

The Building Blocks of Modularity

Jim Weirich
Chief Scientist
EdgeCase LLC
@jimweirich

Copyright 2009, Jim Weirich
All Rights Reserved



Tech Interview

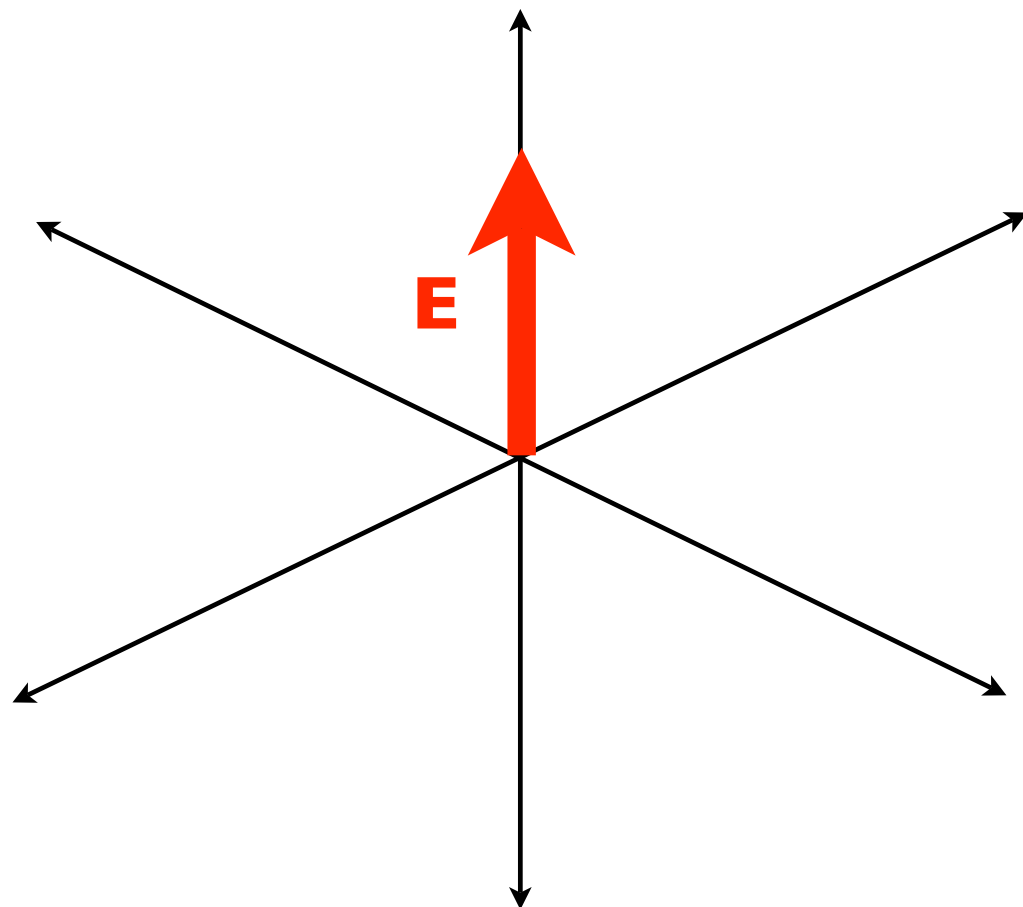
“What do you look for
in a good design?”

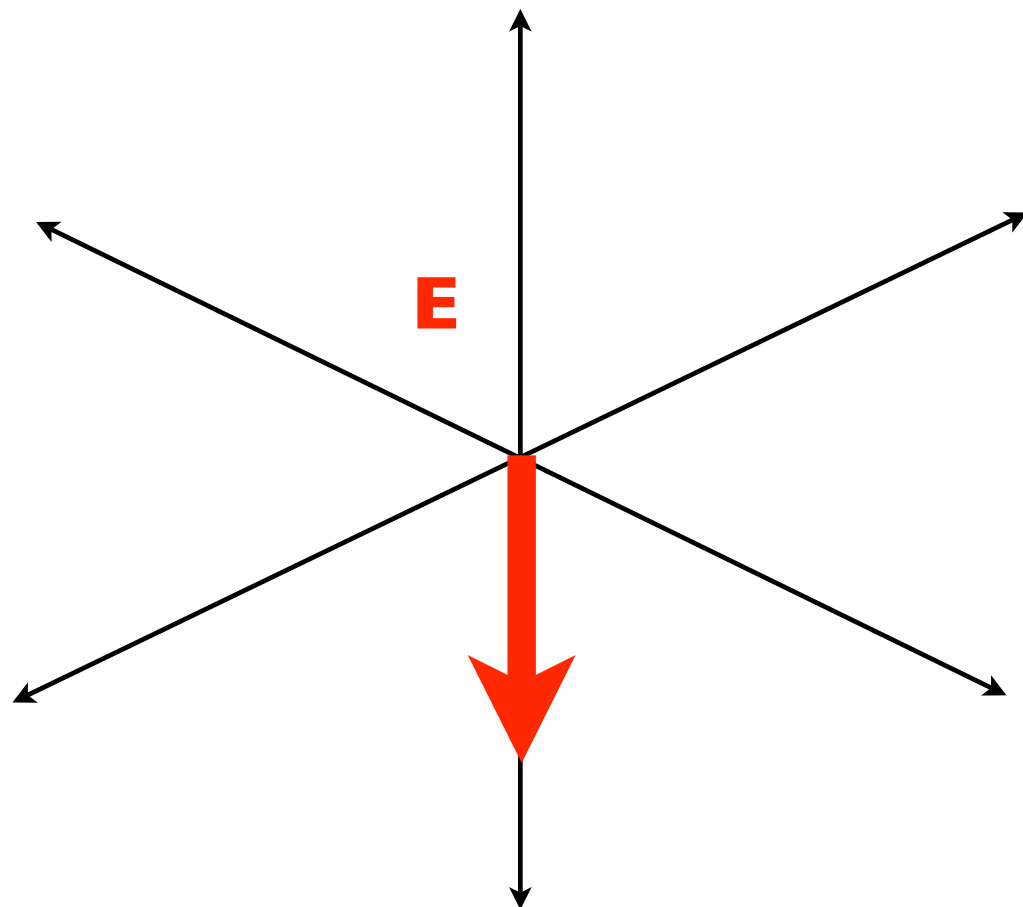
Ummm?

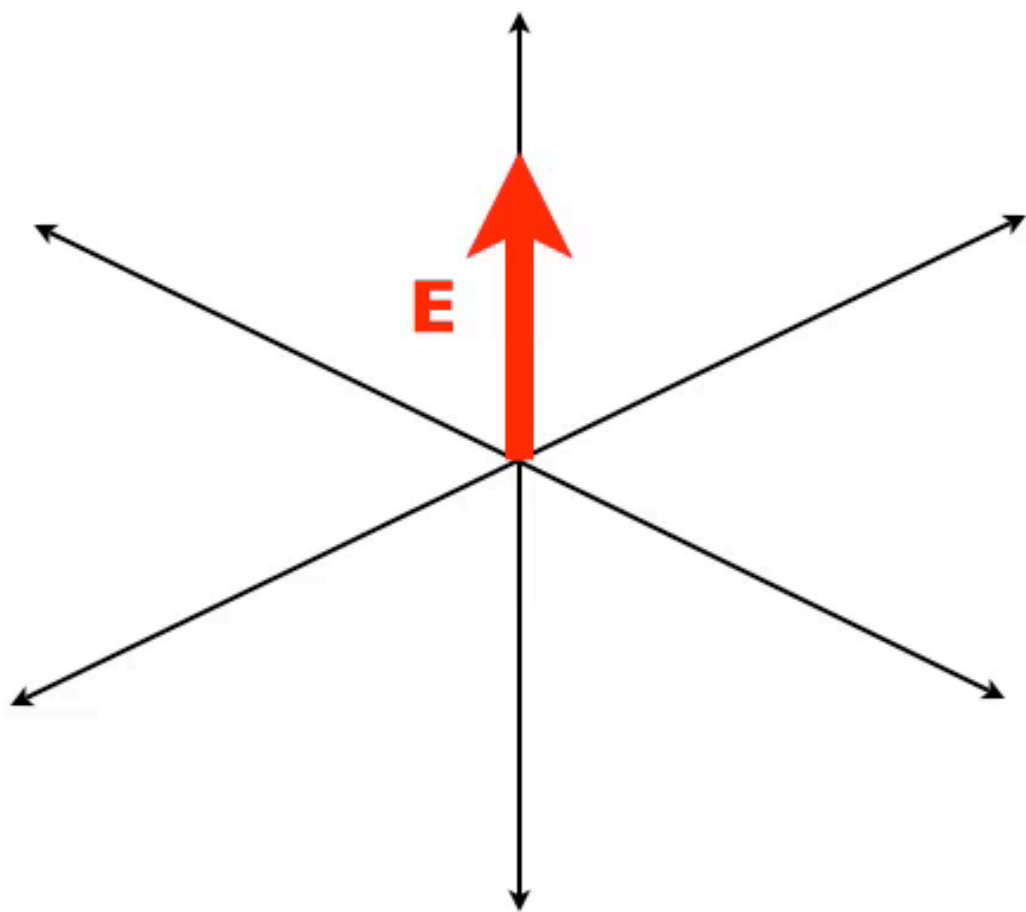


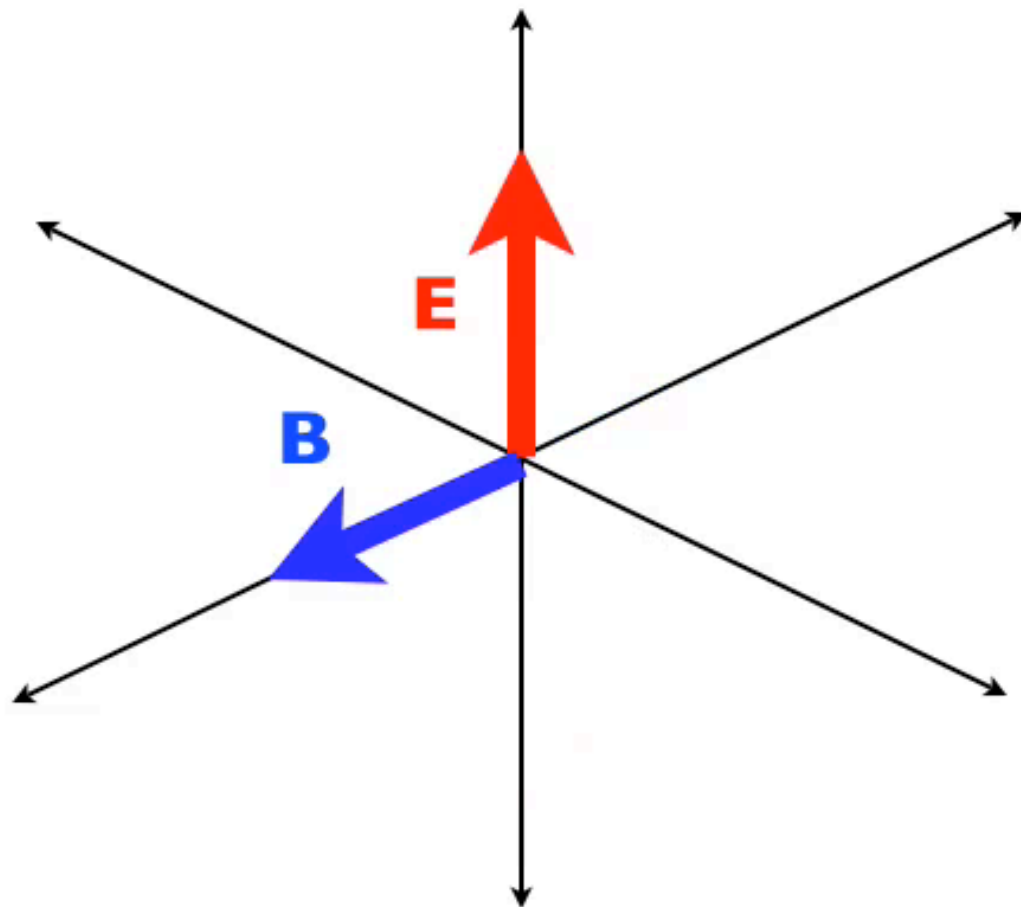












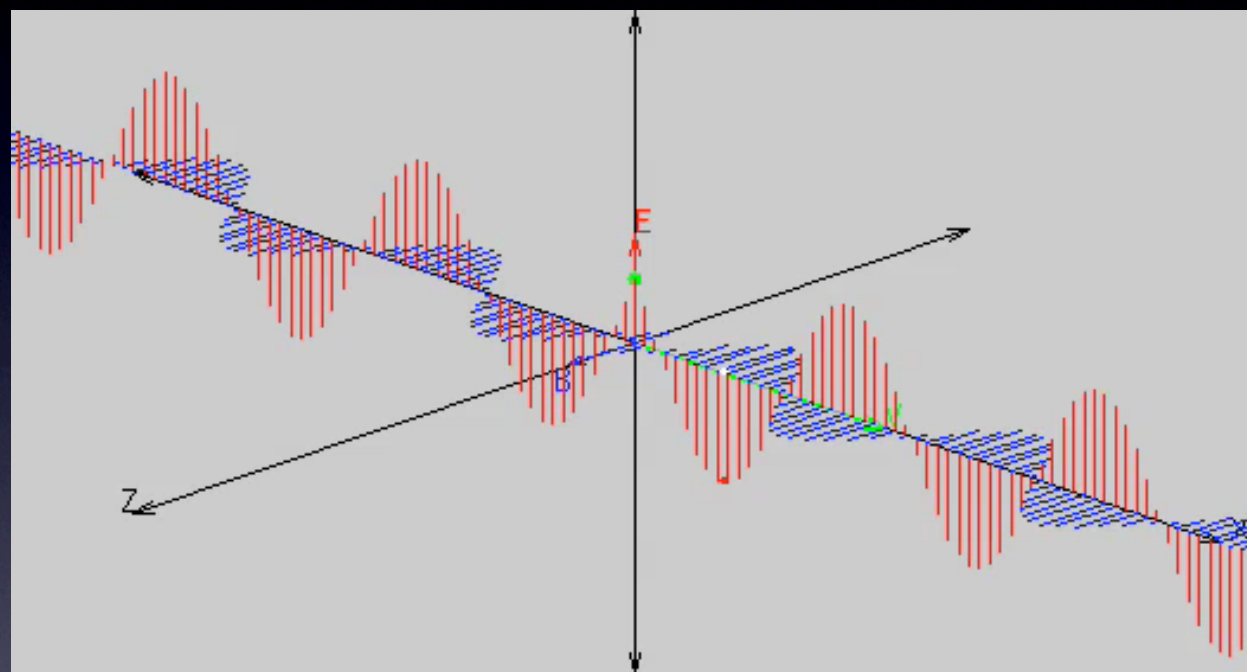
Maxwell's Equations

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t}$$



Unified!

