An Introduction to the Zope 3 Component Architecture

Brandon Craig Rhodes
Georgia Tech

NOLA Plone Symposium 2008
This talk, on the whole, is divided into four parts
1. Enhancing objects
1. Enhancing objects
2. Adapting objects
1. Enhancing objects
2. Adapting objects
3. Component framework
1. Enhancing objects
2. Adapting objects
3. Component framework
4. Adapting for the web
Let's go!
Many programming languages use static typing
float half(int n)
{
    return n / 2.0;
}
float half(int n)
{
    return n / 2.0;
}
Python typing is dynamic
def half(n):
    return n / 2.0
You don't worry about whether an object is of the right type
You simply try using it
“Duck Typing”

(Alex Martelli)
“Duck Typing”

Walks like a duck?
Quacks like a duck?
It's a duck!
def half(n):
    return n / 2.0
def half(n):
    return n / 2.0

(Is $n$ willing to be divided by two? Then it's number-ish enough for us!)
Now, imagine...
Imagine a wonderful duck-processing library to which you want to pass an object
But...

The object you want to pass isn't a duck?
What if it *doesn't* already quack?
What if it bears
not the least resemblance
to a duck!?
Example!
You have a “Message” object from the Python “email” module
from email import message_from_file

e = message_from_file(open('msg.txt'))

print e
<email.message.Message instance at ...>

e.is_multipart()
True

for part in e.get_payload():
...
  print part.get_content_type()

  text/plain
text/html
Messages can be recursive
Imagine that we are writing a GUI email client
And we want to show the parts in a TreeWidget.
The Tree widget needs:

method `name()` - returns name under which this tree node should be displayed

method `children()` - returns list of child nodes in the tree

method `__len__()` - returns number of child nodes beneath this one
How can we add these behaviors to our Message?
(How can we make an object which is not a duck behave like a duck?)
1. Subclassing
Create a “TreeMessage” class that inherits from the “Message” class...
class TreeMessage(Message):

def name(self):
    return self.get_content_type()

def children(self):
    if not self.is_multipart(): return []
    return [ TreeMessage(part) for part in self.get_payload() ]

def __len__(self):
    return len(self.children())
What will the test suite look like?
Remember:

“Untested code is broken code”

— Philipp von Weitershausen, Martin Aspeli
Your test suite must instantiate a “TreeMessage” and verify its tree-like behavior...
Did you read Arts & Letters Daily today?

```python
txt = "From: persephone@gmail.com
To: brandon@rhodesmill.org
Subject: what an article!

m = message_from_string(txt, TreeMessage)
assert m.name() == 'text/plain'
assert m.children == []
assert m.__len__() == 0
```
We were lucky!
Our test can cheaply instantiate Messages.
txt = "" 'From: persephone@gmail.com
To: brandon@rhodesmill.org
Subject: what an article!

Did you read Arts & Letters Daily today?
"""

m = message_from_string(txt, TreeMessage)
assert m.name() == 'text/plain'
assert m.children == []
assert m.__len__() == 0
What if we were subclassing an LDAP connector?!

We'd need an LDAP server just to run unit tests!
We were lucky (#2)!
The “message_from_string()” method let us specify an alternate factory!
txt = "" 'From: persephone@gmail.com
To: brandon@rhodesmill.org
Subject: what an article!

Did you read Arts & & Letters Daily today?
"""

m = message_from_string(txt, TreeMessage)
assert m.name() == 'text/plain'
assert m.children == []
assert m.__len__() == 0
Final note: we have just broken the “Message” class's behavior!
Python library manual

7.1.1.1 defines “Message”:

```
__len__():
    Return the total number of headers, including duplicates.
```
t = "" "From: persephone@gmail.com
To: brandon@rhodesmill.org
Subject: what an article!

Did you read Arts & Letters Daily today?

m = message_from_file(t, Message)
>>> print len(m)
3

m = message_from_file(t, TreeMessage)
>>> print len(m)
0
So how does subclassing score?
No harm to base class
No harm to base class

Cannot test in isolation
No harm to base class

Cannot test in isolation

Need control of factory
No harm to base class
- Cannot test in isolation
- Need control of factory
- Breaks if names collide
No harm to base class

