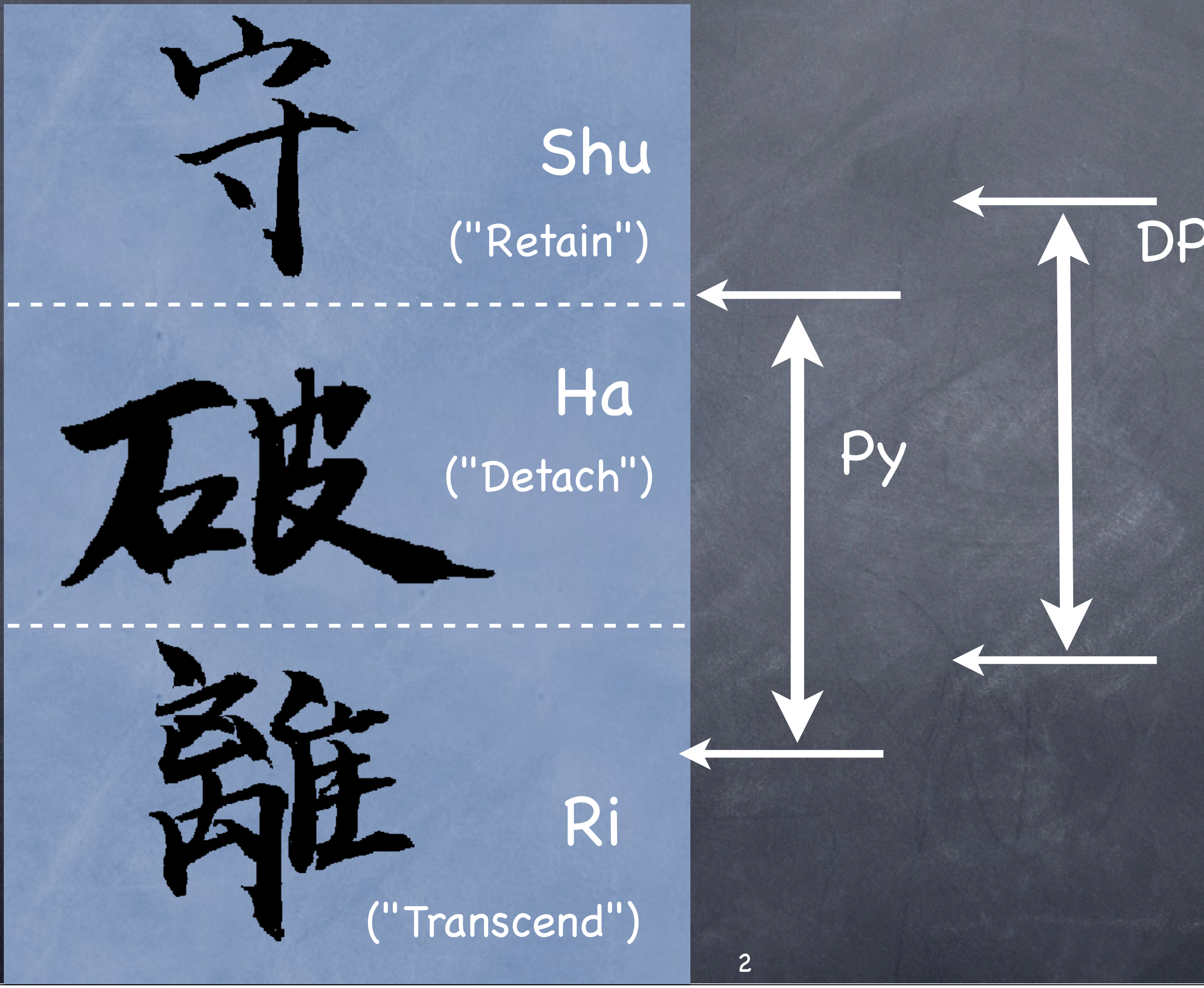
Practical Python Patterns

Alex Martelli (aleax@google.com)

