Python Design Patterns

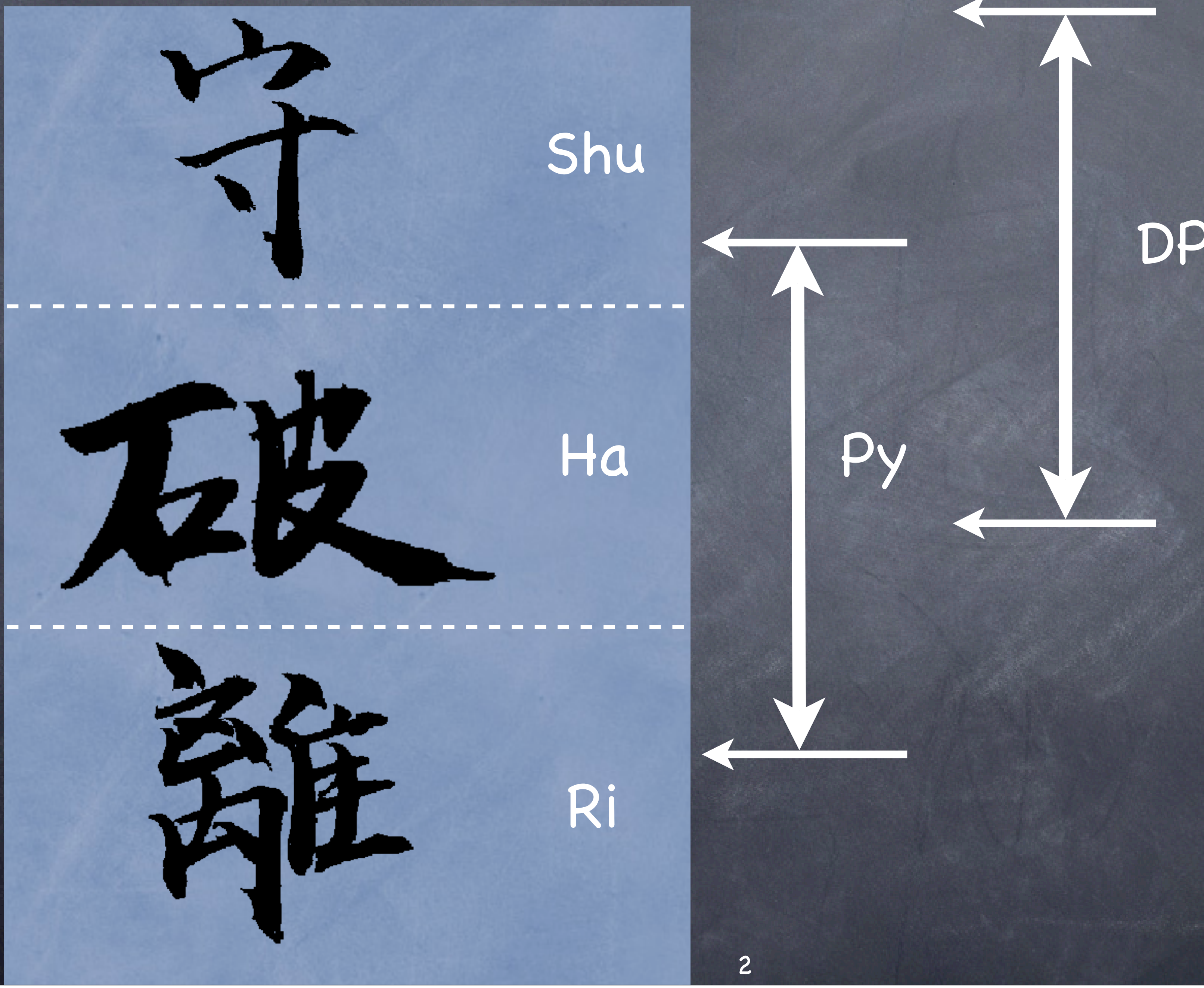
Alex Martelli

http://www.aleax.it/goo_pydp.pdf

Google

©2007 Google -- aleax@google.com

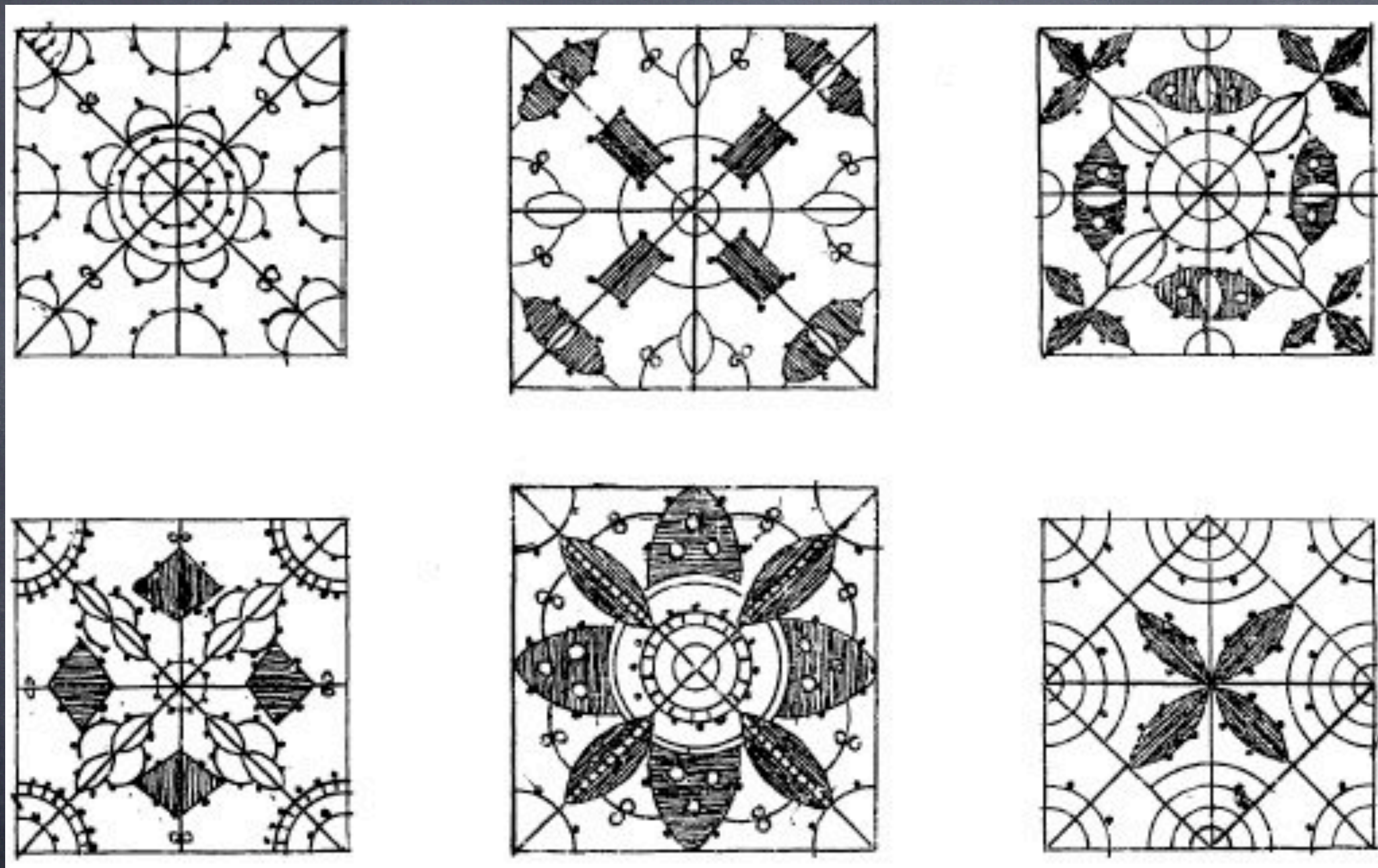
The "levels" of this talk



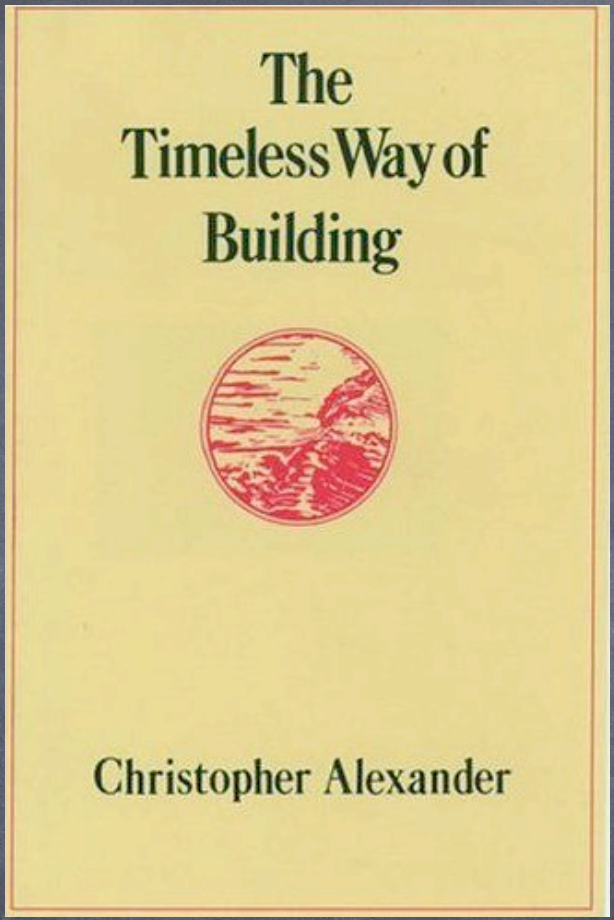
Contents of these talks

- Patterns [11]
- Python: hold or wrap? [3]
- Creational Patterns [8]
- Structural Patterns (M&A) [22]
- Behavioral Patterns (Template &c) [35]
  - Template [9basic+16advanced -> 25]
  - State & Strategy [9]
- Q & A

Patterns



The Birth of Patterns



...every building, every town, is made of certain entities which I call patterns... in terms of these pattern languages, all the different ways ... become similar in general outline.

