PICK THE RIGHT TOOL FOR THE JOB

Languages and Architectures

SERVICE ORIENTED ARCHITECTURE(SOA)

SERVICE ORIENTED ARCHITECTURE

- SOA: SW architecture where all components are designed to be services
- Apps composed of interoperable services
    - Easy to tailor new version for subset of users
    - Also easier to recover from mistake in design
- Contrast to “SW silo” without internal APIs

CEO: AMAZON SHALL USE SOA!

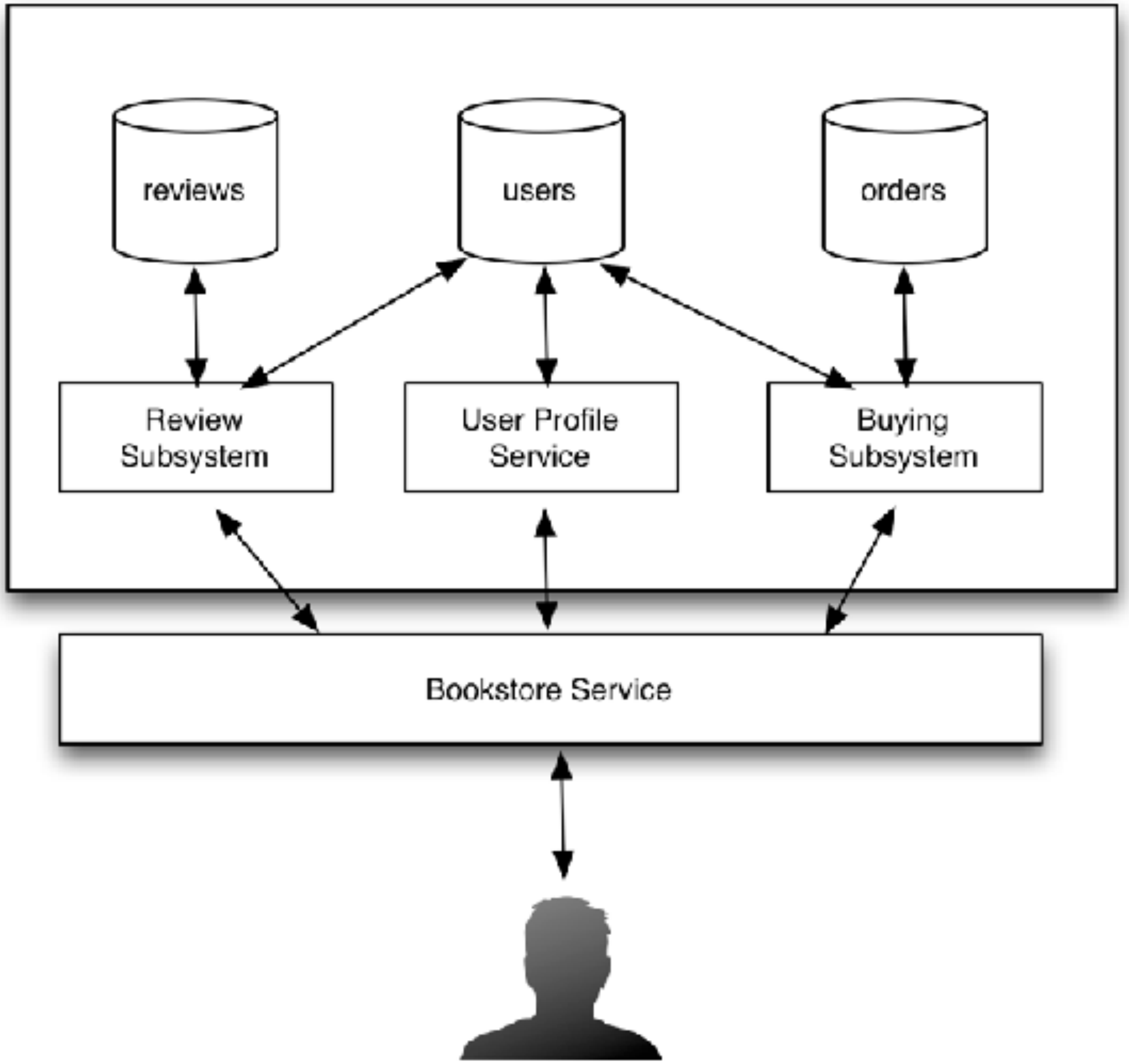
1. “All teams will henceforth expose their data and functionality through service interfaces.”
2. “Teams must communicate with each other through these interfaces.”
3. “There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.”

CEO: AMAZON SHALL USE SOA!

4. “It doesn't matter what [API protocol] technology you use.”
5. “Service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.”
6. “Anyone who doesn't do this will be fired.”
7. “Thank you; have a nice day!”