- Cannot test in isolation
- Need control of factory
- Breaks if names collide

Subclassing: D
2. Using a mixin
Create a “TreeMessage” class that inherits from both “Message” and a “Mixin”...
class `Mixin`(object):
    def `name`(self):
        return self.get_content_type()
    def `children`(self):
        if not self.is_multipart(): return []
        return [ `TreeMessage`(part) for part in self.get_payload() ]
    def `__len__`(self):
        return len(self.children())
  
class `TreeMessage`(Message, Mixin): pass
Your test suite can then inherit from a mocked-up “message”...
class FakeMessage(Mixin):
    def get_content_type(self):
        return 'text/plain'
    def is_multipart(self):
        return False
    def get_payload(self):
        return ''

m = FakeMessage()

assert m.name() == 'text/plain'
assert m.children() == []
assert m.__len__() == 0
How does a mixin rate?
No harm to base class
工序无害，可以单独测试程序混合
No harm to base class
Can test mixin by itself
Need control of factory
✓ No harm to base class
✓ Can test mixin by itself
● Need control of factory
● Breaks if names collide
No harm to base class
Can test mixin by itself
Need control of factory
Breaks if names collide

Mixin: C
3. Monkey patching
To “monkey patch” a class, you add or change its methods dynamically...
def name(self):
    return self.get_content_type()

def children(self):
    if not self.is_multipart(): return []
    return [Message(part) for part in self.get_payload()]

def __len__(self):
    return len(self.children())

Message.name = name
Message.children = children
Message.__len__ = __len__
Is this desirable?
Don't care about factory
Don't care about factory

Changes class itself
Don't care about factory
Changes class itself
Broken by collisions
Don't care about factory

Changes class itself

Broken by collisions

Patches fight each other
Don't care about factory

Changes class itself

Broken by collisions

Patches fight each other

Ruby people do this
Don't care about factory

- Changes class itself
- Broken by collisions
- Patches fight each other
- Ruby people do this

*Monkey patching: F*
4. Adapter
Touted in the Gang of Four book (1994)
Idea: provide “Tree” functions through an entirely separate class

Message
- get_content_type()
- is_multipart()
- get_payload()

MessageTreeAdapter
- name()
- children()
- __len__()
class MessageTreeAdapter(object):
    def __init__(self, message):
        self.m = message
    def name(self):
        return self.m.get_content_type()
    def children(self):
        if not self.m.is_multipart(): return []
        return [TreeMessageAdapter(part) for part in self.m.get_payload()]
    def __len__(self):
        return len(self.children())
How does wrapping look in your code?
IMAP library (or whatever)

Message object

tw = TreeWidget(MessageTreeAdapter(msg))

Adapted object
Test suite can try adapting a mock-up object
class FakeMessage(object):
    def get_content_type(self):
        return 'text/plain'
    def is_multipart(self): return True
    def get_payload(self): return []

m = MessageTreeAdapter(FakeMessage())
assert m.name() == 'text/plain'
assert m.children == []
assert m.__len__() == 0
How does the Adapter design pattern stack up?
No harm to base class
No harm to base class
Can test with mock-up
- No harm to base class
- Can test with mock-up
- Don't need factories
✓ No harm to base class
✓ Can test with mock-up
✓ Don't need factories
✓ No collision worries
No harm to base class
Can test with mock-up
Don't need factories
No collision worries
Wrapping is annoying
No harm to base class
Can test with mock-up
Don't need factories
No collision worries

Wrapping is annoying

Adapter: B
Q: Why call wrapping “annoying”? 
The example makes it look so easy!
IMAP library (or whatever)

Message object

tw = TreeWidget(TreeMessageMessageAdapter(msg))

Adapted object
A: The example looks easy because it only does adaptation once!
But in a real application, it happens all through your code...
Adapters

A
B
C

Your application

objects

msg
C(msg)
A(famtree)
B(msg)

3rd party Producers

Genealogy
IMAP
DB
email

3rd party Consumers

Web
Widget
GUI

Your application

objects

msg
C(msg)
A(famtree)
B(msg)

3rd party Producers

Genealogy
IMAP
DB
email

3rd party Consumers

Web
Widget
GUI
How can you avoid repeating yourself, and scattering information about adapters and consumers everywhere?
IMAP library (or whatever)

Message object

tw = TreeWidget(TreeMessageMessageAdapter(msg))

Adapted object
tw = TreeWidget(TreeMessageMessageAdapter(msg))
tw = TreeWidget(TreeMessageMessageAdapter(msg))

The key is seeing that this code conflates two issues!
tw = TreeWidget(TreeMessageMessageAdapter(msg))

Why does this line work?
It works because a TreeWidget needs what our adapter provides.
tw = TreeWidget(TreeMessageMessageAdapter(msg))

But if we call the adapter then the need = want is hidden inside of our head!
We need to define what the TreeWidget needs that our adapter provides!
An interface is how we specify a set of behaviors.
An *interface* is how we specify a set of behaviors.
For the moment, forget Zope-the-web-framework
Instead, look at Zope the component framework:

```python
zope.interface
zope.component
```
With three simple steps, Zope will rid your code of manual adaptation.
1. Define an interface
2. Register our adapter
3. Request adaptation
from zope.interface import Interface

class ITree(Interface):
    def name():
        """Return this tree node's name."""
    def children():
        """Return this node's children."""
    def __len__():
        """Return how many children."""
from zope.component import provideAdapter

provideAdapter(MessageTreeAdapter,
adapts=Message,
provides=ITree)
from your_interfaces import ITree

class TreeWidget(...):
    def __init__(self, arg):
        tree = ITree(arg)

    ...

