Marco Consolaro

ALCOR academy

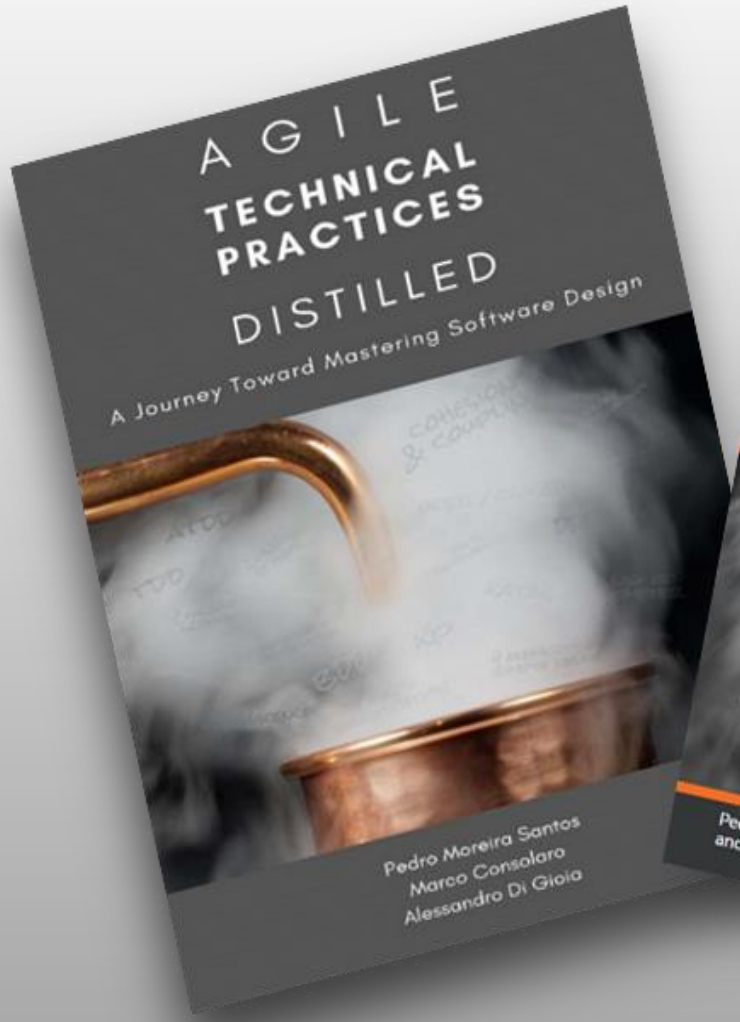
Connascence

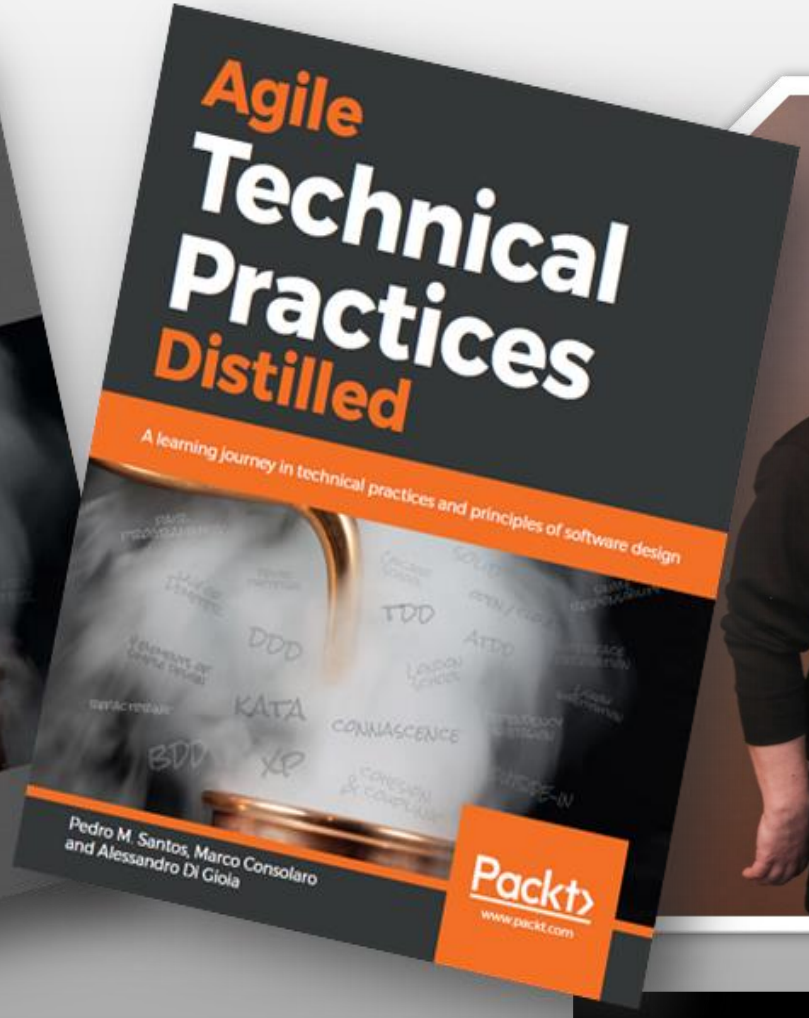
beyond

Coupling &Cohesion

"Great things are done by a series of small things brought together."

Vincent Van Gogh





Bookauthority
BEST NEW AGILE SOFTWARE DEVELOPMENT BOOKS
WINNER

Bookauthority
BEST NEW CONTINUOUS INTEGRATION BOOKS
WINNER

Bookauthority
BEST NEW TDD BOOKS
WINNER

Peti Koch ★★★★★

At the moment (2021) this book is probably unique in the world. It's a book of books! It covers all modern agile technical practices like pair