BOOKSTORE: SILO

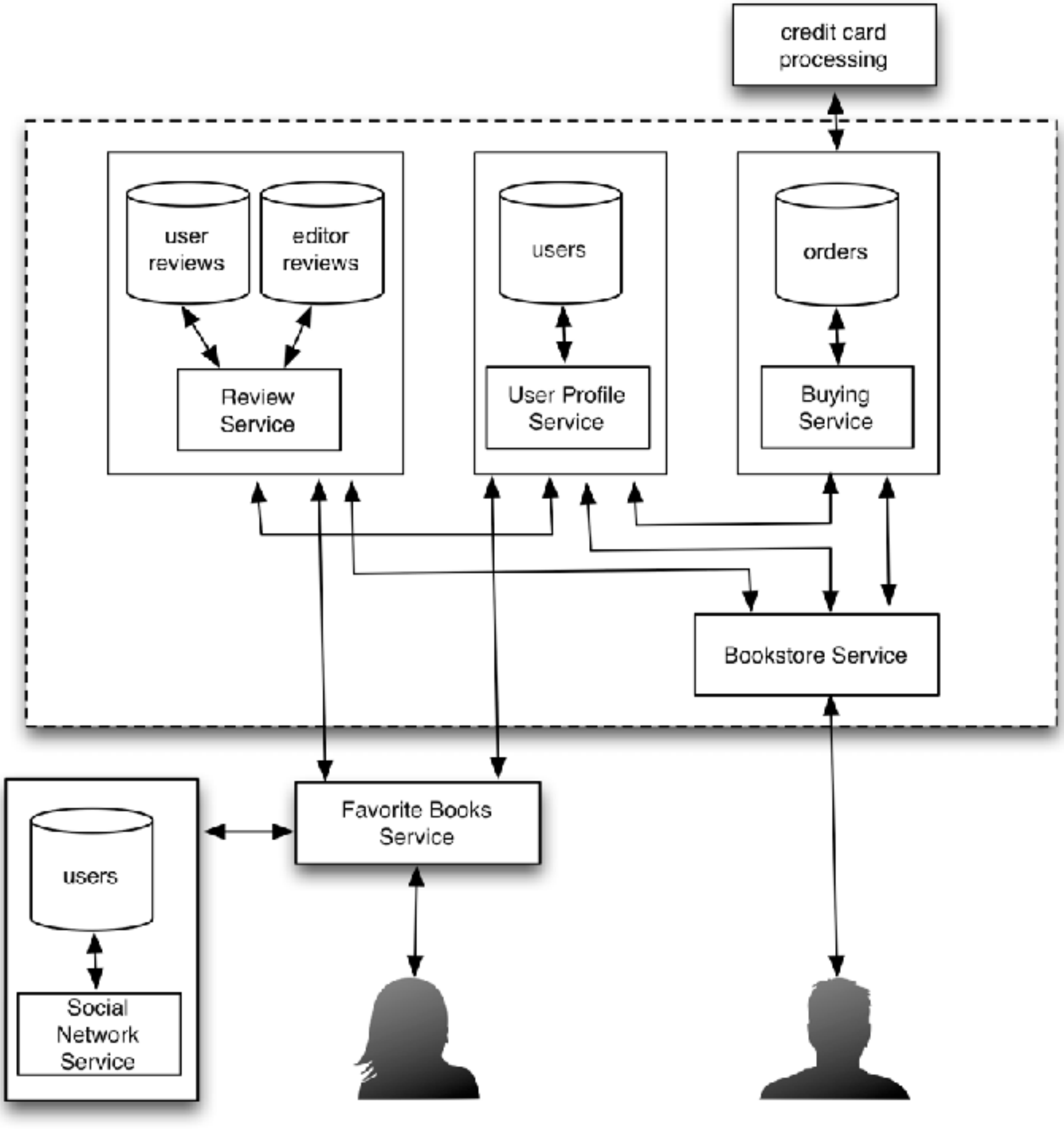
- Internal subsystems can share data directly
  - Review access user profile
- All subsystems inside single API ("Bookstore")



(Figure 1.3, Engineering Long Lasting Software by Armando Fox and David Patterson, Beta edition, 2012.)

BOOKSTORE: SOA

- *Subsystems independent, as if in separate datacenters*
  - *Review Service access User Service API*
- Can recombine to make new service ("Favorite Books")



(Figure 1.4, Engineering Long Lasting Software by Armando Fox and David Patterson, Beta edition, 2012.)

CLOUD COMPUTING

SAAS INFRASTRUCTURE?

- SaaS's 3 demands on infrastructure

1. Communication: allow customers to interact with service
2. Scalability: fluctuations in demand + new services to add users rapidly
3. Dependability: service and communication continuously available 24x7

CLUSTERS OF MACHINES

- Clusters: Commodity computers connected by commodity Ethernet switches

1. More scalable than conventional servers
2. Much cheaper than conventional servers
   - ➤ 20X for equivalent vs. largest servers
3. Few operators for 1000s servers
   - ➤ Careful selection of identical HW/SW
   - ➤ Virtual Machine Monitors simplify operation
4. Dependability via extensive redundancy

