Building Extensible Desktop Applications with Zope 3

Nathan R. Yergler
Software Engineer
Creative Commons

ccPublisher

ccPublisher 2 Goals

- Wanted to make customization easy
- Allow for extensibility
- Leverage common components
  - Reduced maintenance burden
  - We shouldn’t “own” anything that’s not strategic

extensible (n):

“An architectural property of a program that allows its capabilities to expand.”

http://web.mit.edu/oki/learn/gloss.html

component (n):

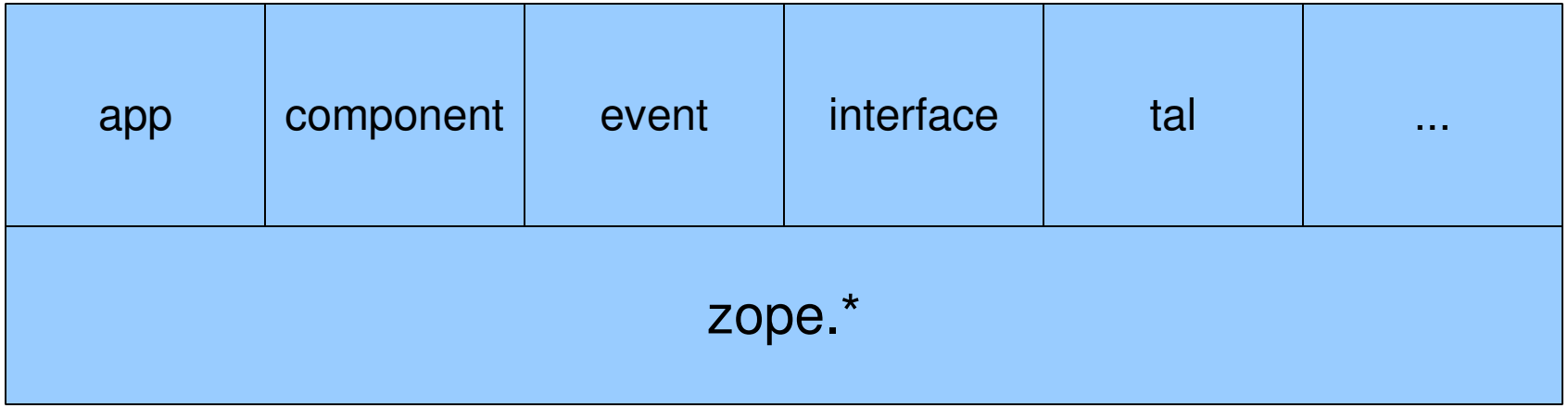
“a system element offering a predefined service and able to communicate with other components”

http://en.wikipedia.org/wiki/Software_component

Zope 3

- Zope is an open source web application server
- Zope 3 includes a component / event model
  - Adapters
  - Utilities
  - Events

Zope 3 Overview



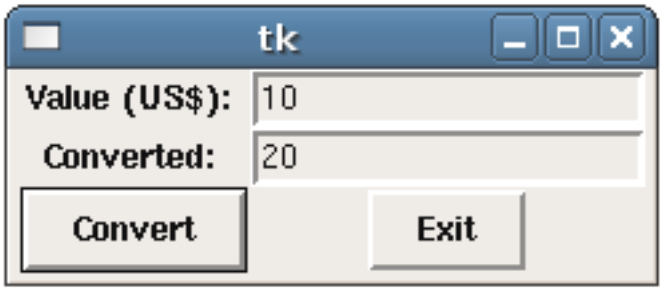
A Simple Example

```
class CurrencyConverterApp:

    def __init__(self, root):
        ...


    def convert(self):
        # clear the output widget
        self.txtOutput.delete(0,END)

        # set the output widget to input * 2
        self.txtOutput.insert(0,
             float(self.txtInput.get()) * 2)
```



Steps to Extensibility

- Separate interface and converter functionality
- Decouple interface and converter using component lookup
- Allow for multiple providers of functionality
- Package core functionality as an extension

