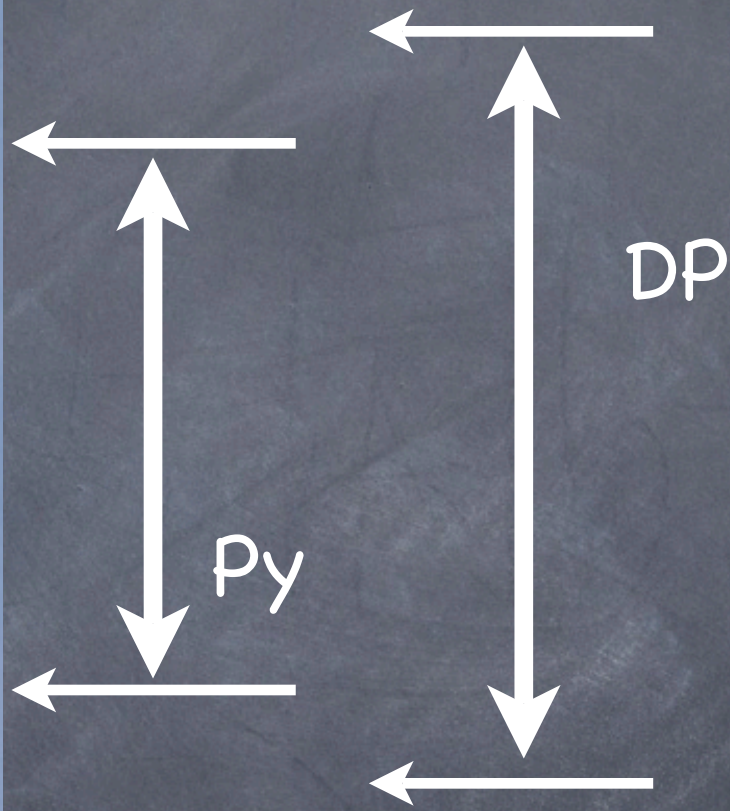
Design Patterns in Python

Alex Martelli (aleax@google.com)

