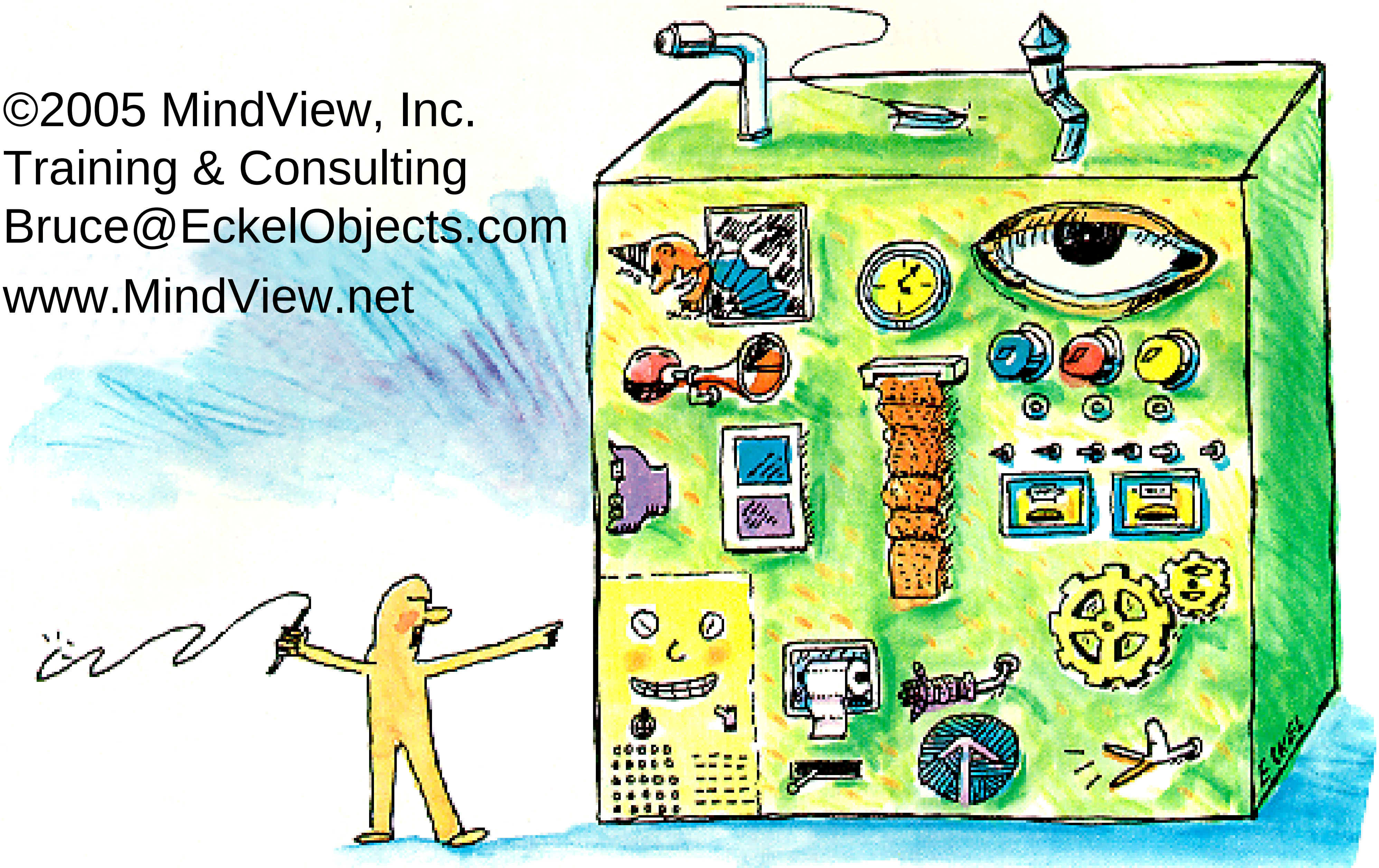
OO Design in Python

Training & Consulting
Bruce@EckelObjects.com
www.MindView.net



OO Design is well understood?

