Masquerading and Adaptation Design Patterns in Python

Alex Martelli <alex@strakt.com>

O'REILLY OPEN SOURCE CONVENTION

JULY 26–30, 2004
PORTLAND, OR

STRAKT

This talk's audience...:

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

STRAKT

What we'll cover...:

- Design Patterns, including "myths and realities"
- the Holder and Wrapper Python idioms
- the Adapter DP
- the Facade DP
- the Currying DP and Python callback systems
- the Decorator DP
- the Protocol Adaptation proposal (PEP 246)
- ...and, iff time allows...:
  - the Proxy DP
  - the Bridge DP

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

Design Patterns

- rich, thriving subculture of the OO development culture
- Gamma, Helms, Johnson, Vlissides, "Design Patterns", Addison-Wesley
  - more introductory: Shalloway, Trott, "Design Patterns Explained" (AW)
- 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!

DP myths and realities (1)

- DPs are **not** independent from language choice, because: design and implementation **must** interact (no to "waterfall"...!)
- in machine-code: "if", "while", "procedure" ... are patterns!
- HLLs embody these, so they are not 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 <u>not</u> indispensable
  - mostly a checklist "don't miss this"
  - and a help to experienced readers
- nor indeed always appropriate
  - always ask: <u>who's the audience?</u>

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

STRAKT

DPs: a summary assessment

- Design Patterns are **not** "silver bullets"
- they **are**, however, quite helpful IRL
- **naming**, 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 with the enemy"
  - and yet drawing up such plans is still indispensable

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

STRAKT

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

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,**k):`
    ```
                return self._S.method(*a,**k)
    ```
  - automatic (typically via `__getattr__`)...:

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


STRAKT

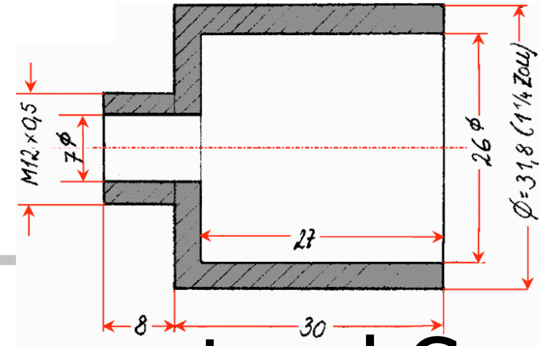
Holder vs Wrapper + and -

- Holder: simpler, more direct and immediate
  - low coupling O ↔ S
  - high coupling between O's <u>clients</u> and S (and O's internals...), lower flexibility
- Wrapper: slightly fancier, somewhat indirect
  - high coupling (and hopefully cohesion...!) O ↔ S
  - automatic delegation can help with that
- Wrapper helps respect "Demeter's Law"
  - basically "only one dot" (or, "not too many dots"!-)

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

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)

NB, "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,params)
```

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

STRAKT

Adapter observations

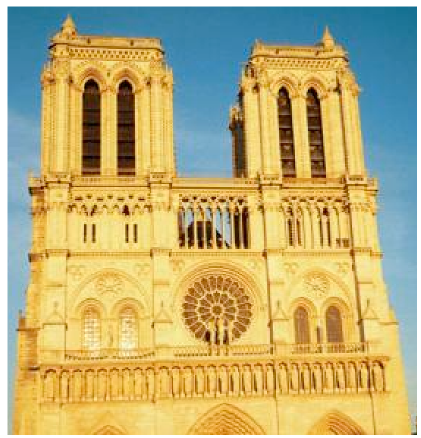
- real-life Adapters may require lots of code
- mixin classes help in adapting to rich protocols (by implementing advanced methods on top of fundamental ones)
- Adapter occurs at all levels of complexity, from tiny `dbhash` to many bigger cases
- in Python, Adapter is <u>not</u> just about classes and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

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

DP "Facade"

- existing supplier code σ provides rich, complex functionality in protocol S
- we need a simpler "subset" C of S
- facade code Φ "sneaks in front of" σ, implements and supplies C by calling S



STRAKT

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`
  - yes, I did also give this as an 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 (inheritance only "widens", can't "restrict")

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

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 a pedantically perfect name: DP names rarely are

"Currying" in Python

- the typical use case: for some object `btn`, `btn.setOnClick(acallable)...`
  - will call `acallable()` "when the button gets clicked"
  - but we have a function `def foo(anumber):` ...
  - how do we ensure a button click calls `foo(23)`?
  - `btn.setOnClick(lambda: foo(23))`
  - very similar issues if the callback is `acallable(evt)` and we want it to call `foo(evt, 23)`
- `lambda` **can** do any sig-adaptation, but...

Currying with a class