Electric Fields
+
Magnetic Fields = Electro-
magnetism



†Natural and artificial flavors

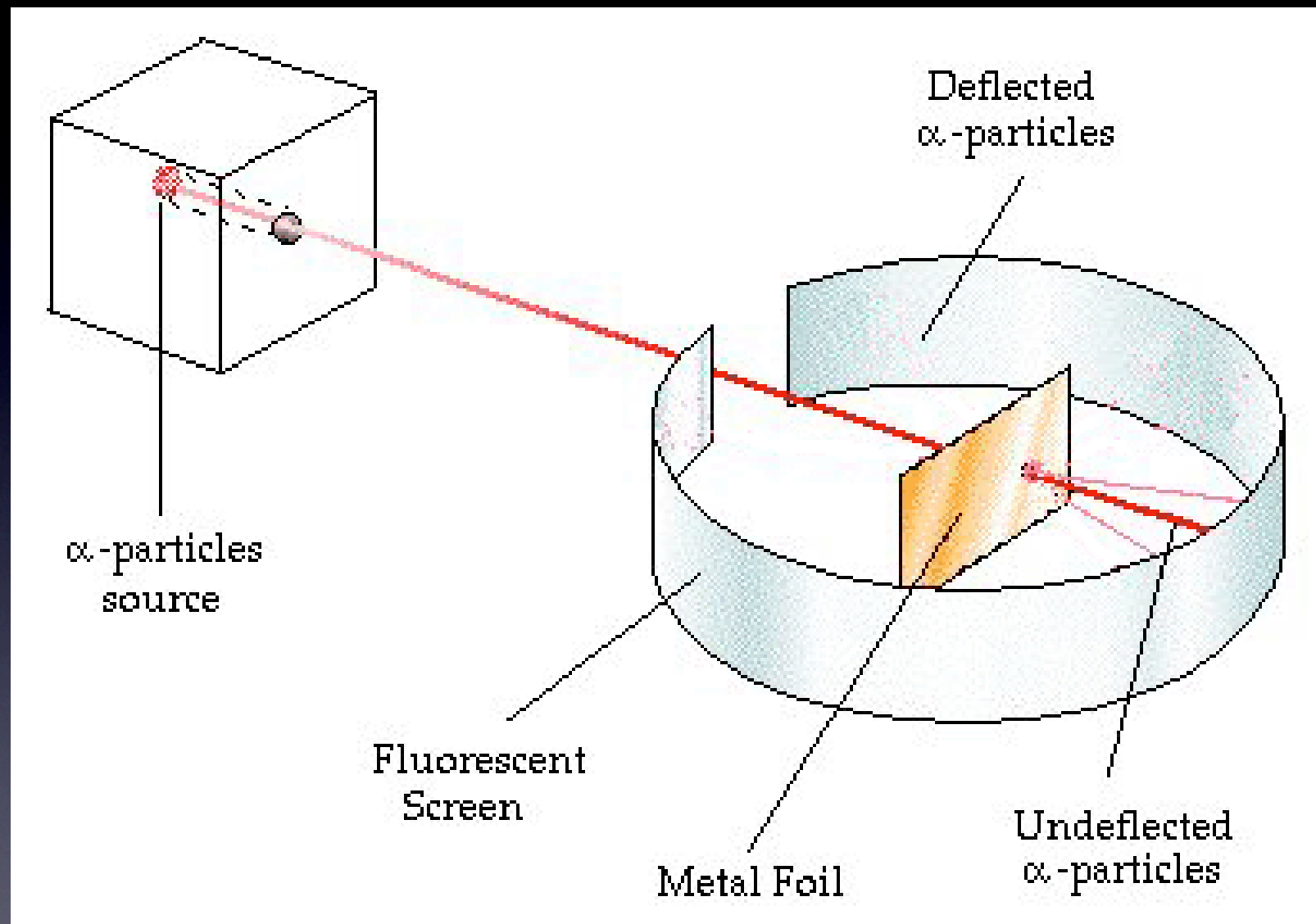


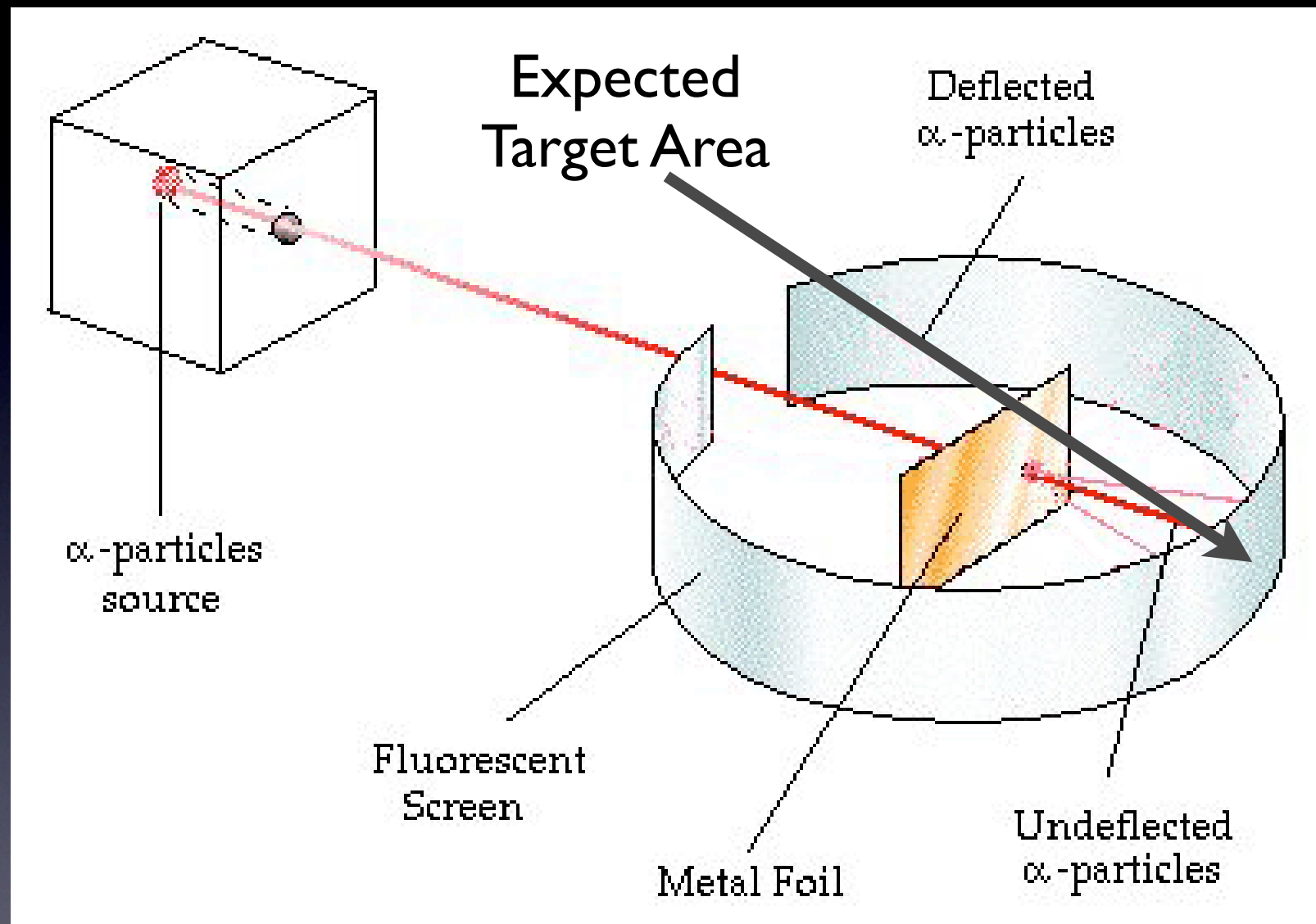
†Natural and artificial flavors

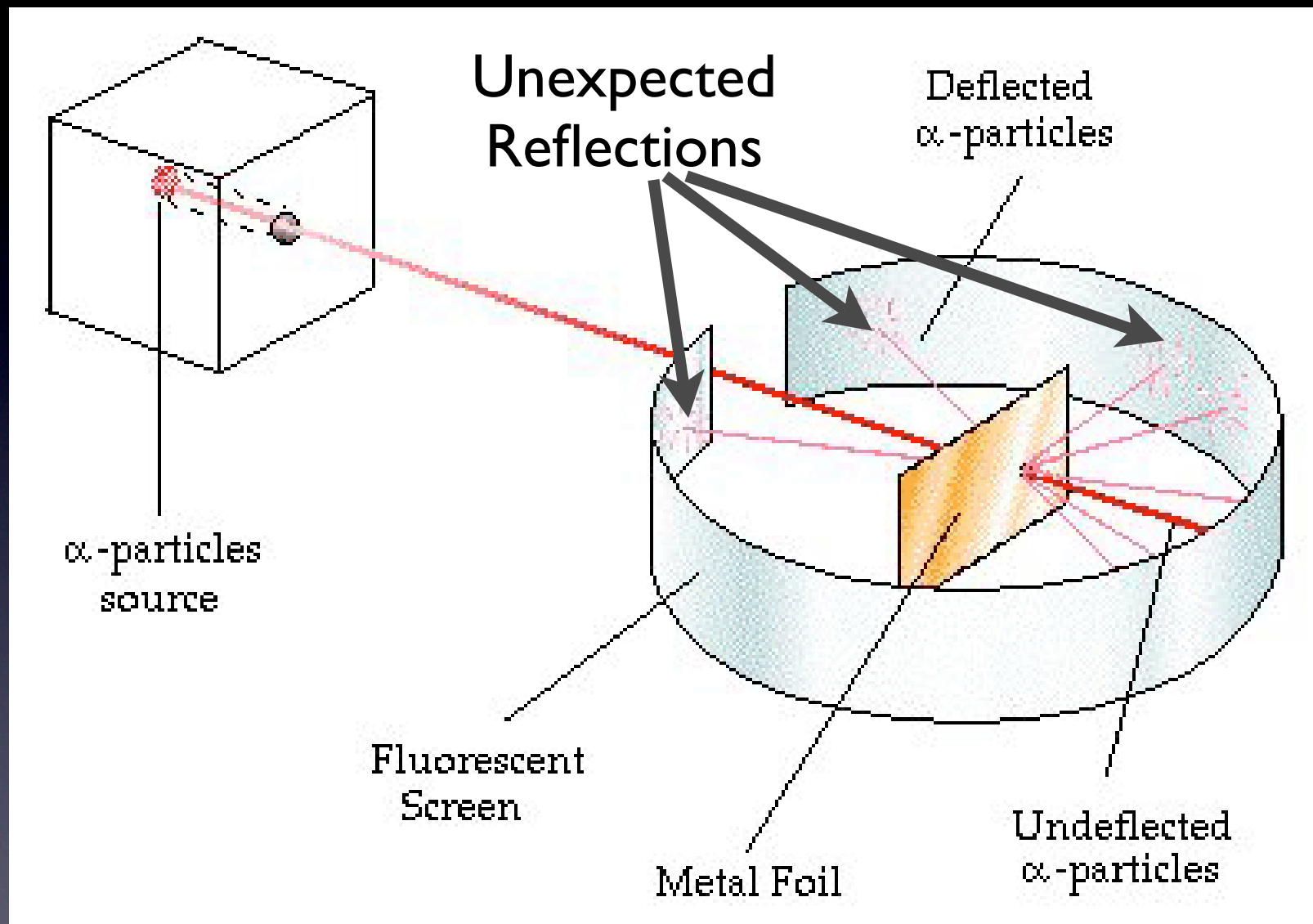
Ernest Rutherford



1909

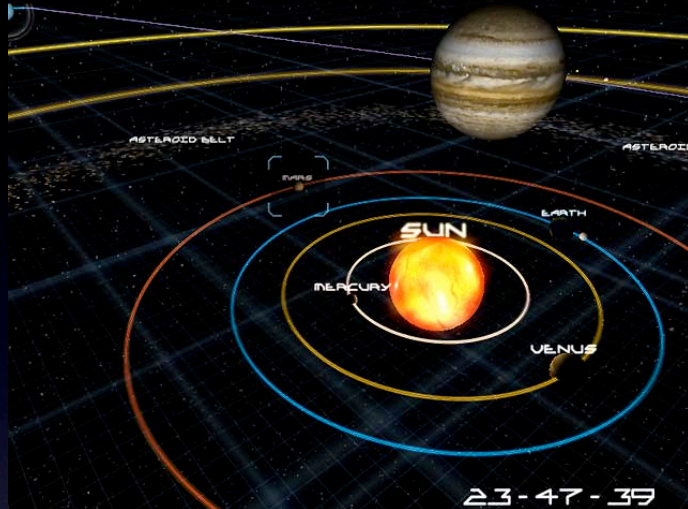




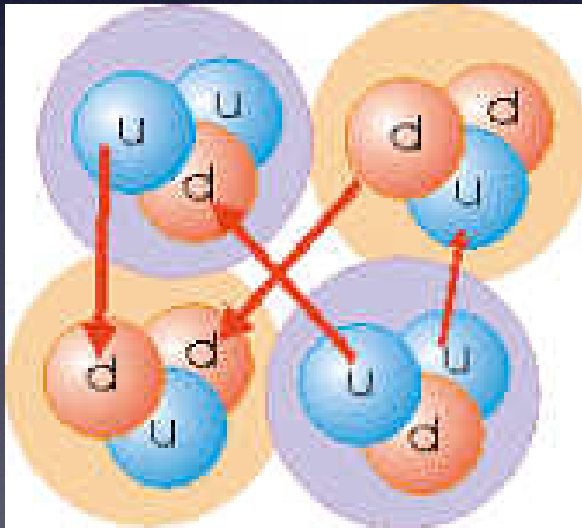
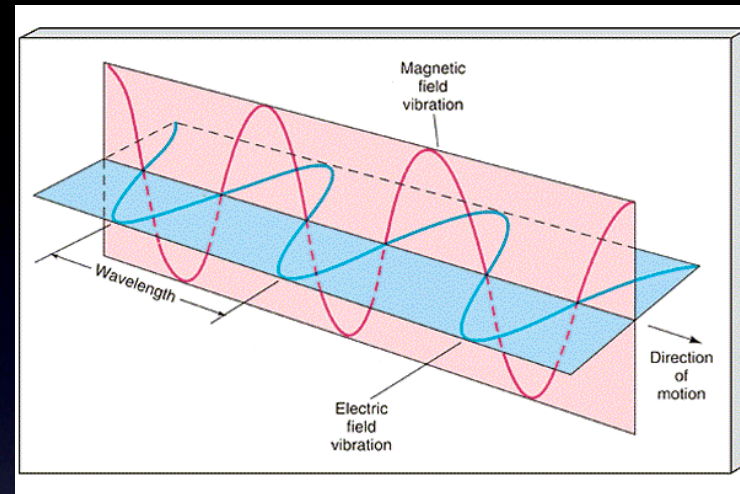




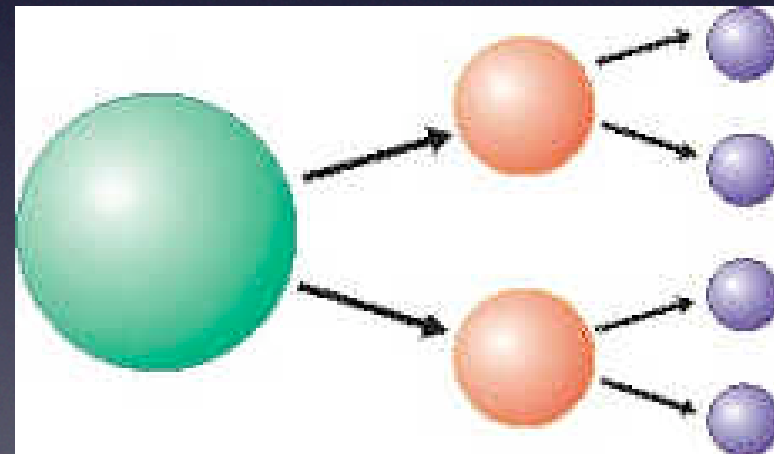
Gravity



Electromagnetism



Strong Nuclear

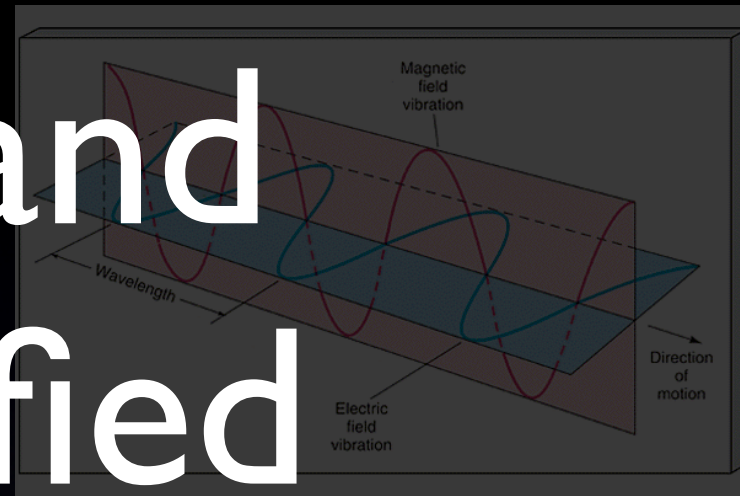


Weak Nuclear

Gravity

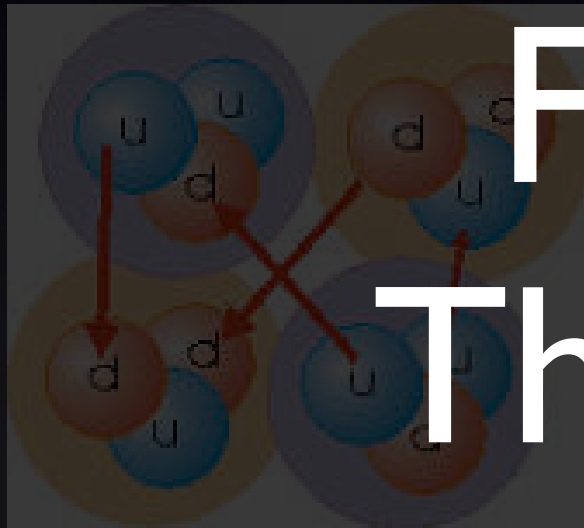


Electromagnetism

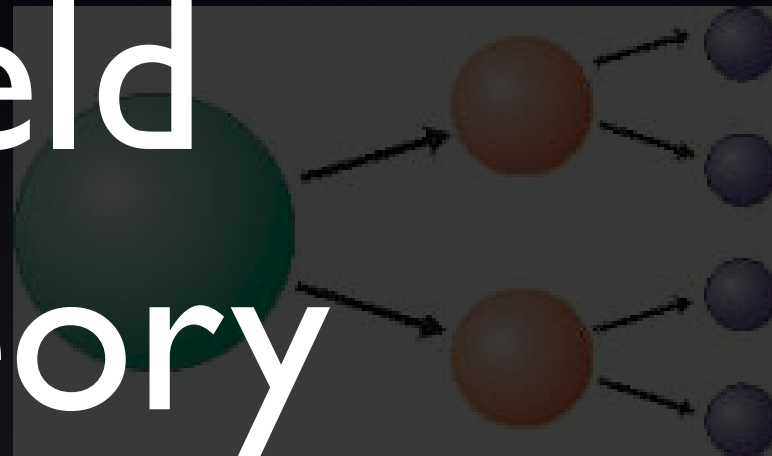


Grand Unified

Field Theory



Strong Nuclear



Weak Nuclear

Some Principles ...

- SOLID
- Law of Demeter
- DRY
- Small Methods
- Design by Contract

Some Principles ...

Grand Unified Theory of Software Development

- Law of Demeter

- DRY

- Small Methods

- Design by Contract

The Grand Unified Theory of Software Development

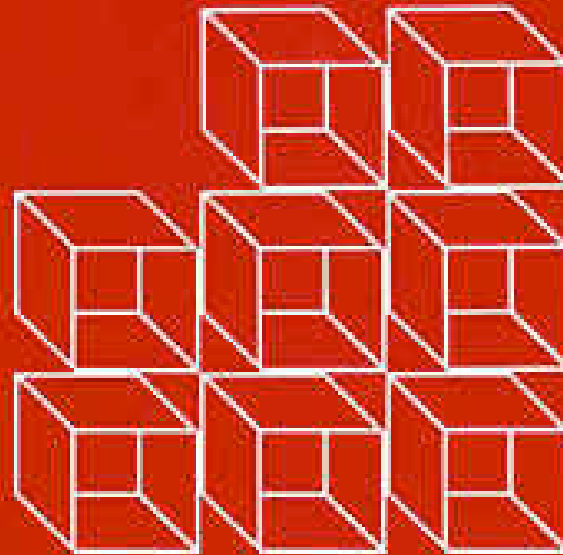
Jim Weirich
Chief Scientist
EdgeCase LLC
@jimweirich

