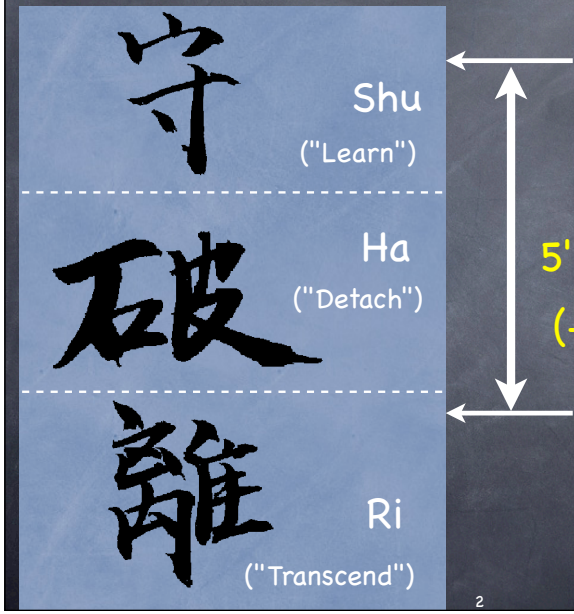
Zen and the Art of Abstraction Maintenance

http://www.aleax.it/osc09_abst.pdf

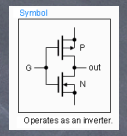
This talk's "level"

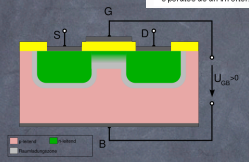


5' Q & A at the end

(+: let's talk later!)

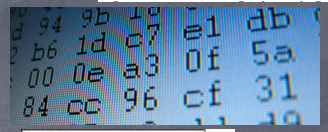
A Tower of Abstractions

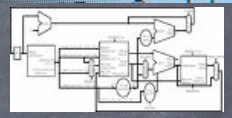


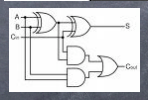


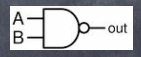
$$-\sum_i \frac{\hbar^2 \nabla_i^2 \Psi}{2m_i} + \sum_{i>j} V_{i,j} \Psi = E\Psi$$

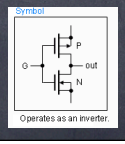
```
push    ebp
mov     ebp, esp
movzx   ecx, [ebp+arg_0]
pop     ebp
movzx   dx, cl
```





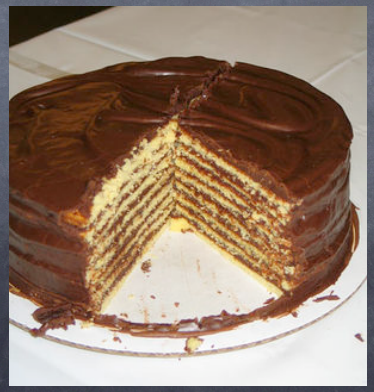






Can't Live Without it...

- programming (& other "knowledge work")
  - USES abstraction layers,
  - often PRODUCES new layers



...can live with it?

- all abstractions "LEAK" (Spolsky's Law)

  - ...bugs, overloads, attacks...
  - ...you MUST "get" a few layers below!
- +, they SHOULD "leak" (sometimes;-)
  - in designed, architected ways

and: abstraction *can slow you down*!

Abstract -> Procrastinate!

- McCrea, S. M., Liberman, N., Trope, Y., & Sherman, S. J. -- Construal level and procrastination. Psychological Science, Volume 19, Number 12, December 2008, pp. 1308-1314(7)
- remote events are mentally construed at higher abstraction levels than "near" ones
- reverse holds: higher-abstraction construal levels lead to > chance of procrastination
- (at least for psych students, typically the only experimental subjects available;-)

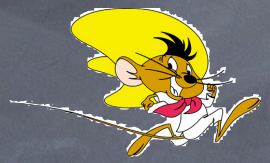
To Achieve, Think Concrete!