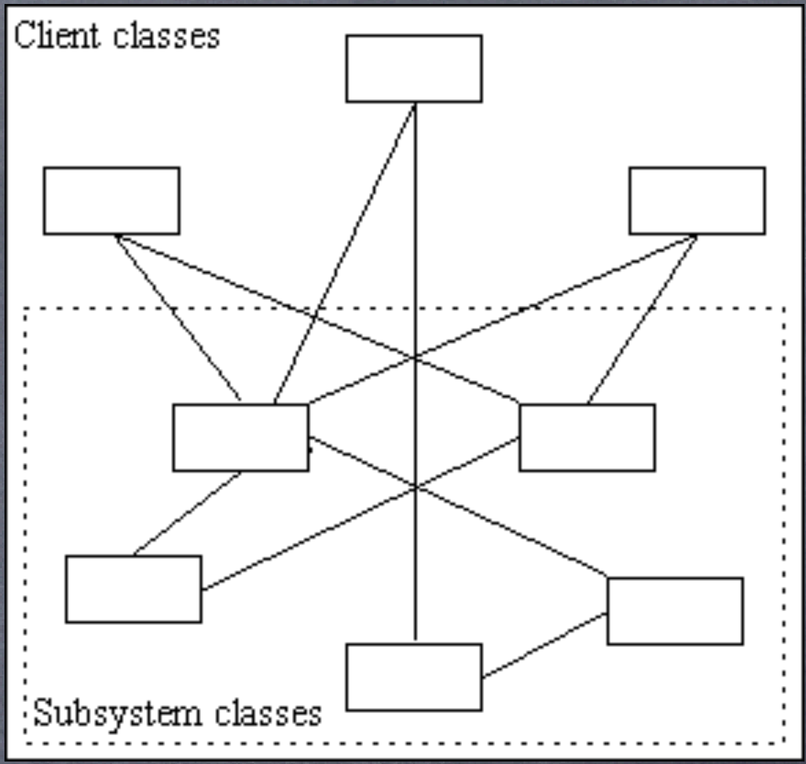
http://www.aleax.it/oscon010_pydp.pdf

The "levels" of this talk



A Pattern Example

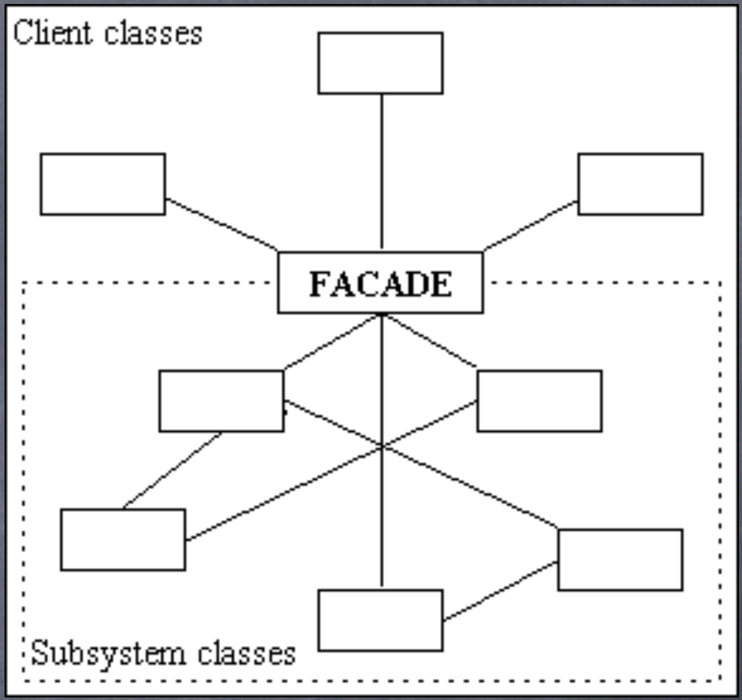
- "Forces": a rich, complex system offers a lot of functionality; client code interacts with many parts of this functionality in a way that's "out of control"
- this causes many problems for client-code programmers AND the system's ones too



(complexity + rigidity)

Solution: the "Facade" DP

- interpose a simpler "Facade" object/class exposing a controlled subset of functionality
  - client code now calls into the Facade, only
  - the Facade implements its simpler functionality via calls into the rich, complex subsystem
- subsystem implementers gains flexibility, clients gain simplicity



Facade is a Design Pattern

- summary of a frequent design problem + structure of a solution to that problem (+ pros and cons, alternatives, ...), and:
  - A NAME (much easier to retain/discuss!)
- "descriptions of communicating objects and classes customized to solve a general design problem in a particular context"
- that's NOT: a data structure, an algorithm, a domain-specific system architecture, a programming-language/library feature
- MUST be studied in a specific context!
- BEST: give Known Uses ("KU"), "stars"

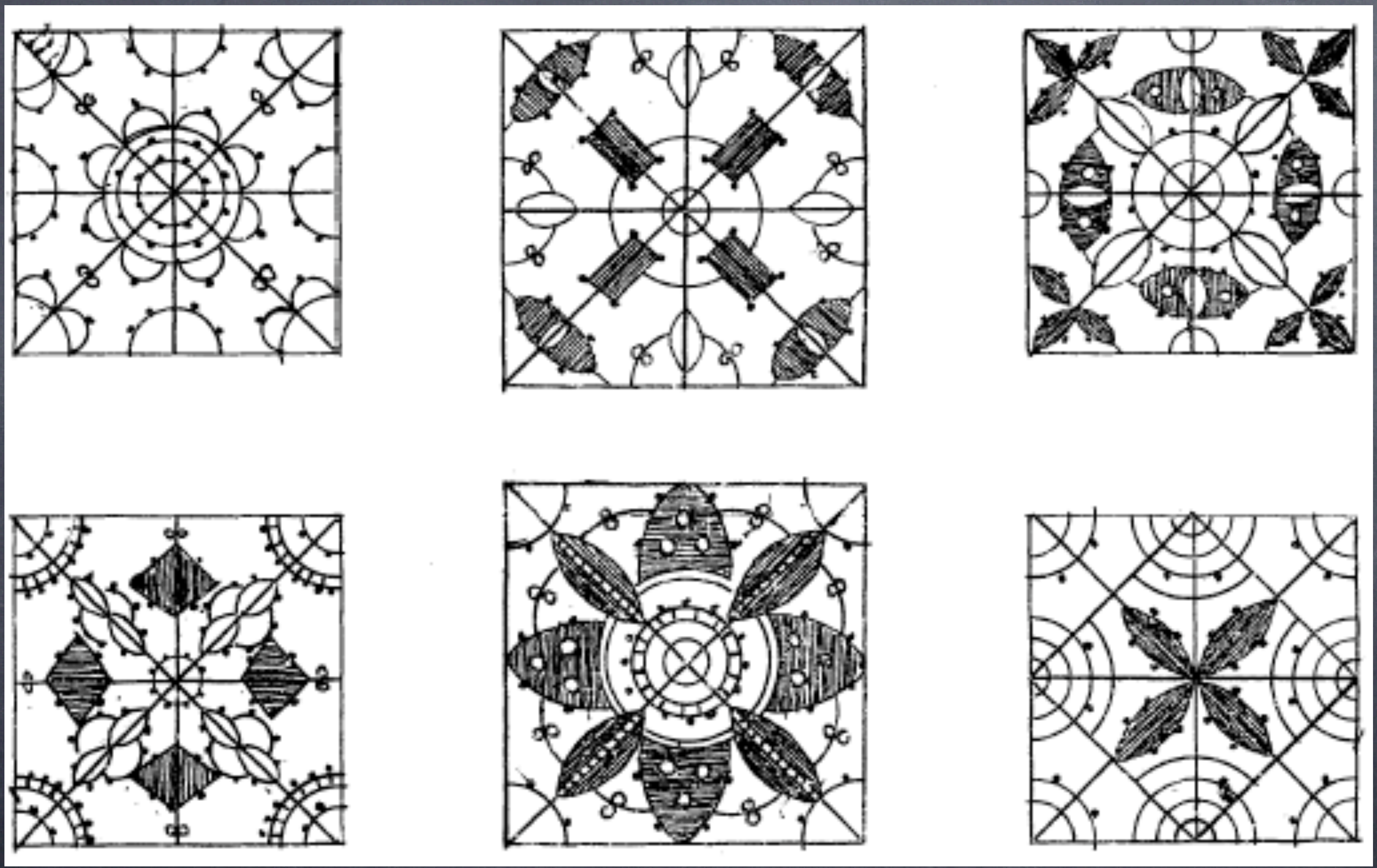
Some Facade KUs

- ...in the Python standard library...:
  - dbhash facades for bsddb
    - highly simplified/subset access
    - also meets the "dbm" interface (thus, also an example of the Adapter DP)
  - os.path: basename, dirname facade for split + indexing; isdir (&c) facade for os.stat + stat.S_ISDIR (&c)
- Facade is a structural DP (we'll see another, Adapter, later; in dbhash, they "merge"!-)

Is Facade a "Pythonic" DP?