programming, TDD, refactoring, code smells, ... Each practice is briefly explained followed by specific katas and how the practice fits into the big picture. Highly recommended!


ALCOR
academy
Helping CTOs to promote a
culture of excellence
with our
INNOVATIVE
SOCIO-TECHNICAL TRAINING

ALCOR academy

TRAINING PROGRAMME

FLYING

Lesson 1

Connascence

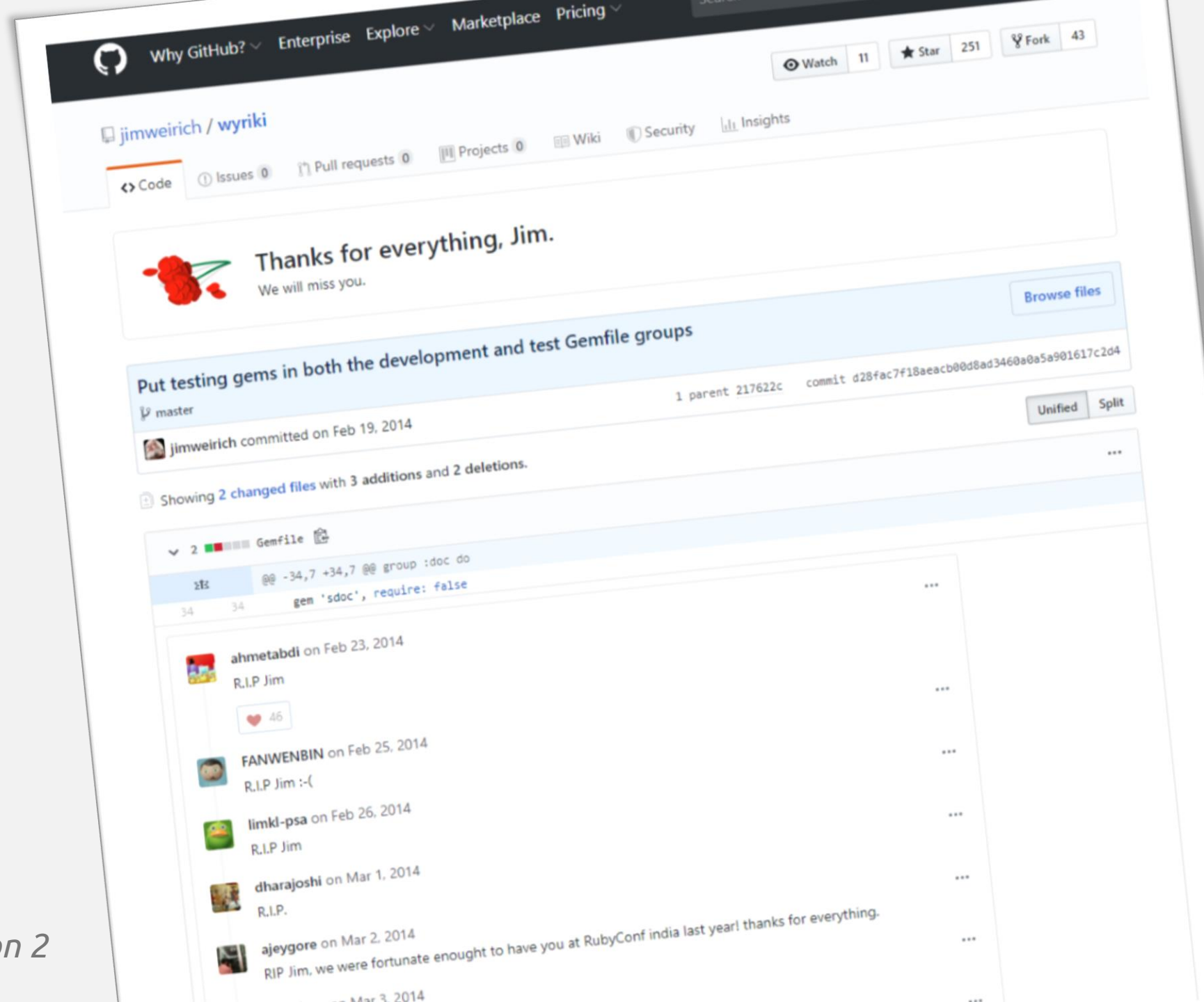
*“Experience by itself teaches nothing. Without theory, experience has no meaning. Without theory, one has no questions to ask. Hence, without theory, there is no learning.”— **Edwards Deming***