Separating Responsibilities

- Write an interface declaration
- Write a component which provides the interface
- Use the component in the user interface

IConverter

```
class IConverter(zope.interface.Interface):


    def currency_name():
        """Return the name of the target currency."""


    def currency_symbol():
        """Return the symbol for the target currency."""


    def convert(usd_value):
        """Convert US$ value to the target currency."""
```

The Converter Component

```
class RealConverter:
    zope.interface.implements(interfaces.IConverter)

    def currency_name(self):
        """Return the name of the target currency."""
        return "Real"

    ...
    def convert(self, usd_value):
        """Convert US$ value to the target currency."""
        return usd_value / 2.247
```

Using the Component

```
def convert(self):
    # clear the output widget
    self.txtOutput.delete(0,END)

    # set the output widget to input * 2
    self.txtOutput.insert(0,
          components.RealConverter().convert(
              float(self.txtInput.get())
              )
          )
```

Decoupling the Pieces

- Register the component as a utility
- Use Zope component model to lookup our converter

Utilities

- Components which provide an interface
- Looked up by interface and optional name
- Provide global functionality
- Applications can be agnostic about how things happen

Registering the Component

```
from zope import component


# register the converter as a utility
converter = RealConverter()
component.provideUtility(converter, IConverter, 'Real')
```

Using Component Lookup

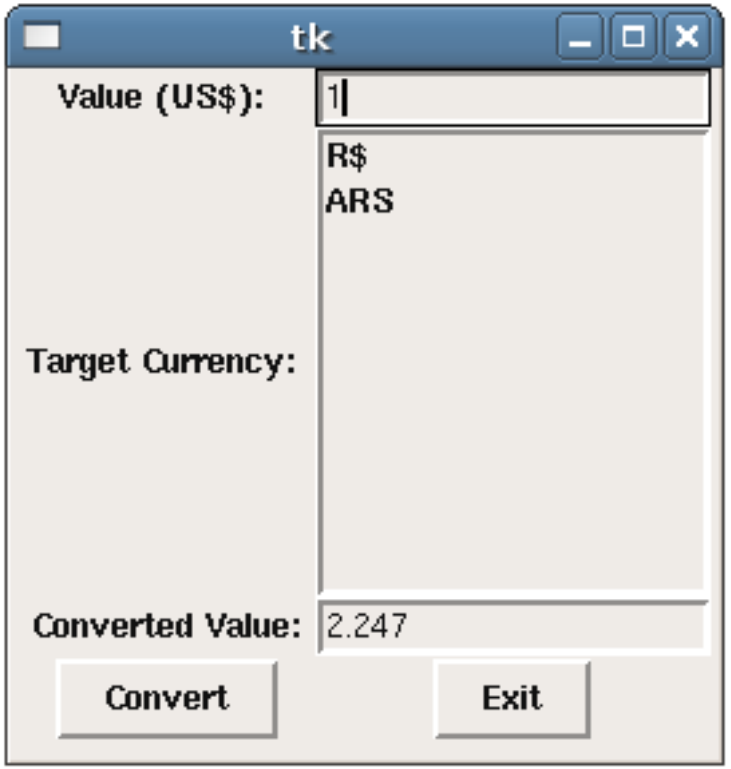
```
def convert(self):
    # lookup our converter
    converter =
      zope.component.getUtility(
          interfaces.IConverter, 'Real')

    # clear the output widget
    self.txtOutput.delete(0,END)

    # set the output widget to input * 2
    self.txtOutput.insert(0,
       converter.convert(float(self.txtInput.get())))
```

Multiple Providers

- Using multiple named utilities is one option
- Requires us to know the names of all converters
- Instead we use adapters
  - Recall utilities provide are unique (Interface, Name)
  - There can be multiple subscription adapters