- yes... and no
  - it works just fine in Python, but...
  - ...it works just as well most everywhere!
  - i.e., it is, rather, a "universal" DP
- points to ponder/debate...:
  - is it "Facade" if it offers all functionality?
  - is it "Facade" if it _adds_ functionality?
    - do taxonomies ever work fully?-)
  - do other DPs/idioms "go well with it"?
    - "above"? "below"? "to the side"?

Design Patterns



What's a Design Pattern

- summary of a frequent design problem + structure of a solution to that problem + pros and cons, alternatives, ..., and:
  - A NAME (much easier to retain/discuss!)
- "descriptions of communicating objects and classes customized to solve a general design problem in a particular context"
- DPs are NOT: data structures, algorithms, domain-specific system architectures, programming language features
- MUST be studied in a language's context!
- Best: supply Known Uses ("KU") & "stars"

Step back: what's a Pattern?

- identify a closely related class of problems
  - if there is no problem, why solve it?-)
- identify a class of solutions to the problems
  - closely related, just like the problems are
- may exist in any one of many different possible scales ("phases of work")
  - just like the problems do
- Design patterns are exactly those patterns whose scale/phase is... design!

A Pattern's "problem(s)"

- each Pattern addresses a problem
  - rather, a closely related <u>class of problems</u>
- a problem is defined by:
  - "forces"
    - constraints, desiderata, side effects, ...
  - "context" (including: what technologies can be deployed to solve the problem)

A Pattern's "solution(s)"

- to write-up a pattern, you must identify a class of solutions to the problems
  - within the context (technologies, &c)
- meaningful name and summary
- a "middling-abstraction" description
- real-world examples (if any!-), "stars"
  - one-star == "0/1 existing examples"
- rationale, "quality without a name"
- how it balances forces / +'s & issues
- pointers to related/alternative patterns

Why bother w/Patterns?

- identifying patterns helps all practitioners of a field "up their game"...
- ...towards the practices of the very best ones in the field
  - precious in teaching, training, self-study
  - precious in concise communication, esp. in multi-disciplinary cooperating groups
  - also useful in enhancing productivity
    - to recognize is faster than to invent
    - structured description helps recognition

"Design" is a vague term...

- most generically, it can mean "purpose"
- or specifically, a plan towards a purpose
- a geometrical or graphical arrangement
- an "arrangement" in a more abstract sense
  - ...
- in saying "Design Patterns", we mean "design" in the sense common to buildings architecture and SW development:
  - work phase "between" study/analysis and "actual building" (not temporally;-)
- (SWers use "architecture" differently;-)

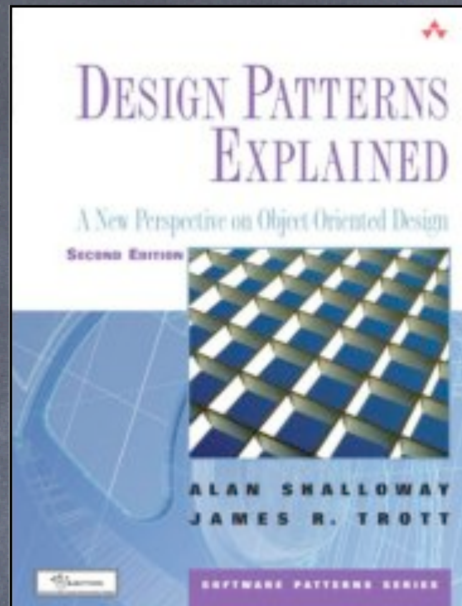
Other kinds of Patterns

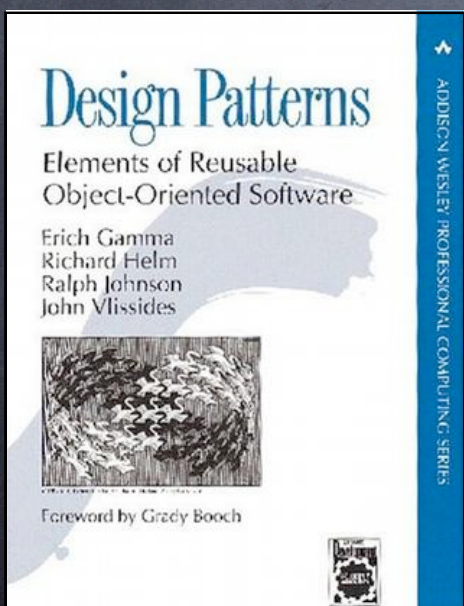
- Analysis: find/identify value-opportunities
- Architecture: large-scale overall-system approaches to let subsystems cooperate
- Human Experience: focus on how a system presents itself and interacts with people
- Testing: how best to verify system quality
- Cooperation: how to help people work together productively to deliver value
- Delivery/Deployment: how to put the system in place (& adjust it iteratively)
- ...

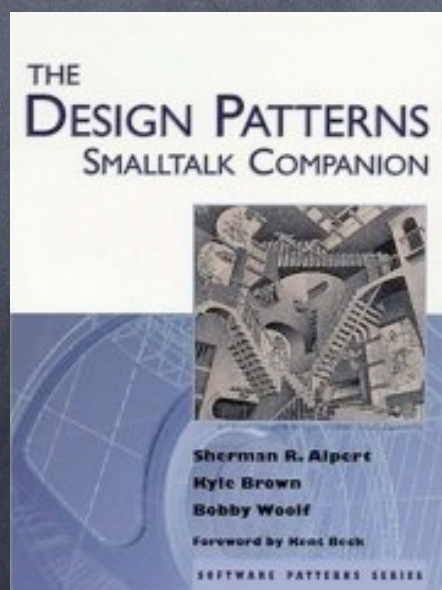
What's a "Pythonic" Pattern?