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

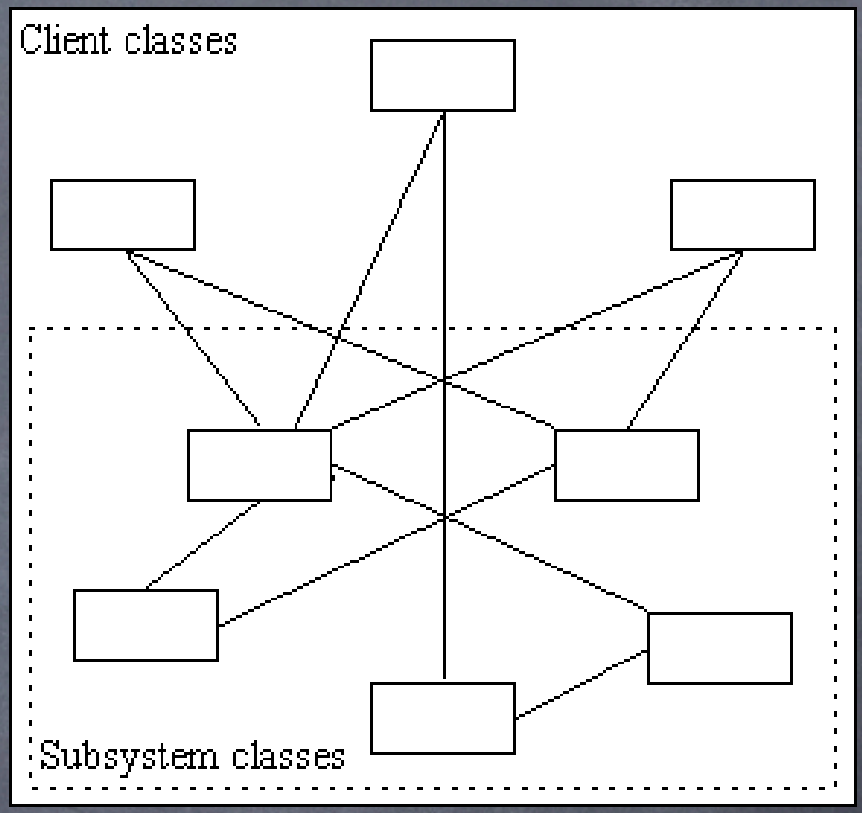
The "levels" of this talk



Google

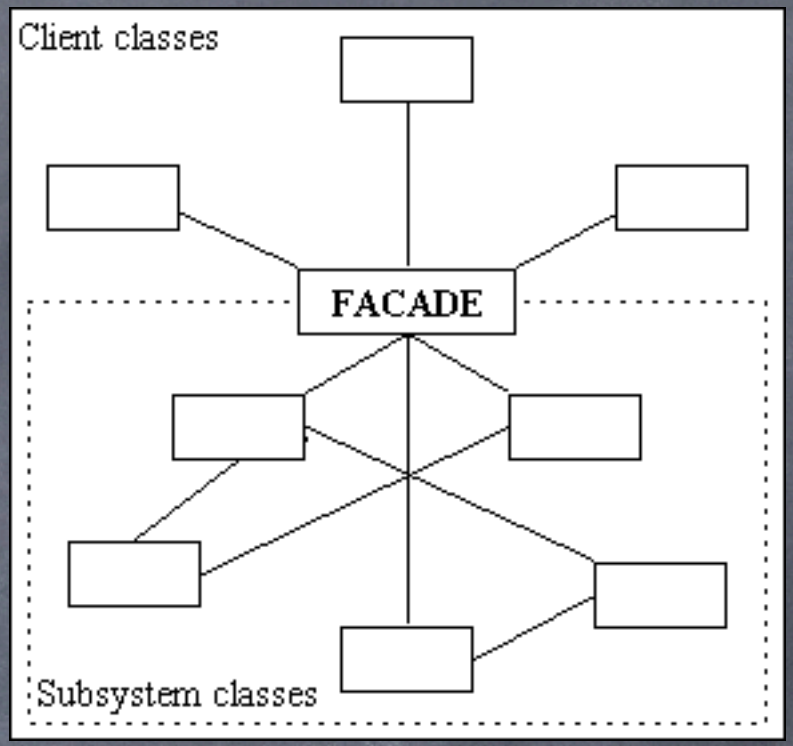
Hit the ground running...

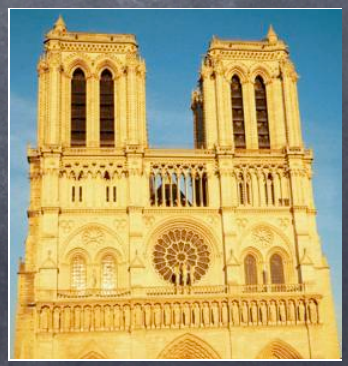
- "Forces": some rich, complex subsystem offers a lot of useful functionality; client code interacts with several parts of this functionality in a way that's "out of control"
- this causes many problems for client-code programmers AND subsystem ones too (complexity + rigidity)



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 implementation gains flexibility, clients gain simplicity





