case if an interceptor performs any changes of the state of the elements of the `call` tuple mentioned before, which will affect next interceptors so that the proper method won't be called). You can also short-circuit the method call by not calling `invocation.invoke()`, return something different, alter the real result value, etc. Interceptors seem to be really powerful concept.

With a component having two interceptors, as in the configuration above, and a typical `intercept()` method, the code will be executed (when some method is called on the component) in the following order:

1. code of *interceptor1* placed before the `invoke()` call
2. code of *interceptor2* placed before the `invoke()` call
3. actual body of the method of *component1*

4. code of *interceptor2* placed after the `invoke()` call
5. code of *interceptor1* placed after the `invoke()` call

Important note: The wrapper object overshadows number of attributes of the component if they have the same names. Most of those attributes are "private" (in a Python sense), but two are not: `invoke` and `call`. If the component has any attributes of those names, they won't be available. Also some special methods are used (e.g. `__getattr__`).

3.7 Custom object factories

By default, PyContainer creates component instances from top-level classes of locally available modules. However, it is often convenient to have more control over the instantiation, for example to retrieve the objects from remote data source. To use custom factories, second configuration file is needed, let's call it `factories.xml`:

```
<factories>
  <factory id="default" class="pycontainer.factories.LocalFactory" />
  <factory id="factory1" class="myfactories.CustomFactory">
    <property name="anAttribute">"a string"</property>
    ...
  </factory>
  <factory id="factory2" class="myfactories.CustomFactory">
    ...
  </factory>
  <factory id="factory3" class="myfactories.AnotherCustomFactory">
    ...
  </factory>
</factories>
```

Each factory is identified by an *id* attribute. The factories are instantiated from locally available modules defined by *class* attribute, just as components. Similarly to components, they may have properties injected (only of built-in types). Hence the presence of different factories of the same class in the example above – they may have different sets of properties. In the example above the *default* factory is specified – it is used to instantiate components that don't have their factory specified. It is not necessary to define this factory if you want the regular components to be instantiated in a normal way.

A factory class should be a subclass of the `Factory` class from the `pycontainer.factories` module and override the `getInstance()` method:

```
def getInstance (self, classpath):
    ...
    return aNewInstance
```

The `classpath` argument is the value of *class* attribute from component definition. The `getInstance()` method is supposed to always return a new instance of an object. Of course, the factory object has the attributes injected as defined in the factory configuration file.

To use factories, also the component configuration file has to be modified:

```
<components factories="factories.xml">
  <component id="component1" class="components.Component1" factory="default">
    ...
  </component>
  <component id="component2" class="components.Component2" factory="factory1">
```

```

    ...
</component>
<component id="component3" class="components.Component3" factory="factory2">
    ...
</component>
<component id="component4" class="components.Component4">
    ...
</component>
</components>

```

The *factory* attribute of the *component* element has to match any of the *id* attributes in the factories configuration file. For default factory, the *default* value can be used, but may as well be omitted.

Note: In future versions the format of the configuration file for components and factories may be somehow unified.

Future versions may also provide a standard library of useful factories.

3.8 Configuration file details

Components configuration file

The root of an XML configuration file is the *components* element. It has an (optional) attribute *factories*, which should be a name (or relative path) of the factories configuration file.

The *components* element can contain any number of *component* elements. Every *component* element has two mandatory attributes and two optional:

- *id* – identifies the component in the container, allows access without specifying the actual class (it can be thought as "interface")
- *class* – it's really a path to class: dot-separated module path followed by dot followed by class name. It's done like this for simplicity, and for the default object factory it implies at least two limitations:
 1. You can't use nested classes as component classes
 2. You can't directly use module as a component. It's sometimes convenient, and there are modules in standard library that work like that. What you can do is wrap the module into something callable, e.g.:

```

import xml.dom.minidom
def mydom():
    return xml.dom.minidom

```

You can create your own object factory to avoid these limitations.

- *type* (optional) – "singleton" or "prototype" (default is singleton)
- *factory* (optional) – factory to be used for instantiation; the value must match any of the *id* attributes of the *factory* elements in the factories configuration file

A *component* element can optionally contain any number of *property* elements. Every *property* element has a *name* attribute, which is mapped to an attribute of the component during instantiation. If the *property* element contains *local* attribute, its value is treated as a reference to other component in the container (it must match the *id* attribute of one of the other components defined in config file). If it doesn't, the text between the start and end tag of the *property* element is treated as a value to be injected into the component. PyContainer uses the evil `eval` built-in function to do that (yes, I know). Example:

```
<property name="products">["bread", "butter", "milk"]</property>
<property name="address">"A string"</property>
<property name="number">6</property>
```

Factories configuration file

The format of the factories configuration file is similar to the components' file. The root is the *factories* element, which can contain any number of *factory* elements. Each *factory* element has to have two attributes, *id* and *class*, with the same meaning as for component configuration. A *factory* element can contain any number of *property* elements (like for components, but only simple ones, with the value evaluated from between the start and end tag).

Note: In future versions the format of the configuration file for components and factories may be somehow unified.

4 License

Copyright © 2004, Rafal Sniezynski

PyContainer is distributed under original Python license. Included in the distribution is the pxdom 1.1 module. PyContainer uses it when it has trouble importing the Expat module (happens on older Python distributions). Pxdom is:

Copyright © 2004, Andrew Clover. All rights reserved.

License details for PyContainer and pxdom are in the `license.txt` file in the distribution. PyContainer also uses slightly modified version of David Mertz's great `gnosis.xml.objectify` package (public domain).

5 History

- 0.4 (2004.09.05)
 - added support for interceptors
 - added support for custom object factories
 - added support for arbitrary lifecycle methods
- 0.3 (2004.08.25)
 - added support for prototype components
 - lifecycle management is now performed in a much more reasonable way
- 0.2 (July 2004) – initial public release

6 To do

- provide a standard library of object factories
- investigate the thread safety (or lack thereof)
- add more exception handling
- bugfixes, etc.
- probably a lot more, any suggestions are welcome

7 Contact

My email address: thirdeye at interia pl

Any feedback is welcome.

Pycontainer homepage is <http://www.pycontainer.glt.pl>

8 Links

PicoContainer

HiveMind

Spring Framework

Webwork 2

[Sorry for my English]