Copyright 2009, Jim Weirich
All Rights Reserved





**COMPOSITE/STRUCTURED
DESIGN** GLENFORD J. MYERS



1978

Coupling & Cohesion

Types of Coupling

**No
Coupling**

**Data
Coupling**

**Stamp
Coupling**

**Control
Coupling**

**External
Coupling**

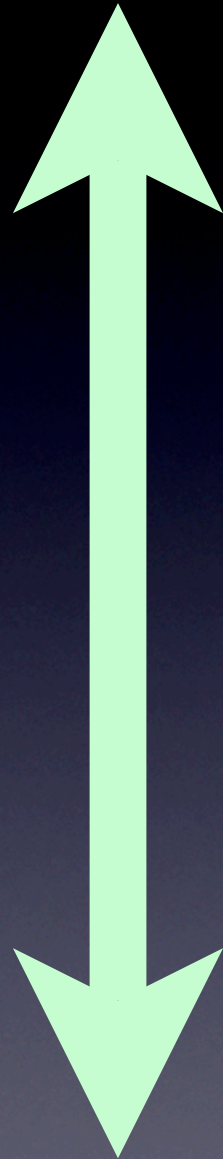
**Common
Coupling**

**Content
Coupling**

Types of Coupling

**Less
Coupling
(good)**

**More
Coupling
(bad)**



**No
Coupling**

**Data
Coupling**

**Stamp
Coupling**

**Control
Coupling**

**External
Coupling**

**Common
Coupling**

**Content
Coupling**

**No
Coupling**

**Data
Coupling**

**Stamp
Coupling**

**Control
Coupling**

**External
Coupling**

**Common
Coupling**

**Content
Coupling**

**No
Coupling**

**Data
Coupling**

**Stamp
Coupling**

**Control
Coupling**

**External
Coupling**

**Common
Coupling**

**Content
Coupling**

**No
Coupling**

**Data
Coupling**

**Stamp
Coupling**

**Control
Coupling**

**External
Coupling**

**Common
Coupling**

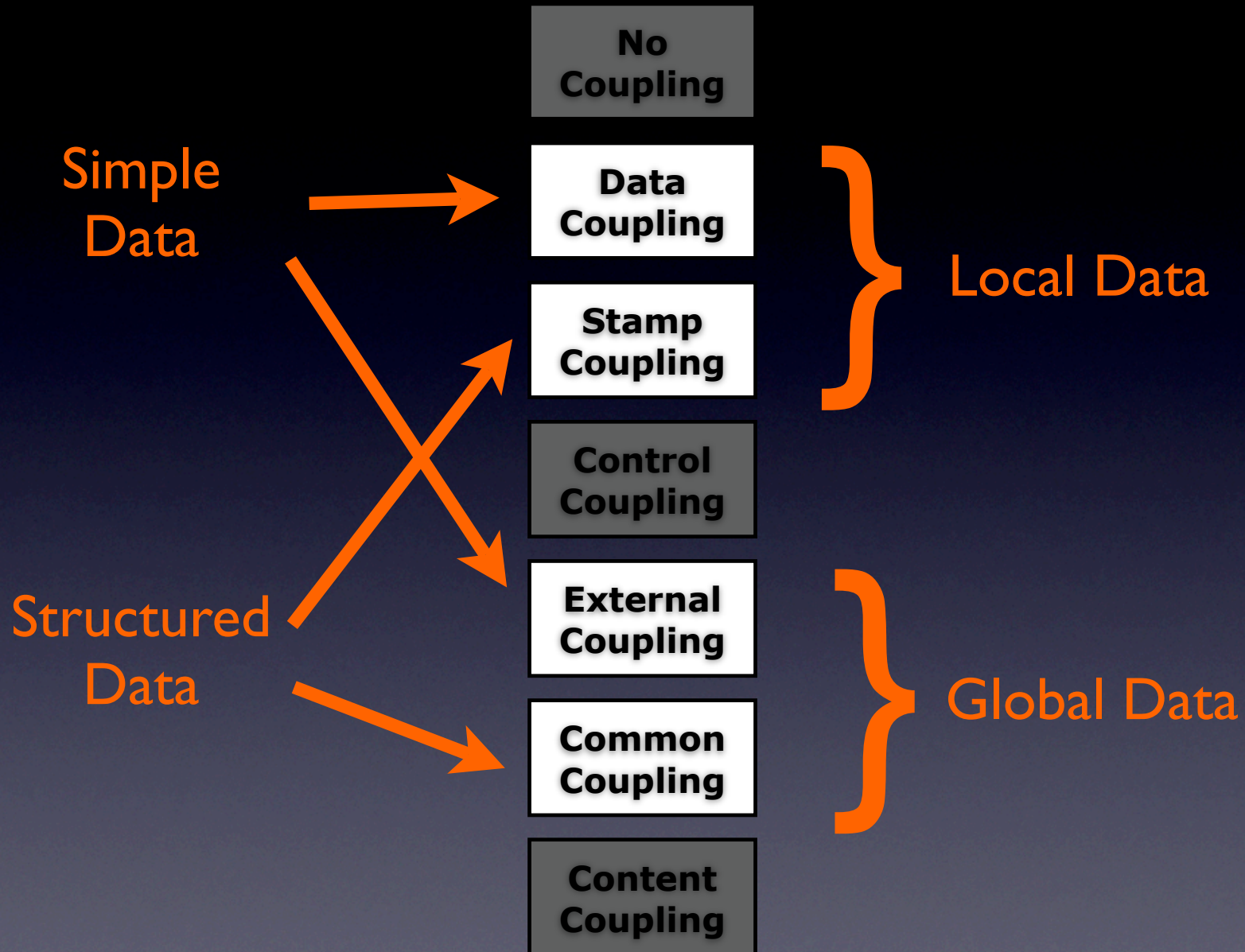
**Content
Coupling**



Local Data



Global Data



**No
Coupling**

**Data
Coupling**

**Stamp
Coupling**

**Control
Coupling**

**External
Coupling**

**Common
Coupling**

**Content
Coupling**

Control Coupling

- Method has a “flag” parameter
- The flag controls which algorithm to use

Control Coupling

- Method has a “flag” parameter
- The flag controls which algorithm to use
- **Symptoms**
 - The word “OR” in description
 - Flag value is arbitrary and not related to problem domain.

Control Coupling

```
Array.instance_methods
```

Control Coupling

```
Array.instance_methods
```

```
Array.instance_methods(true)
```

```
Array.instance_methods(false)
```


Control Coupling

```
Array.instance_methods  
Array.instance_methods(true)  
Array.instance_methods(false)
```

... the instance methods in *mod*
are returned, otherwise the
methods in *mod* and *mod's*
superclasses are returned.

Control Coupling

```
Array.instance_methods  
Array.instance_methods(true)  
Array.instance_methods(false)
```

... the instance methods in *mod*
are returned, **otherwise** the
methods in *mod* and *mod*'s
superclasses are returned.

Another Example?

Control Coupling

```
Customer.find(:first, ...)  
Customer.find(:all, ...)
```


Control Coupling

Returns object

```
Customer.find(:first, ...)  
Customer.find(:all, ...)
```

Returns list of objects

Myer's Classifications
were 'OK'

Failed to extend
well to Objects and
Dynamic Languages

WHAT EVERY PROGRAMMER SHOULD KNOW ABOUT OBJECT- ORIENTED DESIGN



MEILIR PAGE-JONES

Foreword by
Larry L. Constantine



1996

Connascence

1. The common birth of two or more at the same time; production of two or more together.
2. That which is born or produced with another.
3. The act of growing together.

Connascence

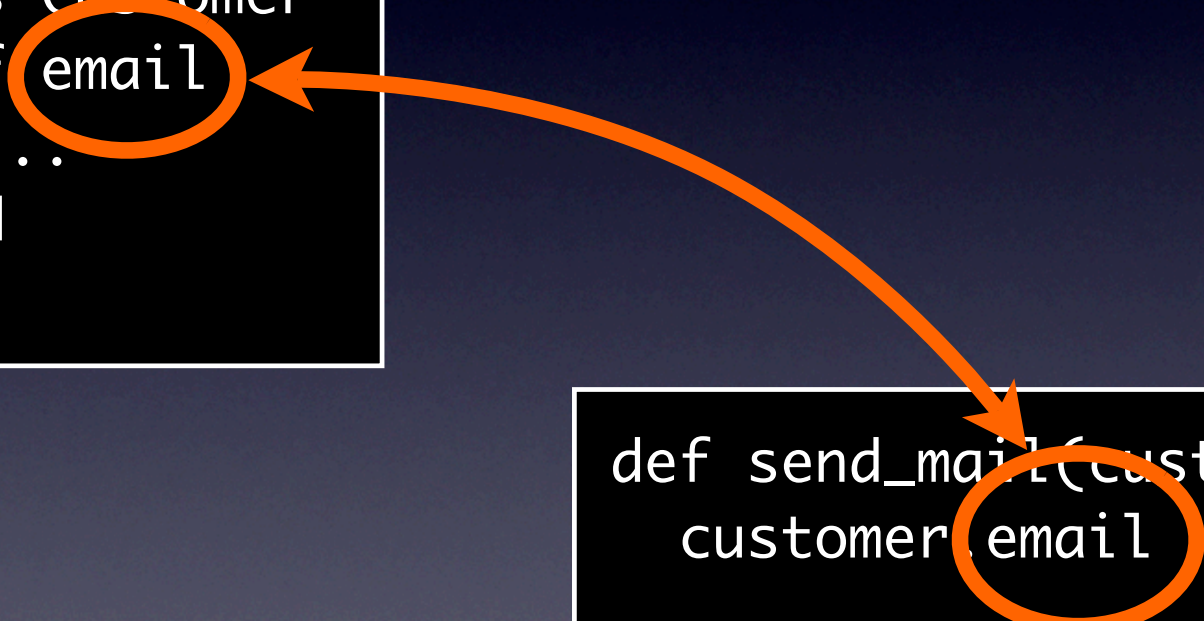
Two pieces of software share *connascence* when a change in one requires a corresponding change in the other.

CoN

```
class Customer
  def email
    ...
  end
end
```

```
def send_mail(customer)
  customer.email
  ...
end
```

```
class Customer
  def email
    ...
  end
end
```

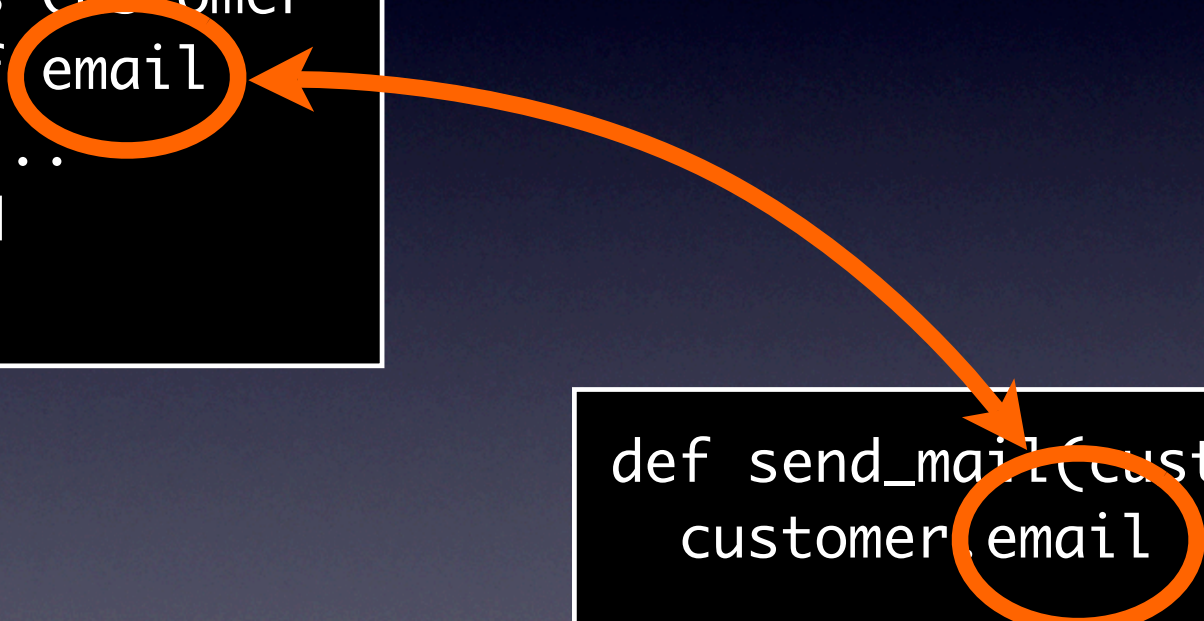


The diagram illustrates a method call. An orange arrow originates from the `email` attribute access in `customer.email` within the `send_mail` function and points to the `email` method definition inside the `Customer` class. Both the `email` text in the function call and the `email` text in the class definition are circled in orange.

```
def send_mail(customer)
  customer.email
  ...
end
```


Connascence of Name

```
class Customer
  def email
    ...
  end
end
```



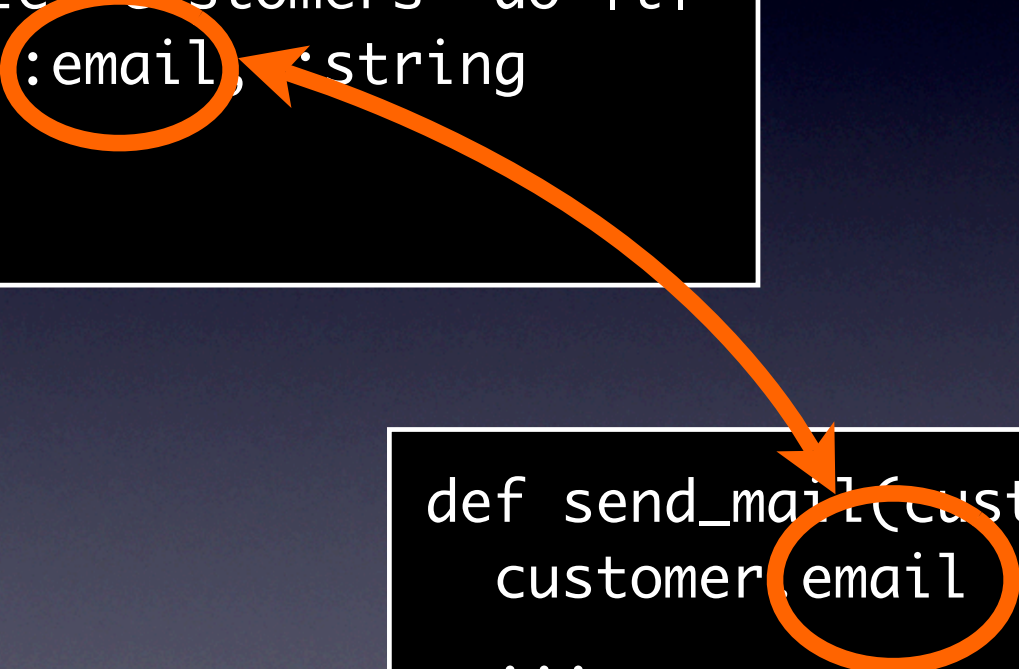
The diagram illustrates the concept of 'Connascence of Name'. It shows two code snippets. The first snippet is a Ruby class definition for 'Customer' with a method 'email'. The second snippet is a function definition 'send_mail(customer)' that calls 'customer.email'. An orange arrow points from the 'email' method in the first snippet to the 'email' attribute access in the second snippet. Both 'email' occurrences are circled in orange, highlighting that they share the same name but are accessed differently (one as a method, one as an attribute).

```
def send_mail(customer)
  customer.email
  ...
end
```

Connascence of Name

```
create_table "customers" do |t|  
  t.column :email, :string  
  ...  
end
```

```
def send_mail(customer)  
  customer.email  
  ...  
end
```



Connascence of Name

Another example?

```
class Customer
  def email
    ...
  end
end
```

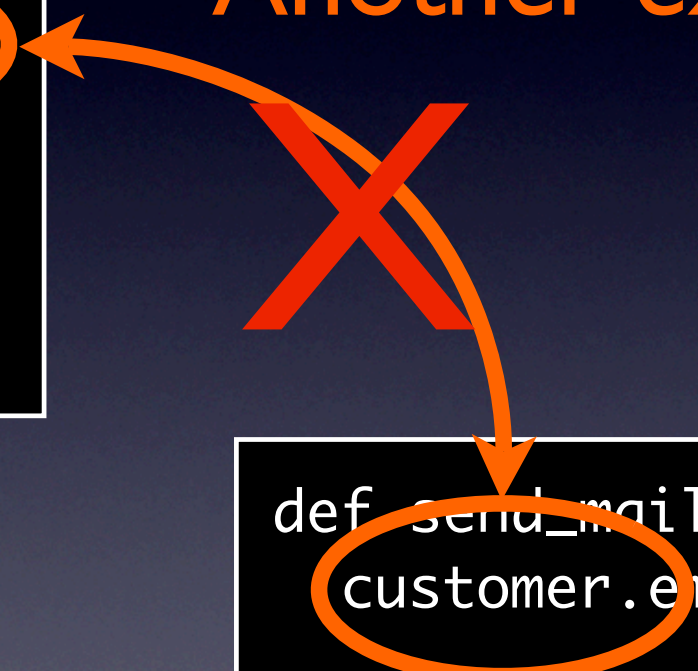
```
def send_mail(customer)
  customer.email
  ...
end
```