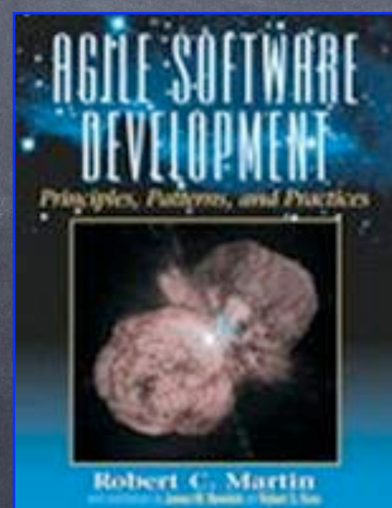
- a Design Pattern arising in contexts where (part of) the technology in use is Python
- well-adapted to Python's strengths, if and when those strengths are useful
- dealing with Python-specific issues, if any
- basically, all the rest of this tutorial!

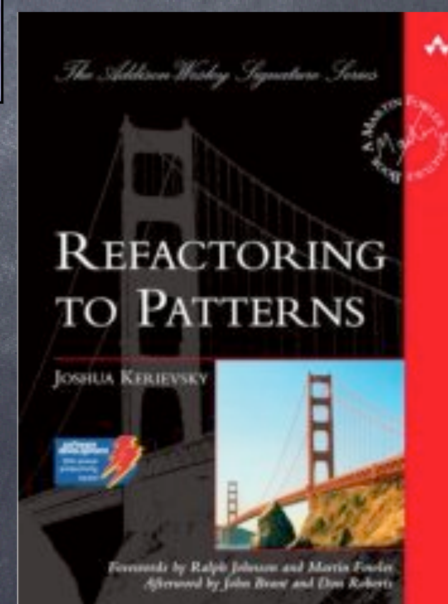
Many Good DP Books











(biblio on the last slide)

Classic (Gof4) DP Categories

- Creational: ways and means of object instantiation
- Structural: mutual composition of classes or objects (the Facade DP is Structural)
- Behavioral: how classes or objects interact and distribute responsibilities among them
- Each can be class-level or object-level

Prolegomena to DPs

- "program to an interface, not to an implementation"
  - that's mostly done with "duck typing" in Python -- rarely w/"formal" interfaces
    - in 2.6+, ABCs can change that a bit!
  - pretty similar to "signature-based polymorphism" in C++ templates

Duck Typing Helps!

Teaching the ducks to type takes a while, but saves you a lot of work afterwards!-)

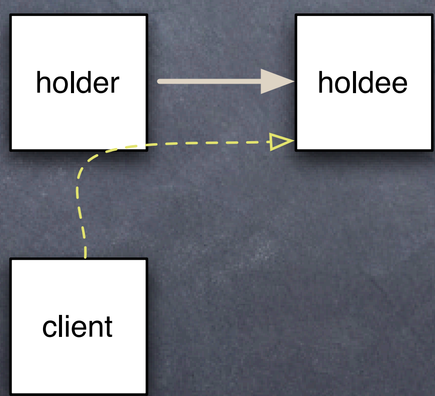
Prolegomena to DPs

- "favor object composition over class inheritance"
  - in Python: hold, or wrap
  - inherit only when it's really convenient
    - expose all methods in base class (reuse + usually override + maybe extend)
    - but, it's a very strong coupling!
    - 2.6+ ABCs can help with this, too

Python: hold or wrap?

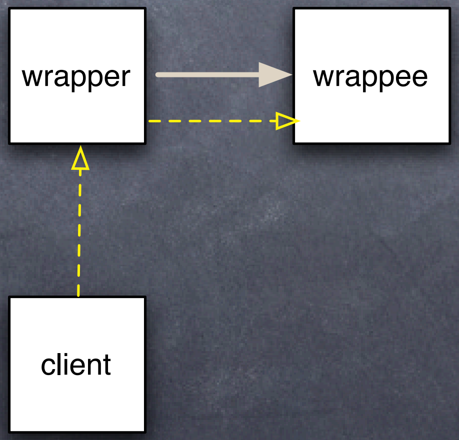
Python: hold or wrap?

- "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



Python: hold or wrap?

- "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)