Patterns in general [1]

- "Each pattern describes a problem which occurs over and over in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander et al, "A Pattern Language"]

Patterns in general [2]

- Design (and thus patterns) is not independent from the implementation's technology -- when building with bricks, vs concrete, vs wood, &c, many patterns remain (often w/small changes), but many appear, disappear, or change deeply
- "Point of view affects one's interpretation of what is and isn't a pattern... choice of programming language is important because it influences one's point of view" [Gamma et al, "Design Patterns"]

Design Patterns in SW

- rich, thriving culture and community
  - mostly a subculture of OO development
- Gamma, Helms, Johnson, Vlissides (1995)
  - "the gang of 4" AKA "Gof4"
- PLOP conferences & proceedings thereof
- DP once risked becoming a fad, or fashion
  - there is no silver bullet...
  - ...but, we now know, DP are here to stay
- ...and, NOT independent from the choice of programming language!-)

What are classic SW DPs

- not data structures, nor algorithms
- not domain-specific architectures for entire subsystems
- just: "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [Gof4]
- scope: sometimes class, mosty object
- purpose: general category describing what the pattern is about

DP write-up components

- NAME, context, problem
- forces, solution, examples
- results, rationale, related DPs
- KNOWN USES (KU for short)
  - DP are discovered, NOT invented
- DP are about description (and helpful related suggestions), NOT prescription
- formal fixed schema not a must
  - but helpful as a checklist
  - somewhat audience-dependent

Google

SW DP Books

Your Brain on Design Patterns
Head First Design Patterns
O'REILLY
Dating Design Patterns
Elements of Reusable Objective-Oriented Paired Programming
Ericha Gordon
Rickie Hanson
Rhonda Jackson
Jonna Disvlisses
Modern C++ Design
Generic Programming and Design Patterns Applied
Andrei Alexandrescu
Pattern Hatching
Design Patterns Applied
John Vlissides
Design Patterns Explained
A New Perspective on Object-Oriented Design
Second Edition
Alan Shalloway
James R. Trott
Software Patterns Series
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
Addison Wesley Professional Computing Series
The Design Patterns Smalltalk Companion
Sherman R. Alpert
Kyle Brown
Bobby Woolf
Software Patterns Series
Agile Software Development
Principles, Patterns, and Practices
Robert C. Martin
The Addison-Wesley Signature Series
Refactoring to Patterns
Joshua Kerievsky

DP myths and realities

- many "classic" DP (for C++ or Java) are "workarounds against static typing" (cfr: Alpert, Brown, Woolf, "The DPs Smalltalk Companion", Addison-Wesley DP Series)
- in Python: classic DP, minus WaFST, plus (optionally...:-) specific exploits of Python's dynamic and introspection strengths
- no silver bullet, but, quite helpful IRL
  - NAMES matter more than you'd think
  - "the guy with the hair, you know, the Italian" vs "Alex"...

Categories of SW DP

- Creational
  - concern the ways and means of object instantiation
- Structural
  - deal with the mutual composition of classes or objects
- Behavioral
  - analyze the ways in which classes or objects interact and distribute responsibilities among them

Prolegomena to SW DPs

- "program to an interface, not to an implementation"
  - usually done "informally" in Python
- "favor object composition over class inheritance"
  - in Python: hold, or wrap
  - inherit only when it's really convenient
    - very direct way to expose all methods in the base class (reuse + usually override + maybe extend)
    - but, it's a rather strong coupling!

Python: hold or wrap?

Python: hold or wrap?

- "Hold": object O has subobject S as an attribute (maybe property) -- that's all
  - use self.S.method or O.S.method
  - simple, direct, immediate, but... pretty strong coupling, often on the wrong axis
- "Wrap": hold (often via private name) plus delegation (so you directly use O.method)
  - explicit (def method(self...)...self.S.method)
  - automatic (delegation in __getattr__)
  - gets coupling right (Law of Demeter)