Facade is a Design Pattern

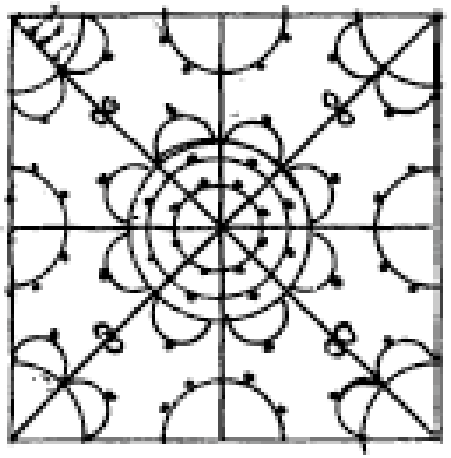
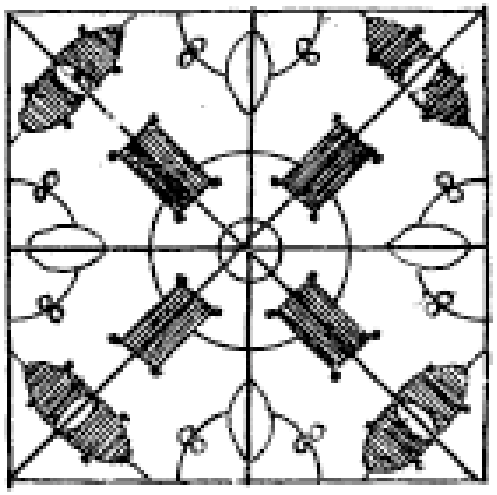
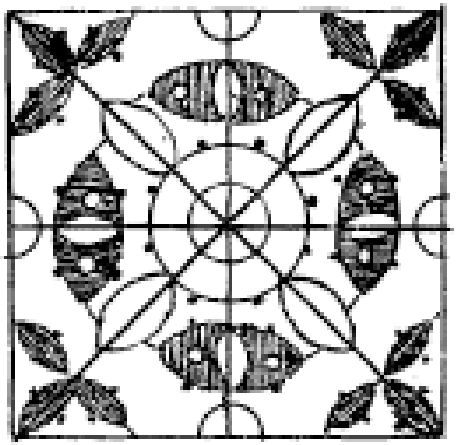
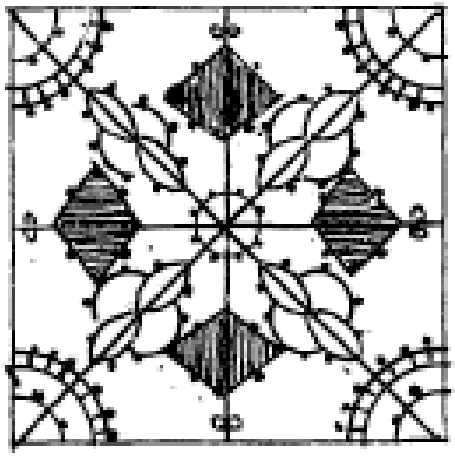
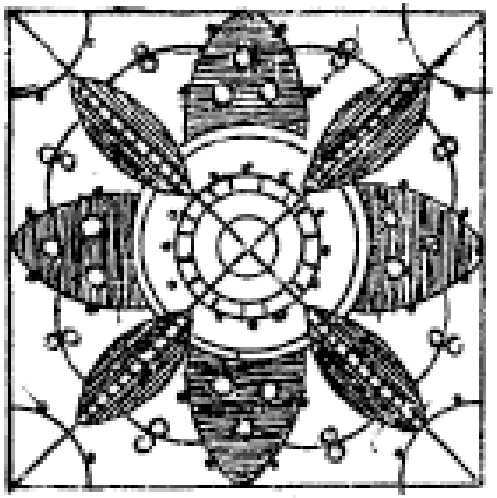
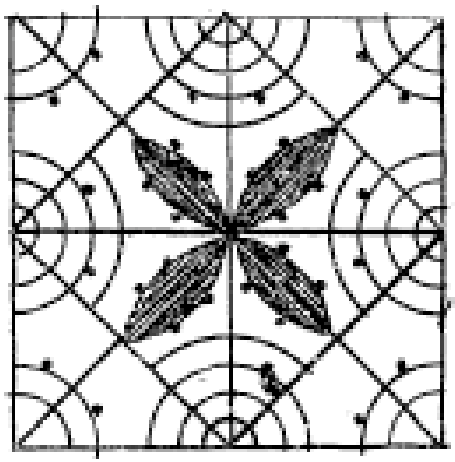
- summary of a frequent design problem + structure of a solution to that problem (+ pros and cons, alternatives, ...), and:
  - A NAME (much easier to retain/discuss!)
- "descriptions of communicating objects and classes customized to solve a general design problem in a particular context"
- that's NOT: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

Google

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"!-)