Connascence of Name

Another example?

```
class Customer
  def email
    ...
  end
end
```

```
def send_email(customer)
  customer.email
  ...
end
```

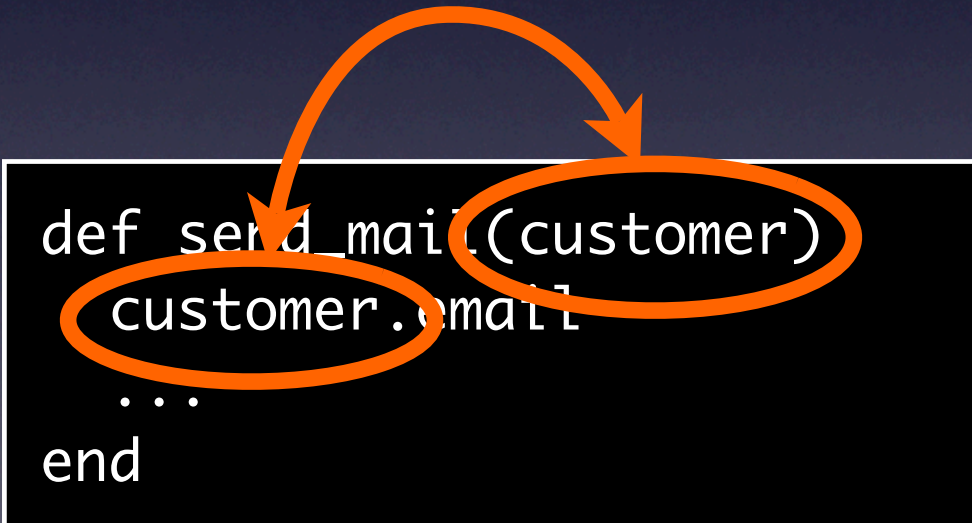


The diagram illustrates a lack of connascence of name. A red 'X' is placed between the two code blocks. An orange arrow points from the 'Customer' class in the first block to the 'customer.email' method call in the second block. The 'Customer' class and 'customer.email' are both circled in orange, highlighting the mismatch between the class name and the method name used in the call.

Connascence of Name

Another example?

```
class Customer
  def email
    ...
  end
end
```

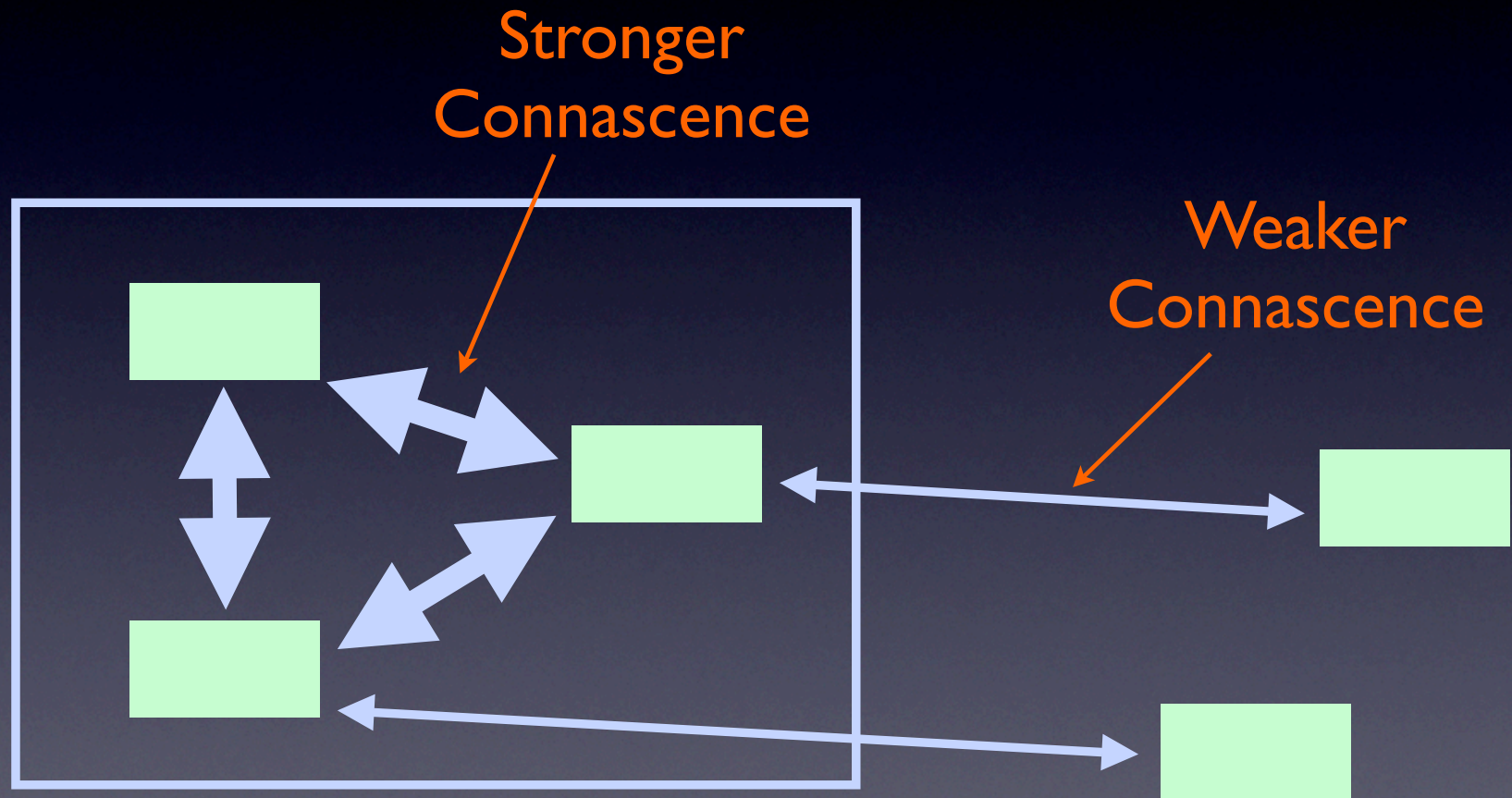


The diagram illustrates a call to the `email` method on a `customer` object. An orange arrow originates from the `customer` parameter in the `send_mail` function definition and points to the `customer` object in the `customer.email` expression. Both the `customer` parameter and the `customer` object are circled in orange.

```
def send_mail(customer)
  customer.email
  ...
end
```

Locality Matters

Rule of Locality



Rule of Locality

As the distance between software elements increases, use weaker forms of connascence.

CoP


```
:orders => {  
  "3" => "1",  
  "5" => "2"  
}
```

Translate params hash
to
A List of Pairs



```
[  
  [ Order.find(3), true ],  
  [ Order.find(5), false ],  
]
```


```
def process_orders(list_of_pairs)
  list_of_pairs.each do |order, expedite|
    # handle an order
  end
end
```

Order of the data
within the pair
is significant

```
[
  [ Order.find(3), true ],
  [ Order.find(5), false ],
]
```

```
class OrdersController
  def build_order_list(params)
    [order, flag]
  end
end
```


```
class Orders
  def process_orders(pairs)
    pairs.each do |order, flag| ... end
  end
end
```



Connascence of Position

```
class OrdersController
  def build_order_list(params)
    [order, flag]
  end
end
```

```
class Orders
  def process_orders(pairs)
    pairs.each do |order, flag| ... end
  end
end
```



Consider

Low Degree of CoP

[order, expedite]

High Degree of CoP

```
[  
  order, expedite, confirmation_number,  
  ordered_date, expiration, special  
]
```

CoP → CoN

```
class OrderDisposition
  attr_reader :order,
    :expedite,
    :confirmation_number,
    :ordered_date,
    :expiration,
    :special
  ...
end
```

Degree Matters

CoN < CoP

Rule of Degree

Convert high degrees of connascence
into
weaker forms of connascence

Another Example?



```
Customers.find(  
  ["last_name = ?", "Weirich"], "age")
```

```
def find(conditions, ordered_by)  
  ...  
end
```

Two orange double-headed arrows connect the two code blocks. One arrow points from the 'find' method call in the top block to the 'def find' definition in the bottom block. The other arrow points from the 'age' argument in the top block to the 'ordered_by' parameter in the bottom block, illustrating how the argument is mapped to the parameter.

```
Customers.find(  
  ["last_name = ?", "Weirich"], "age",  
  12, 24, ['first_name', 'last_name'])
```

```
def find(conditions, ordered_by,  
  limit, offset, selected)  
  ...  
end
```



The diagram consists of five orange arrows pointing from the arguments of the `Customers.find` call in the top box to the parameters of the `def find` method in the bottom box. The arrows are as follows: 1. From the first argument `["last_name = ?", "Weirich"]` to the parameter `conditions`. 2. From the second argument `"age"` to the parameter `ordered_by`. 3. From the third argument `12` to the parameter `limit`. 4. From the fourth argument `24` to the parameter `offset`. 5. From the fifth argument `['first_name', 'last_name']` to the parameter `selected`.

CoP → CoN

```
Customers.find(  
  :conditions => ["last_name = ?", "Weirich"],  
  :order_by => "age",  
  :limit => 12,  
  :offset => 24,  
  :select => ['first_name', 'last_name'])
```

```
def find(options={})  
  ...  
end
```


Another Example?

Connascence of Position

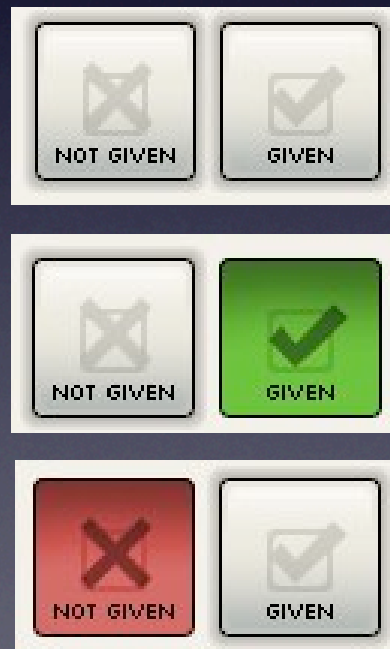
```
def test_user_can_do_something_interesting  
  user = User.find(:first)  
  ...  
end
```

Connascence of Position

```
def test_user_can_do_something_interesting
  user = User.find_by_name("Jim")
  ...
end
```


CoM

```
<input type="checkbox" value="2" />  
<input type="checkbox" value="1" />
```



```
<input type="checkbox" value="2" />  
<input type="checkbox" value="1" />
```

<input type="checkbox"/> NOT GIVEN	<input checked="" type="checkbox"/> GIVEN
<input type="checkbox"/> NOT GIVEN	<input checked="" type="checkbox"/> GIVEN
<input checked="" type="checkbox"/> NOT GIVEN	<input checked="" type="checkbox"/> GIVEN

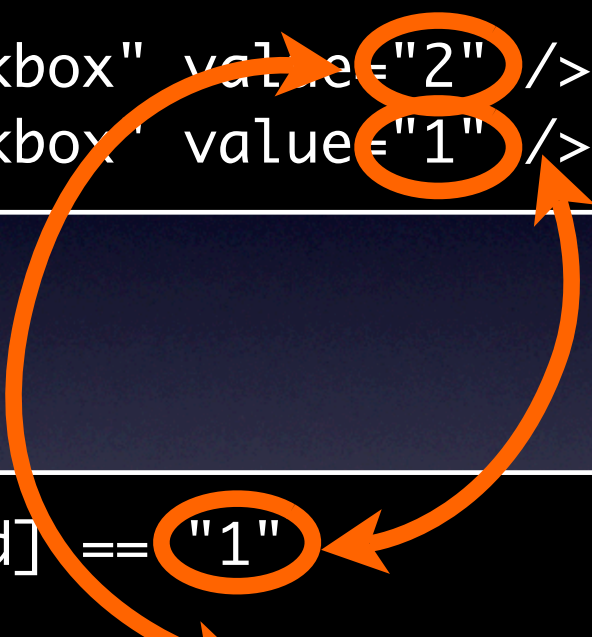

```
<input type="checkbox" value="2"/>  
<input type="checkbox" value="1"/>
```



```
if params[:med][id] == "1"  
  mark_given(id)  
elsif params[:med][id] == "2"  
  mark_not_given(id)  
end
```

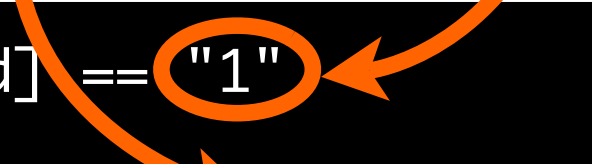
Connascence of Meaning

```
<input type="checkbox" value="2" />  
<input type="checkbox" value="1" />
```



A diagram consisting of two orange curved arrows. The first arrow starts from the value "2" in the first HTML line and points to the value "1" in the second HTML line. The second arrow starts from the value "1" in the second HTML line and points to the value "1" in the code block below.

```
if params[:med][id] == "1"  
  mark_given(id)  
elsif params[:med][id] == "2"  
  mark_not_given(id)  
end
```



A diagram consisting of two orange curved arrows. The first arrow starts from the value "1" in the code block and points to the value "2" in the first HTML line. The second arrow starts from the value "2" in the code block and points to the value "1" in the second HTML line.

