Using Grok to Walk Like a Duck

Brandon Craig Rhodes
Georgia Tech

for PyCon 2008
in the Windy City
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 the Message displayed in a TreeWidget

awesome article!!! (persephone@gmail.com)

- multipart/mixed
  - text/plain
- multipart/alternative
  - text/plain
  - text/html
- image/jpeg

From: persephone@gmail.com
To: brandon@rhodesmill.org
Subject: awesome article!!!

Hey Brandon - I haven't heard from you for a while, but did you catch this awesome article that they linked to in "Arts & Letters Daily"? It's about an English teacher that sets up a "poetry stand" up in New Jersey somewhere, and it's just really inspiring and incredible, you've got to stop whatever you're doing...
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...
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
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 library?! 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?
" ""
Final note: we have just broken the “Message” class's behavior!
Python library manual
7.1.1 defines “Message”:

```
__len__():
    Return the total number of headers, including duplicates.
```
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 TreeMessageMessage(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
No harm to base class
Can test mixin by itself
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 need factory
Don't need factory

Changes class itself
Don't need factory

- Changes class itself
- Broken by collisions
Don't need factory
- Changes class itself
- Broken by collisions
- Patches fight each other
Don't need factory
• Changes class itself
• Broken by collisions
• Patches fight each other
• Ruby people do this
Don't need 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

call

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(MessageTreeTreeAdapter(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...
Your application

Adapters

A
B
C

3rd party Producers

IMAP
Genealogy
DB
email

objects

A(famtree)

msg

B(msg)

C(msg)

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!
Why does this line work?

tw = TreeWidget(TreeMessageMessageAdapter(msg))
tw = TreeWidget(TreeMessageMessageAdapter(msg))

It works because a TreeWidget needs what our adapter provides.
tw = TreeWidget(TreeMessageMessageAdapter(msg))

But this line of code keeps that information *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:

```
zoope.interface
zoope.component
```
With three simple steps, Zope will rid your code of manual adaptation
Define an interface

Register our adapter

Request adaptation
Define

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
Request

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
Zope framework is cool

Registered adapter: A
What adapters provide

What consumers need

A(famtree)

B(msg)

C(msg)

IMAP

Genealogy

DB

email

Web

Widget

GUI
To conclude:

3 practical tips
3 closing statements
Practical tip #1: you can provide a default argument for adaptation
from your_interfaces import ITree

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


from your_interfaces import ITree

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

Q: What if Zope doesn't know how to adapt the object?
from your_interfaces import ITree

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

Q: What if Zope doesn't know how to adapt the object?
A: It throws an exception!
What if that annoys you?

What if some objects “just work” natively?
Right way out
and an
Easy way out
Right way:
Mark up other classes that already provide interface

```python
from zope.interface import alsoProvides
alsoProvides(GenealogyTree, ITree)
alsoProvides(FileSystemTree, ITree)
```
from your_interfaces import ITree
class TreeWidget(...):
    def __init__(self, arg):
        tree = ITree(arg)
        ...

i = int(3)
f = float(3.1415)

(Look! Zope is Pythonic!)
Fast way:
Provide a default for when there is no adapter
from your_interfaces import ITree

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

from your_interfaces import ITree

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

...
from your_interfaces import ITree

class TreeWidget(...):
    def __init__(self, arg):
        tree = ITree(arg, arg)
        ...  
        item = mydict.get(32, None)
        attr = getattr(obj, 'name', '')

(Look! Zope is Pythonic!)
Practical tip #2: your adapter can announce what it adapts
class MessageTreeAdapter(object):
    def __init__(self, message):
        ...

from zope.component import provideAdapter
provideAdapter(MessageTreeAdapter, adapts=Message, provides=ITree)
Define / Register

class MessageTreeAdapter(object):
    adapts(Message)
    implements(ITree)
    def __init__(self, message):
        ...

from zope.component import provideAdapter
provideAdapter(MessageTreeAdapter)
Practical tip #3: There are actually three ways to register
class MessageTreeAdapter(object):
    adapts(Message)
    implements(ITree)
    def __init__(self, message):
        ...

from zope.component import provideAdapter
provideAdapter(MessageTreeAdapter)
b. Use ZCML

<configure
    xmlns="http://namespaces.zope.org/zope"
    i18n_domain="zope"
  >

    <adapter factory="MessageTreeAdapter"
            for="Message"
            provides="ITree"
    />

</configure>
c. Use Grok!

```python
class MessageTreeAdapter(grok.Adapter):
    adapts(Message)
    provides(ITree)
    def __init__(self, message):
        ...
```
Closing Statement #1:

Grok is cool
Grok lets you define View adapters that prep your objects for the Web
Grok lets you create *space suits* so your objects can survive the web.
Closing Statement #2:

Dynamic adaptation might feel like a type declaration, but it's not!
from your_interfaces import ITree

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

Isn't this an evil old-fashioned type declaration, like in C?
A: No, it's not!

It specifies a behavior, not a type; it's dynamic; it's optional.
Think of adapters as “two-storey” attributes and methods!
In the old days attributes were just names:

def gather_info(arg):
    title = arg.title
    content = arg.content
    encoding = arg.encoding
Now we ask for an adapter.attribute:

def gather_info(arg):
    author = IAnnotations(arg).author
    content = ITextContent(arg).content
    encoding = IEncoded(arg).encoding
Closing Statement #3:

This is the future!
Sprint with me!

Grok for the masses!
http://rhodesmill.org/brandon/adapters
http://rhodesmill.org/brandon/adapters

Thank you!