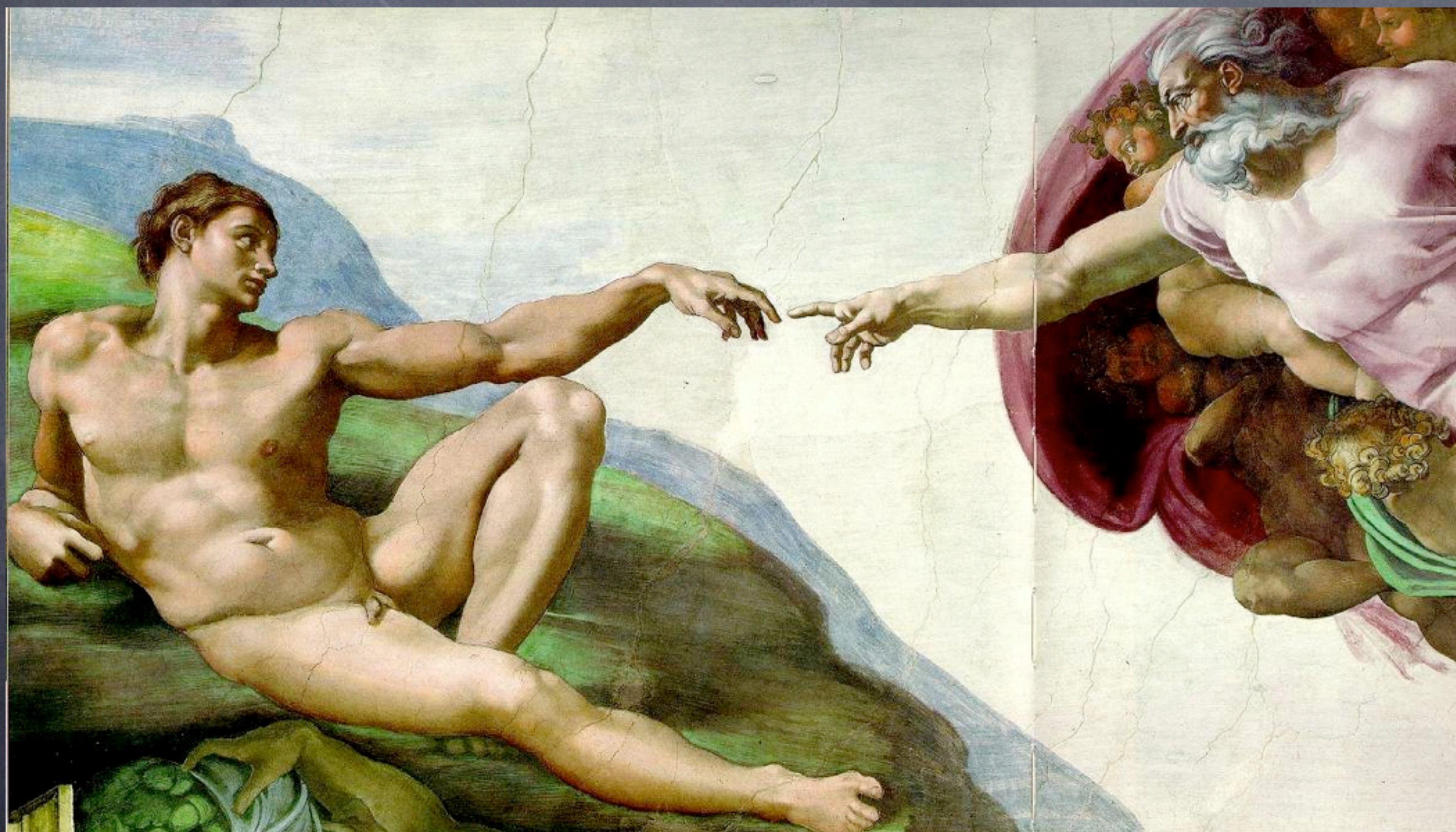


E.g: wrap to "restrict"

```
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)
  ...
```

<u>Inheritance</u> cannot restrict!

Creational Patterns



Creational DPs: "Just One"

- "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 can never be 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 always 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 do...:

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

data overriding to the rescue!

Creational DPs: "Flexibility"

- "we don't want to commit to instantiating a specific concrete class"
  - "Dependency Injection" DP
    - 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)
    - factory classes (abstract & not)

DI: why we want it

```
class Scheduler(object):
  def __init__(self):
    self.i = itertools.count().next
    self.q = somemodule.PriorityQueue()
  def AddEvent(self, when, c, *a, **k):
    self.q.push((when, self.i(), c, a, k))
  def Run(self):
    while self.q:
      when, n, c, a, k = self.q.pop()
      time.sleep(when - time.time())
      c(*a, **k)
```

Side note...:

```
class PriorityQueue(object):
  def __init__(self):
    self.l = []
  def __len__(self):
    return len(self.l)
  def push(self, obj):
    heapq.heappush(self.l, obj)
  def pop(self):
    return heapq.heappop(self.l)
```

Fine, but...

- ...how to test Scheduler without long waits?
- ...how to integrate it with other subsystems' event loops, simulations, ...?

Core issue: Scheduler "concretely depends" on concrete objects (time.sleep, time.time).

Possible solutions:

1. Template Method (Structural, see later)
2. "Monkey Patching" (idiom)
3. Dependency Injection

Template Method vs DI

See later, but, a summary:

```
    ...
    when, n, c, a, k = self.q.pop()
    self.WaitFor(when)
    c(*a, **k)
    ...
def WaitFor(self, when):
  time.sleep(when - time.time())
```

To customize: subclass, override WaitFor

TM-vs-DI example

```
class sq(ss):
  def __init__(self):
    ss.__init__(self)
    ss.mtq = Queue.Queue()
  def WaitFor(self, when):
    try:
      while when>time.time():
        c, a, k = self.mtq.get(true,
                    time.time() - when)
        c(*a, **k)
    except Queue.Empty:
        return
```

TM-vs-DI issues

- inheritance → strong, inflexible coupling
  - per-class complex, specialized extra logic
- not ideal for testing
  - if another subsystem makes a scheduler, how does it know to make a test-scheduler instance vs a simple one?
- multiple integrations even harder than need be (but, there's no magic bullet for those!-)

Monkey-patching...

```
import ss
class faker(object): pass
fake = faker()
ss.time = fake
fake.sleep = ...
fake.time = ...
```

- handy in emergencies, but...
- ...easily abused for NON-emergencies!
  - "gives dynamic languages a bad name"!-)
- subtle, hidden "communication" via secret, obscure pathways (explicit is better!-)

Dependency Injection

```
class Scheduler(object):
  def __init__(self, tm=time.time,
                     sl=time.sleep):
    self.tm = tm
    self.sl = sl
  ...
      self.sl(when - self.tm())
```

- a known use: standard library sched module!

With DI, "faking" is easy

```
class faketime(object):
  def __init__(self, t=0.0): self.t = t
  def time(self): return self.t
  def sleep(self, t): self.t += t

f = faketime()
s = Scheduler(f.time, f.sleep)
...
```

DI/TM "coopetition"

Not mutually exclusive...:

```
class Scheduler(object):
  def __init__(self, tm=time.time,
                     sl=time.sleep):
  ... # move key operation to a method:
  def WaitFor(self, when):
    self.sl(when-self.tm())
```

then may use either injection, or subclassing and overriding, (or both!-), for testing, integration, &c

DI design-choice details

- inject by constructor (as shown before)
  - with, or without, default dep. values?
  - ensure just-made instance is consistent
  - choose how "visible" to make the inject...
- inject by setter
  - automatic in Python (use non-_ names)
  - very flexible (sometimes too much;-)
- "inject by interface" (AKA "IoC type 1")
  - not very relevant to Python
- DI: by code or by config-file/flags?