- Allen, "Getting Things Done":
  - what's my SINGLE NEXT ACTION?
- interaction (& user-centered) design:
  - NOT "the user", BUT "John, newbie trader, vast videogame experience" and "Mark, seasoned trader, started in Hammurabi's time, STILL prefers cuneiform on clay tablets"
- "prefer action to abstr-action" (J. Fried, founder of "37 signals")

Abstraction Penalty

- when a language allows low- and high-abstraction approaches, there can be a penalty for abstraction (Stepanov, http://std.dkuug.dk/JTC1/SC22/WG21/docs/PDTR18015.pdf & much later research)
- an issue of quality of implementation, not always true: in Python we're used to an abstraction *bonus*, not *penalty*!

Itertools FLIES!



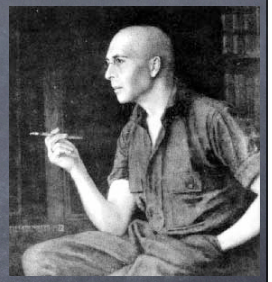
```
$ python -mtimeit 'for x in range(42): pass'
100000 loops, best of 3: 5.13 usec per loop

$ python -mtimeit 'for x in xrange(42): pass'
100000 loops, best of 3: 4.17 usec per loop

$ python -mtimeit -s'import itertools' \
> 'for x in itertools.repeat(None, 42): pass'
100000 loops, best of 3: 3.4 usec per loop
```

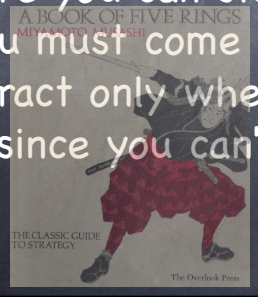
The "Martian Smilie" rocks!

```
$ python -mtimeit -s'x="abracadabra"' \
> 'y="".join(reversed(x))'
100000 loops, best of 3: 5.96 usec per loop
$ python -mtimeit -s'x="abracadabra"' \
> 'y=x[::-1]'
1000000 loops, best of 3: 0.597 usec per loop
```

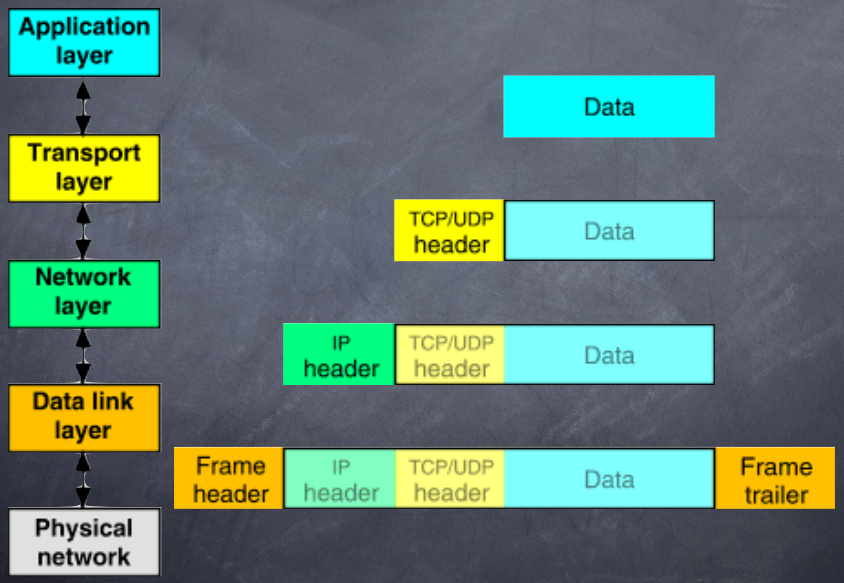
All Abstractions Leak

- all abstractions leak, because...
  - ...*all abstractions LIE*!
  - the map is not the territory
- before you can abstract,
  - you must grok the details
- before you can step back,
  - you must come close
- abstract only when you know ALL details
  - →since you can't, be humble & flexible!





A great abstraction: TCP/IP



TCP leaks: *TRUST*!

- TCP/IP, superb stack of abstractions, BUT...
  - ...designed in an ancient era of trust!
- The whole stack "leaks" all over the place in terms of security attacks from:
  - "below" (ARP cache poisoning),
  - "above" (DNS cache poisoning),
  - "beside" (mendacious BGP),
  - "within" (sniffing, pwd FTP/Telnet, ...)
  - ...etc, etc...