WAREHOUSE SCALE COMPUTERS

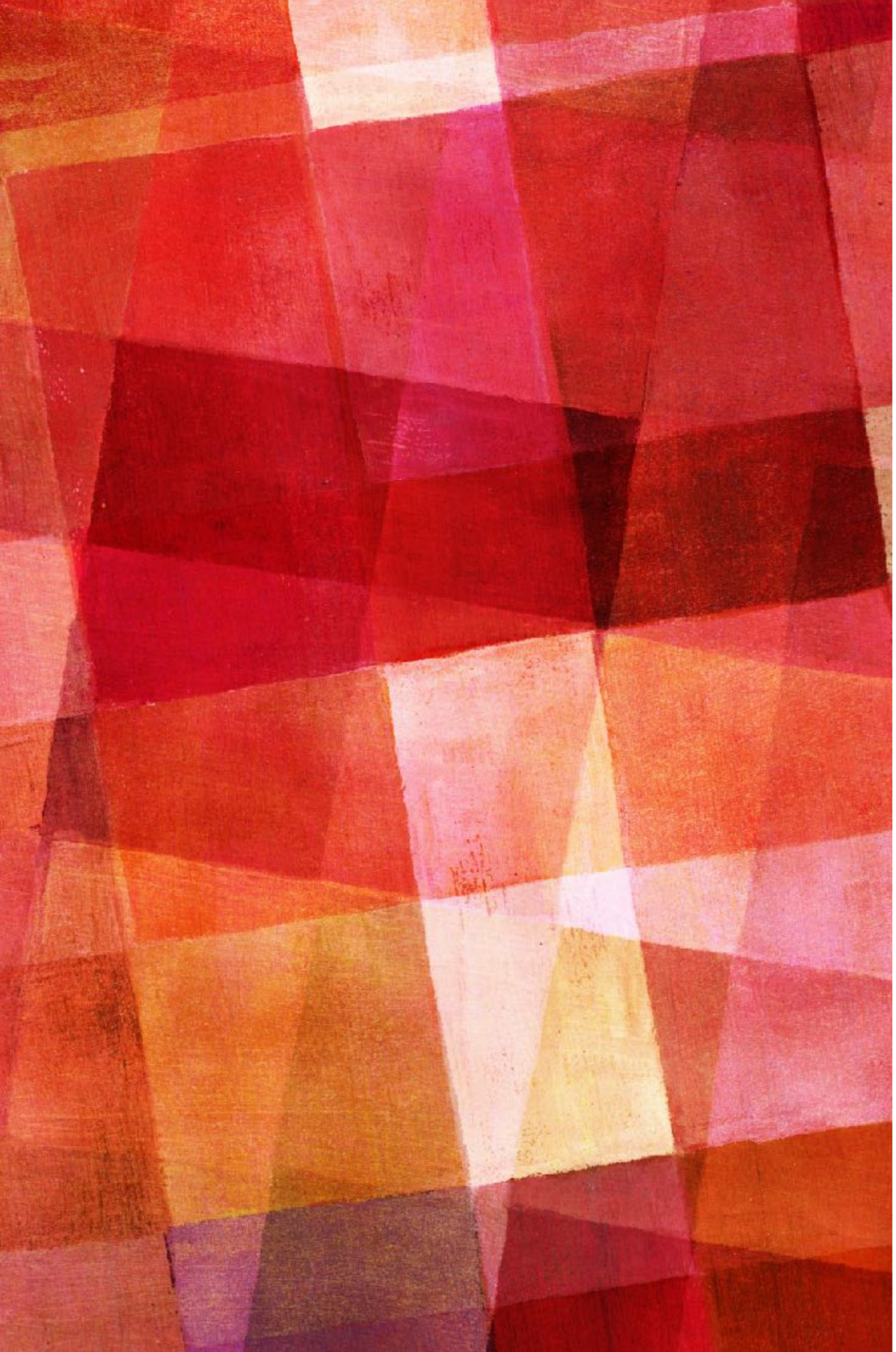
- Clusters grew from 1000 servers to 100,000 based on customer demand for SaaS apps
- Economies of scale pushed down cost of largest datacenter by factors 3X to 8X
  - Purchase, house, operate 100K v. 1K computers
- Traditional datacenters utilized 10% - 20%
- Make profit offering pay-as-you-go use at less than your costs for as many computers as you need

UTILITY COMPUTING / PUBLIC CLOUD COMPUTING

- Offers computing, storage, communication at pennies per hour
  +
- No premium to scale:

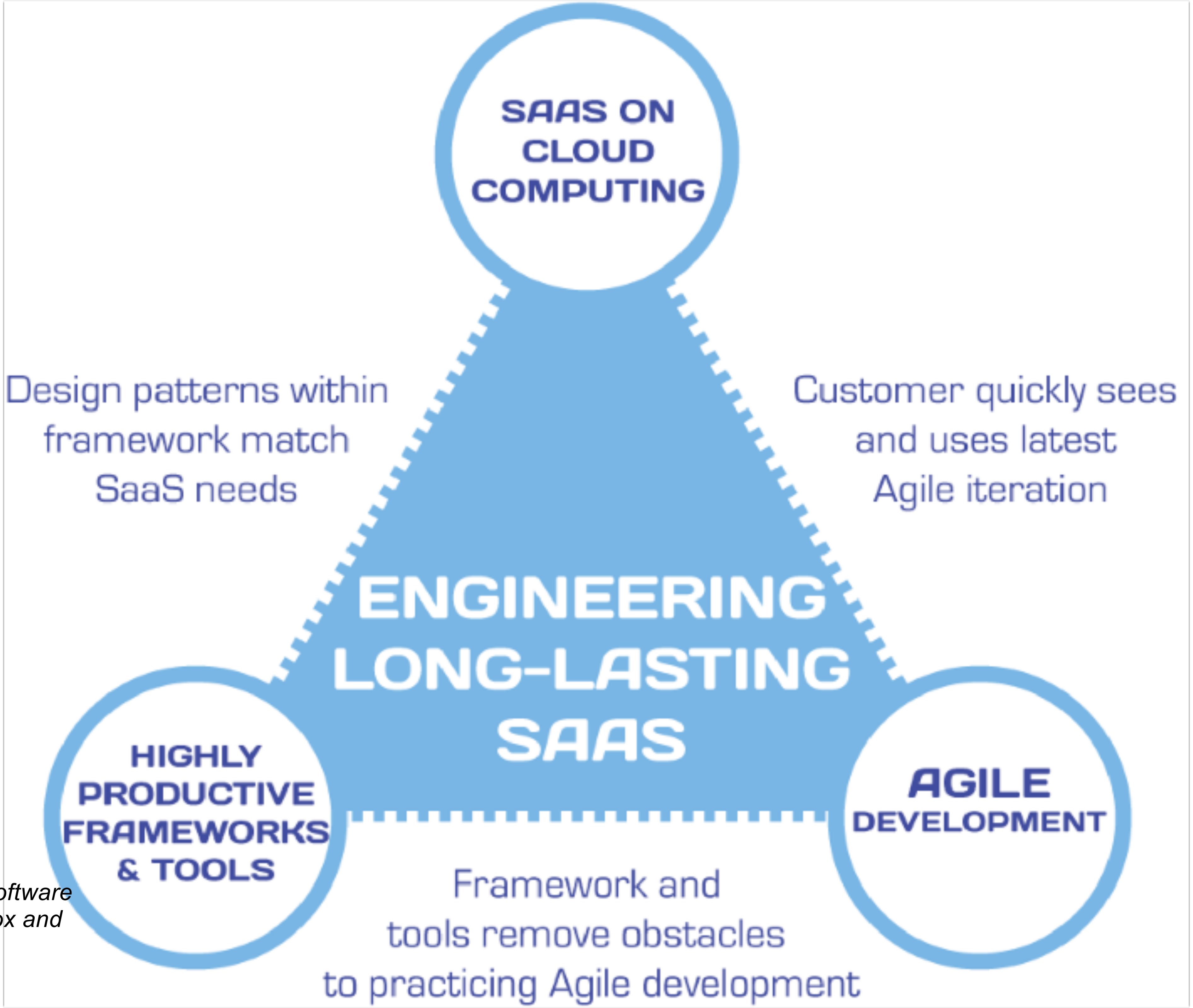
```
      1000 computers @      1 hour
  =       1 computer   @ 1000 hours
```

- Illusion of infinite scalability to cloud user
  - As many computers as you can afford
- Leading examples: Amazon Web Services, Google App Engine, Microsoft Azure



- SOA (Service Oriented Architectures)
  - SOA versus siloed- why?
  - Differences? Pros and Cons
    - Pros- Modularity, accessibility, communication, reliability, scalability, reuse
    - Cons- Execution time
- SAAS (Software as a Service)
  - On-demand, fast updates, less hardware dependency, etc…

SUMMARY: ENGINEERING SW IS MORE THAN PROGRAMMING


SAAS ON CLOUD COMPUTING
Design patterns within framework match SaaS needs
Customer quickly sees and uses latest Agile iteration
ENGINEERING LONG-LASTING SAAS
HIGHLY PRODUCTIVE FRAMEWORKS & TOOLS
AGILE DEVELOPMENT
Framework and tools remove obstacles to practicing Agile development

(Figure 1.11, Engineering Software as a Service by Armando Fox and David Patterson, 20124)

GETTING STARTED

- Project Discussion:
  - Workout journal
  - Track your journey
  - Beer Here
  - Concert/Venue Tracker
  - Parking Lot Helper
  - Band Tracker
  - Food/Nutrition Tracker
  - Zombie Survival
    - etc…
- Turn in at least 1 idea that your group all likes! Avoid stuff that will include a lot of Javascript or GUI.

REMINDER

- HW1 due in less than 2 weeks!
  - Start now!
  - Ask questions!

TIME TO START WORKING… DECISIONS, DECISIONS, DECISIONS

- What do the customers want?
- How should it be deployed?
- What resources are available?
  - Are there constraints?
  - Can resources be reused?
- What architecture should we use?
  - High level (ex: SOA vs Silo)
  - Mid level (ex: MVC, template, etc…)
- What about language?

SELECTING A LANGUAGE- SOME CONSIDERATIONS

- Compiled or Interpreted?
- Object oriented? Procedural?
- Level of complexity?
- Overhead?
- Ability to interact with other tools/languages/architectures?
- What’s special about the language?
- Is there a requirement?

- How do you decide?

Getting Started with a New Language
Beginning Ruby

Recommended: http://rubylearning.com/satishtalim/tutorial.html

Many other tutorials out there! Post great ones on Piazza!