```
class Curry(object):
  def __init__(self, f, *a, **k):
    self.f, self.a, self.k = f, a, k
  def __call__(self, *b, **kk):
    d = self.k.copy()
    d.update(kk)
    return self.f(*(b+self.a), **d)

btn.setOnClick(Curry(foo, 23))
```

Currying with a closure

```
def curry(f, *a, **k):
  def curried(*b, **kk):
    d = k.copy()
    d.update(kk)
    return f(*(b+a), **d)
  return curried
btn.setOnClick(curry(foo, 23))
```

- `**k w/lambda` possible though a bit tricky:

```
lambda *b,**kk: f(*(b+a), **dict(k,**kk))
```

STRAKT

Currying-on-callback-setting

- best way to design callback-settings in Python: stash away extra args w/callable

```
def setOnClick(self, f, *a, **k): ...
```

- then just use as: `btn.setOnClick(foo, 23)`
- known uses:
  - `atexit.register`, `urllib.addclosehook`, `Tkinter.after`
  - `threading.Timer`, `sched.scheduler.enter`, `optparse.Option` (w/o `*`/`**` in signatures)

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

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
- <u>decorator</u> 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

STRAKT

Python toy-example Decorator

```
class uppercasingfile:
    def __init__(self, *a, **k):
        self.f = file(*a, **k)
    def write(self, data):
        self.f.write(data.upper())
    def __getattr__(self, name):
        return getattr(self.f, name)
```

Decorator: some known uses

- `gzip.GzipFile` decorates file with compression / decompression (using `zlib`)
- `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 in Python (we have other dynamisms...)

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

Protocol Adaptation (PEP 246)

- given protocol (type, interface, ...) P and object O, *who* knows how to adapt O to P...?
  - maybe O already "belongs to" / "implements" P, e.g. `isinstance(O, P)` when `type(P) is type`
  - maybe, given O's type/value, P can adapt O to itself, e.g. `P(O)` could return a suitable value or wrapper
  - maybe P and O know nothing about each other, but a 3rd-party factory `makePfromO(O)` could return a suitable "adapter to P" wrapper or value
- **why** should my *application* code care?!

What is P in PEP 246's context?

- could be anything, really
  - a type
  - a zope.interface
  - a PyProtocols' Protocol
  - ...something else again...
  - **it doesn't really matter all that much!**
- should **mean** a Protocol, **not** just an Interface (syntax+semantics+pragmatics, **not** just syntax such as method names and signatures)

PEP 246: the adapt function (1)

```
def adapt(obj, prot, default=None):
    """NB: approximate semantics only!"""
    if type(obj) is prot: return obj
    try: return _adapt_obj_prot(obj, prot)
    except TypeError: pass
    try: return _adapt_prot_obj(prot, obj)
    except TypeError: pass
    if isinstance(obj, prot): return obj
    return _adapt_byreg(obj, prot, default)
```

PEP 246: the adapt function (2)

```
def _adapt_obj_prot(obj, prot):
  c = getattr(obj, '__conform__', None)
  return c(prot)
def _adapt_prot_obj(prot, obj):
  a = getattr(prot, '__adapt__', None)
  return a(obj)
def _adapt_byreg(obj, prot, default):
  a = _adapt_regis.get((type(obj), prot))
  try: return a(obj, prot)
  except TypeError: return default
```

STRAKT

The Adaptation Registry

```
_adapt_regis = {}
def reg_adapt(atype, prot, factory):
  _adapt_regis[(atype,prot)] = factory
```

just for example... :

```
def by_coercion(obj, prot):
  return prot(obj)
reg_adapt(str, int, by_coercion)
# now adapt('23',int) is 23,
# but adapt('foo',int) raises
```

Protocol Adaptation usage

- Mr Xer writes function `X`, requiring an argument `a` which meets protocol `P`
  - but carefully uses `a=adapt(a,P)` on entry
- Ms Yer writes function `Y` returning an instance `q` of type `Q`
- Mr Zer writes an adapter factory `Z`, Q → P
- application writer Aer, once `Z` is registered, just calls `X(Y())` w/o a care in the world

A quote from PEP 246

"""

The typical Python programmer is an integrator, someone who is connecting components from various suppliers. Often the interfaces between these components require an intermediate adapter. Usually the burden falls upon the application programmer to study the interfaces exposed by one component and required by another, determine if they are directly compatible, or develop an adapter.

"""

Protocol Adaptation removes this burden!

Protocol Adaptation vs typecheck