Connascence of Meaning

MED_GIVEN = "1"

MED_NOT_GIVEN = "2"

CoM → CoN

```
MED_GIVEN = "1"  
MED_NOT_GIVEN = "2"
```

```
<input type="checkbox" value="<%= MED_GIVEN %>" />  
<input type="checkbox" value="<%= MED_NOT_GIVEN %>" />
```

```
if params[:med][id] == MED_GIVEN  
  mark_given(id)  
elsif params[:med][id] == MED_NOT_GIVEN  
  mark_not_given(id)  
end
```

CoM → CoN

```
MED_GIVEN = "1"  
MED_NOT_GIVEN = "2"
```

```
<input type="checkbox" value="<%= MED_GIVEN %>" />  
<input type="checkbox" value="<%= MED_NOT_GIVEN %>" />
```

```
if params[:med][id] == MED_GIVEN  
  mark_given(id)  
elsif params[:med][id] == MED_NOT_GIVEN  
  mark_not_given(id)  
end
```

CN

Revisit

```
MED_GIVEN = "1"
```

```
MED_NOT_GIVEN = "2"
```

MED_GIVEN = "1"

MED_NOT_GIVEN = "2"



MED_GIVEN = "1"

MED_NOT_GIVEN = "1"

Contranascence

MED_GIVEN = "1"

MED_NOT_GIVEN = "1"

Another Example?

Contranascence

My XML Library

```
class Node  
  ...  
end
```


Contranascence

My XML Library

```
class Node  
  ...  
end
```

Your Graphing Library

```
class Node  
  ...  
end
```

Contranascence

My XML Library

```
class Node  
  ...  
end
```

Your Graphing Library

```
class Node  
  ...  
end
```



Contranascence

My XML Library

```
module MyXml
  class Node
    ...
  end
end
```

Your Graphing Library

```
module YourGraphing
  class Node
    ...
  end
end
```


Contranascence

```
irb/slex.rb:92:      class Node
tkextlib/blt/tree.rb:15:  class Node < TkObject
tkextlib/blt/treeview.rb:18:  class Node < TkObject
tkextlib/blt/treeview.rb:966:  class Node < TkObject
tkextlib/bwidget/tree.rb:13:  class Node < TkObject
tkextlib/bwidget/tree.rb:262:  class Node
xmlrpc/parser.rb:17:      class Node
yaml/syck.rb:14:      class Node
```

Another Example?

Contranascence

My XML Library

```
module Kernel
  def to_node
    ...
  end
end
```

Your Graphing Library

```
module Kernel
  def to_node
    ...
  end
end
```


Contranascence

My XML Library

```
module Kernel
  def to_node
    ...
  end
end
```

Your Graphing Library

```
module Kernel
  def to_node
    ...
  end
end
```

Contranascence

Selector Namespaces

(Ruby 2 ?)

CoA

add_check_digit("31415972") → "314159728"

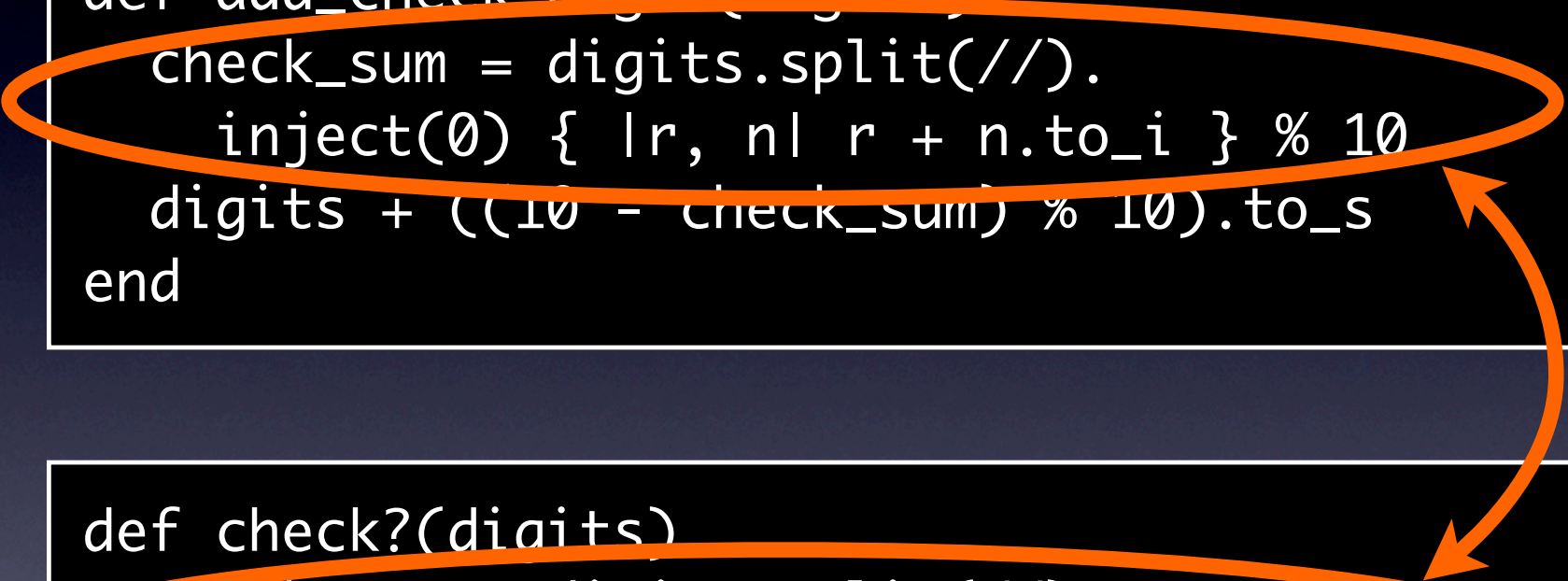
check?(" 314159728") → true

check?(" 314159723") → false

```
def add_check_digit(digits)
  check_sum = digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
  digits + ((10 - check_sum) % 10).to_s
end
```

```
def check?(digits)
  check_sum = digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
  check_sum == 0
end
```

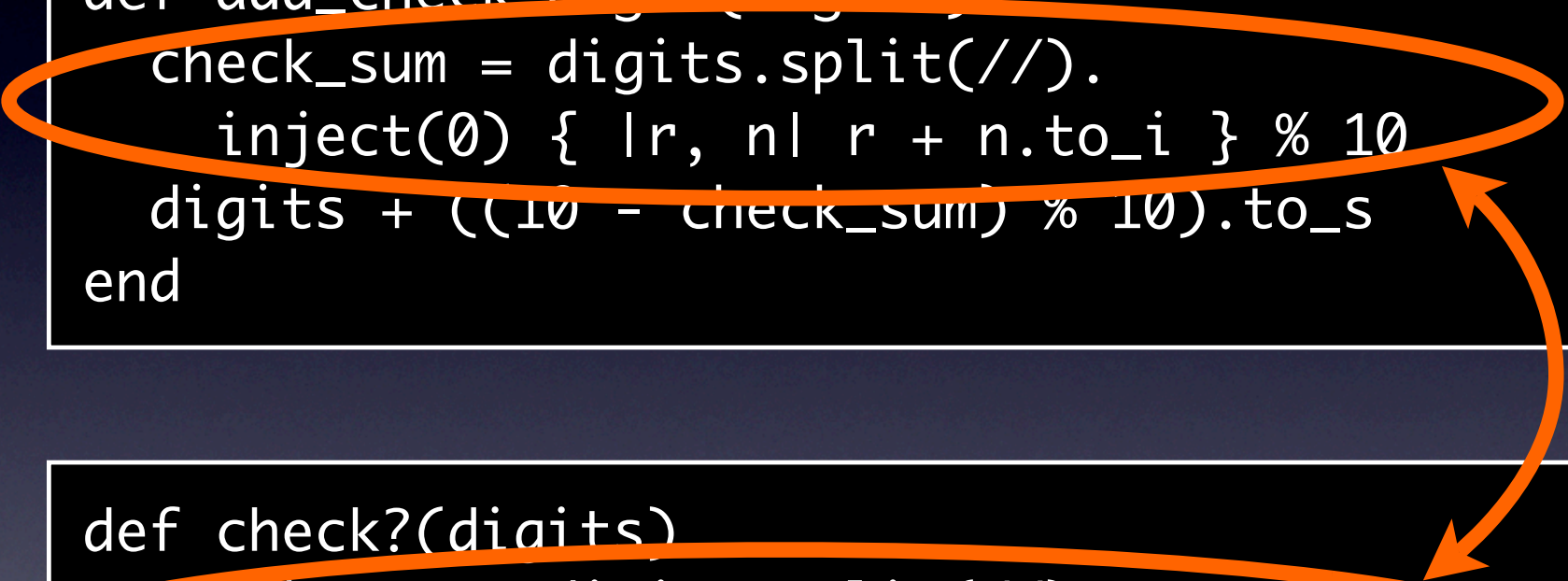
```
def add_check_digit(digits)
  check_sum = digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
  digits + ((10 - check_sum) % 10).to_s
end
```




```
def check?(digits)
  check_sum = digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
  check_sum == 0
end
```


Connascence of Algorithm

```
def add_check_digit(digits)
  check_sum = digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
  digits + ((10 - check_sum) % 10).to_s
end
```



```
def check?(digits)
  check_sum = digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
  check_sum == 0
end
```



CoA → CoN

```
def add_check_digit(digits)
  digits + ((10 - check_sum(digits)) % 10).to_s
end
```

```
def check?(digits)
  check_sum(digits) == 0
end
```

```
def check_sum(digits)
  digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
end
```

DRY

```
def add_check_digit(digits)
  digits + ((10 - check_sum(digits)) % 10).to_s
end
```

```
def check?(digits)
  check_sum(digits) == 0
end
```

```
def check_sum(digits)
  digits.split(//).
    inject(0) { |r, n| r + n.to_i } % 10
end
```


CoT

```
class Account
  def credit(amount)
    @amount += amount
  end
end
```

```
threads = (0...Threads).map {  
  Thread.new do  
    A.credit(1)  
  end  
}
```