Design Patterns

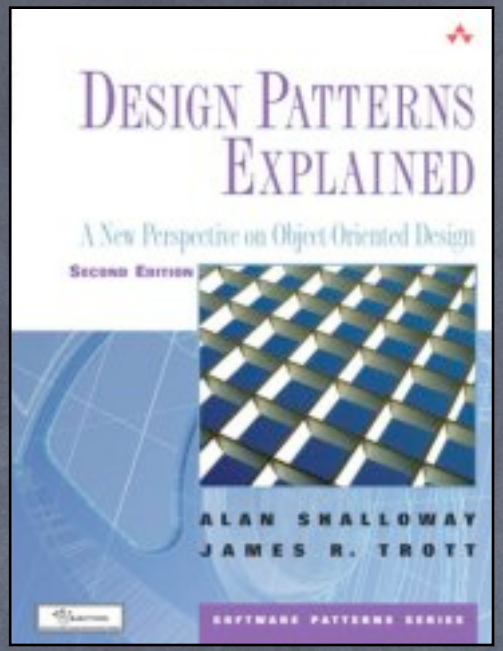







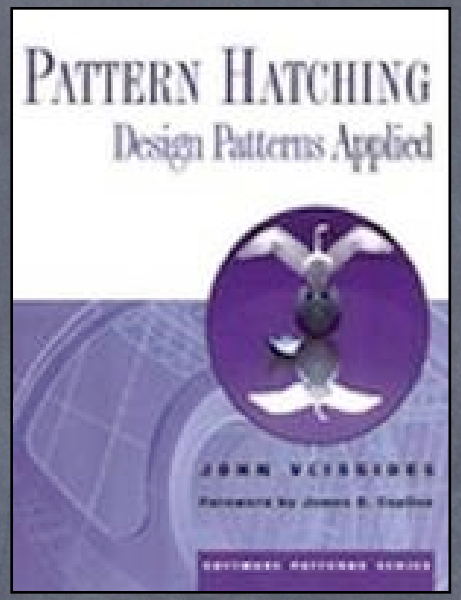
What's a Design Pattern

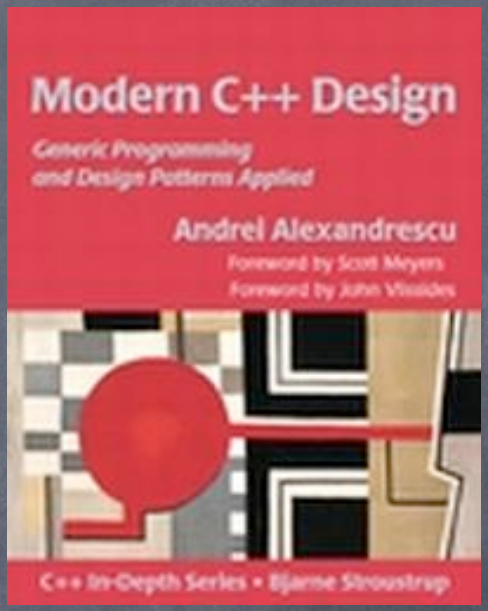
- summary of a frequent design problem + structure of a solution to that problem + pros and cons, alternatives, ..., and:
  - A NAME (much easier to retain/discuss!)
- "descriptions of communicating objects and classes customized to solve a general design problem in a particular context"
- DPs are NOT: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

