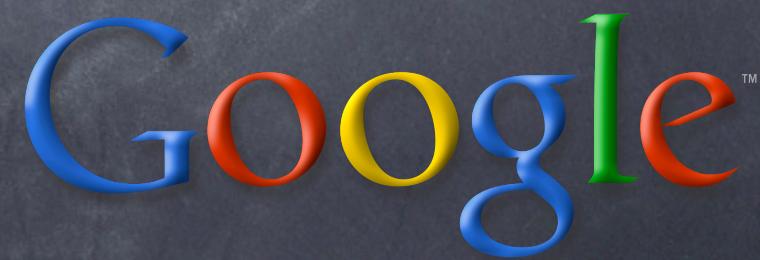


Design Patterns in Python

Alex Martelli (aleax@google.com)

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



Copyright ©2007, Google Inc

Questa presentazione...:

守

Shu

("Impara")

破

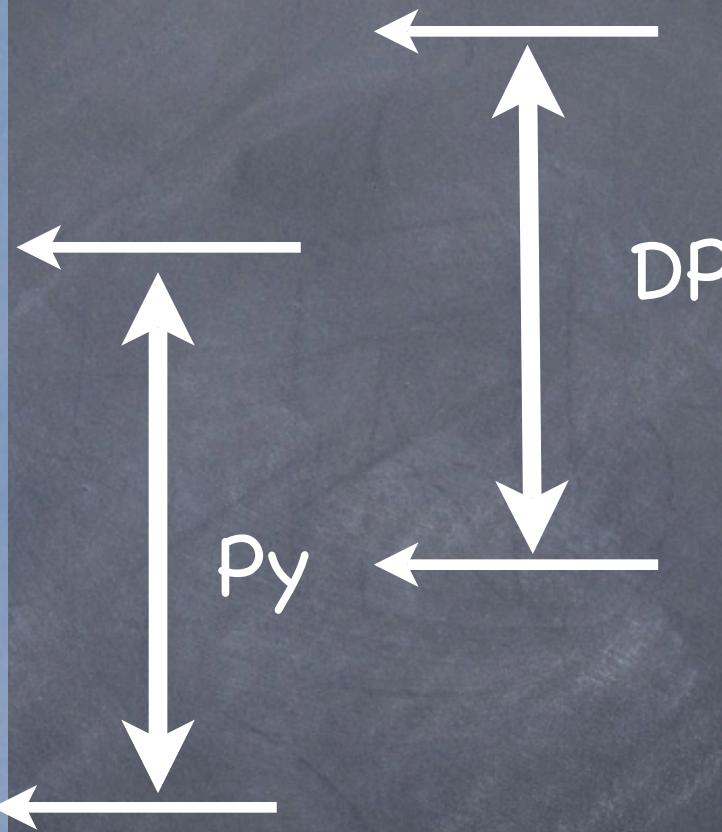
Ha

("Distacca")

離

Ri

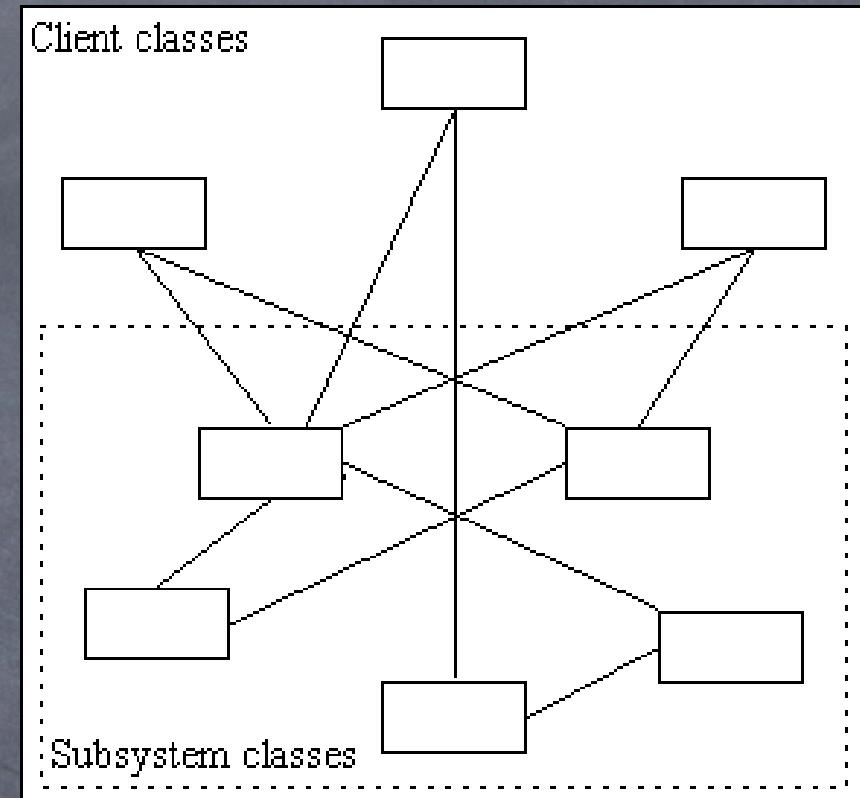
("Trascendi")