```
def fooit(x):
    if isinstance(x,int): return fooI(x)
    elif isinstance(x,...
```

replace fooit's body with something like:

```
    adapt(x,fooer)(x)
```

with an initialization that goes roughly like:

```
class fooer: pass
def int_foo(x): return fooI(x)
reg_adapt(int,fooer,lambda *a:int_foo)
```

A PEP 246 trial implementation

- Phillip Eby's PyProtocols
  - http://peak.telecommunity.com/PyProtocols.html
- several nifty little extras wrt PEP 246
- spells `reg_adapt` as `declareAdapter`
- also supports `IBar(foo)` like `adapt(foo,IBar)`
  - only for instances of `protocols.interface` & `c`
- uses "metaprotocols" extensively
- **warning:** <u>transitive</u> adaptation
  - strong risk of "too much black magic"...

STRAKT

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

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
- <u>proxy</u> 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 JustInTimeCreator:
    def __init__(self, cls, *a, **k):
        self._m = cls, a, k
    def __getattr__(self, name):
        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

Next up...:

- Design Patterns, myths and realities
- Holder and Wrapper
- Adapter
- Facade
- Currying
- Decorator
- Protocol Adaptation (PEP 246)
- Proxy
- Bridge

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...

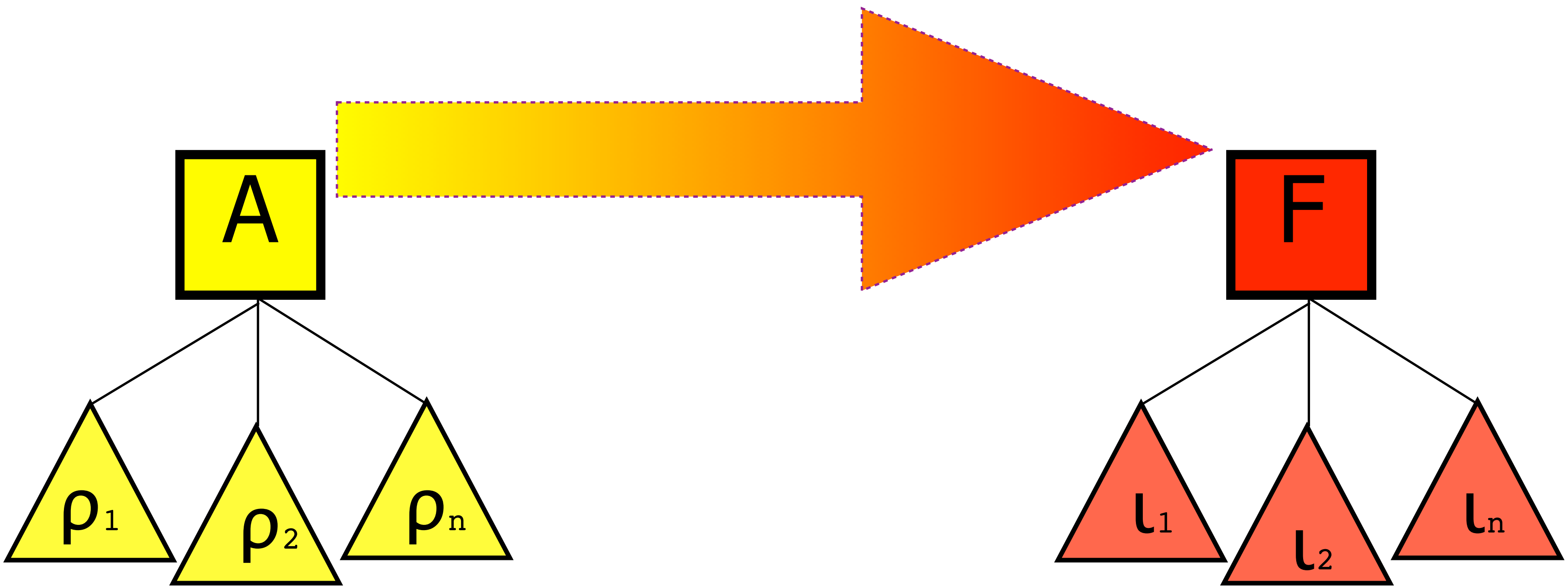
STRAKT

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

STRAKT

Why "bridge"?



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 <u>mostly</u> 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)
```

  - respects "Demeter's Law", see "Holder vs Wrapper"

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 <u>class</u> F, instantiates it per-request, not just an <u>instance</u> of F

Bridge observations

- Bridge occurs mostly for substantially complex and rich cases
- inheritance used only occasionally in Python Bridge cases
  - when used, may be from a not-truly-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