“CONNASCENCE EXAMINED”

Jim Weirich - 2012

https://www.youtube.com/watch?v=22vYwcfQnk8

James Nolan (Jim) Weirich
(1956 – 2014)



CONNASCENCE

from Latin: **co-** + **nascence**

“together”

nascentem ⇒ “arising young, immature”,
present participle of ***nasci*** ⇒ **“to be born”**

“The birth and growth of two or more things at the same time”

two or more elements

(fields, methods, classes, parameters, variables, etc.)

are **connascent** if

a change in one element would require also a change in the others

Taxonomy in 3 dimensions

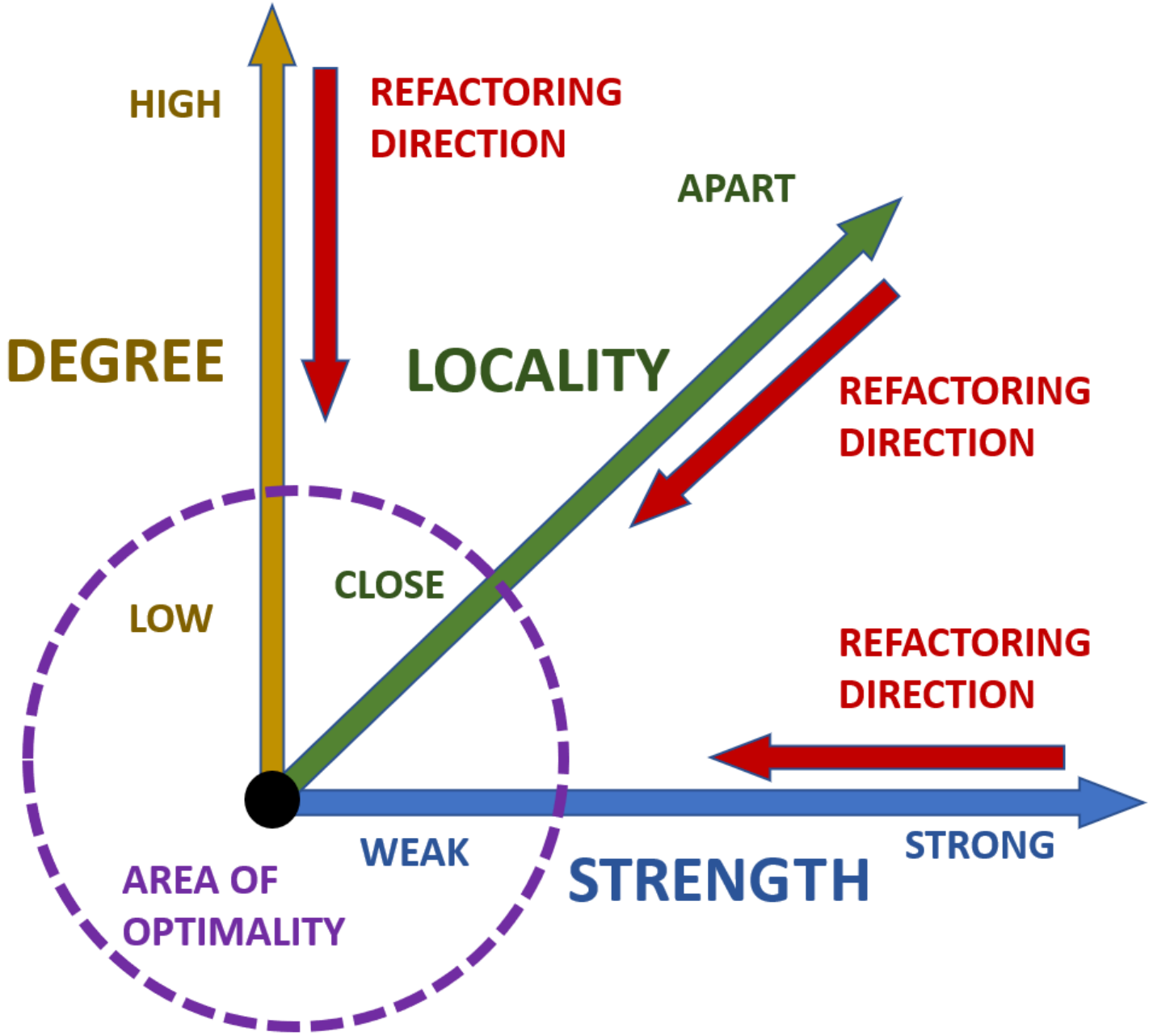
→ **Degree**: the size of its impact, estimable by the number of entities it affects.