DI and factories

```
class ts(object):
  ...
  def Delegate(self, c, a, k):
    q = Queue.Queue()
    def f(): q.put(c(*a,**k))
    t = threading.Thread(target=f)
    t.start()
    return q
```

- each call to Delegate needs a new Queue and a new Thread; how do we DI these objects...?
- easy solution: inject factories for them!

DI and factories

```
class ts(object):
  def __init__(self, q=Queue.Queue,
                     t=threading.Thread):
    self.q = q
    self.t = t
  ...
  def Delegate(self, c, a, k):
    q = self.q()
    ...
    t = self.t(target=f)
```

- pretty obvious/trivial solution when each class is a factory for its instances, of course;-)

The Callback Pattern

- AKA "the Hollywood Principle"...:
  - "Don't call us, we'll call you!"

The "Callback" concept

- it's all about library/framework code that "calls back" into YOUR code
  - rather than the "traditional" (procedural) approach where YOU call code supplied as entry points by libraries &c
- AKA, the "Hollywood principle":
  - "don't call us, we'll call you"
    - by: Richard E. Sweet, in "The Mesa Programming Environment", SigPLAN Notices, July 1985
- for: customization (flexibility) and "event-driven" architectures ("actual" events OR "structuring of control-flow" ["pseudo" events])

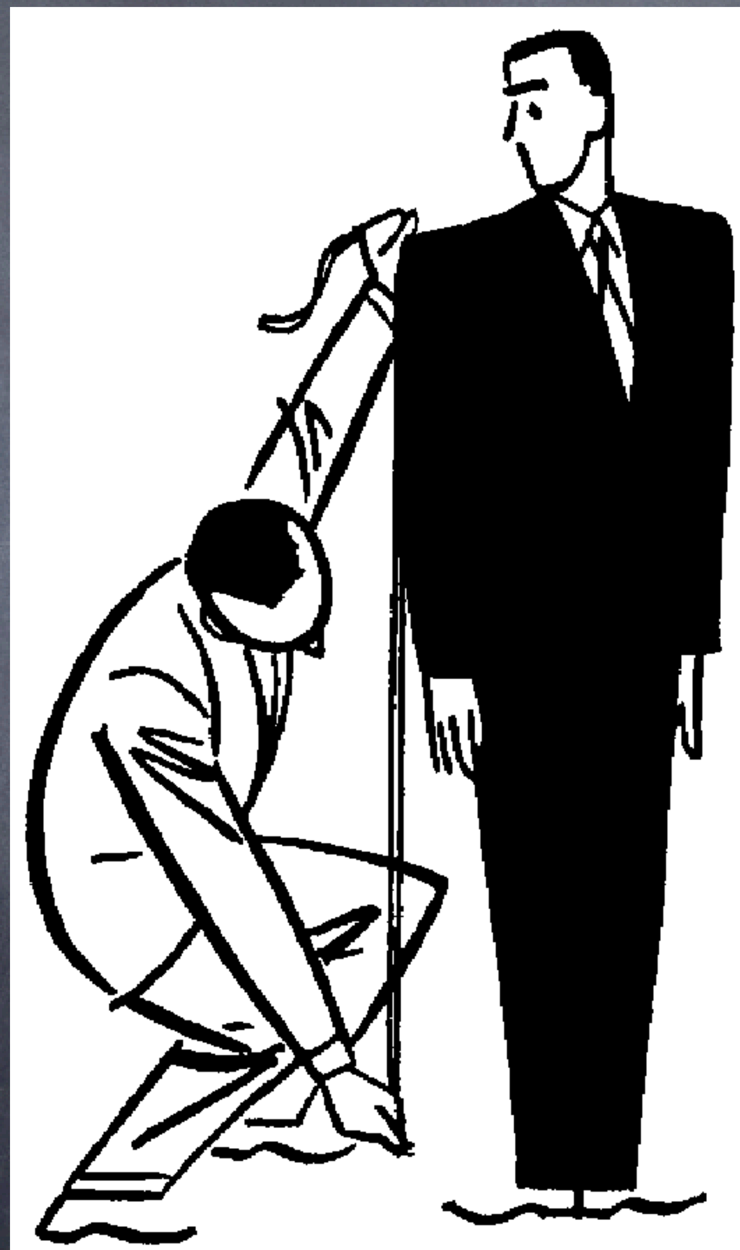
"Callback" implementation

- hand a callable over to "somebody"
- the "somebody" may store it "somewhere"
  - a container, an attribute, whatever
  - or even just keep it as a local variable
- and calls it "when appropriate"
  - when it needs some specific functionality (i.e., for customization)
  - or, when appropriate events "occur" (state changes, user actions, network or other I/O, timeouts, system events, ...) or "are made up" (structuring of control-flow)

Lazy-loading Callbacks

```
class LazyCallable(object):
  def __init__(self, name):
    self.n, self.f = name, None
  def __call__(self, *a, **k):
    if self.f is None:
      modn, funcn = self.n.rsplit('.', 1)
      if modn not in sys.modules:
        __import__(modn)
      self.f = getattr(sys.modules[modn],
                       funcn)
    self.f(*a, **k)
```

Customization



Customizing sort (by key)