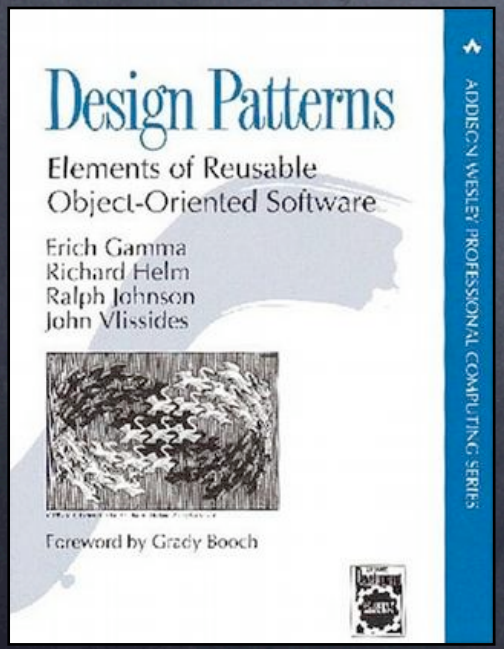
Google

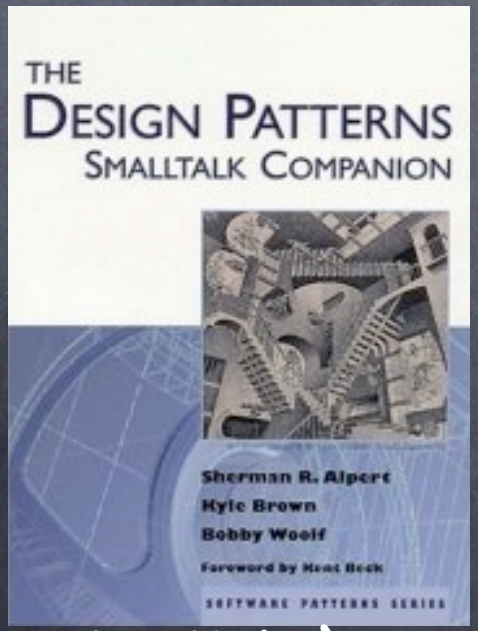
Many Good DP Books

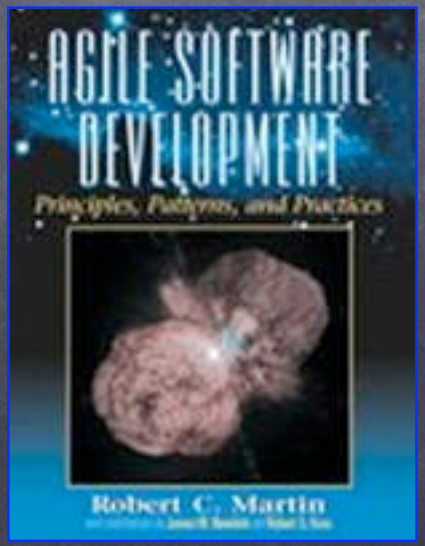


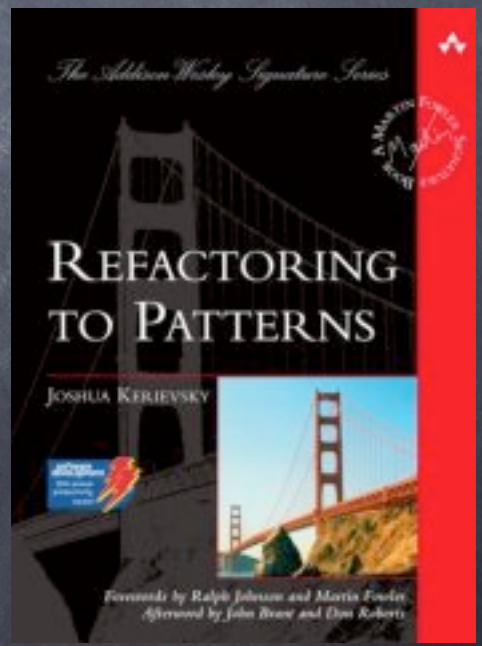












(biblio on the last slide)

Google

Classic 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 of the categories can be expressed at class-level or object-level in a given DP

Prolegomena to DPs [1]

- "program to an interface, not to an implementation"
  - that's mostly done with "duck typing" in Python -- rarely w/"formal" interfaces
  - actually similar to "signature-based polymorphism" in C++ templates
  - however, Python 3000 is adding ABCs, and some major Python frameworks (Zope, Twisted, PEAKS) already choose to use more "formal" interfaces anyway

But, Duck Typing Still Rocks!

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

Google

Prolegomena to DPs [2]

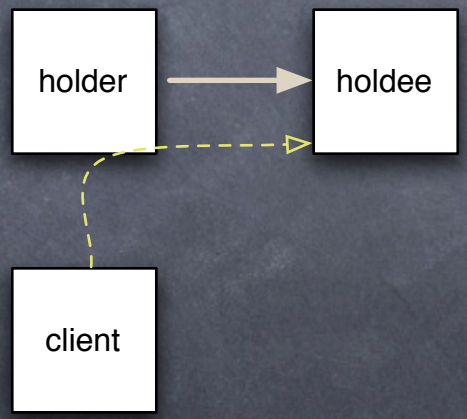
- "favor object composition over class inheritance"
  - in Python, basic idioms are hold and wrap
  - inherit only when it's really convenient
    - expose all methods in base class (reuse + usually override + maybe extend)
    - but, it's a very strong coupling!
  - object-level approaches are more fluid
    - (but: need class-level ones for special methods/operator overriding)

Python: Hold and Wrap

Google

The "Hold" Basic Idiom

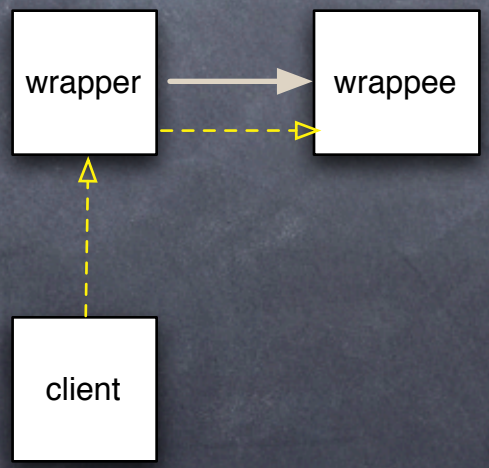
- "Hold": object O has subobject S as an attribute (maybe property) -- that's all
  - use self.S.method or O.S.method
  - simple, direct, immediate, but... pretty strong coupling, often on the wrong axis