Tanto X cominciare...:

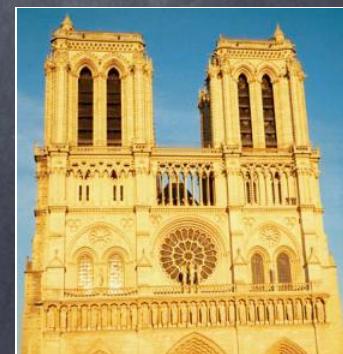
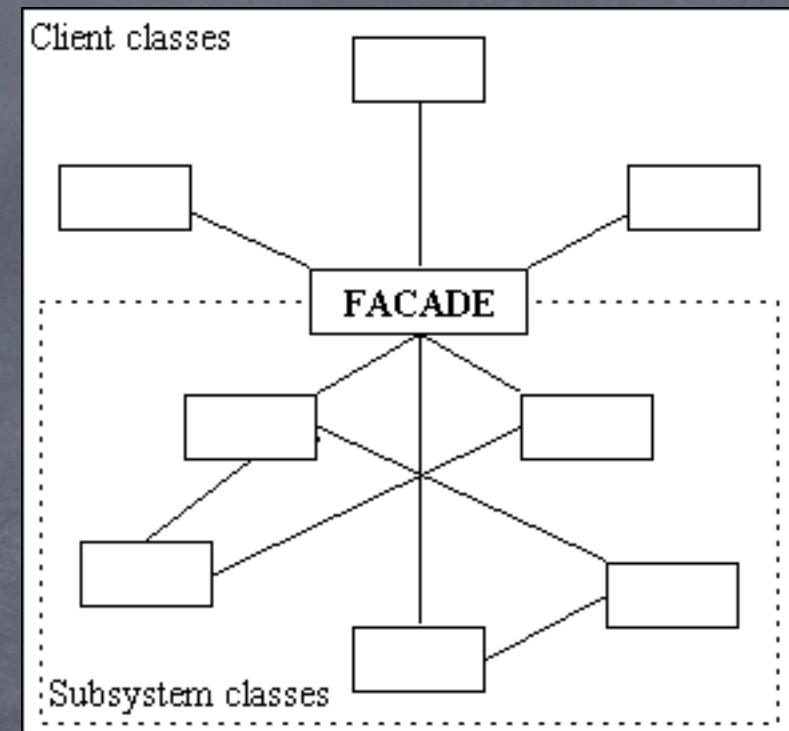
- "Forze": un ricco e complesso sottosistema offre tante funzionalità utili; il codice-cliente interagisce con varie parti della funzionalità in modi "fuori controllo"
- ciò causa problemi ai programmatore di codice cliente e agli autori del sottosistema stesso

(complessità + rigidità)



Soluzione: la DP "Facade"

- interponiamo un oggetto o classe "Facciata" che espone un sottoinsieme controllato e semplificato della funzionalità
- il codice-cliente "vede" solo la Facciata
- la Facciata implementa la sua funzionalità con chiamata al sottosistema
- il sottosistema guadagna in **flessibilità**, i clienti in **semplicità**



