Masquerading and Adaptation Design Patterns in Python

Alex Martelli

STRAKT

This talk's audience...:

- "fair" to "excellent" grasp of Python and OO development
- "none" to "good" grasp of Design Patterns in general
- wants to learn more about: DP, masquerading, adaptation, DPs for Python, DP/language issues

Design Patterns

- rich, thriving subculture of the OO development culture
- Gamma, Helms, Johnson, Vlissides: "Design Patterns", Addison-Wesley 1995 ("gang of 4" == "Gof4")
- PLoP conferences & books

...but also...

- Design Patterns risked becoming a "fad" or "fashion" recently
  - cause: the usual, futile search for the "silver bullet"...!
  -

- let's not throw the design patterns out with the silver bullet!

© 2004 AB Strakt

STRAKT

DP myths and realities (1)

- DPs are **not** independent from language choice, because: design and implementation **must** interact (<u>no</u> to "waterfall"...!)
- in machine-code: "if", "while", "procedure" ... <u>are patterns</u>!
- HLLs embody these, so they are <u>not</u> patterns in HLLs

DP myths and realities (2)

- many DPs for Java/C++ are "workarounds for static typing"
- cfr Alpert, Brown, Woolf, "The DPs Smalltalk Companion" (AW)
- Pythonic patterns = classic ones, <u>minus</u> the WfST, <u>plus</u> (optionally) exploits of Python's strengths

DP myths and realities (3)

- formal-language presentation along a fixed schema **is** useful
- it is not indispensable
  - mostly a checklist "don't miss this"
  - and a help to experienced readers
- nor indeed always appropriate
  - always ask: who's the audience?

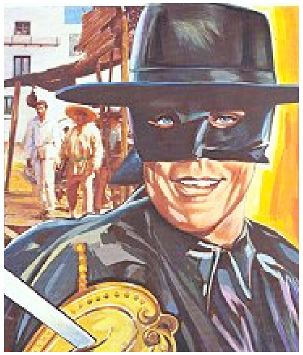
DP myths and realities (4)

- Design Patterns are **not** "silver bullets"
- they **are**, however, quite helpful IRL
- the **name** by itself already helps a lot!
  - like "that guy with the hair, you know, the Italian..."
  - vs "**Alex**"
- even when the DPs themselves dont help,
- **study** and **reflection** on them still does
  - "no battle plan ever survives contact wit the enemy"
  - and yet drawing up such plans is still indispensable

DP write-up components:

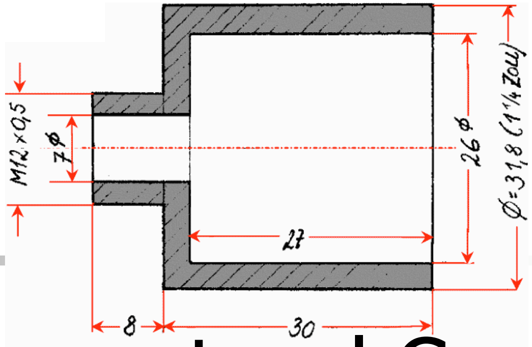
- **name**, context, problem
- forces, solution, (examples)
- results, (rationale), related DPs
- known uses: DPs are discovered, not invented!
- DPs are about description (and suggestion), not prescription

Two groups of structural DPs



- Masquerading: an object "pretends to be" (possibly fronts/proxies for...) another
- Adaptation: correct "impedance mismatches" between what's provided and what's required

DP "Adapter"



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

("interface" vs "protocol": "syntax" vs "syntax + semantics + pragmatics")

Python toy-example Adapter

- C requires: method `foobar(foo, bar)`
- S provides: method `barfoo(bar, foo)`
- a non-OO context is of course possible:

```
def foobar(foo,bar):
    return barfoo(bar,foo)
```

- in OO context, say we have available as σ:

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

Object Adapter

- per-instance, by wrapping & delegation:

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

foobarer = FobaringWrapper(barfooer)
```

Class Adapter

- per-class, by subclassing & self-delegation:

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

foobarer = Foobarer(some,init,parms)
```

Adapter: some known uses

- `shelve`: adapts "limited `dict`" (`str` keys and values, basic methods) to fuller `dict`:
  - non-str values via `pickle` + `UserDict.DictMixin`
- `socket._fileobject`: `socket` to filelike
  - has lot of code to implement buffering properly
- `doctest.DocTestSuite`: adapts doctest's tests to `unittest.TestSuite`
- `dbhash`: adapts `bsddb` to `dbm`
- `StringIO`: adapts `str` or `unicode` to filelike

Adapter observations

- real-life Adapters may require lots of code
- mixin classes help adapting to rich protocols (by implementing advanced methods on top of fundamental ones)
- Adapter occurs at all levels of complexity, from tiny dbhash to many bigger cases
- in Python, Adapter is <u>not</u> just about classes and their instances (by a long shot...)