TCP/IP today…:-(

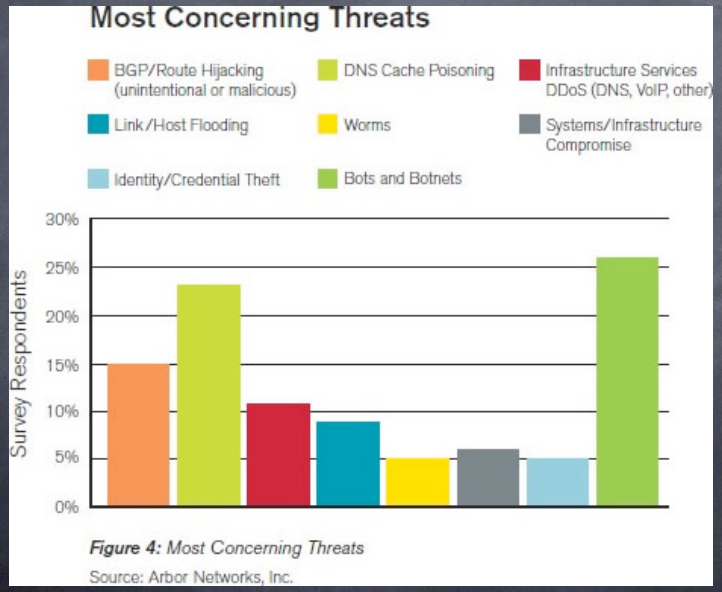
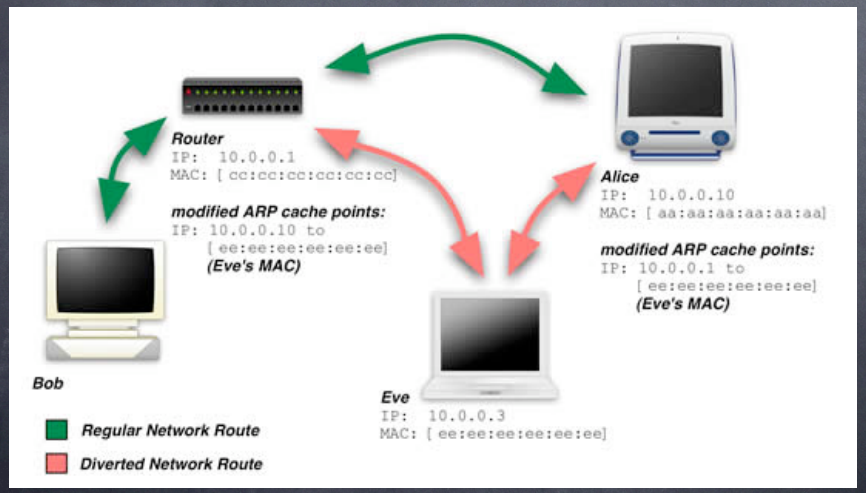


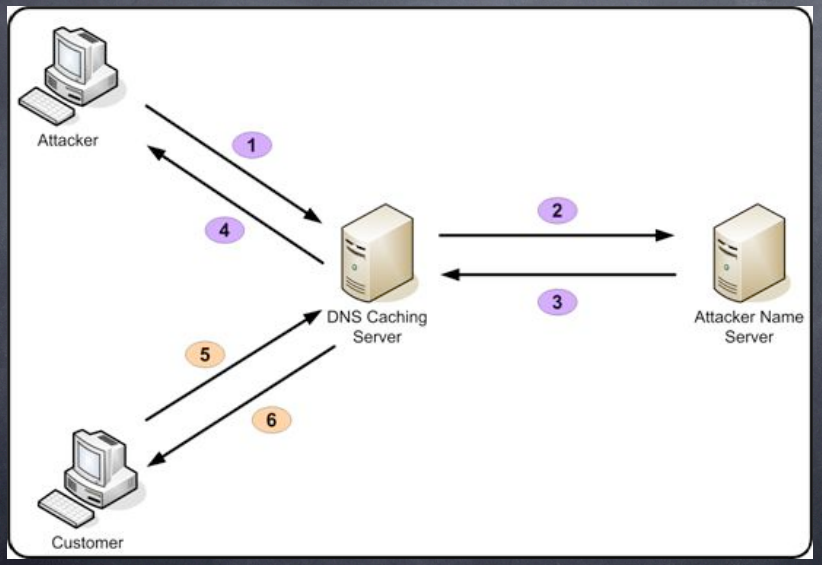
Figure 4: *Most Concerning Threats*

Source: Arbor Networks, Inc.