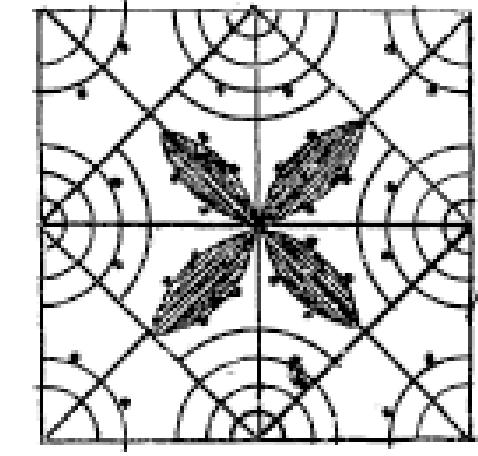
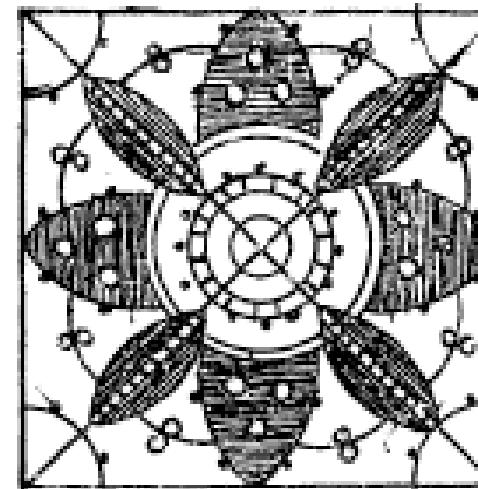
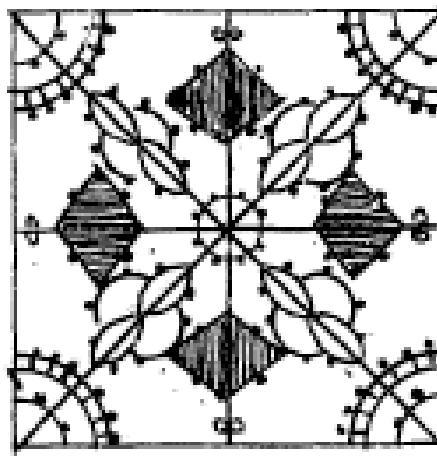
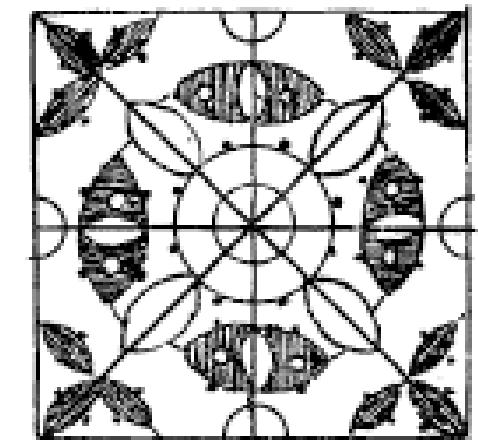
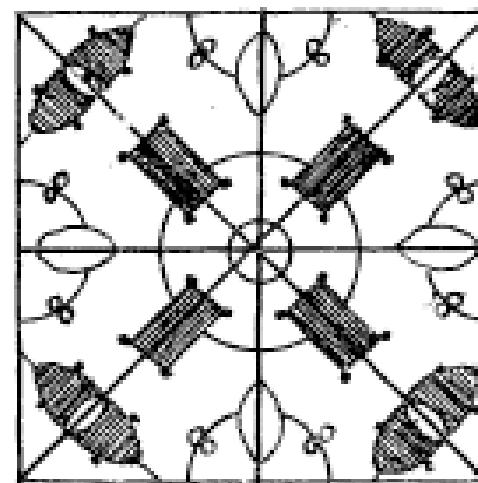
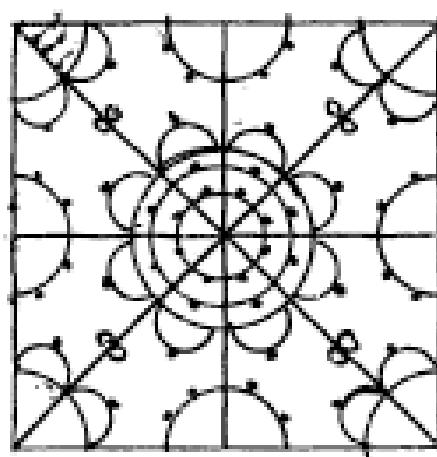
Facade è una Design Pattern

- riassume un frequente problema di progetto + struttura di una soluzione del problema (+ pro e contro, alternative, ...), e:
 - **UN NOME** (da ricordare/discutere!)
- "descrizioni di oggetti e classi comunicanti personalizzate per risolvere un problema generale in un particolare contesto"
- NON: strutture dati, algoritmi, architetture di sistema specifiche di un dominio, feature di linguaggi di programmazione e librerie, ...
- DEVONO elencare vari "usì conosciuti" (UC)

Alcuni UC di Facade

- ...nella libreria standard di Python...:
 - dbhash: facade per bsddb
 - sottoinsieme molto semplificato
 - implementa l'interfaccia dbm (quindi e` anche un esempio della DP "Adapter")
 - os.path: basename, dirname facade per split + indicizzazione; isdir (&c) facade per os.stat + stat.S_ISDIR (&c)
 - Facade e` un DP **strutturale** (come Adapter, v. oltre; in dbhash, si "fondono"!-)