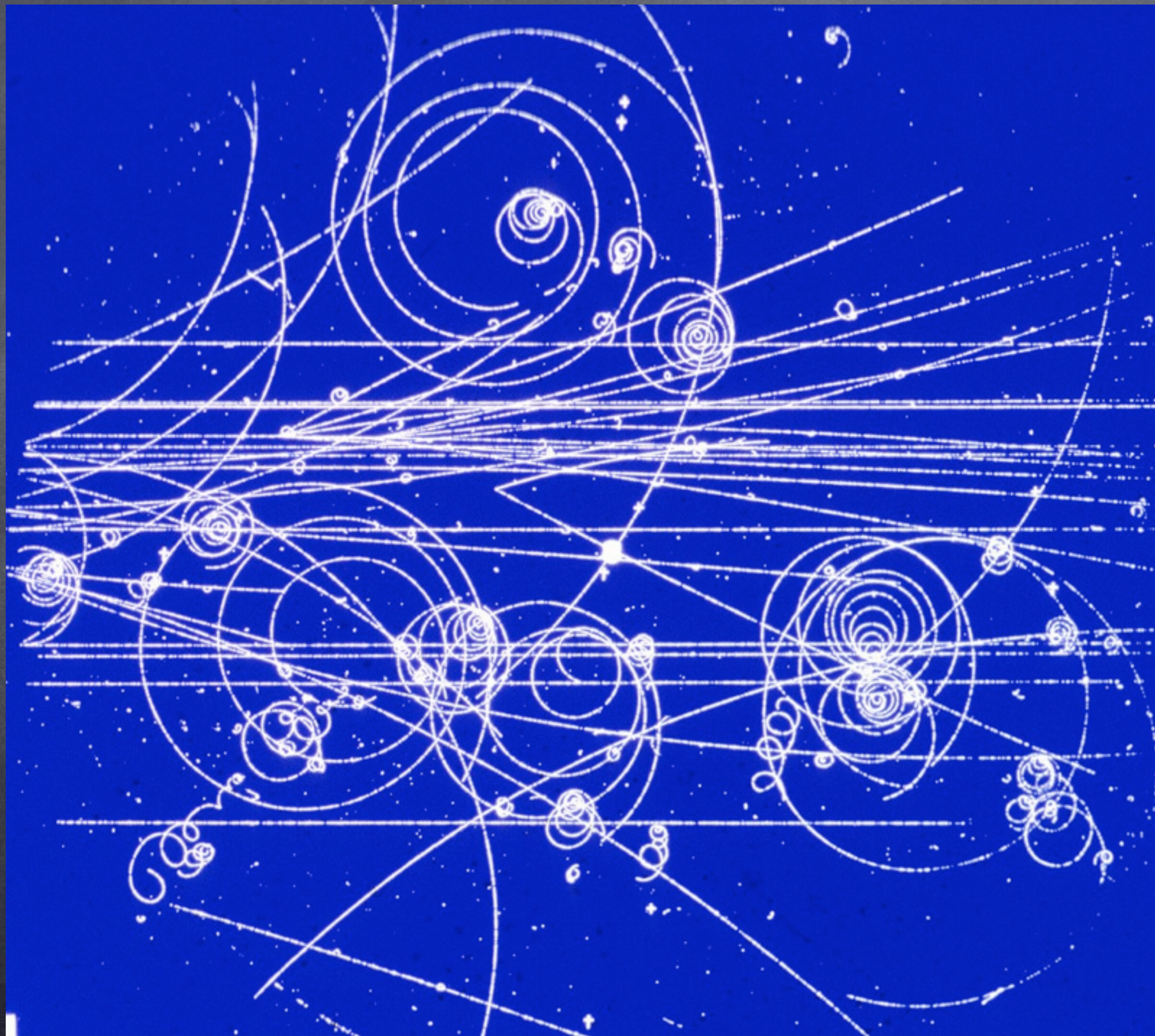
```
mylist.sort(key=str.toupper)
```

handily, speedily embodies the DSU pattern:

```
def DSU_sort(mylist, key):
  aux = [ (key(v), j, v)
           for j, v in enumerate(mylist)]
  aux.sort()
  mylist[:] = [v for k, j, v in aux]
```

Note that a little "workaround" is needed wrt the usual "call a method on each object" OO idiom...

Events



Kinds of "Event" callbacks

- Events "proper"...:
  - GUI frameworks (mouse, keyboard, ...)
  - Observer/Observable design pattern
  - asynchronous (event-driven) I/O (net &c)
  - "system-event" callbacks
- Pseudo-events for "structuring" execution:
  - "event-driven" parsing (SAX &c)
  - "scheduled" callbacks (sched)
  - "concurrent" callbacks (threads &c)
  - timing and debugging (timeit, pdb, ...)

Events in GUI frameworks

- the most classic of event-driven fields
- e.g, consider Tkinter:
- elementary callbacks e.g. for buttons:
  - b=Button(parent, text='boo!', command=...)
- flexible, advanced callbacks and events:
  - wgt.bind(event, handler)
    - event: string describing the event (e.g. '<Enter>', '<Leave>', '<Key>', ...)
    - handler: callable taking Event argument (w. attributes .widget, .x, .y, .type, ...)
  - can also bind by class, all, root window...

The Observer DP

- a "target object" lets you add "observers"
  - could be simple callables, or objects
  - object == "collection of callable"
- when the target's state changes, it calls back to "let the observers know"
- design choices: "general" observers (callbacks on ANY state change), "specific" observers (callbacks on SPECIFIC state changes; level of specificity may vary), "grouped" observers (objects with >1 methods for kinds of state-change), ...

Callback issues

- what arguments are to be used on the call?
  - no arguments: simplest, a bit "rough"
  - in Observer: pass as argument the target object whose state just changed
    - lets 1 callable observe several targets
  - or: a "description" of the state changes
    - saves "round-trips" to obtain them
  - other: identifier or description of event
- but -- what about other arguments (related to the callable, not to the target/event)...?

Fixed args in callbacks

- functools.partial(callable, *a, **kw)
  - pre-bind any or all arguments
- however, note the difference...:
  - x.setCbk(functools.partial(f, *a, **kw))
    - vs
  - x.setCbk(f, *a, **kw)
- ...having the set-callback itself accept (and pre-bind) arguments is a neater idiom
- sombunall[1] Python callback systems use it

[1]: Robert Anton Wilson

Callback "dispatching"

- what if more than one callback is set for a single event (or, Observable target)?
  - remember and call the latest one only
    - simplest, roughest
  - or, remember and call them all
    - LIFO? FIFO? or...?
    - how do you _remove_ a callback?
    - can one callback "preempt" others?
- can events (or state changes) be "grouped"?
  - use object w/methods instead of callable

Callbacks and Errors

- are "errors" events like any others?
- or are they best singled-out?
  http://www.python.org/pycon/papers/deferex/
- Twisted Matrix's "Deferred" pattern: one Deferred object holds...
  - N "chained" callbacks for "successes" +
  - M "chained" callbacks for "errors"
  - each callback is held WITH opt *a, **kw
  - plus, argument for "event / error identification" (or, result of previous callback along the appropriate "chain")

System-events callbacks

- for various Python "system-events":
  - atexit.register(callable, *a, **k)
  - oldhandler = signal.signal(signum, callable)
  - sys.displayhook, sys.excepthook, sys.settrace(callable), sys.setprofile(callable)
- some extension modules do that, too...:
  - readline.set_startup_hook, set_pre_input_hook, set_completer

"Pseudo" events

- "events" can be a nice way to structure execution (control) flow
  - so in some cases "we make them up" (!) just to allow even-driven callbacks in otherwise non-obvious situations;-)
- parsing, scheduling, concurrency, timing, debugging, ...