Google

The "Wrap" Basic Idiom

- "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)



Google

E.g: wrap to "restrict"

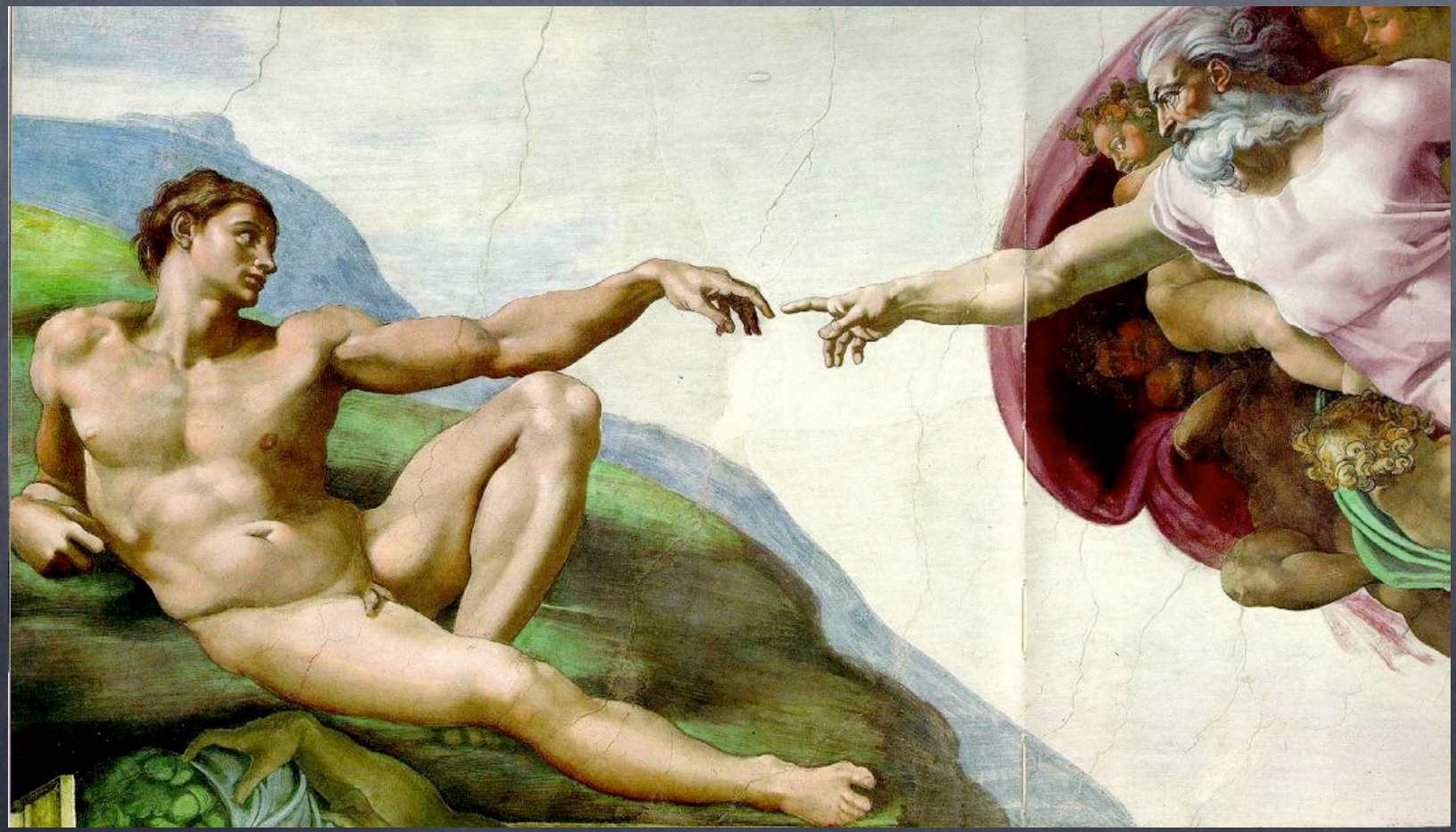
```
class RestrictingWrapper(object):
  def __init__(self, w, block):
    self._w = w
    self._block = block
  def __getattr__(self, n):
    if n in self._block:
      raise AttributeError, n
    return getattr(self._w, n)
```

...since inheritance cannot restrict!