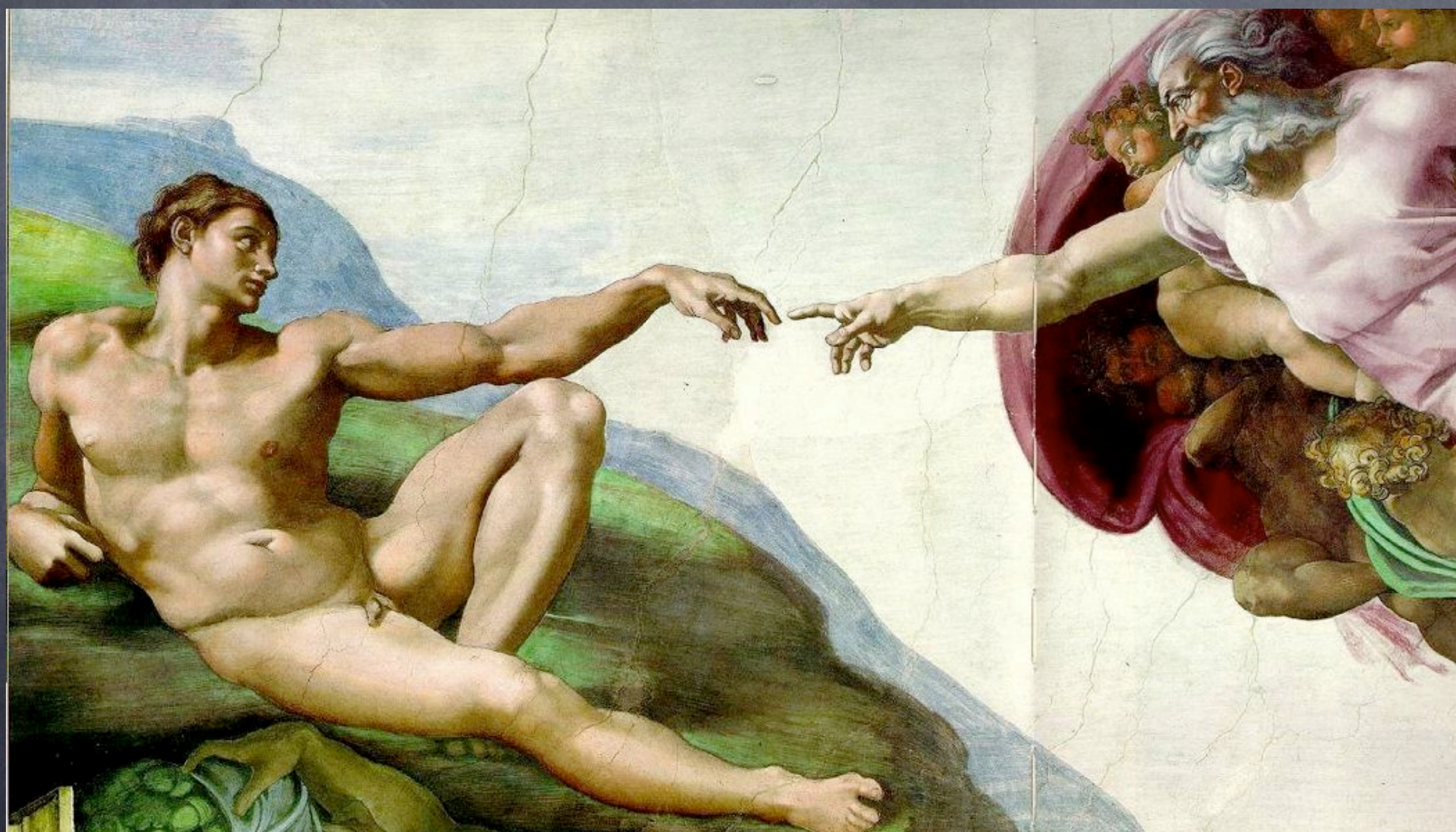
Wrapping to "restrict"

```
class RestrictingWrapper(object):
  def __init__(self, w, block):
    self._w = w
    self._block = block
  def __getattr__(self, n):
    if n in self._block:
      raise AttributeError, n
    return getattr(self._w, n)
  ...
```

Inheritance cannot restrict!

Creational Patterns

- not very common in Python...
- ...because "factory" is essentially built-in!-)



Creational Patterns [1]

- "we want just one instance to exist"
  - use a module instead of a class
    - no subclassing, no special methods, ...
  - make just 1 instance (no enforcement)
    - need to commit to "when" to make it
  - singleton ("highlander")
    - subclassing not really smooth
  - monostate ("borg")
    - Guido dislikes it

Singleton ("Highlander")

```
class Singleton(object):
  def __new__(cls, *a, **k):
    if not hasattr(cls, '_inst'):
      cls._inst = super(Singleton, cls
                 ).__new__(cls, *a, **k)
    return cls._inst
```

subclassing is a problem, though:

```
class Foo(Singleton): pass
class Bar(Foo): pass
f = Foo(); b = Bar(); # ...???...
```

problem is intrinsic to Singleton

Monostate ("Borg")

```
class Borg(object):
  _shared_state = {}
  def __new__(cls, *a, **k):
    obj = super(Borg, cls
               ).__new__(cls, *a, **k)
    obj.__dict__ = cls._shared_state
    return obj
```

subclassing is no problem, just:

```
class Foo(Borg): pass
class Bar(Foo): pass
class Baz(Foo): _shared_state = {}
```

data overriding to the rescue!

Google

Creational Patterns [2]

- "we don't want to commit to instantiating a specific concrete class"
  - dependency injection
    - no creation except "outside"
    - what if multiple creations are needed?
  - "Factory" subcategory of DPs
    - may create w/ever or reuse existing
    - factory functions (& other callables)
    - factory methods (overridable)
    - abstract factory classes

Factories in Python

- each type/class is intrinsically a factory
  - internally, may have __new__
  - externally, it's just a callable, interchangeable with any other
  - may be injected directly (no need for boilerplate factory functions)
- modules can be kinda "abstract" factories w/o inheritance ('os' can be 'posix' or 'nt')

KU: type.__call__

```
def __call__(cls,*a,**k):
  nu = cls.__new__(cls,*a,**k)
  if isinstance(nu, cls):
    cls.__init__(nu,*a,**k)
  return nu
```

(An instance of "two-phase construction")

factory-function example

```
def load(pkg, obj):
  m = __import__(pkg,{},{},[obj])
  return getattr(m, obj)

# example use:
# cls = load('p1.p2.p3', 'c4')
```

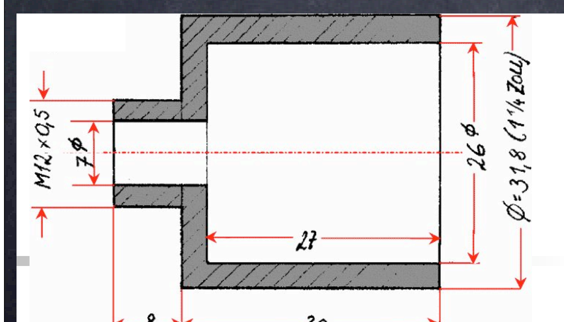
Structural Patterns

The "Masquerading/Adaptation" subcategory:

- Adapter: tweak an interface (both class and object variants exist)
- Facade: simplify a subsystem's interface
- Bridge: let many implementations of an abstraction use many implementations of a functionality (without repetitive coding)
- Decorator: reuse+tweak w/o inheritance
- Proxy: decouple from access/location

Adapter

- client code γ requires a protocol C
- supplier code σ provides different protocol S (with a superset of C's functionality)
- adapter code α "sneaks in the middle":
  - to γ, α is a supplier (produces protocol C)
  - to σ, α is a client (consumes protocol S)
  - "inside", α implements C (by means of appropriate calls to S on σ)



Toy-example Adapter

- C requires method foobar(foo, bar)
- S supplies method barfoo(bar, foo)
- e.g., σ could be:

```
class Barfooer(object):
    def barfoo(self, bar, foo):
        ...
```

Object Adapter

- per-instance, with wrapping delegation:

```
class FoobarWrapper(object):
  def __init__(self, wrappee):
    self.w = wrappee
  def foobar(self, foo, bar):
    return self.w.barfoo(bar, foo)

foobarer=FoobarWrapper(barfooer)
```

Class Adapter

- per-class, w/subclasing & self-delegation:

```
class Foobarer(Barfooer):
  def foobar(self, foo, bar):
    return self.barfoo(bar, foo)

foobarer=Foobarer(...w/ever...)
```

Adapter KU

- socket._fileobject: from sockets to file-like objects (w/much code for buffering)
- doctest.DocTestSuite: adapts doctest tests to unittest.TestSuite
- dbhash: adapt bsddb to dbm
- StringIO: adapt str or unicode to file-like
- shelve: adapt "limited dict" (str keys and values, basic methods) to complete mapping
  - via pickle for any <-> string
  - + UserDict.DictMixin

Adapter observations

- some RL adapters may require much code
- mixin classes are a great way to help adapt to rich protocols (implement advanced methods on top of fundamental ones)
- Adapter occurs at all levels of complexity
- in Python, it's _not_ just about classes and their instances (by a long shot!-)

Facade

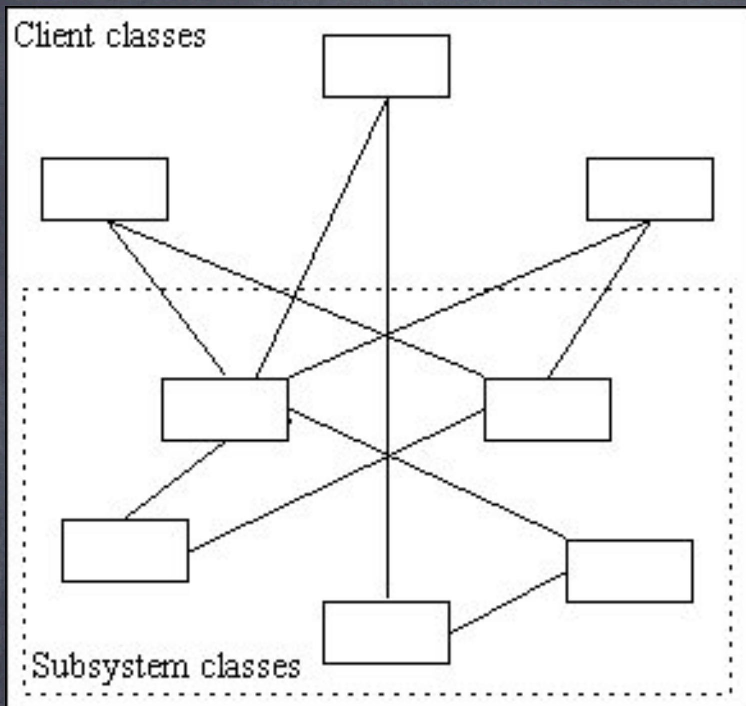
- supplier code σ provides rich, complex functionality in protocol S
- we need simple subset C of S
- facade code Φ implements and supplies C (by means of appropriate calls to S on σ)

Google

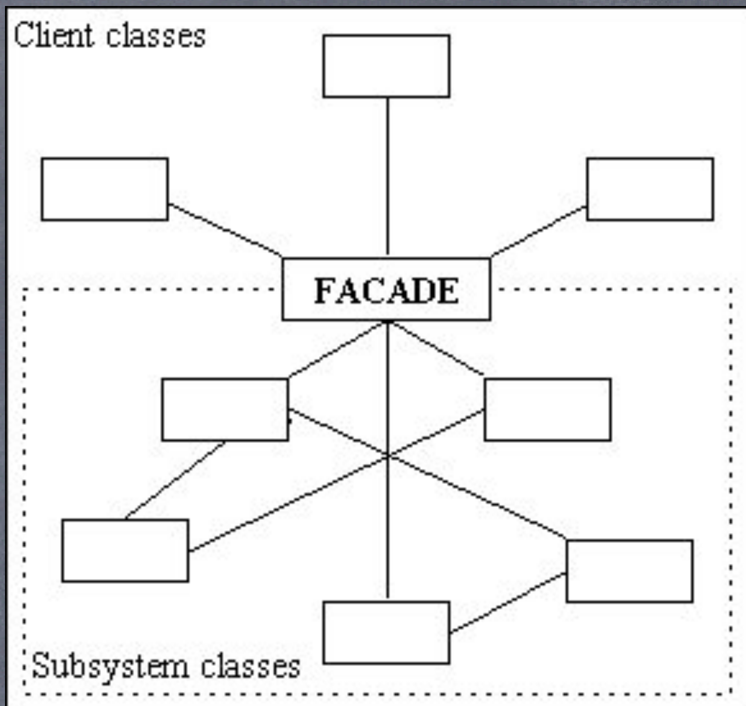
Facade vs Adapter

- Adapter's about supplying a given protocol required by client-code
  - or, gain polymorphism via homogeneity
- Facade is about simplifying a rich interface when just a subset is often needed
- Facade most often "fronts" for many objects, Adapter for just one