The Setup

@amount += 1

@amount += 1

23

Step 1

@amount += 1

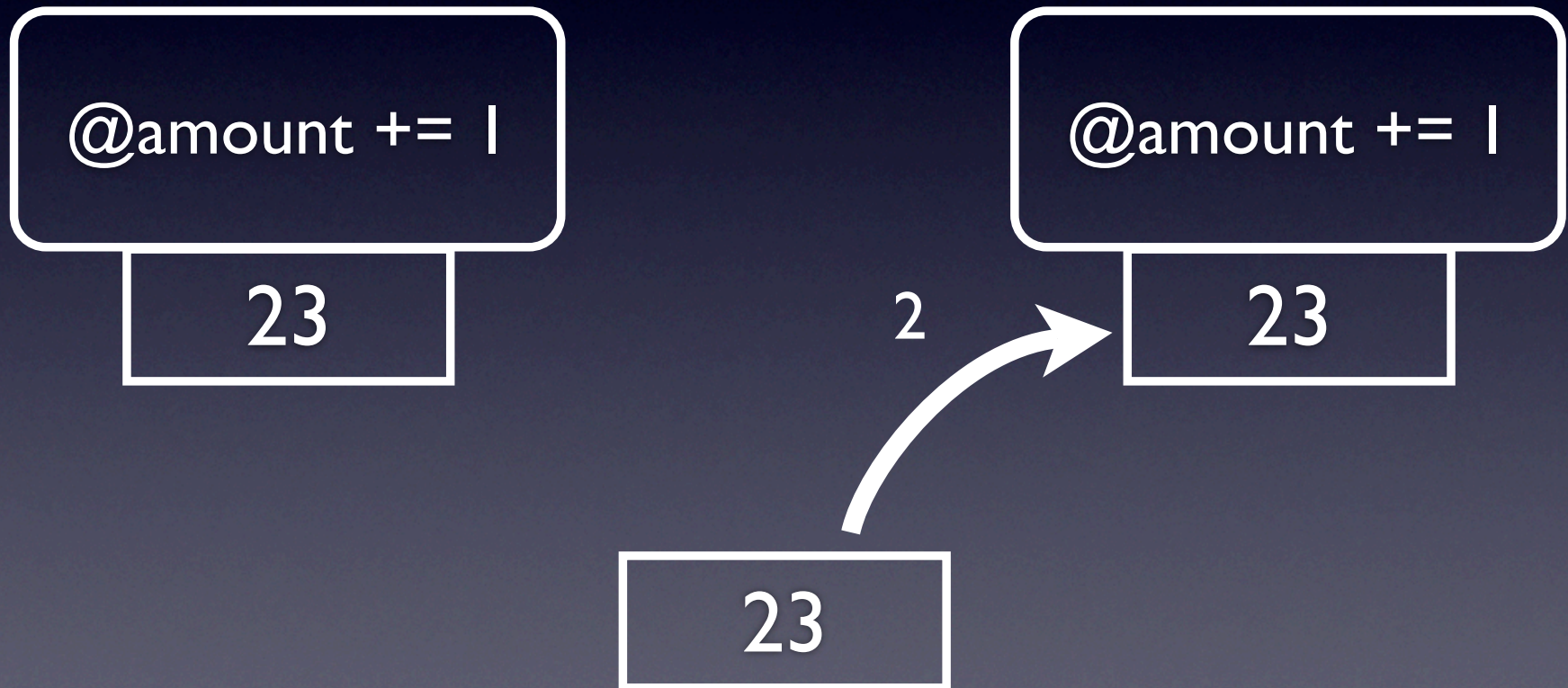
23

@amount += 1

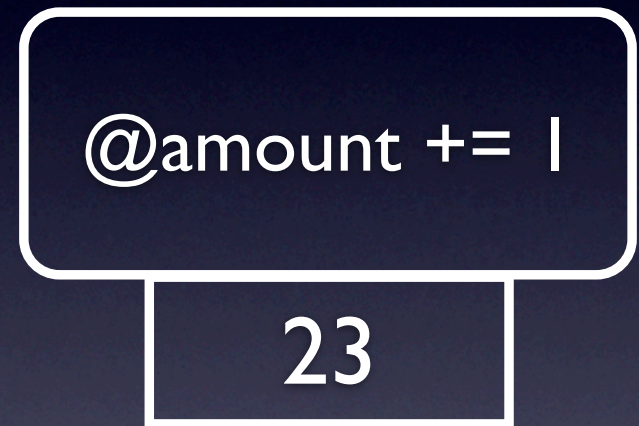
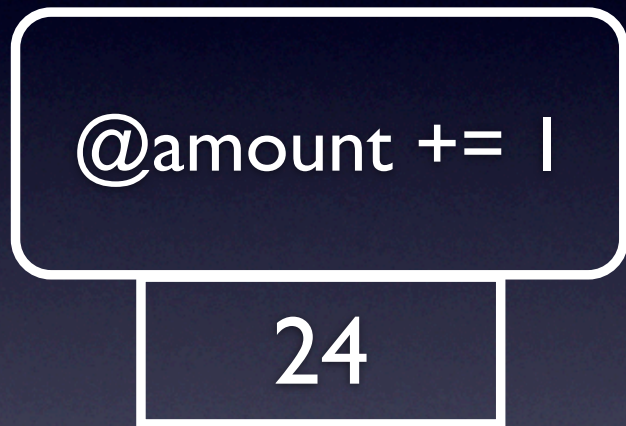
23



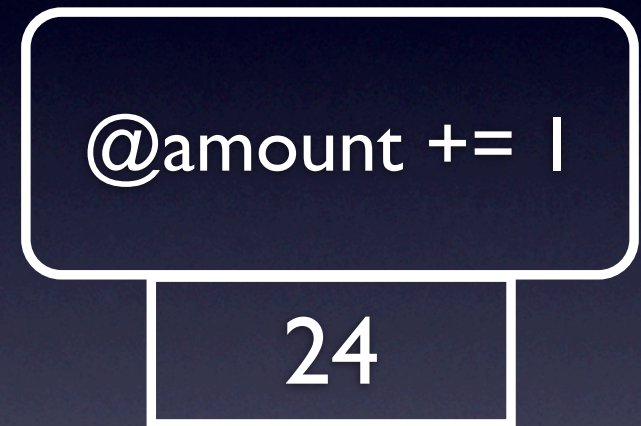
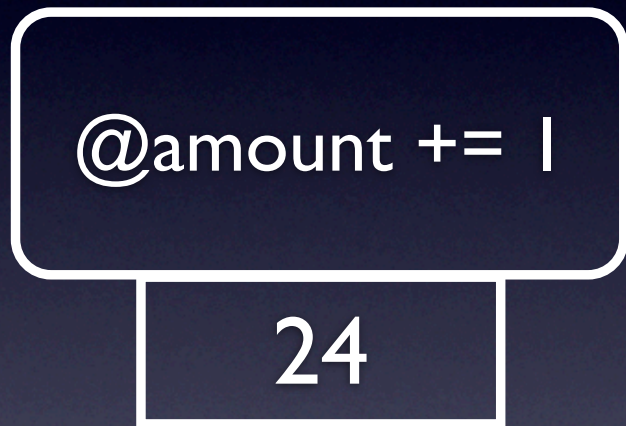
Step 2