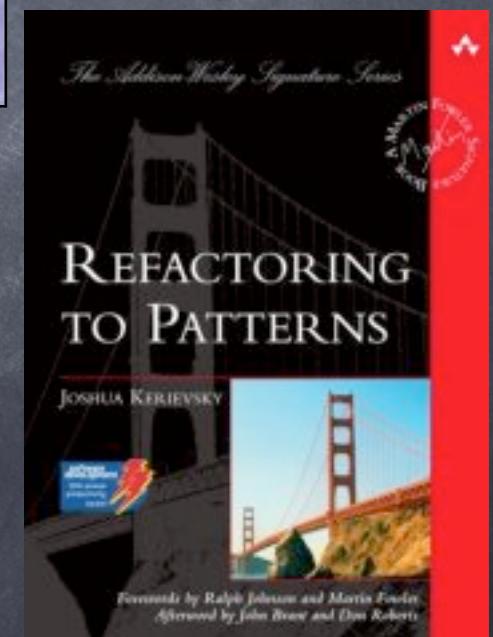
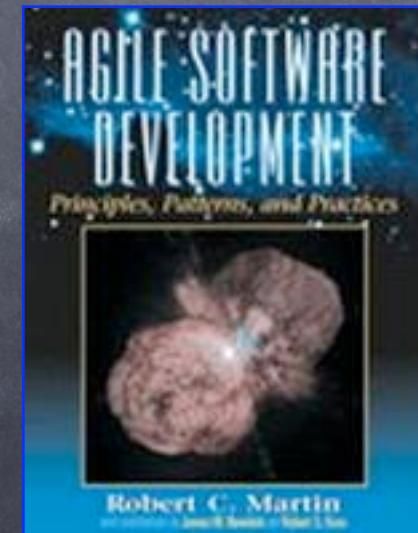
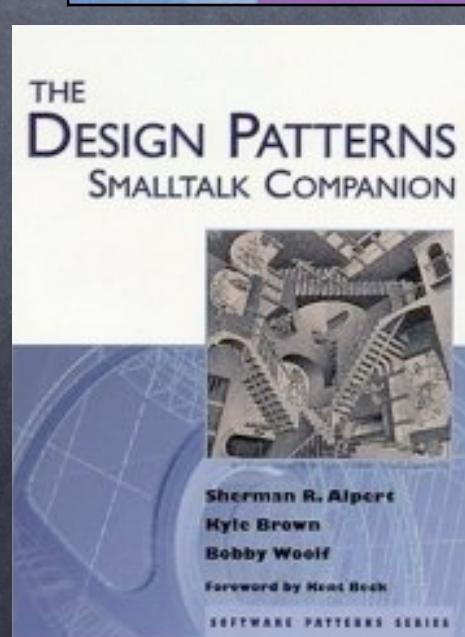
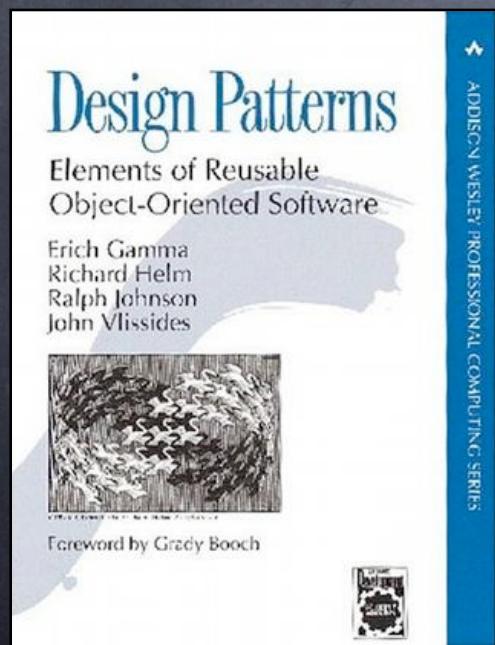
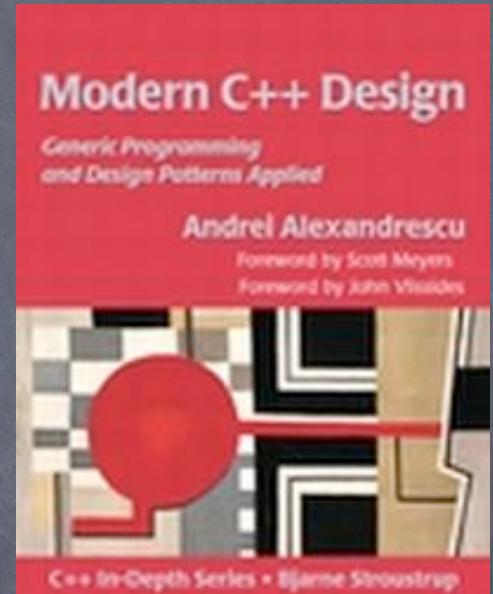
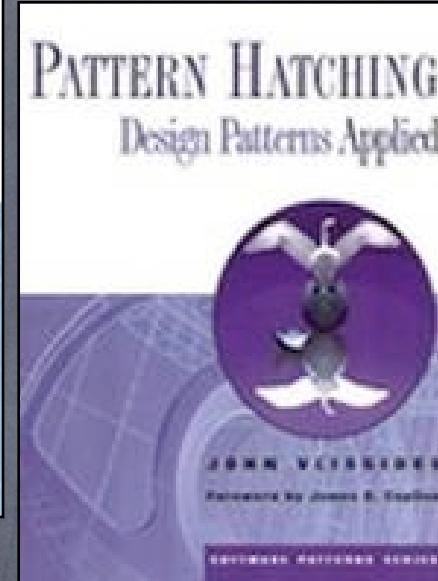
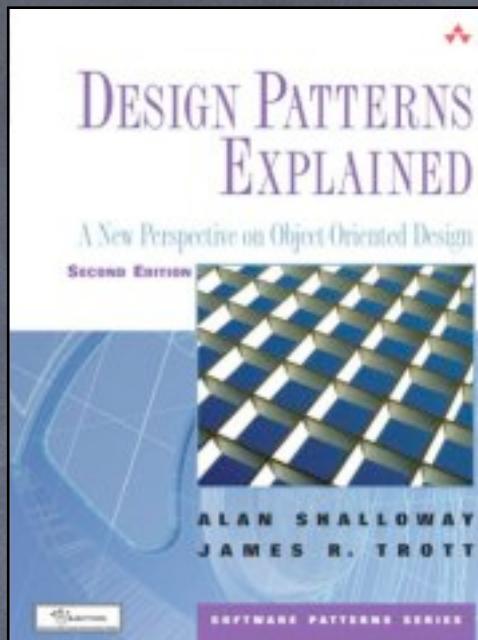
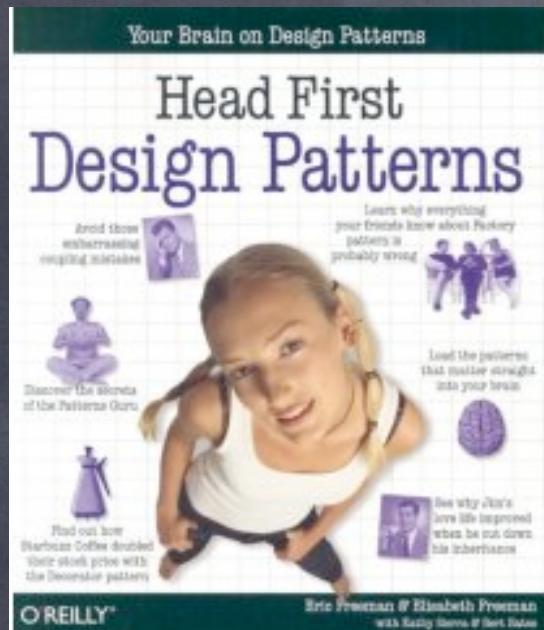
Design Pattern