Request
from your_interfaces import ITree
class TreeWidget(...):
    def __init__(self, arg):
        tree = ITree(arg)
        ...

Zope will:
1. Recognize need
2. Find the registered adapter
3. Wrap and return the Message
from your_interfaces import ITree

class TreeWidget(...):
    def __init__(self, arg):
        tree = ITree(arg)

...  (Look! Zope is Pythonic!)

i = int(32.1)
l = list('abc')
f = float(1024)
And that's it!
And that's it!

Define an interface
Register our adapter
Request adaptation
No harm to base class
Can test with mock-up
Don't need factories
No collision worries
Adapters now dynamic!

Registered adapter: A
What adapters provide

Genealogy

DB

email

IMAP

Web

Widget

GUI

What consumers need
The finale

Adapting for the Web
dum ... dum ... dum ...

DAH DUM!
Grok
Web framework built atop Zope 3 component architecture
Grok makes
Zope 3 simple to use
(and to present!)
Imagine a **Person** class
The **Person** class was written by someone else
The **Person** class is full of business logic, and stores instances in a database.
We want to browse Person objects on the Web
What might the Web need the object to do?
This page presents the basic data we have regarding Joe.

1. What's at a URL
   http://person_app/Joe

2. HTML document
   <HTML>
   <HEAD>
   <TITLE>Person JOE</TITLE>
   </HEAD>
   <BODY>
   This page presents the basic data we have regarding Joe.
   ...
   </BODY>
   </HTML>

3. What is its URL
   http://person_app/Joe
What's at this URL?
What's at this URL?

http://person_app/Joe

# how Zope processes this URL:

```python
r = root
j = ITraverser(r).traverse('person_app')
k = ITraverser(j).traverse('Joe')
return k
```
# what we write:
class PersonTraverser(grok.Traverser):
    grok.context(PersonApp)
    def traverse(self, name):
        if person_exists(name):
            return get_person(name)
        return None

What's at this URL?

http://person_app/Joe
What's at this URL?

http://person_app/Joe

# what we write:

class PersonTraverser(grok.Traverser):
    grok.context(PersonApp)
    def traverse(self, name):
        if person_exists(name):
            return get_person(name)
        return None
# what we write:

class PersonTraverser(grok.Traverser):
    grok.context(PersonApp)
    def traverse(self, name):
        if person_exists(name):
            return get_person(name)
How does a Person render?
How does a Person render?

app.py

class PersonIndex(grok.View):
    grok.context(Person)
    grok.name('index')

app_templates/personindex.pt

<html><head><title>All about
    <tal tal:replace="context/name" />
</title></head>...
3.
What is a person's URL?
What is a person's URL?

class PersonURL(grok.MultiAdapter):
    grok.adapts(Person, IHTTPRequest)
    grok.implements(IAbsoluteURL)
    def __init__(self, person, req):
        self.person, self.req = person, req
    def __call__(self):
        base = grok.url(grok.getSite())
        return base + '/\' + self.person.name
5 + 3 + 8 = 16 lines
This page presents the basic data we have regarding Joe.

1. What's at a URL

http://person_app/Joe

2. HTML Document

<Person>
    <title>Person JOE</title>
    <body>
        This page presents the basic data we have regarding Joe. ...
    </body>
</Person>

3. What is its URL

PersonURL

http://person_app/Joe
Other Zope adapter uses
Other Zope adapter uses

Indexing — Index, Query, Search, ...
Data schemas — Schema, Vocabulary, DublinCore ...
Form generation — AddForm, EditForm, ...
Security — SecurityPolicy, Proxy, Checker, ...
Authentication — Login, Logout, Allow, Require, ...
Copy and paste — ObjectMover, ObjectCopier, ...
I18n — TranslationDomain, Translator, ...
Appearance — Skins, macros, viewlets, ...

Much, much more!
Adapters can be local!

Global adapters

Local adapters
five.grok
five.grok

Lennart Regebro
Thank you!

http://zope.org/Products/Zope3
http://grok.zope.org/
http://rhodesmill.org/brandon/adapters
http://regebro.wordpress.com/
zope-dev@zope.org mailing list
grok-dev@zope.org mailing list

Web Component Development with Zope 3 by PvW