Event-driven parsing

- e.g. SAX for XML
  - "events" are start and end of tags
  - handlers are responsible for keeping stack or other structure as needed
    - often not necessary to keep all...!
- at the other extreme: XML's DOM
- somewhere in-between: "pull DOM"...
  - events as "stream" rather than callback
  - can "expand node" for DOMy subtrees

Scheduled callbacks

- standard library module sched
- s = sched.Sched(timefunc, delayfunc)
  - e.g, Sched(time.time, time.sleep)
- evt = s.enter(delay, priority, callable, arg)
  - or s.enterabs(time, priority, callable, arg)
  - may s.cancel(evt) later
- s.run() runs events until queue is empty (or an exception is raised in callable or delayfunc: it propagates but leaves s in stable state, s.run can be called again later)

"Concurrent" callbacks

- threading.Thread(target=..,args=..,kwargs=..)
  - call backs to target(*args,**kwargs)
  - at the t.start() event [or later...!]
  - *in a separate thread* (the key point!-)
- multiprocessing.Process
- stackless: stacklet.tasklet(callable)
  - calls back according to setup
  - when tasklet active and front-of-queue
  - channels, reactivation, rescheduling

Timing and debugging

- timeit.Timer(stmt, setup)
  - *string* arguments to compile & execute
  - a dynamic-language twist on callback!-)
  - "event" for callback:
    - setup: once, before anything else
    - stmt: many times, for timing
- the pdb debugger module lets you use either strings or callables...:
  - pdb.run and .runeval: strings
  - pdb.runcall: callable, arguments

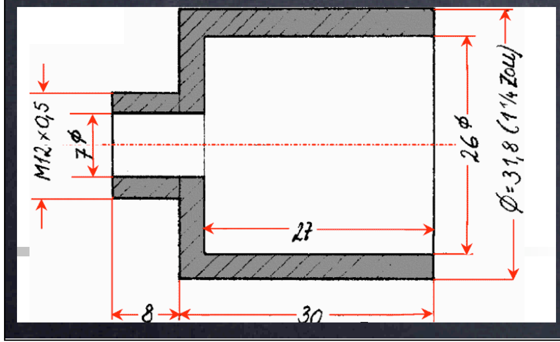
Structural Patterns

"Masquerading/Adaptation" subcategory:

- Adapter: tweak an interface (both class and object variants exist)
- Facade: simplify a subsystem's interface
- ...and many others I don't cover, such as:
  - Bridge: many implementations of an abstraction, many implementations of a functionality, no 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...)
```

Class Adapter (mixin)

- flexible, good use of multiple inheritance:

```
class BF2FB:
  def foobar(self, foo, bar):
    return self.barfoo(bar, foo)

class Foobarer(BF2FB, Barfooer):
  pass

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!-) -- often _callables_ are adapted (via decorators and other HOFs, closures, functools, ...)

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 a subsystem made up of many classes/objects, Adapter "front" for just one single object or class

Behavioral Patterns

- Template Method: self-delegation
  - ..."the essence of OOP"...
  - some of its many Python-specific variants

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!-)

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): ...
```

KU: cmd.Cmd.cmdloop

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

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.

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()
```

A Priority/FIFO Queue

```
class PriorityQueue(Queue):
  def _init(self, maxsize):
    self.maxsize = maxsize
    self.q = list()
    self._n = 0
  def put(self, priority, item):
    Queue.put(self, (priority, item))
  def _put(self, (p,i)):
    self._n += 1
    heapq.heappush(self.q, (p,self._n,i))
  def _get(self):
    return heapq.heappop(self.q)[-1]
```

"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)
- very handy for event-driven programming when you can't (or do not want to...!) "predict" all possible events in the ABC (e.g., event-driven parsing of HTML or XML)

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)
```

A multi-style TM case

- classic + factored + introspective
  - multiple "axes" to separate three carefully distinguished "variabilities"
- DP equivalent of a "3-Subjects Fugue"
  - "all arts aspires to the condition of Music" (Pater, Pound, Santayana...?-)

UC: unittest.TestCase

```
def__call__(self, result):
  method = getattr(self, ...)
  try: self.setUp()
  except: result.addError(...)
  try: method()
  except self.failException, e:...
  try: self.tearDown()
  except: result.addError(...)
  ...result.addSuccess(...)...
```

KU: ABCs

Simple example, collections.Sequence:

```
class Sequence(Sized,Iterable,Container):
  def count(self, value):
    the_count = 0
    for item in self:
      if item == value:
        the_count += 1
    return the_count
  ...
```

See also module abc.

Questions & Answers

http://www.aleax.it/oscon010_pydp.pdf

Q?

A!

1. Design Patterns: Elements of Reusable Object-Oriented Software -- Gamma, Helms, Johnson, Vlissides -- advanced, very deep, THE classic "Gang of 4" book that started it all (C++)
2. Head First Design Patterns -- Freeman -- introductory, fast-paced, very hands-on (Java)
3. Design Patterns Explained -- Shalloway, Trott -- introductory, mix of examples, reasoning and explanation (Java)
4. The Design Patterns Smalltalk Companion -- Alpert, Brown, Woolf -- intermediate, very language-specific (Smalltalk)
5. Agile Software Development, Principles, Patterns and Practices -- Martin -- intermediate, extremely practical, great mix of theory and practice (Java, C++)
6. Refactoring to Patterns -- Kerievsky -- introductory, strong emphasis on refactoring existing code (Java)
7. Pattern Hatching, Design Patterns Applied -- Vlissides -- advanced, anecdotal, specific applications of ideas from the Gof4 book (C++)
8. Modern C++ Design: Generic Programming and Design Patterns Applied -- Alexandrescu -- advanced, very language specific (C++)