(but: special method take a lot of work here)

Creational Patterns

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



Google

Creational Patterns [1]

- "we want just one instance to exist"

1. use a module instead of a class
   - no subclassing, no special methods, ...
2. make just 1 instance (no enforcement)
   - need to commit to "when" to make it
3. singleton ("highlander")
   - subclassing not really smooth
4. 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!

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")

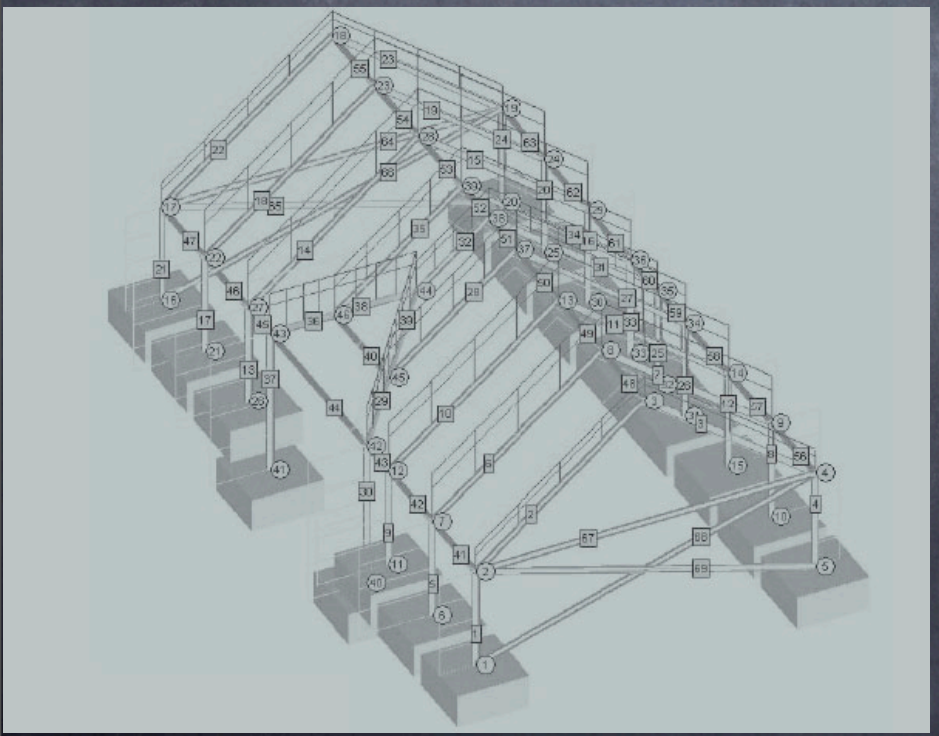
factory-function example

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

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

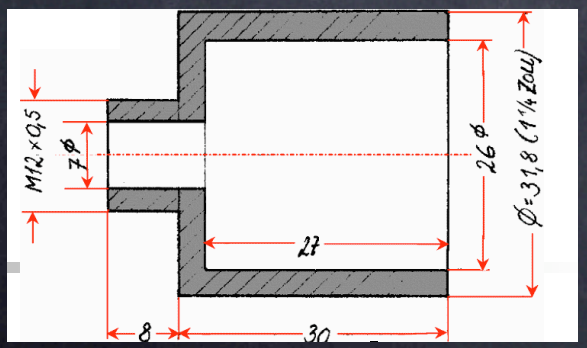
Structural Patterns

- focus on Adapter: tweak an interface (both class and object variants exist)
  - we've already seen Facade: simplify a subsystem's interface
  - (won't cover many other important ones, such as Proxy, Bridge, Decorator)



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 (direct)

- 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, ...)

Adapter vs Facade

- 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

- focus on Template Method: self-delegation
- (won't cover many, many important others)

Google

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 (injected, 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
  - embodies "Hollywood Principle": "don't call us, we'll call you" (framework vs lib)

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.

Google

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()
```

TM+Adapter: 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(...)...
```

Questions & Answers

Q?

A!

Google

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 idea from the Gof4 book (C++)

8.Modern C++ Design: Generic Programming and Design Patterns Applied -- Alexandrescu -- advanced, very language specific (C++)