Facade, before & after



Without Facade



With Facade

http://www.tonymarston.net/php-mysql/design-patterns.html

Facade KU

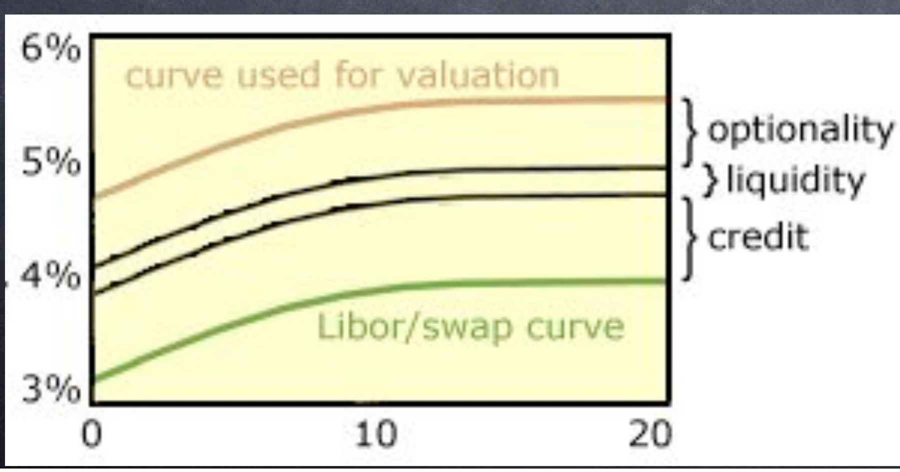
- asynchat.fifo facades for list
- dbhash facades for bsddb
  - ...also given as example of Adapter...!-)
- old sets.Set used to facade for dict
- Queue facades for deque + lock
  - well...:-)
- os.path: basename, dirname facade for split + indexing; isdir &c facade for os.stat + stat.S_ISDIR &c.

Facade observations

- some RL facades may have substantial code
  - simplifying the _protocol_ is the key
  - interface simplifications are often accompanied by minor functional additions
- Facade occurs at all levels of complexity, but it's most important for _complicated subsystems_ (and reasonably-simple views)
- inheritance is never useful for Facade, because it can only "widen", never "restrict" (so, wrapping is the norm)

Adapting/Facading callables

- callables (functions &c) play a large role in Python programming -- so you offen may need to adapt or facade them
- functools.partial to preset some args (AKA "currying") + bound-methods special case
- decorator syntax to adapt functions or methods by wrapping in HOFs
- closures for simple HOF needs
  - classes w/__call__ for complex ones



Google

Bridge

- have N1 realizations ρ of abstraction A,
- each using any one of N2 implementations ι of functionality F,
- <u>without</u> coding N1*N2 cases ...
- have abstract superclass A of all ρ hold a reference R to the interface F of all ι,
- ensure each ρ uses any functionality of F (thus, from a ι) <u>only</u> by delegating to R

A Toy Bridge Example

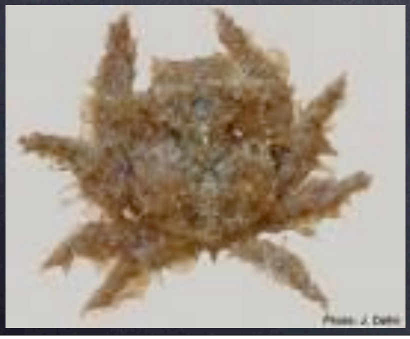
```
class AbstractParser(object):
  def __init__(self, scanner):
    self.scanner = scanner
  def __getattr(self, name):
    return getattr(self.scanner, name)
class ExpressionParser(AbstractParser):
  def expr(self):
    ... token = self.next_token() ...
    ... self.push_back(token) ...
```

Bridge KU: SocketServer

- BaseServer is the "abstraction" A
- BaseRequestHandler is F (the abstract superclass for functionality implementation)
- (also uses mixins for threading, forking...)
- note: A holds the very class F, and instantiates it per-request (shades of a Factory DP...) -- typical in Bridge DP in Python (also e.g. email: Parser -> Message)
- -> Bridge is typical of somewhat "rich", complicated situations

Decorator

- client code γ requires a protocol C
- supplier code σ does provide protocol C
- but we want to insert some semantic tweak
  - often dynamically plug-in/plug-out-able
- decorator code δ "sneaks in the middle":
  - γ uses δ just as it would use σ
  - δ wraps σ, and it may intercept, modify, add (a little), delegate, ...