DP "Facade"

- existing supplier code σ provides rich, complex functionality in protocol S
- we need a simpler "subset" C of S
- <u>facade</u> code Φ implements and supplies C (by calling S on σ)

Python toy-example Facade

```
class LifoStack:
    def __init__(self):
        self._stack = []
    def push(self, datum):
        self._stack.append(datum)
    def pop(self):
        return self._stack.pop()
```

Facade vs Adapter

- Adapter is mostly about supplying a "given" protocol required by client-code
  - (sometimes, it's about homogeinizing existing suppliers in order to gain polymorphism)
- Facade is mostly about simplifying a rich interface of which only a subset is needed
- of course they do "shade" into each other
- Facade often "fronts for" several objects, Adapter typically for just one

Facade: some known uses

- `asynchat.fifo` facades for `list`
- `dbhash` facades for `bsddb`
  - ...also given as Adapter known-use...!-)
- `sets.Set` mostly facades for `dict`
  - also adds some set-operations functionality
- `Queue` facades for `list + lock`
- `os.path`: `basename` and `dirname` facade for `split` + indexing; `isdir` &c facade for `os.stat + stat.S_ISDIR` &c

Facade observations

- real-life Facades may contain substantial code (simplifying the <u>protocol</u> is key...)
- interface-simplification is often mixed in with some small functional enrichments
- Facade occurs at all levels of complexity, from tiny `os.path.dirname` to richer cases
- inheritance is never really useful here (since it can only "widen", not "restrict")

Adapting/facading callables

- callables (functions, methods, ...) play a very large role in Python programming
  - they're first-class objects
  - Python doesn't force you to only use classes...!
- a frequently needed adaptation (may be seen as facade): pre-fix some arguments
- most often emerges in callback systems
- widely known as the "Currying" DP
  - not pedantically perfect, but then, DP naming...

"Currying" in Python

- typical case: `btn.setOnClick(acallable)`
  - will call `acallable()` [[maybe `acallable(evt)`]]
  - how do we make it call `foo(23)`?
  - `btn.setOnClick(lambda: foo(23))`

```
def curry(f, *a):
    def g(*b): return f(*(a+b))
    return g
```

  - `btn.setOnClick(curry(foo, 23))`
- best design...: `btn.setOnClick(foo, 23)`

Bridge

♠	AKQJT987		♠	AKQJT987
♥	62		♥	62
♦	54	vs	♦	543
♣	73		♣	7

- "The Bridge World" January and February 2000 issues, "How Shape Influences Strength" by A. Martelli

...oops!...

- ah, not **that** Bridge...?!

...that's [just a bit] more like it...

DP "Bridge"

- several (N1) realizations ρ of abstraction A,
- may each use any one of several (N2) implementations ι of functionality F
- we don't want to code N1 * N2 cases
- so we make abstract superclass A of all ρ hold a reference R to (an instance of) abstract superclass F of all ι, and...
- ...make each ρ use any functionality from F (thus, from a ι) only through R

Python toy-example Bridge

```
class AbstractParser:
  def __init__(self, scanner):
    self.scanner = scanner

class ExprParser(AbstractParser):
  def expr(self):
      ...t = self.scanner.next()...
      ...self.scanner.push_back(t)...
```

Pythonic peculiarities of Bridge

- often no real need for an abstract base class for the "implementation"
  - just rely on signature-based polymorphism
  - Python inheritance is mostly about handy code reuse
- each ρ can access self.R.amethod directly
- or you can proxy with A.amethod...:
  - def amethod(self,*a): return self.R.amethod(*a)
- then have each ρ access self.amethod
  - respects "Demeter's Law" ("only one dot")

Bridge: some known uses

- `htmllib`: `HTMLParser` → `Formatter`
  - but: not really meant for subclassing
- `formatter`: formatter → writer
  - `NullFormatter` / `AbstractFormatter` "unrelated"
  - `NullWriter` baseclass not technically "abstract" (provides empty implementations of methods)
- `xml.sax`: reader(parser) → handlers
  - multiple Bridge's -- one per handler
- `email`: `Parser -> Message`
  - holds class, not instance

Advanced known-use of Bridge

- `SocketServer` std library module:
- `BaseServer` is the abstraction
- `BaseRequestHandler` is the implementation abstract-superclass
- ...with some typical pythonic peculiarities:
  - also uses mix-ins (for threading, forking, ...)
  - A holds the very class F, instantiates it per-request, not just an instance of F