Design Pattern e linguaggi

- nella gran parte dei casi, una DP va studiata entro un particolare linguaggio o categoria di linguaggi
 - Facade è` un buon controesempio!-)
- analogia in architettura: abbiamo le stesse "pattern" se costruiamo in giunchi, legno, mattoni, marmo, cemento precompresso...?
 - alcune si` (se relative all'"uso dello spazio" e al comportamento umano...)
 - altre no (dipendono dal materiale...!)
- Gamma &c (la "Gof4") lo spiegano benissimo, ma tanti, troppi hanno frainteso...!

Buoni Libri sulle DP



(biblio alla fine, ma: in inglese; non in Python)

Classiche Categorie di DP

- **Creazionali:** relative all'istanziare oggetti
- **Strutturale:** mutua composizione di classi od oggetti
- **Comportamentali:** come le classi od oggetti interagiscono e si dividono le responsabilità
- Ciascuna puo` essere a livello di classi o di oggetti (quando possibile, e` normalmente meglio puntare su oggetti che su classi!)

Prolegomeni alle DP

- "programma appoggiandoti a una interfaccia, non a una implementazione"
- in Python, normalmente, si fa col "duck typing" -- raramente con interfacce "formali" (ma: Zope, Twisted, Py3K...)
- il "duck typing" è simile al "polimorfismo basato sulla signature" dei template usati per il Generic Programming in C++

Il Vantaggio del Duck Typing



Insegnare alle anatre a battere a macchina
e` faticoso, ma poi risparmia molto lavoro!-)

Prolegomeni alle DP

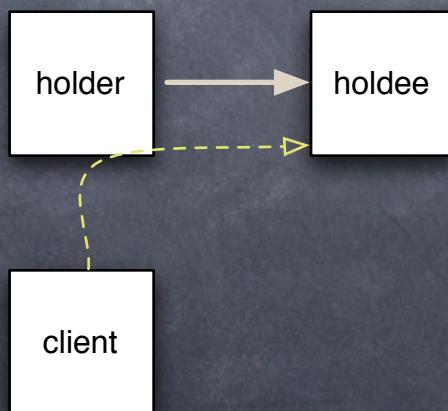
- "preferisci la composizione di oggetti rispetto all'eredità di classi"
 - in Python: **tieni**, o **avvolgi**
 - eredita solo quando è **davvero** comodo
 - espone tutti i metodi della classe base (riusa + generalmente override + eventualmente estende)
 - ma, è un accoppiamento molto stretto!

Python: tieni o avvolgi



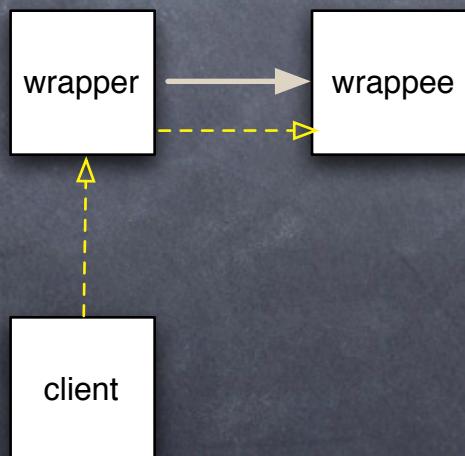
Python: tieni o avvolgi

- "tieni": l'oggetto O ha un oggetto S come attributo (magari proprietà) -- e` tutto!
- usa self.S.method o O.S.method
- semplice, diretto, immediato, ma... accoppiamento forte, e spesso lungo ("asse" sbagliato!)