Toy Example Decorator

```
class WriteFullLinesOnlyFile(object):
  def __init__(self, *a, **k):
    self._f = open(*a, **k)
    self._b = ''
  def write(self, data):
    lns = (self._b+data).splitlines(True)
    if lns[-1][-1]=='\n': self._b = ''
    else: self._b = lns.pop(-1)
    self._f.writelines(lns)
  def __getattr__(self, name):
    return getattr(self._f, name)
```

KU of Decorator

- gzip.GzipFile decorates a file with compress/decompress functionality
- threading.RLock decorates thread.Lock with reentrancy & ownership functionality
- codecs classes decorate a file with generic encoding and decoding functionality

Proxy

- client code γ needs to access an object τ
- however, something interferes w/that...:
  - τ lives remotely, or in persisted form
  - access restrictions may apply (security)
  - lifetime or performance issues
- proxy object π "sneaks in the middle":
  - π wraps τ, may create/delete it at need
  - may intercept, call, delegate, ...
  - γ uses π as it would use τ

Toy Example Proxy

```
class RestrictingProxy(object):
  def __init__(self, block, f, *a, **k):
    self._makeit = f, a, k
    self._block = block
  def __getattr__(self, name):
    if name in self._block:
      raise AttributeError, name
    if not hasattr(self, '_wrapped'):
      f, a, k = self._makeit
      self._wrapped = f(*a, **k)
    return getattr(self._wrapped, name)
```

KU of Proxy

- the values in a shelve.Shelf proxy for persisted objects (get instantiated at need)
- weakref.proxy proxies for any existing object but doesn't "keep it alive"
- idlelib.RemoteDebugger uses proxies (for frames, code objects, dicts, and a debugger object) across RPC to let a Python process be debugged from a separate GUI process

Q & A on part 1

Q?

A!

Behavioral Patterns

- Template Method: self-delegation
  - "the essence of OOP"...
- State and Strategy as "factored out" extensions to Template Method

Template Method

- great pattern, lousy name
  - "template" very overloaded
    - generic programming in C++
    - generation of document from skeleton
    - ...
- a better name: self-delegation
  - directly descriptive
  - TM tends to imply more "organization"

Classic TM

- abstract base class offers "organizing method" which calls "hook methods"
- in ABC, hook methods stay abstract
- concrete subclasses implement the hooks
- client code calls organizing method
  - on some reference to ABC (injecter, or...)
  - which of course refers to a concrete SC

TM skeleton

```
class AbstractBase(object):
  def orgMethod(self):
    self.doThis()
    self.doThat()

class Concrete(AbstractBase):
  def doThis(self): ...
  def doThat(self): ...
```

TM example: paginate text

- to paginate text, you must:
  - remember max number of lines/page
  - output each line, while tracking where you are on the page
  - just before the first line of each page, emit a page header
  - just after the last line of each page, emit a page footer

AbstractPager

```
class AbstractPager(object):
  def __init__(self, mx=60):
    self.mx = mx
    self.cur = self.pg = 0
  def writeLine(self, line):
    if self.cur == 0:
      self.doHead(self.pg)
    self.doWrite(line)
    self.cur += 1
    if self.cur >= self.mx:
      self.doFoot(self.pg)
      self.cur = 0
      self.pg += 1
```

Concrete pager (stdout)

```
class PagerStdout(AbstractPager):
  def doWrite(self, line):
    print line
  def doHead(self, pg):
    print 'Page %d:\n\n' % pg+1
  def doFoot(self, pg):
    print '\f',    # form-feed character
```

Concrete pager (curses)

```
class PagerCurses(AbstractPager):
  def __init__(self, w, mx=24):
    AbstractPager.__init__(self, mx)
    self.w = w
  def doWrite(self, line):
    self.w.addstr(self.cur, 0, line)
  def doHead(self, pg):
    self.w.move(0, 0)
    self.w.clrtobot()
  def doFoot(self, pg):
    self.w.getch()    # wait for keypress
```

Classic TM Rationale

- the "organizing method" provides "structural logic" (sequencing &c)
- the "hook methods" perform "actual ``elementary'' actions"
- it's an often-appropriate factorization of commonality and variation
  - focuses on objects' (classes') responsibilities and collaborations: base class calls hooks, subclass supplies them
  - applies the "Hollywood Principle": "don't call us, we'll call you"

A choice for hooks

```
class TheBase(object):
  def doThis(self):
    # provide a default (often a no-op)
    pass
  def doThat(self):
    # or, force subclass to implement
    # (might also just be missing...)
    raise NotImplementedError
```

Default implementations often handier, when sensible; but "mandatory" may be good docs.

Overriding Data

```
class AbstractPager(object):
  mx = 60
...
class CursesPager(AbstractPager):
  mx = 24
...
```

access simply as self.mx -- obviates any need for boilerplate accessors self.getMx()...

KU: Queue.Queue

```
class Queue:
  ...
  def put(self, item):
    self.not_full.acquire()
    try:
      while self._full():
        self.not_full.wait()
      self._put(item)
      self.not_empty.notify()
    finally:
      self.not_full.release()
  def _put(self, item): ...
```

Queue's TMDP

- Not abstract, often used as-is
  - thus, implements all hook-methods
- subclass can customize queueing discipline
  - with no worry about locking, timing, ...
  - default discipline is simple, useful FIFO
  - can override hook methods (_init, _qsize, _empty, _full, _put, _get) AND...
  - ...data (maxsize, queue), a Python special

Customizing Queue

```
class LifoQueueA(Queue):
  def _put(self, item):
    self.queue.appendleft(item)

class LifoQueueB(Queue):
  def _init(self, maxsize):
    self.maxsize = maxsize
    self.queue = list()
  def _get(self):
    return self.queue.pop()
```