Outline

- Three pillars of Ruby
- Everything is an object, and every operation is a method call
- OOP in Ruby
- Reflection and metaprogramming
- Functional idioms and iterators

Ruby is…

- Interpreted
- Object-oriented
  - *Everything is an object*
  - *Every operation is a method call on some object*
- Dynamically typed: objects have types, but variables don’t
- Dynamic
  - *add, modify code at runtime (metaprogramming)*
  - *ask objects about themselves (reflection)*
  - *in a sense all programming is metaprogramming*

Naming conventions

- ClassNames use UpperCamelCase

```
class FriendFinder …  end
```

- methods & variables use snake_case

```
def learn_conventions  …  end
def faculty_member?  …  end
def charge_credit_card!  …  end
```

- CONSTANTS (scoped) & $GLOBALS (not scoped)

```
    TEST_MODE = true        $TEST_MODE = true
```

- *symbols:* immutable string whose value is itself

```
favorite_framework = :rails
:rails.to_s == "rails"
"rails".to_sym == :rails
:rails == "rails"  # => false
```

Variables, Arrays, Hashes

- There are no declarations!
    - local variables must be assigned before use
    - instance & class variables ==nil until assigned
- OK: x = 3; x = 'foo'
- Wrong: Integer x=3
- Array: x = [1,'two',:three]
  x[1] == 'two' ; x.length==3
- Hash: w = {'a'=>1, :b=>[2, 3]}
  w[:b][0] == 2
  w.keys == ['a', :b]

Methods

- Everything (except fixnums) is pass-by-reference

```
def foo(x,y)
  return [x,y+1]
end

def foo(x,y=0)  # y is optional, 0 if omitted
  [x,y+1]       # last exp returned as result
end

def foo(x,y=0) ; [x,y+1] ; end
```

- Call with: `a,b = foo(x,y)` or `a,b = foo(x)` when optional arg used

Basic Constructs

- ➤Statements end with ';' or newline, but can span line if parsing is unambiguous

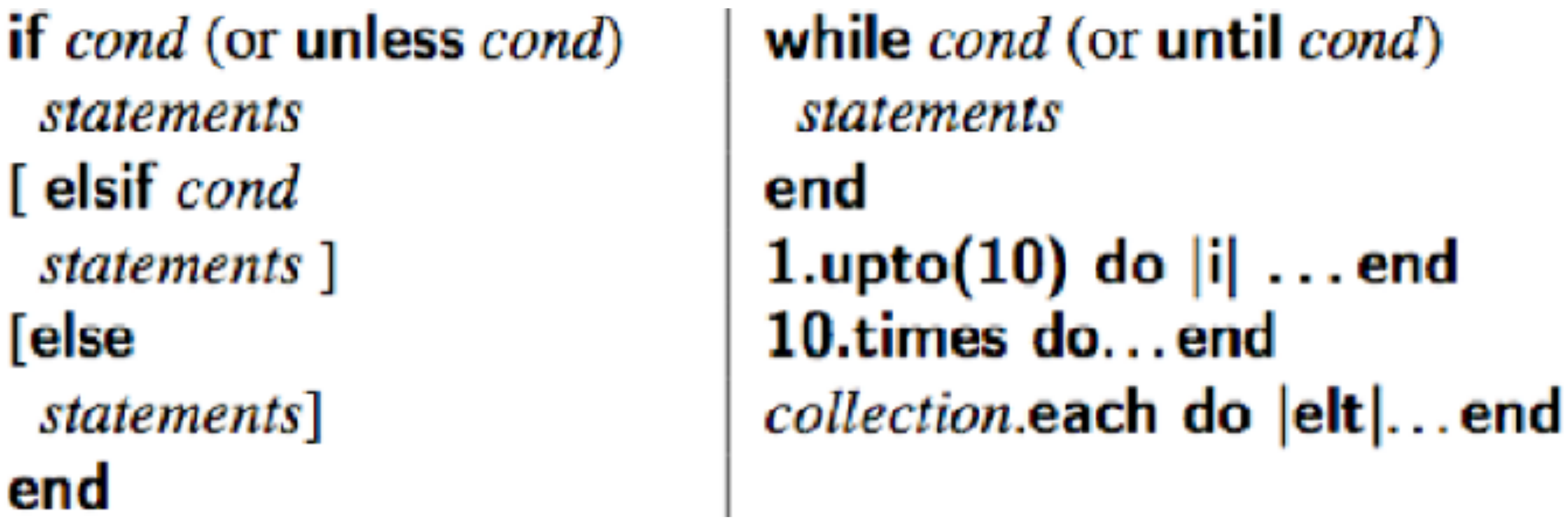
✔raise("Boom!") unless (ship_stable)

✖ raise("Boom!")
unless (ship_stable)

- ➤Basic Comparisons & Booleans:

== != < > =~ !~ true   false nil

- ➤ Th



```
if cond (or unless cond)
  statements
[ elsif cond
  statements ]
[else
  statements]
end
```

```
while cond (or until cond)
  statements
end
1.upto(10) do |i| ...end
10.times do...end
collection.each do |elt|...end
```

Strings & Regular Expressions

"string", %Q{string}, 'string', %q{string}

a=41 ; "The answer is #{a+1}"

- match a string against a regexp:

"kjustice@uccs.EDU" =~ /(.*)@(.*)\.edu$/i

/(.*)@(.*)\.edu$/i =~ "kjustice@uccs.EDU"

- If no match, value is false
- If match, value is non-false, and $1...$n *capture* parenthesized groups ($1 == 'kjustice', $2 == 'uccs')

/(.*)$/i or %r{(.*)$}i

or Regexp.new('(.*)$', Regexp::IGNORECASE)

- A real example...

http://pastebin.com/hXk3JG8m

MEET YOUR GROUPS

- Customer Ideas due Sunday
  - Title and 3-4 sentence summary of at least one project your group would like another group to develop
    - Make sure your whole team “cares” about the topic
    - Do not propose GUI or Javascript intensive apps
    - Need for external services encouraged
  - Communication Options (other than in person)
    - Groups in Piazza, Groups in BB
    - Skype, **Google Hangouts**
    - Avoid email, txts, etc… except for little questions

PAST IDEAS

- Workout journal
- Track your journey
- Beer Here
- Concert/Venue Tracker
- Parking Lot Helper
- Band Tracker
- Food/Nutrition Tracker
- Zombie Survival
    - etc…

EVERY THING IS AN OBJECT,
EVERY OPERATION IS A METHOD CALL

EVERYTHING IS AN OBJECT; (ALMOST) EVERYTHING IS A METHOD CALL

- Even lowly integers and nil are true objects:

```
57.methods
57.heinz_varieties
nil.respond_to?(:to_s)
```

- Rewrite each of these as calls to send:
  - Example: my_str.length => my_str.send(:length)

```
1 + 2
my_array[4]
my_array[3] = "foo"
if (x == 3)  ....
my_func(z)
```

1.send(:+, 2)
my_array.send(:[], 4)
my_array.send(:[]=, 3,"foo")
if (x.send(:==, 3)) ...
self.send(:my_func, z)

- in particular, things like “implicit conversion” on comparison is *not in the type system, but in the instance methods*

REMEMBER!

- a.b means: call method b on object a
  - a is the <u>*receiver*</u> to which you <u>*send*</u> the method call, assuming a will <u>*respond to*</u> that method

✘ *does not mean:* b is an instance variable of a

✘ *does not mean*: a is some kind of data structure that has b as a member

Understanding this distinction will save you from much grief and confusion

EXAMPLE: EVERY OPERATION IS A METHOD CALL

```
y = [1,2]
y = y + ["foo",:bar] # => [1,2,"foo",:bar]
y << 5               # => [1,2,"foo",:bar,5]
y << [6,7]           # => [1,2,"foo",:bar,5,[6,7]]
```

- “<<” *destructively modifies* its receiver, “+” does not
  - destructive methods often have names ending in “!”
- Remember! These are nearly all *instance methods* of Array—*not* language operators!
- So `5+3`, `"a"+"b"`, and `[a,b]+[b,c]` are all *different* methods named '+'
  - `Numeric#+`, `String#+`, and `Array#+`, to be specific

HASHES & POETRY MODE

```
h = {"stupid" => 1, :example=> "foo" }
h.has_key?("stupid") # => true
h["not a key"]     # => nil
h.delete(:example) # => "foo"
```

- Ruby idiom: “poetry mode”
  - using hashes to pass “keyword-like” arguments
  - omit hash braces when **last** argument to function is hash
  - omitting parens around function arguments

```
link_to("Edit”,{:controller=>'students', :action=>'edit'})
link_to "Edit", :controller=>'students', :action=>'edit'
```

- When in doubt, parenthesize defensively

- Ambiguous example: method taking 2 hashes

RUBY OOP AND OTHER PILLARS

POETRY MODE (+ SYNTACTIC SUGAR) IN ACTION

```
a.should(be().send(:>=,7))
a.should(be() >= 7)
a.should be >= 7
```

```
(redirect_to(login_page)) and return()
 unless logged_in?
redirect_to login_page and return unless
 logged_in?
```

Ruby OOP

Classes & inheritance

```
class SavingsAccount < Account    # inheritance
 # constructor used when SavingsAccount.new(...) called
 def initialize(starting_balance=0) # optional argument
   @balance = starting_balance
 end
 def balance   # instance method
   @balance   # instance var: visible only to this object
 end
 def balance=(new_amount)  # note method name: like setter
   @balance = new_amount
 end
 def deposit(amount)
   @balance += amount
 end
 @@bank_name = "MyBank.com"    # class (static) variable
 # A class method
 def self.bank_name   # note difference in method def
   @@bank_name
 end
 # or: def SavingsAccount.bank_name ; @@bank_name ; end
end
```

http://pastebin.com/m2d3myyP

Instance variables: shortcut

```
class SavingsAccount < Account
  def initialize(starting_balance)
    @balance = starting_balance
  end
  def balance
    @balance
  end
  def balance=(new_amount)
    @balance = new_amount
  end
end
```

Instance variables: shortcut

```
class SavingsAccount < Account
  def initialize(starting_balance)
    @balance = starting_balance
  end

  attr_accessor :balance


end
```

*attr_accessor is just a plain old method that uses metaprogramming...**not** part of the language!*