Python: tieni o avvolgi

- “avvolgi”: tieni (spesso con nome privato) e delega (quindi usi direttamente O.method)
- diretto (def method(self...)...self.S.method)
- automatico (con __getattr__)
- e` l'accoppiamento giusto (Legge di Demetra)



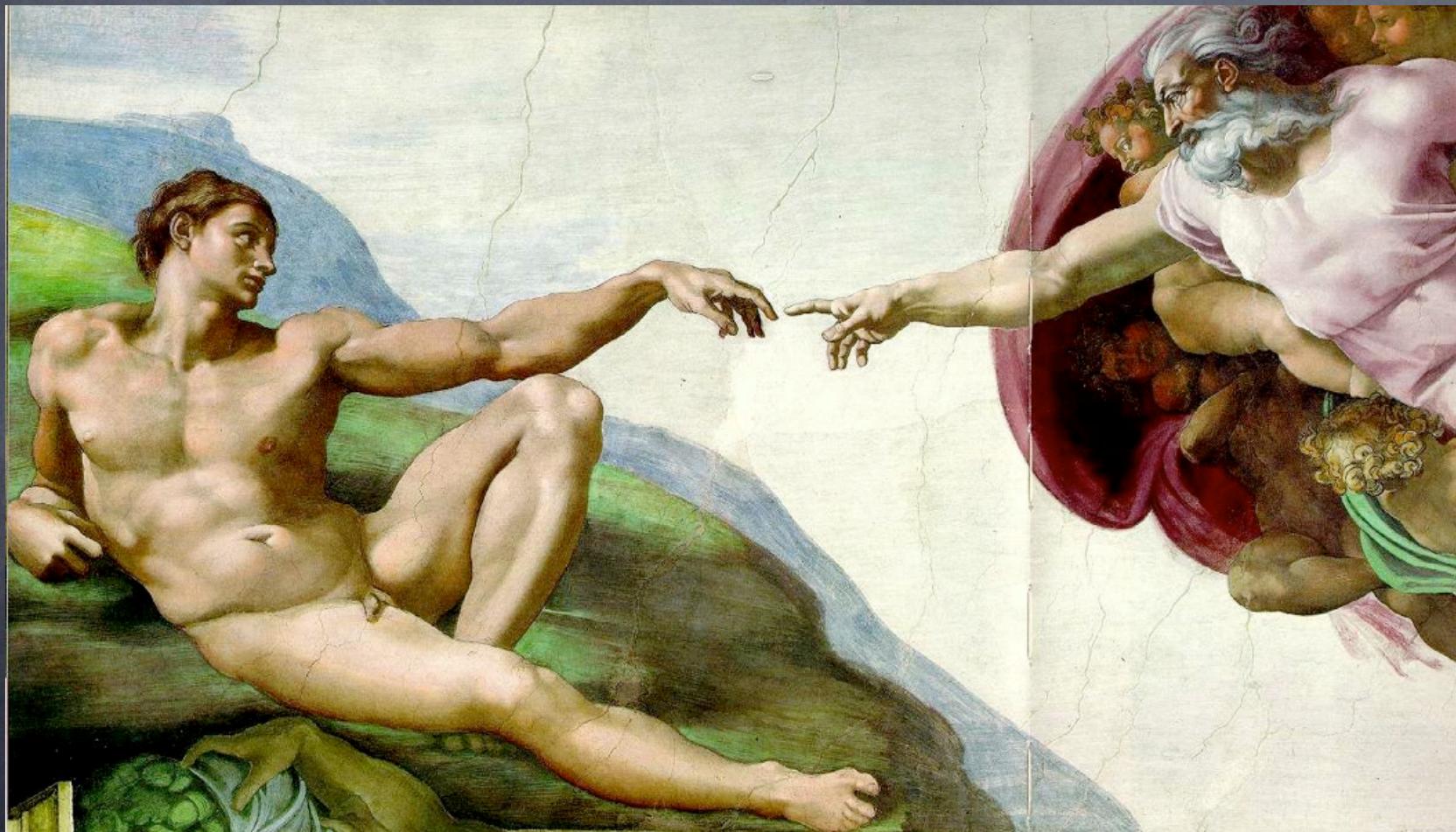
E.g: avvolgi e "restringi"

```
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)  
    ...
```

L'eredità non puo` mai restringere!

Pattern Creazionali

- non comuni in Python...
- ...perche` "factory" e` in pratica built-in!-)



Pattern Creazionali [1]

- "voglio che esista una sola istanza"
 - usa un modulo invece di una classe
 - ne` subclassing, ne` metodi speciali
 - fai solo 1 istanza (senza "meccanismi")
 - devi decidere _quando_ farla... e di che esatta sottoclasse...
 - singleton ("highlander")
 - il subclassing e` sempre problematico
 - monostate ("borg")
 - non piace a Guido

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
```

il subclassing e` sempre un problema:

```
class Foo(Singleton): pass
```

```
class Bar(Foo): pass
```

```
f = Foo(); b = Bar(); # ...???
```

...ed e` un problema intrinseco di 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
```

il subclassing non e` affatto un problema...:

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

il **data overriding** risolve!