→ **Locality**: how close are the affected entities in respect of each other in terms of abstraction.

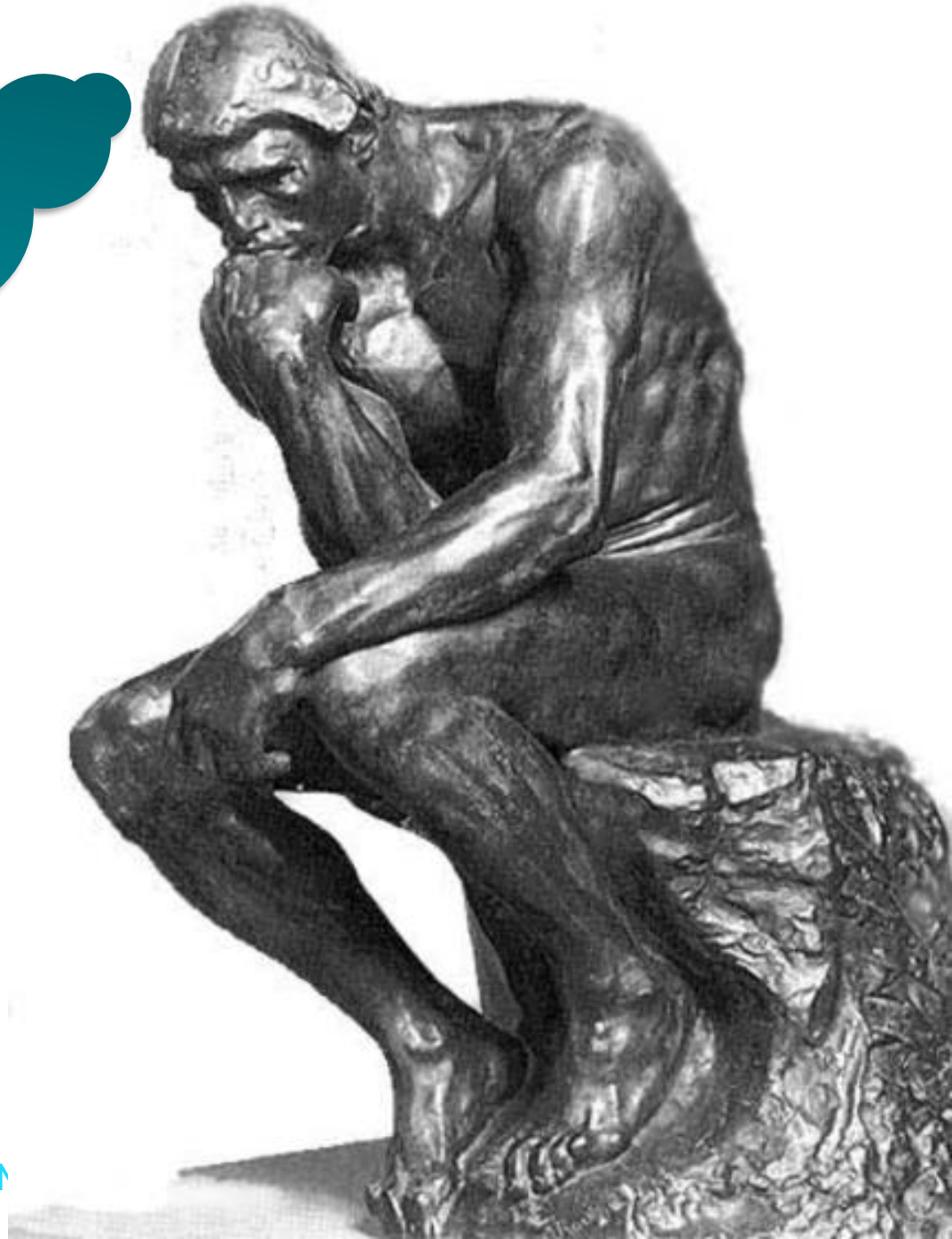
→ **Strength**: how likely it requires compensating changes in connascent elements. The stronger the form of connascence, the more difficult and costly it is to refactor the elements in the relationship.