Review: Ruby's Distinguishing Features (So Far)

- Object-oriented with **no** multiple-inheritance
  - *everything is an object, even simple things like integers*
  - *class,instance variables invisible outside class*
- Everything is a method call
  - usually, only care if *receiver responds to method*
  - most "operators" (like +, ==) actually instance methods
  - Dynamically typed: objects have types; variables don't
- Destructive methods
  - Most methods are nondestructive, returning a new copy
  - Exceptions: <<, some destructive methods (eg merge vs. merge! for hash)
- Idiomatically, {} and () sometimes optional

FINISHING RUBY PILLAR CONCEPTS

All Programming is Metaprogramming

An international bank account!

```
acct.deposit(100)    # deposit $100

acct.deposit(euros_to_dollars(20))
  # about $25
```

An international bank account!

```
acct.deposit(100)     # deposit $100
acct.deposit(20.euros)    # about $25
```

- No problem with open classes....

```
class Numeric
  def euros ; self * 1.292 ; end
end
```

http://pastebin.com/f6WuV2rC

- But what about

```
acct.deposit(1.euro)
```

http://pastebin.com/WZGBhXci

METHOD MISSING

```
class Numeric
  def method_missing(method_id,*args,&block)  # capture all args in case need to call super
    if method_id.to_s == 'euro'
      self.send('euros')
    else
      super
    end
  end
end
```

The power of method_missing

- But suppose we also want to support

```
acct.deposit(1000.yen)

acct.deposit(3000.rupees)
```

- Surely there is a DRY way to do this?

http://pastebin.com/agjb5qBF

http://pastebin.com/HJTvUid5

MODIFYING METHOD MISSING

```
class Numeric
  @@currencies = {'yen' => 0.013, 'euro' => 1.292, 'rupee' => 0.019}
  def method_missing(method_id, *args, &block)   # capture all args in case
have to call super
    singular_currency = method_id.to_s.gsub( /s$/, '')
    if @@currencies.has_key?(singular_currency)
      self * @@currencies[singular_currency]
    else
      super
    end
  end
end
```

Introspection & Metaprogramming

- You can ask Ruby objects questions about themselves at runtime
- You can use this information to *generate new code* (methods, objects, classes) at runtime
- You can “reopen” any class at any time and add stuff to it.
    - *this is in addition to extending/subclassing*

Blocks, Iterators, Functional Idioms

Loops—but don't think of them that way

```
["apple", "banana", "cherry"].each do |string|
  puts string
end

for i in (1..10) do
  puts i
end

1.upto 10 do |num|
  puts num
end

3.times {  print "Rah, " }
```

If you're iterating with an index, you're probably doing it wrong

- *Iterators* let objects manage their own traversal
- (1..10).each do |x| ... end
  (1..10).each { |x| ... }
  1.upto(10) do |x| ... end => range traversal
- my_array.each do |elt| ... end=> array traversal
- hsh.each_key do |key| ... end
  hsh.each_pair do |key,val| ... end=> hash traversal
- 10.times {...} *# => iterator of arity zero*
- 10.times do ... end

- Pattern: Object on left -> ask it via the method `each` to manage traversal of object's elements one at a time

“Expression orientation”

```
x = ['apple','cherry','apple','banana']
x.sort # => ['apple','apple','banana','cherry']
x.uniq.reverse # => ['banana','cherry','apple']
x.reverse!  # => modifies x
x.map do |fruit|
  fruit.reverse
end.sort
  # => ['ananab','elppa','elppa','yrrehc']
x.collect { |f| f.include?("e") }
x.any? { |f| f.length > 5 }
```

- A real life example....