Pattern Creazionali [2]

- "non vogliamo vincolarci all'istanziare una specifica classe concreta"
- DP "Dependency Injection"
 - lascia le istanziazioni "fuori"
 - ma: se servono N (variabile) istanze...
- sotto-categoria "Factory" delle DP
 - puo` creare se serve, o riutilizzare "funzioni fabbrica" (& altri chiamabili)
 - o metodi (per permettere override)
 - "classi fabbrica" (astratte o meno)

Pattern Strutturali

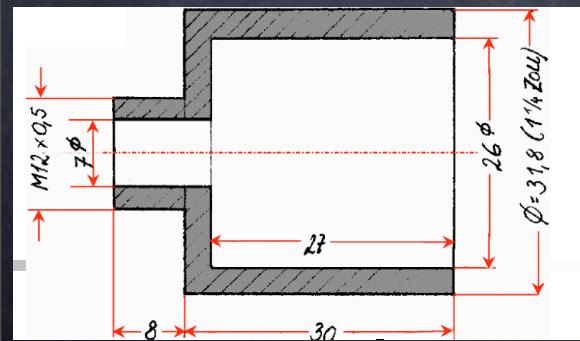
Sottocategorie "Masquerading/Adaptation":

- ⦿ Adapter: modifica una interface (esistono varianti a livello di classe e di oggetto)
- ⦿ Facade: semplifica l'interfaccia di un sottosistema
- ⦿ ...e molte altre che non tratto, come ad es:
 - ⦿ Bridge
 - ⦿ Decorator
 - ⦿ Proxy



Adapter

- il codice-cliente γ richiede un protocollo C
- il codice-fornitore σ fornisce un diverso protocollo S (con funzionalità sufficiente)
- l'adapter α "si mette in mezzo":
 - per γ , α è un fornitore (del protocollo C)
 - per σ , α è un cliente (del protocollo S)
 - "dentro", α implementa C (con opportune chiamate all' S di σ)



Adapter: esempio giocattolo

- C richiede un metodo foobar(foo, bar)
- S fornisce un metodo barfoo(bar, foo)
- ad esempio, σ potrebbe essere:
class Barfooer(object):
 def barfoo(self, bar, foo):

...

Adapter di oggetto

- per-istanza, con "avvolgi"/delegazione:

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

```
foobarer=FoobarWrapper(barfooer)
```

Adapter di classe (diretto)

- per-classe, con subclassing e auto-delegazione:

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

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

Adapter di classe (mixin)

- flessibile, buon uso di eredità multipla:

```
class BF2FB:
```

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

```
class Foobarer(BF2FB, Barfooer):  
    pass
```

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

UC di Adapter

- socket._fileobject: da socket a oggetto file-like (+ molto codice per fare il buffering)
- doctest.DocTestSuite: da test in stile doctest a unittest.TestSuite
- dbhash: da bsddb a dbm (e` anche Facade!)
- StringIO: da str o unicode a file-like
- shelve: da "dict limitato" (chiavi e valori str, metodi di base) a mappa (quasi) completa
- usa pickle per serializzare any <-> string
- + UserDict.DictMixin

Note su Adapter

- alcuni adapter richiedono tanto codice
- le classi mixin sono un ottimo modo per implementare protocolli "ricchi" (se i metodi avanzati sono costruibili da quelli semplici)
- Adapter esiste a tutti i livelli di complessità
- in Python, non si adattano solo classi e istanze -- spesso si adattano anche oggetti chiamabili (con decoratori e altre FOS, "chiusure", functools, ...)

Facade ed Adapter

- Adapter serve a fornire un dato protocollo richiesto dal codice-cliente
 - o a volte a omogenizzare x polimorfismo
- Facade serve a semplificare una ricca interfaccia quando ne serve un sottoinsieme
- Facade di solito si pone davanti a un sottosistema di molteplici classi/oggetti, mentre Adapter si pone davanti ad un singolo oggetto o classe

Domande e Risposte (p.1/2)

D?

R!

Pattern Comportamentali

- Template Method: auto-delegazione
- ... "l'essenza dell'OOP"...
- alcune sue varianti Python-specifiche



Template Method

- ottima pattern, pessimo nome
 - "template" è una parola "sovraffabbricata"!
 - generic programming in C++
 - generazione di documenti da scheletro
 - ...
- un nome migliore: **self-delegation**
 - direttamente descrittivo (in inglese e in Python... meno in italiano, visto che "auto-delegazione" fa pensare piuttosto a un gruppo di macchine in visita ufficiale:-)

TM Classico

- una classe-base astratta offre un "metodo organizzante" che chiama "metodi gancio"
- nella CBA, i metodi gancio restano astratti
- le sottoclassi concrete implementano i ganci
- il codice cliente chiama il metodo organizzante
- generalmente su di un riferimento alla CBA (iniettato o ottenuto da "fabbrica")
- naturalmente dietro a quel riferimento c'e` un'istanza di una classe concreta

TM: esempio-giocattolo

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

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

Esempio di TM: paginatore

- per "paginare" del testo, occorre:
 - ricordare il numero max di righe/pagina
 - emettere ciascuna riga, tenendo traccia della posizione sulla pagina
 - subito prima della prima riga di ciascuna pagina, emettere un header di pagina
 - subito dopo l'ultima riga di ciascuna pagina, emettere un footer di pagina

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
```

Paginatore su 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',      # carattere form-feed
```

Paginatore cn 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()      # aspetta un tasto
```

UC: 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()
```

I Perche` del TM Classico

- il metodo organizzante fornisce la "logica strutturale" di un'operazione
- i metodi gancio eseguono le effettive azioni elementari "concrete" (varie possibili forme)
- e` una fattorizzazione spesso appropriata di "cio` che cambia" (le forme concrete) e "cio` che resta valido" (la logica strutturale)
- identifica le responsabilita` e collaborazioni fra CBA e classi concrete
- il "Principio di Hollywood": "non ci chiami, la chiameremo noi!"