Bridge observations

- Bridge occurs mostly for substantially complex and rich cases
- inheritance used only occasionally in Python Bridge cases (and when used may be from a technically non-abstract class)
- often reference R is to class, not instance
    - affords easy repeated instantiation
    - no KU found, but: state might be kept in a Memento

Pydioms: Holder vs Wrapper

- Holder: object O has subobject S as an attribute (may be a property), that's all
  - use as `self.S.method` or `O.S.method`
- Wrapper: holder (often via a private attribute) plus delegation (use `O.method`)
  - explicit: `def method(self,*a):`

    ```
                return self._S.method(*A)
    ```
  - automatic (typically via `__getattr__`)...:

```
def __getattr__(self, name):
    return getattr(self._S, name)
```

STRAKT

Holder vs Wrapper + and -

- Holder: simpler, more direct and immediate
- low coupling (and doubtful cohesion...!) between O and S
- high coupling between O's clients and S (and O's internals...), lower flexibility
- Wrapper: slightly fancier, somewhat indirect
- high coupling (and hopefully cohesion...!) between O and S
  - automatic delegation helps with that

DP "Decorator"

- client code γ requires a certain protocol C
- supplier code σ provides exactly protocol C
- **however**, we also want to insert some small addition or semantic modification
  - quite possibly "pluggable" in/out during runtime
- decorator code δ "sneaks in the middle":
  - δ wraps σ, both consumes and produces C
  - may intercept, modify, (add a little), delegate, ...
  - γ uses δ, just as it would use σ

Python toy-example Decorator

```
class fullinesfile:
    def __init__(self, *a, **k):
        self.f = file(*a, **k)
        self.buf = ''
    def write(self, data):
        lns=(self.buf+data).splitlines(True)
        if lns[-1][-1]=='\n': self.buf=''
        else: self.buf = lns.pop(-1)
        self.f.writelines(lns)
```

Decorator: some known uses

- `gzip.GzipFile` decorates file with compression / decompression (using `zlib`)
- `multifile.MultiFile` decorates a MIME multipart file (each part read separately)
- `threading.RLock` decorates `thread.Lock` with re-entrancy (and "ownership" concept)
  - `Semaphore`, even `Condition`, also kinda decorators
- codecs stream classes decorate `file` with generic encoding and decoding

Decorator observations

- "pure" decorator (without some small additions to the protocol) is rare in Python
- file/stream objects are favourite targets for Python decorator uses
- Decorator typically occurs in reasonably simple cases
- dynamic on/off snap-ability not often used

DP "Proxy"

- client code ϒ would be just about fine with accessing some "true" object τ
- however, some kind of issue interferes:
  - we need to restrict access (e.g. for security)
  - object τ "lives" remotely or in some persisted form
  - we have lifetime/performance issues to solve
- proxy object π "sneaks in the middle":
  - π wraps τ, may create/delete it at need
  - may intercept, check calls, delegate, ...
  - γ uses π, just as it would use τ

Python toy-example Proxy

```
class ProxyFor:
    def __init__(self, cls, forb=(),*a,**k):
        self._m = cls, a, k; self._f = forb
    def __getattr__(self, name):
        if name in self._f:
            raise AttributeError
        if not hasattr(self, '_x'):
            cls, a, k = self._m
            self._x = cls(*a, **k)
        return getattr(self._x, name)
```

Proxy: some known uses

- `Bastion` used to proxy for any other object in a restricted-execution context
- `shelve.Shelf`'s values proxy for persisted objects (getting instantiated at-need)
- `xmlrpclib.ServerProxy` proxies for a remote server (not for a Python object...)
- `weakref.proxy` proxies for any existing object but doesn't "keep it alive"

Proxy observations

- a wide variety of motivations for use:
  - controlling access
  - remote or persisted objects
  - instantiating only at-need
  - other lifetime issues
- correspondingly wide range of variations
- Python's automatic delegation and "type agnosticism" make Proxy a real snap
- wrapping and proxying are quite close

Protocol Adaptation

- PEP 246
- any object might "embody" a protocol
  - e.g. Zope 3's zope.interface -- or *anything else, really*...
- adapt(component, protocol[, default])
  - checks if <u>component</u> directly implements protocol
  - checks if <u>protocol</u> knows how to adapt component
  - else falls back to a <u>registry</u> of adapters indexed by type(component) [[or otherwise, e.g. by URI]]
  - last ditch: returns default or raises an exception