Taxonomy
in 3 dimensions

- DEGREE
- LOCALITY
- STRENGTH



SHOW ME SOME CODE, PLEASE!

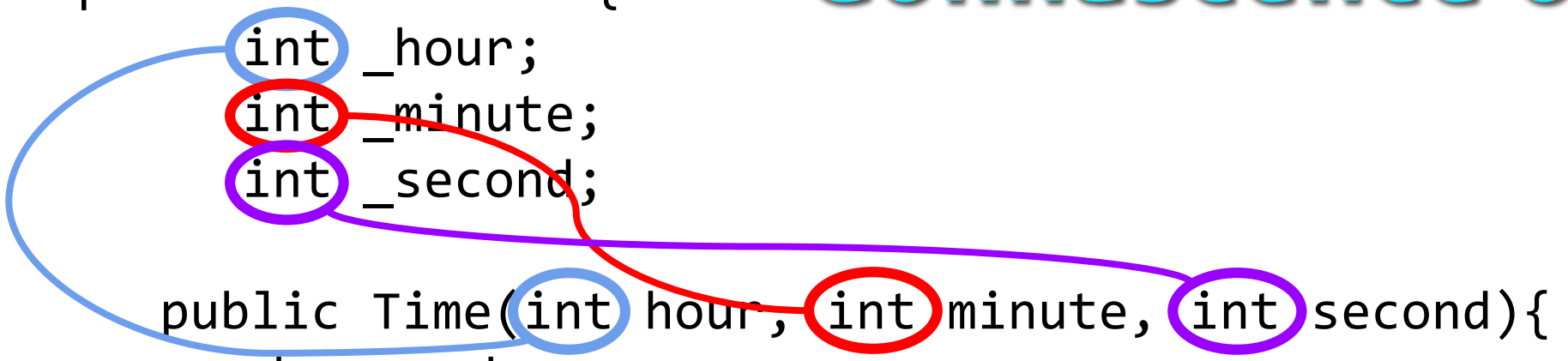


Connascence of Name

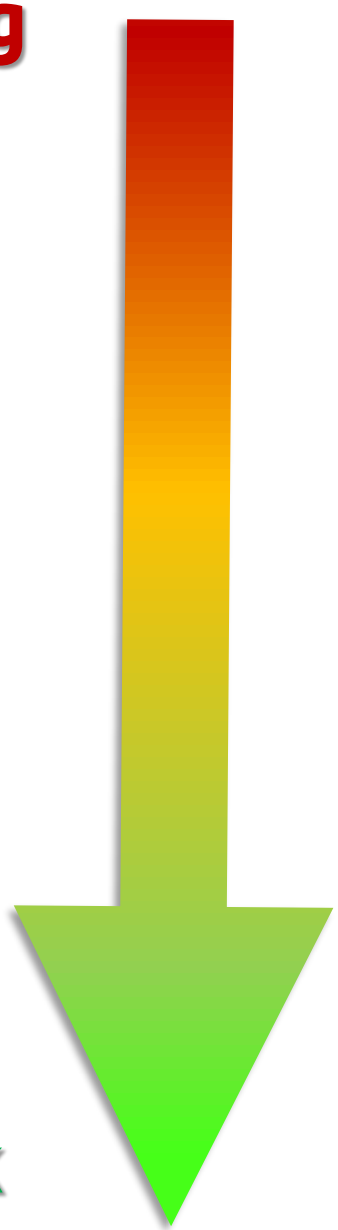
```
public class Time{
    int _hour;
    int _minute;
    int _second;

  public Time(int hour, int minute, int second){
    _hour = hour;
    _minute = minute;
    _second = second;
  }

  public string Display(){
    return _hour + ":" + _minute + ":" + _second + ":";
  }
}
```

Connascence of Type

```
public class Time{
    int _hour;
    int _minute;
    int _second;

    public Time(int hour, int minute, int second){
        _hour = hour;
        _minute = minute;
        _second = second;
    }

    public string Display(){
        return _hour + ":" + _minute + ":" + _second + ":";
    }
}
```



Strong



Weak

2 ***Type***

1 ***Name***

Static
(discoverable at compile time)