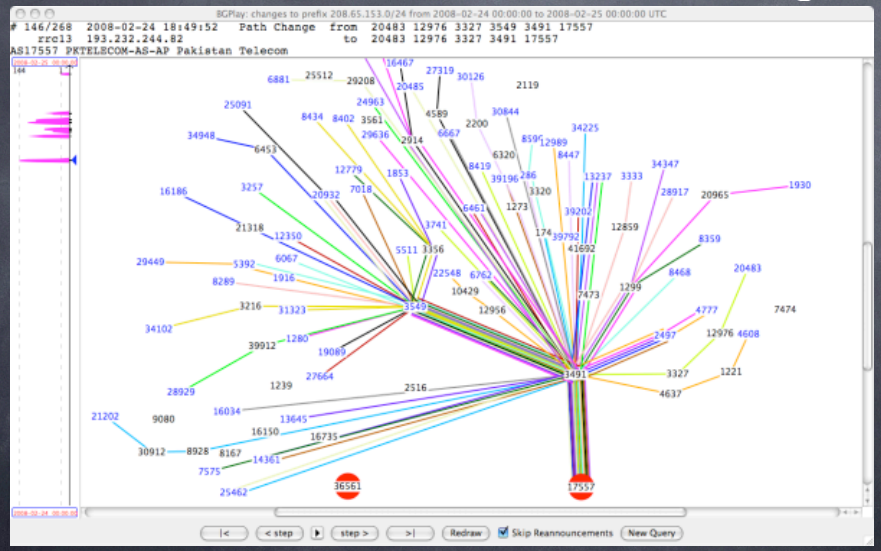
One "leak": ARP poisoning



Other "leak": DNS poisoning



The worst: BGP Hijacking



..."leaks" may be *good*!

- e.g.: remote/distributed filesystems trying to "emulate" local ones
  - "less local" → the costlier "abstraction"
    - semantics, locking, reliability, ...
  - "filesystem", splendid abstraction...
  - "local filesystem", NOT!
    - "never subclass a concrete class" [Haahr]
- doesn't mean "abstraction is a bad thing"
  - JUST the abstraction isn't enough
  - needs systematic, usable LEAKS!

How to Abstract Wrong

- small scale: 1 class → 1 interface
  - always "surfaces" implementation details
- mid-scale: "subclassing concrete classes"
  - concrete class (== implementation) → NEVER the right base for subclassing
- mid-scale: encapsulation errors
  - windows vs toolbars in MFC 4.*
- large scale: "floating framework"
  - "framework" with just 1 application...

How to Abstract Well

- master at least 1-2 layers BELOW
- to DESIGN an excellent abstraction:
  - DEEP familiarity with SEVERAL possible implementations ("layers below")
  - DEEP familiarity with SEVERAL intended uses ("layers above" which will USE it)
  - no blinders, no shortcuts!
- YOU can be the next user or implementer!
  - Golden Rule's really a must;-)
- http://c2.com/cgi/wiki?TooMuchAbstraction

Donald Knuth: yes, you can!

- the psychological profiling [[of the programmer]] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large. [[...]]
- Computer scientists see things simultaneously at the low level and the high level [[of abstraction]]

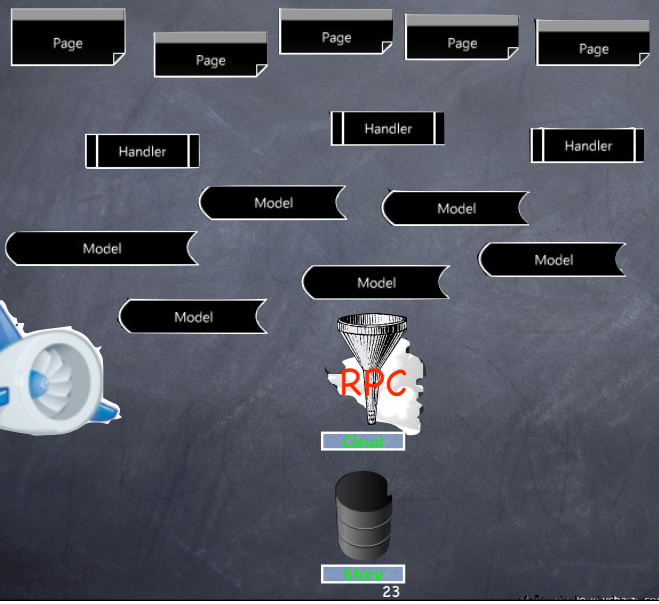
http://www.ddj.com/184409858

Jason Fried: and you must!