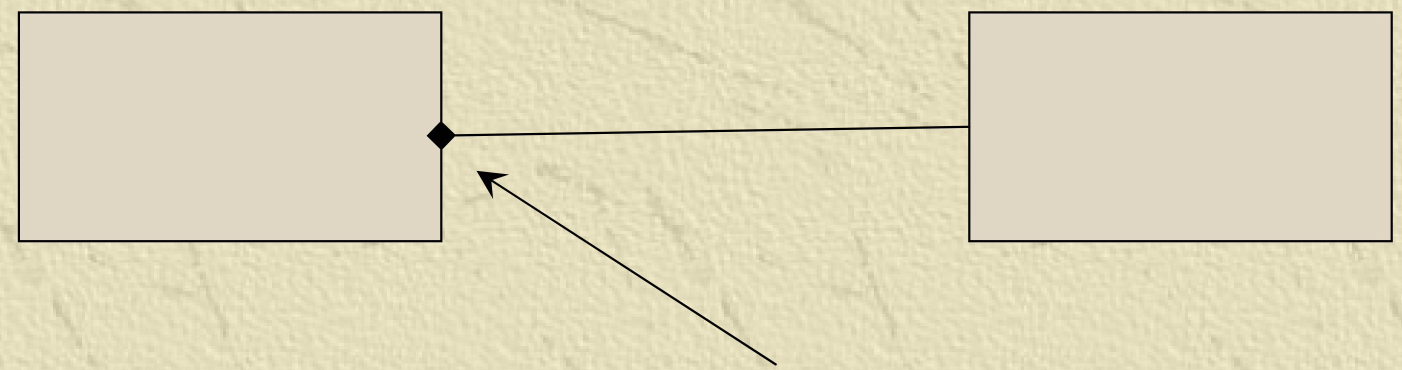
- Reasonably … from a certain point of view
- Stories, scheduling, object discovery, design, feedback, iteration
  - All fairly similar
- But…

- In the last few years, my hobby has been “challenging accepted knowledge”
- (Usually causing a lot of trouble in the process)
- A consultant’s job is to ask the hard questions

The point I want to make

- Design techniques are influenced by the languages they are about
- …Or that the technique-creators know
- Much of OO design is the same
- What is different because we’re using a dynamic language?

Example: UML



Filled or not? (Is the object "owned" or shared?) Meaningful in C++, not so much in Java, not at all in Python

Example: Design Patterns

- The GoF book was written primarily with C++ in mind, and pre-template C++ at that
- One Smalltalk programmer, very limited influence
- One pattern in particular was the favorite of one author, least favorite of another

Review of OOP in Python

```python
class Pet:
    count = 0  # Static field
    def __init__(self, petName = "Nameless"):
        self.name = petName # Normal field
        Pet.count += 1 # Access a static field
    def __del__(self):
        print self.name, "destructor"
    def __str__(self):
        return  self.__class__.__name__ + \
                " " + self.name
    def speak(self): print self, "speaking"
    def __add__(self, other) :
        print "mating", self, "with", other
    @staticmethod
    def getCount(): return Pet.count

p = Pet()
p.speak() # Pet Nameless speaking
```

Inheritance

```
class Dog(Pet): pass

# Constructors inherit too!
d = Dog("Bosco")
d.speak() # Dog Bosco speaking
```

Multiple inheritance

```
class Amphibian:
    def swim(self):
        print self, "swimming"

# Multiple inheritance:
class Gecko(Amphibian, Pet): pass

g = Gecko("Frank")
g.swim() # Gecko Frank swimming
```

Operator overloading & destructors

(Turn on your V-chips)

```
g + d # mating Gecko Frank with Dog Bosco
# Automatically reflexive:
d + g # mating Dog Bosco with Gecko Frank
# Calling a static method:
print Pet.getCount() # 3
# Bosco destructor
# Frank destructor
# Nameless destructor
```

Type-class unification

```
class MyList(list): pass
l = MyList([1,2,3])
print l # [1, 2, 3]
class MyInt(int): pass
```

Interface Inheritance vs. Implementation Inheritance

- Why do we inherit?
- Statically typed languages: to allow polymorphism by creating a common interface – upcast to “forget” specific type
- Dynamically typed & Duck typed: the language doesn’t care what the interface is
  - You inherit to reuse the implementation, adding to it and/or modifying the behavior
- Much code vanishes when you can just “send messages to objects.”

Does Python need interfaces?

- I've thought:
  - Interfaces are for static type checking
  - Python is dynamically typed
- But from yesterday's conversation:
  - Interfaces allow you to find out more about the type before you call a method
  - Can produce less coupling at the point of creation
  - Can be a helpful way to communicate *about* design, ala design patterns
- But what about this:

```
class Interface(object):
    def method1(self): raise NotImplementedError
    def method2(self): raise NotImplementedError
    def method3(self): raise NotImplementedError

class Implementation1(Interface):
    def method1(self): print "Implementation1.method1"
    def method2(self): print "Implementation1.method2"
    def method3(self): print "Implementation1.method3"

class Implementation2(Interface):
    def method1(self): print "Implementation2.method1"
    def method2(self): print "Implementation2.method2"
    def method3(self): print "Implementation2.method3"

def f(iface):
    print iface.__class__.__name__,
    if isinstance(iface, Interface):
        print "implements Interface“
    else:
        print "doesn't implement Interface"
```

```
f(Interface())
f(Implementation1())
f(Implementation2())
f(1)

output = """
Interface implements Interface
Implementation1 implements Interface
Implementation2 implements Interface
int doesn't implement Interface
"""
```

- PEP 245 effectively formalizes this
- Slightly less work to create & use
- What else would builtins buy you?