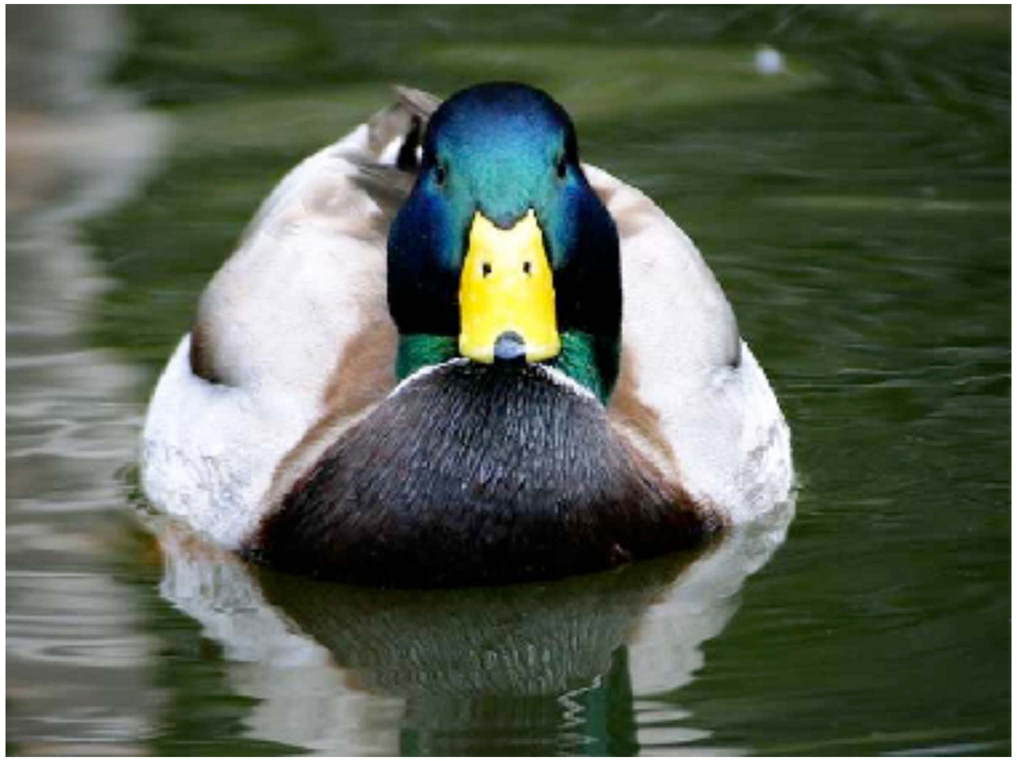
http://pastebin.com/Aqgs4mhE

Mixins and Duck Typing

What is “duck typing”?

- If it responds to the same methods as a duck…it might as well be a duck
- More than just overloading; similar to Java Interfaces
- Example: `my_list.sort`

```
[5, 4, 3].sort
["dog", "cat", "rat"].sort
[:a, :b, :c].sort
IO.readlines("my_file")
```



Modules

➤A *module* is a collection of class & instance methods that are not actually a class

➤you can't instantiate it

➤Some modules are *namespaces*, similar to Python: Math::sin(Math::PI / 2.0)

➤The more interesting ones let you *mix the methods into* a class: class A < B ; include MyModule ; end

➤A.foo will search A, then MyModule, then B

➤sort is actually defined in module Enumerable,

which is *mixed into* Array by default

A Mix-in Is A Contract

- Example: Enumerable assumes objects of target class respond to each
  - ...provides all?, any?, collect, find, include?, inject, map, partition, ...
- Example: Comparable assumes that objects of target class respond to <=>
  - provides < <= => > == between? for free
- Enumerable also provides sort, which requires *elements* of target class (things returned by each) to respond to <=>

Class of objects doesn't matter: only methods to which they respond

Example: sorting a file

- Sorting a file
- File.open returns an IO object
- IO objects respond to each by returning each line as a String
- So we can say File.open('filename.txt').sort
- relies on IO#each and String#<=>
- Which lines of file begin with vowel?

```
File.open('file').
select { |s| s =~ /^[aeiou]/i }
```

Making accounts comparable

- Just define <=> and then use the Comparable module to get the other methods

- Now, an Account quacks like a numeric ☺

http://pastebin.com/itkpaqMh

```
class Account
  include Comparable
  def <=>(other)
    self.balance <=> other.balance
  end
```

When Module? When Class?

- Modules reuse *behaviors*
  - *high-level behaviors that could conceptually apply to many classes*
  - *Example: Enumerable, Comparable*
  - *Mechanism: mixin (include Enumerable)*
- Classes reuse *implementation*
  - *subclass reuses/overrides superclass methods*
  - *Mechanism: inheritance (class A < B)*
- Remarkably often, we will *prefer composition over inheritance*

yield()

Blocks (anonymous λ)

```
(map '(lambda (x) (+ x 2)) mylist )
mylist.map { |x| x+2 }
```

```
(filter '(lambda (x) (even? x)) mylist)
mylist.select do |x| ; x.even? ; end
```

```
(map
  '(lambda (x) (+ x 2))
  (filter '(lambda (x) (even? x)) mylist))
mylist.select {|x| x.even?}.map {|x| x+2 }
```

Turning iterators inside-out

- Java:
- You hand me each element of that collection in turn.
- I'll do some stuff.
- Then I'll ask you if there's any more left.
- Ruby:
- Here is some code to apply to every element of the collection.
- *You* manage the iteration or data structure traversal.
- Let's do an example…

http://pastebin.com/T3JhV7Bk

Iterators are just one nifty use of yield

```
# in some other library
def before_stuff
  ...before code...
end
def after_stuff
  ...after code...
end

# in your code
def do_everything
  before_stuff()
  my_custom_stuff()
  after_stuff()
end
```

Without yield(): expose 2 calls in other library

```
# in some other library
def around_stuff
  ...before code...
  yield
  ...after code...
end

# in your code
def do_everything
  around_stuff do
    my_custom_stuff()
  end
end
```

Blocks are Closures

- A *closure* is the set of all variable bindings you can “see” at a given point in time
  - In Scheme, it’s called an *environment*
- *Blocks are closures:* they carry their environment around with them

- Result: blocks can help reuse by separating *what to do* from *where & when to do it*
  - We’ll see various examples in Rails

http://pastebin.com/zQPh70NJ

Summary

- Duck typing for re-use of behaviors
    - In Ruby, it's achieved with "mix-ins" via the Modules mechanism, and by the "everything is a message" language semantics
- Blocks and iterators
    - Blocks are anonymous lambdas that *carry their environment around with them*
    - Allow "sending code to where an object is" rather than passing an object to the code
    - Iterators are an important special use case