KU: cmd.Cmd.cmdloop

```
def cmdloop(self):
  self.preloop()
  while True:
    s = self.doinput()
    s = self.precmd(s)
    f = self.docmd(s)
    f = self.postcmd(f,s)
    if f: break
  self.postloop()
```

KU: asyncore.dispatcher

```
# several organizing-methods, e.g:
def handle_write_event(self):
  if not self.connected:
    self.handle_connext()
    self.connected = 1
  self.handle_write()
```

"Class TM": DictMixin

- Abstract, meant to multiply-inherit from
  - does not implement hook-methods
- subclass must supply needed hook-methods
  - at least __getitem__, keys
  - if R/W, also __setitem__, __delitem__
  - normally __init__, copy
  - may override more (for performance)

TM in DictMixin

```
class DictMixin:
  ...
  def has_key(self, key):
     try:
        # implies hook-call (__getitem__)
        value = self[key]
     except KeyError:
        return False
     return True
  def __contains__(self, key):
     return self.has_key(key)
  ...
```

Exploiting DictMixin

```
class Chainmap(UserDict.DictMixin):
  def __init__(self, mappings):
    self._maps = mappings
  def __getitem__(self, key):
    for m in self._maps:
      try: return m[key]
      except KeyError: pass
    raise KeyError, key
  def keys(self):
    keys = set()
    for m in self._maps: keys.update(m)
    return list(keys)
```

"Factoring out" the hooks

- "organizing method" in one class
- "hook methods" in another
- KU: HTML formatter vs writer
- KU: SAX parser vs handler
- adds one axis of variability/flexibility
- shades towards the Strategy DP:
  - Strategy: 1 abstract class per decision point, independent concrete classes
  - Factored TM: abstract/concrete classes more "grouped"

TM + introspection

- "organizing" class can snoop into "hook" class (maybe descendant) at runtime
  - find out what hook methods exist
  - dispatch appropriately (including "catch-all" and/or other error-handling)

KU: cmd.Cmd.docmd

```
def docmd(self, cmd, a):
  ...
  try:
    fn = getattr(self, 'do_' + cmd)
  except AttributeError:
    return self.dodefault(cmd, a)
  return fn(a)
```

Interleaved TMs KU

- plain + factored + introspective
  - multiple "axes", to separate carefully distinct variabilities
- a DP equivalent of a "Fuga a Tre Soggetti"
  - "all art aspires to the condition of music" (Pater, Pound, Santayana...?-)

KU: unittest.TestCase

```
def__call__(self, result=None):
  method = getattr(self, ...)
  try: self.setUp()
  except: result.addError(...)
  try: method()
  except self.failException, e:...
  try: self.tearDown()
  except: result.addError(...)
  ...result.addSuccess(...)...
```

State and Strategy DPs

- Not unlike a “Factored-out” TMDP
  - OM in one class, hooks in others
  - OM calls self.somedelegate.dosomehook()
- classic vision:
  - Strategy: 1 abstract class per decision, factors out object behavior
  - State: fully encapsulated, strongly coupled to Context, self-modifying
- Python: can switch __class__, methods

Strategy DP

```
class Calculator(object):
  def __init__(self):
    self.strat = Show()
  def compute(self, expr):
    res = eval(expr)
    self.strat.show('%r=%r'% (expr, res))
  def setVerbosity(self, quiet=False):
    if quiet: self.strat = Quiet()
    else: self.strat = Show()
```

Strategy classes

```
class Show(object):
  def show(self, s):
    print s

class Quiet(Show):
  def show(self, s):
    pass
```

State DP: base class

```
class Calculator(object):
  def __init__(self):
    self.state = Show()
  def compute(self, expr):
    res = eval(expr)
    self.state.show('%r=%r'% (expr, res))
  def setVerbosity(self, quiet=False):
    self.state.setVerbosity(self, quiet)
```

State classes

```
class Show(object):
  def show(self, s):
    print s
  def setVerbosity(self, obj, quiet):
    if quiet: obj.state = Quiet()
    else: obj.state = Show()

class Quiet(Show):
  def show(self, s):
    pass
```

Ring Buffer

- FIFO queue with finite memory: stores the last MAX (or fewer) items entered
  - good, e.g., for logging tasks
- intrinsically has two macro-states:
  - early (<=MAX items entered yet), just append new ones
  - later (>MAX items), each new item added must overwrite the oldest one remaining (to keep latest MAX items)
- switch from former macro-state (behavior) to latter is massive, irreversible

Switching __class__ (1)

```
class RingBuffer(object):
  def __init__(self):
    self.d = list()
  def tolist(self):
    return list(self.d)
  def append(self, item):
    self.d.append(item)
    if len(self.d) == MAX:
      self.c = 0
      self.__class__ = _FullBuffer
```

Switching __class__ (2)

```
class _FullBuffer(object):
  def append(self, item):
    self.d[self.c] = item
    self.c = (1+self.c) % MAX
  def tolist(self):
    return ( self.d[self.c:] +
             self.d[:self.c] )
```

Switching a method

```
class RingBuffer(object):
  def __init__(self):
    self.d = list()
  def append(self, item):
    self.d.append(item)
    if len(self.d) == MAX:
      self.c = 0
      self.append = self.append_full
  def append_full(self, item):
    self.d.append(item)
    self.d.pop(0)
  def tolist(self):
    return list(self.d)
```

Q & A on part 2

Q?

A!