2 Stili per i Ganci

```
class TheBase(object):
    def doThis(self):
        # fornire un default (spesso un no-op)
        pass
    def doThat(self):
        # o forzare le sottoclassi ad avere
        # un'implementazione, omettendo, o:
        raise NotImplementedError
```

Quando ha senso, l'implementazione di default
è più pratica; `NotImplementedError` può
essere buona diagnosi di alcuni errori.



Mixin come TM di classe

- Consideriamo di nuovo UserDict.DictMixin...
- E` astratto, fatto per ereditarne classi concrete (in eredita` multipla, o meno)
 - non implementa i metodi gancio
- la sottoclassse deve fornire i metodi gancio:
 - almeno `__getitem__`, `keys`
 - se R/W, anche `__setitem__`, `__delitem__`
 - normalmente `__init__`, `copy`
 - puo` fare altre override (per prestazioni):
`__contains__`, `__iter__`, `iteritems`

TM in DictMixin

```
class DictMixin:  
    ...  
    def has_key(self, key):  
        try:  
            # chiama un gancio __getitem__  
            value = self[key]  
        except KeyError:  
            return False  
        return True  
    def __contains__(self, key):  
        return self.has_key(key)  
    ...
```

Use `DictMixin`

```
class R0_Chainmaps(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)
```

UC di TM: 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 e la DP TM

- Queue non è astratta, viene spesso usata così com'è
 - quindi, implementa tutti i metodi gancio
- le sottoclassi possono cambiare la disciplina d'accodamento (spesso: code con priorità)
- nessun problema di locking, tempi, &c
- il default è la semplice e utile FIFO
- possibile override di metodi gancio (`__init__`,
`__qsize`, `__empty`, `__full`, `__put`, `__get`) E...
- ...essendo Python, anche dei dati (`maxsize`,
`queue`), se occorre

Esempi: due Queue LIFO

```
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()
```

Queue con priorita`+FIFO

```
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]
```

"Fattorizzare fuori" i ganci

- metodo organizzante in una classe
- metodi gancio in un'altra
- UC: HTML formatter rispetto a writer
- UC: SAX parser rispetto a handler
- aggiunge un asse di variabilità/flessibilità
- assomiglia un po' alla DP **Strategy**:
 - Strategy: 1 classe astratta per punto di decisione, classi concrete indipendenti
 - TM fattorizzato: classi astratte e concrete più "raggruppate"

TM + introspezione

- la classe "organizzante" puo` esplorare quella "implementante" (sia discendente o meno) a runtime
- scoprire che metodi gancio esistono
- gestire appropriatamente eventuali metodi desiderati ma assenti
 - ad es. chiamando un metodo di default
- tecnica spesso molto comoda nel programmare "ad eventi" quando non si possono prevedere tutti i possibili eventi (ad es., parsing ad eventi di HTML o XML)

UC: cmd.Cmd.docmd

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

NB: docmd è un metodo gancio di cmd.Cmd, ma è anche un metodo organizzante in questa implementazione che cmd.Cmd offre come default!

Un UC di TM multi-stile

- classico + fattorizzato + introspettivo
- molteplici "assi", per separare tre "variabilità" accuratamente distinte
- equivalente DP di una "Fuga a Tre Soggetti"
- "tutta l'arte aspira alla condizione della musca" (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(...)...
```

Domande e Risposte

D?

R!

- 1.Design Patterns: Elements of Reusable Object-Oriented Software --
Gamma, Helms, Johnson, Vlissides -- avanzato, molto profondo, IL
classico "Gof4" che ha "lanciato" la popolarita` delle DP (C++)
- 2.Head First Design Patterns -- Freeman -- introduttivo, veloce, molto
"hands-on" (Java)
- 3.Design Patterns Explained -- Shalloway, Trott -- introduttivo, mix di
esempi, ragionamenti ed applicazioni (Java)
- 4.The Design Patterns Smalltalk Companion -- Alpert, Brown, Woolf
-- intermedio, specifico al linguaggio (Smalltalk)
- 5.Agile Software Development, Principles, Patterns and Practices --
Martin -- intermedio, estremamente pratico, ottimo mix di teoria e
pratica (Java, C++)
- 6.Refactoring to Patterns -- Kerievsky -- introduttivo, enfasi sul
refactoring di codice esistente (Java)
- 7.Pattern Hatching, Design Patterns Applied -- Vlissides -- avanzato,
anedottico, specifici esempi di applicazione delle pattern Gof4 (C++)
- 8.Modern C++ Design: Generic Programming and Design Patterns
Applied -- Alexandrescu -- avanzato, specifico al linguaggio (C++)