Adapters

- Components computed from other components
- Easily convert from one type of object to another
- “Normal” Zope 3 Adaptation is one-to-one
  - one object in, one adapted object returned
- Subscription Adapters are different:
  - Return all adapters from A to B

Refactoring the Interfaces

```
class IUSD(zope.interface.Interface):
    def get_value():
        """Return the stored value."""

class ICurrency(zope.interface.Interface):
    def currency_name():
        """Return the name of the target currency."""

    def currency_symbol():
        """Return the symbol for the target currency."""

class IConverter(zope.interface.Interface):
    def convert():
        """Convert US$ value to the target currency."""
```

Implementing the Component

```
class RealConverter(object):
    component.adapts(interfaces.IUSD)

    interface.implements(interfaces.IConverter,
                         interfaces.ICurrency)

    def __init__(self, usd):
        self.usd = usd

    # implementation for IConverter
    def convert(self):
        return self.usd.get_value() * 2

    # implementation for ICurrency
    ...
```

Registering the Component

```
# Register as an ICurrency implementation for the list
component.provideSubscriptionAdapter(RealConverter,
     provides=interfaces.ICurrency)

# Register as an Adapter for the specific currency
component.provideAdapter(RealConverter,
                         provides=interfaces.IConverter,
                         name='R$')
```

User Interface: Target List

```
def __init__(self, root):

    ...

    # get a list of target currencies
    u = components.USDCurrency(None)
    currencies = [c.currency_symbol() for c in
                  component.subscribers([u],
                                        interfaces.ICurrency)]

    self.lstTargets = Listbox(frame)
    ...
```

User Interface: Converting

```
def convert (self):
    # create an object to hold the US$
    usd = components.USDCurrency(
            float(self.txtInput.get())
          )

    # get the target currency symbol
    target = self.lstTargets.get(
                self.lstTargets.curselection()[0])

    # look up the converter by target name
    converter = component.queryAdapter(usd,
                        interfaces.IConverter, target)

    self.txtOutput.insert(END, converter.convert())
```

Extensions

- Depend on application policy...
  - Where do they live?
  - How are they registered?
- ...and application code...
  - What code is responsible for loading?
  - How are they allowed to interact with the app?

Loading Extensions

- ccPublisher uses ZCML slugs
  - ZCML is an XML-based configuration language
  - ccPublisher looks for small ZCML files in the `extensions` directory and loads them
- setuptools provides Entry Points which you can iterate over

Using entry points for loading

- Move each “extension” into separate modules
- Provide a register() function
- Create a simple setup.py
- Generate a Python Egg for each
- Dynamically load extensions at run time

Extension Module: real.py

```
class RealConverter(object):
    component.adapts(interfaces.IUSD)

    interface.implements(interfaces.IConverter,
                         interfaces.ICurrency)


...


def register():

    component.provideSubscriptionAdapter(RealConverter,
           provides=interfaces.ICurrency)

    component.provideAdapter(RealConverter,
           provides=interfaces.IConverter, name='R$')
```

setup_real.py

```
from setuptools import setup

setup(
    name='real',
    py_modules=['real'],

    entry_points = {'currency.register':
                      ['real = real:register'],
                    }
    )
```

```
$ python setup_real.py bdist_egg
```

Loading the Extensions

```
# load pkg_resources, which is part of setuptools
import pkg_resources

...

def loadExtensions():
    """Load the extensions using the
    currency.register entry point."""

    # iterate over available implementations
    for e in pkg_resources.iter_entry_points(
                                        'currency.register'):

        # load the entry point
        register = e.load()

        # call the registration function
        register()
```

Component-Based Results

- Debugging is different
  - Cohesive components = shallower bugs
  - “Over-reaching” components = break out pdb
- Our “application” is less than 10% of the total code – in other words, 90% of code can be shared between customized applications
- Developers can easily extend the system
  - New storage providers
  - Extensions (i.e., Bit Torrent, blog pinging, etc)

Thanks

Questions?

http://wiki.creativecommons.org/OSCON2006