- "Here's the problem with copying:
  - Copying skips understanding.
  - Understanding is how you grow.
  - You have to understand why something works or why something is how it is.
  - When you copy it, you miss that.
  - You just repurpose the last layer instead of understanding the layers underneath."
- Just '%s/copy/use existing high-level abstractions blindly/g' ...;-)

http://www.37signals.com/svn/posts/1561-why-you-shouldnt-copy-us-or-anyone-else

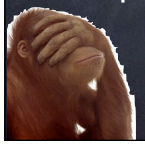
App Engine "Hacks"



Monkey-patch Hacking

- all operations go through an RPC layer, `apiproxy_stub_map.MakeSyncCall`
- not advisable: *monkey-patching*...:

```
from google.appengine.api import \
    apiproxy_stub_map
_org = apiproxy_stub_map.MakeSyncCall
def fake(svc, cal, req, rsp):
  ...
  x = _org(svc, cal, req, rsp)
  ...
apiproxy_stub_map.MakeSyncCall = fake
```



Why the Monkey is Sad

```
class Client(object):
"""Memcache client object... """

def __init__(self, servers=None, debug=0,
     pickleProtocol=pickle.HIGHEST_PROTOCOL,
     pickler=pickle.Pickler,
     unpickler=pickle.Unpickler,
     pload=None,
     pid=None,
     make_sync_call=apiproxy_stub_map.MakeSyncCall):
 """Create a new Client object.... """
 ...
 self._make_sync_call = make_sync_call
```

Better: use "Hooks"!

http://blog.appenginefan.com/2009/01/hacking-google-app-engine-part-1.html (with THANKS to Jens Scheffler!-)

```
from google.appengine.api import apiproxy_stub_map

def prehook(svc, cal, req, rsp):
  ...
apiproxy_stub_map.apiproxy.GetPreCallHooks(
  ).Append('unique_name', prehook, 'opt_api_id')
```

How to Supply "Hooks"?

- ...without a "natural funnel" such as RPC?
- use key semantical "bottlenecks"
  - if your system does SQL queries,
    - pre-hooks w/SQL, post-hooks w/results
- "event/callback" approaches (Qt signal/slot)
- design patterns:
  - pre/post hooks & events ~ Observer
  - Template Method (e.g., Queue.Queue)
  - Dependency Injection

Making Hooks: scheduler

```
class ss(object):
  def __init__(self):
    self.i = itertools.count().next
    self.q = somemodule.PriorityQueue()
  def add_event(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)
```

(PQ is "obvious"...):

```
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)
```

Nice abstraction, but...

- ...how to test ss without long waits?
- ...how to integrate it with event-loops of other systems, simulations, etc...?

Problem: `ss` "concretely depends" on specific objects (time.sleep and time.time).

To "make the abstraction leak", you can...:

1. leave it for "Monkey Patching"
2. design pattern: Dependency Injection

Monkey-patching...

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

- useful in emergencies, but...
  - ...too often an excuse for lazy design!-)
- subtle, hidden "communication" via dark byways (explicit is better than implicit!-)
- broken by optimizations &c...

Dependency Injection

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

- i.e., just like sched in the standard library!-)

DI is a handy hook!

```
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 = ss(f.time, f.sleep)
...
```

DI example (app engine:-)

```
class Client(object):
"""Memcache client object... """

def __init__(self, servers=None, debug=0,
     pickleProtocol=pickle.HIGHEST_PROTOCOL,
     pickler=pickle.Pickler,
     unpickler=pickle.Unpickler,
     pload=None,
     pid=None,
     make_sync_call=apiproxy_stub_map.MakeSyncCall):
 """Create a new Client object.... """
 ...
 self._make_sync_call = make_sync_call
```

Q & A

http://www.aleax.it/osc09_abst.pdf