Does Python need adapters?

- No, adapters are "just" a convenience
- The basic idea: connecting two incompatible objects
- But adapters do make it easier
- Especially when working with larger systems like frameworks
  - Reduce the handwork to make classes work with a framework
- Also make adaptation more commonplace and natural by formalizing them in the language

Delegation (structural)

- Midway between composition and inheritance
- Inheritance: you get the whole interface
- Composition: underlying object is hidden
- Delegation: Some or all of the interface is exposed
- "Fronting" for an object happens often in design patterns
  - Proxy: you can insert operations before and after the call

```
# Must inherit from object for new-style behavior:
class Service(object):
  def a(self): print "Service.a"
  def b(self, arg): print "Service.b with argument", arg
  def c(self): print "Service.c"

def exercise(s):
    print "==== " + s.__class__.__name__ + " ===="
    try:
        s.a()
        s.b("Howdy")
        s.c()
    except NotImplementedError, e:
        print "not implemented:", e

exercise(Service())
output = """
==== Service ====
Service.a
Service.b with argument Howdy
Service.c
"""
```

```
# The whole interface:

class Inheritor(Service): pass

exercise(Inheritor())
output = """
==== Inheritor ====
Service.a
Service.b with argument Howdy
Service.c
"""
```

```
# Same interface minus c():
class Delegator:
    def __init__(self):
        self.service = Service()
    def __getattr__(self, name):
        if name == 'c':
            raise NotImplementedError, "c()"
        if hasattr(Service, name):
            return getattr(self.service, name)

exercise(Delegator())
output = """
==== Delegator ====
Service.a
Service.b with argument Howdy
not implemented: c()
"""
```

```
# Since 2.2, you can subtract after inheritance:

class SubtractionInheritance(Service):
    def __getattribute__(self, name):
        if name == 'c':
            raise NotImplementedError, "c()"
        return Service.__getattribute__(self, name)

exercise(SubtractionInheritance())
output = """
==== SubtractionInheritance ====
Service.a
Service.b with argument Howdy
not implemented: c()
"""
```

```
# Performing operations before/after call (Proxy):

class Proxy:
    def __init__(self):
        self.service = Service()
    def __getattr__(self, name):
        if hasattr(Service, name):
            print "Entering", name
            return getattr(self.service, name)
    def c(self):
        print "Pre-call operation"
        result = self.service.c()
        print "Post-call operation"
```

```
exercise(Proxy())
output = """
==== Proxy ====
Entering a
Service.a
Entering b
Service.b with argument Howdy
Pre-call operation
Service.c
Post-call operation
"""
```

Generators

- Special case of a factory
- Still a pattern, but language support changes the sense of it
- Iterator pattern also built into a number of places

```
def fibonacci(count):
    def fib(n):
        if n < 2: return 1
        return fib(n-2) + fib(n-1)
    n = 0
    while n < count:
        yield fib(n)
        n += 1

for f in fibonacci(20): # Automatically iterable
    print f,

output="""
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
1597 2584 4181 6765
"""
```

Aspect-oriented programming/crosscutting

- I've become convinced that this is only a subset of metaclasses
- See the example of adding "synchronized" to Python methods in *Thinking in Python*
  - I seem to remember that Alex helped with this

Only touched on the issues

- Many more questions to ask. For example:
- Visitor pattern allows you to dynamically add new methods to a fixed hierarchy of classes. What does “fixed hierarchy” and “dynamically add new methods” mean in the context of Python?

Dynamic languages always better?

- Tempting to say so, but I’m not always sure
- For one thing, that statement may assume that all programmers are at the same experience level
- It’s easy to produce sheet rockers, more difficult to produce plumbers and electricians, and good finish carpenters are not so common

“Thinking in Python” wiki project

- Languishing far too long
- Idea: put it into a wiki, let the community add ideas, examples, and correct details
- When it’s stable, I go through and rewrite it into a book, publish
- Need to work out legal issues for taking community contributions and turning them into a book, but there’s precedent in the “Python Cookbook”

Open Space on OO design

6:00 – 6:30 Room 310