Step 3



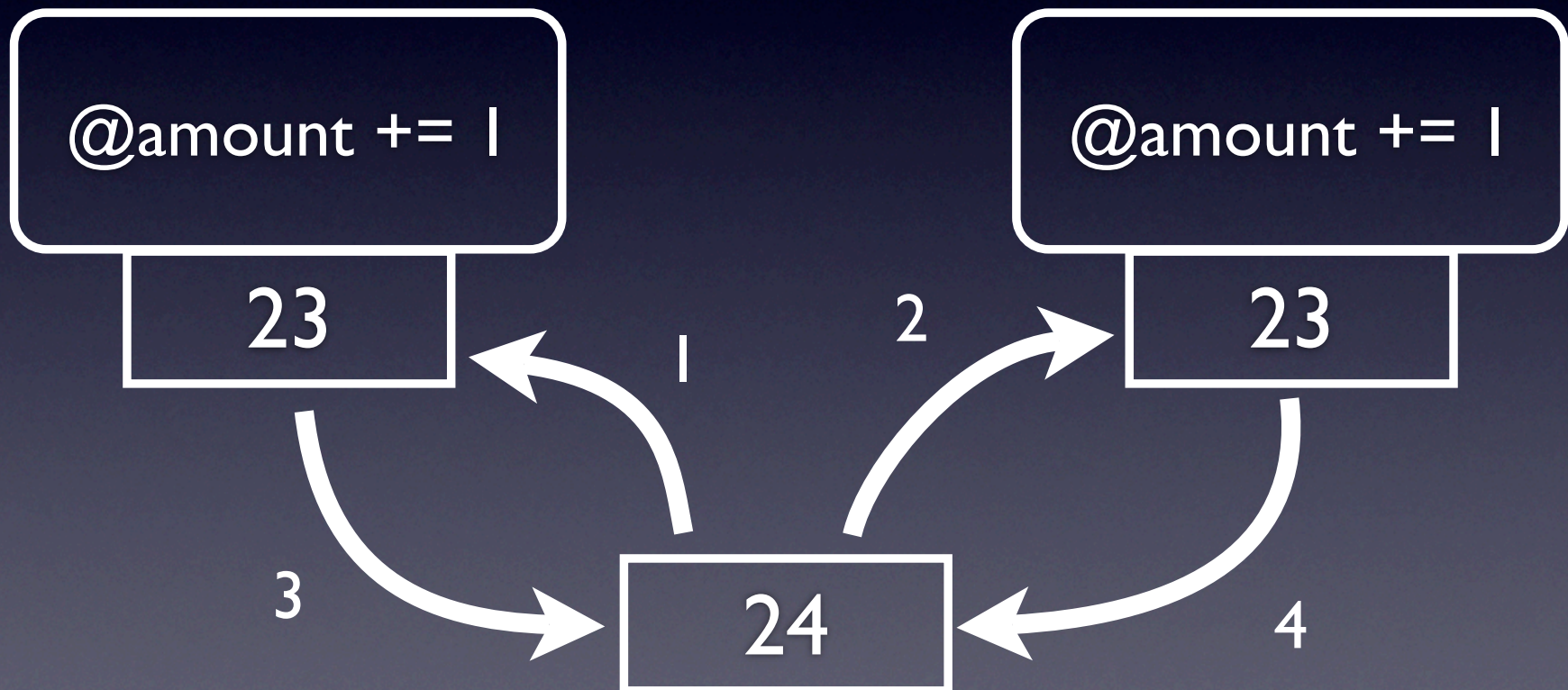
Step 4



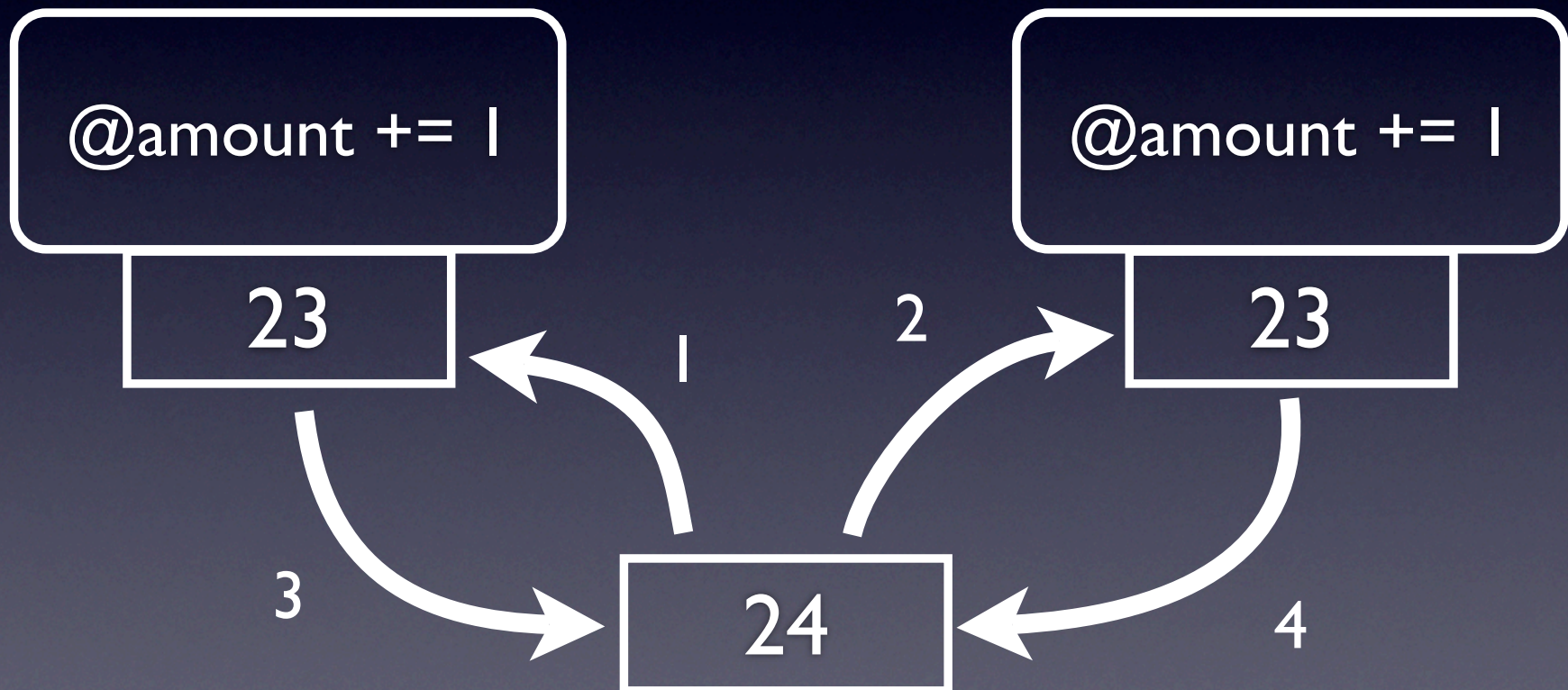
Should be 25
at this point! →



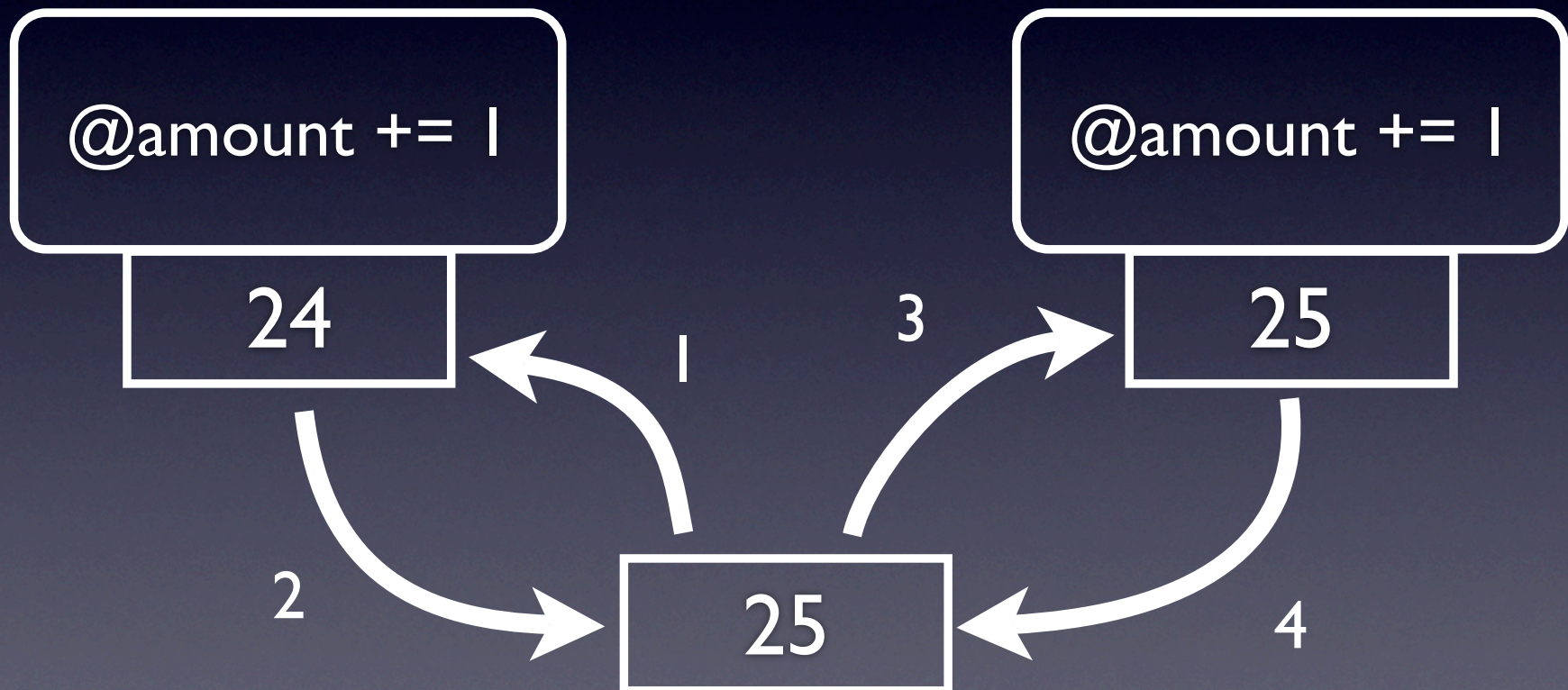
Race Condition



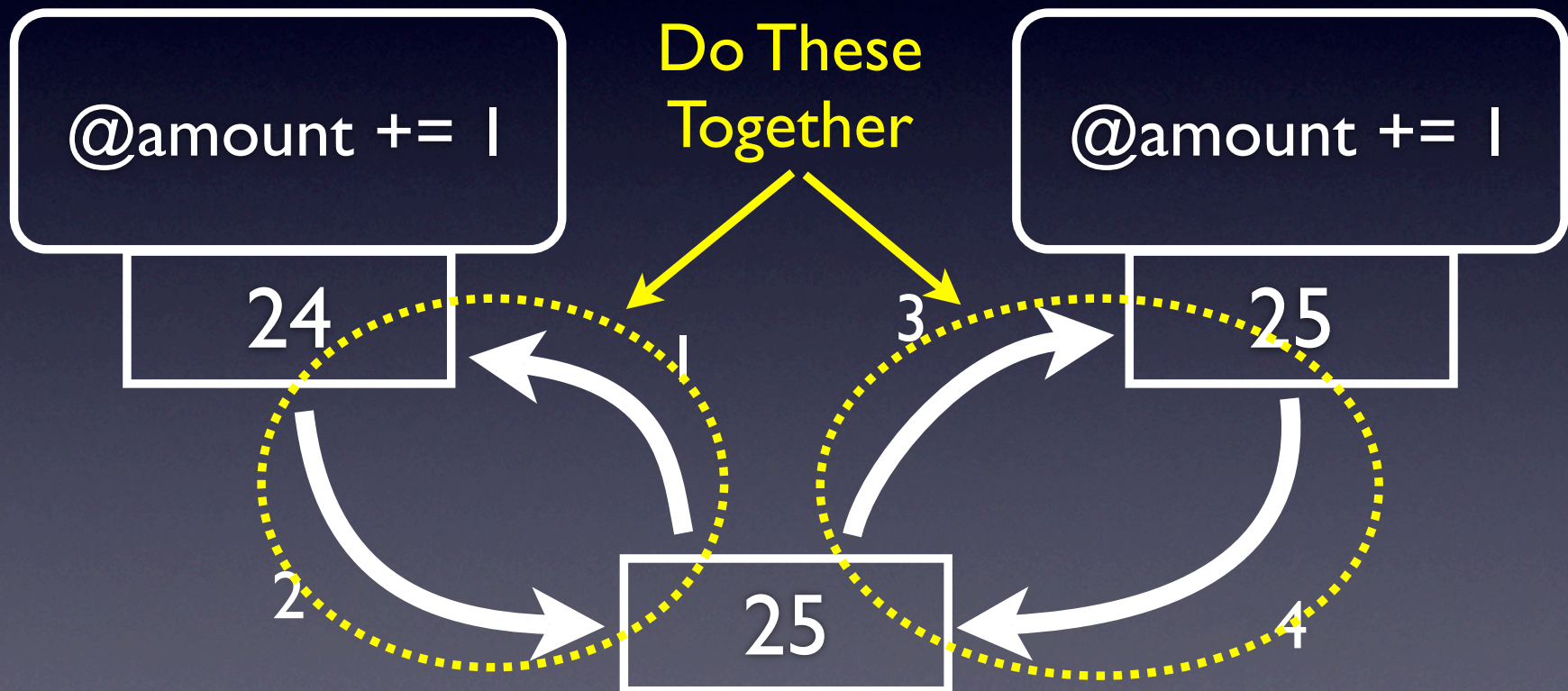
Connascence of Timing



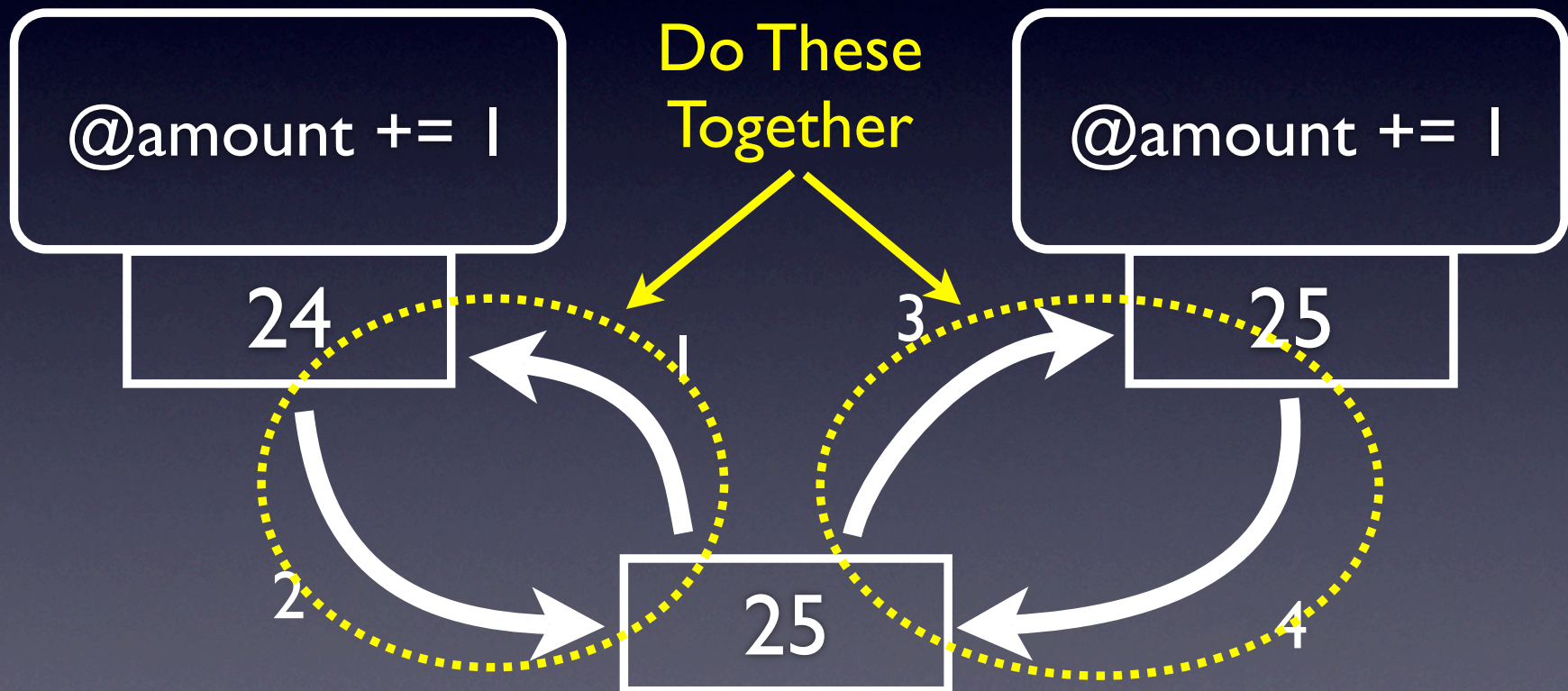
Reordering Steps



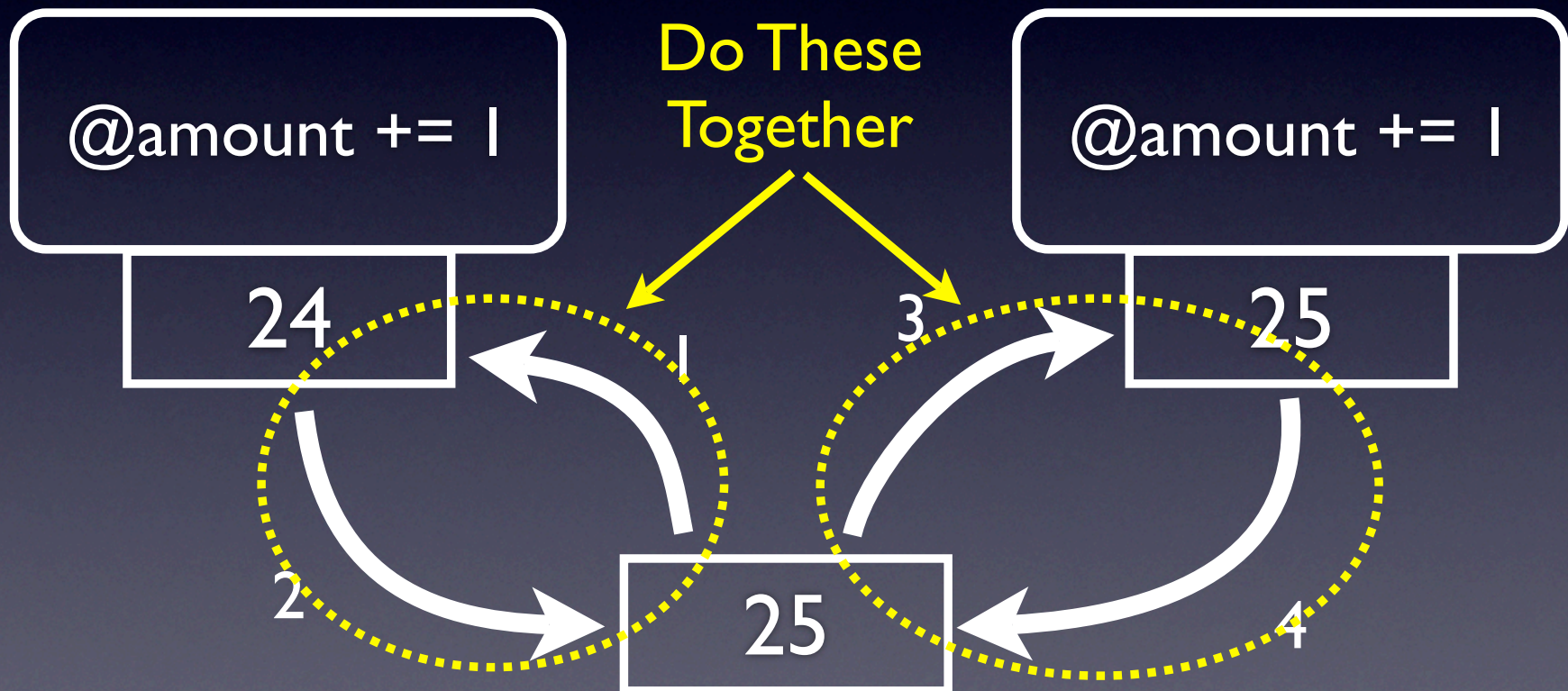
Mutual Exclusion



Connascence of Timing



Connascence of Timing



```
m = Mutex.new
threads = (0...Threads).map {
  Thread.new do
    m.synchronize do
      A.credit(1)
    end
  end
}
```


Summary



Connascence

- Static
 - Connascence of Name
 - Connascence of Type
 - Connascence of Meaning
 - Connascence of Algorithm
 - Connascence of Position
- Dynamic
 - Connascence of Execution
 - Connascence of Timing
 - Connascence of Value
 - Connascence of Identity
 - Contranascence

Rules

- Rule of Locality
- Rule of Degree

References

- *What Every Programmer Should Know About Object Oriented Design*, Meilir Page-Jones
- *Fundamentals of Object-Oriented Design in UML*, Meilir Page-Jones
- *Composite/Structured Design*, Glenford Myers
- *Reliable Software Through Composite Design*, Glenford Myers
- *Agile Software Development, Principles, Patterns, and Practices*, Robert Martin
- *Object-Oriented Software Construction*, Bertrand Meyer
- *The Pragmatic Programmer: From Journeyman to Master*, Andy Hunt & Dave Thomas

Questions?

Thank You!



[git://github.com/jimweirich/presentation_connascence.git](https://github.com/jimweirich/presentation_connascence.git)

Copyright 2009 by Jim Weirich, Some Rights Reserved



Attribution-NonCommercial-ShareAlike 2.0