Connascence of Position

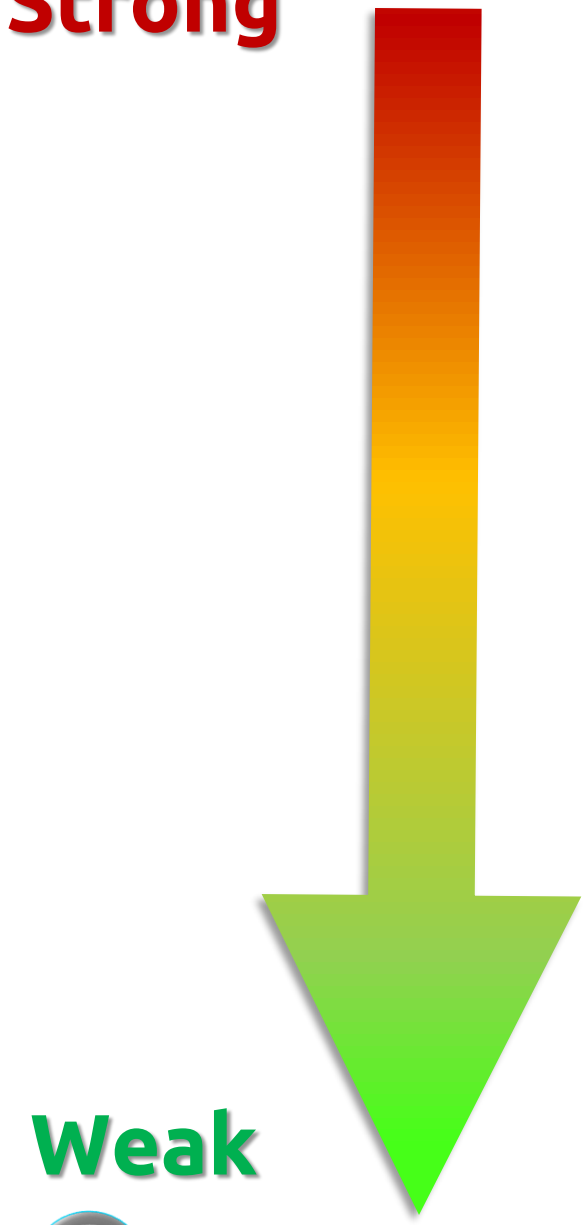
```
public class Notificator
{
  public void SendEmail(string recipient, string sender, string message)
  {
    Email email = new Email();
    email.To(recipient);
    email.From(sender);
    email.Body(message);
    _smtpService.SendEmail(email);
  }
}
```

1st 2nd 3rd

```
_notificator.SendEmail("address1@mail.com", "address2@mail.com", "message");
```

Connascence of Position

when multiple components must be adjacent or must appear in a particular order – often when passed within a positional structure like an array or a tuple.



5 ***Position***

2 ***Type***

1 ***Name***

Static
(discoverable at compile time)

```
public class Notificator
{
  public void SendEmail(Recipient recipient, Sender sender, Message message)
  {
    Email email = new Email();
    email.To(recipient.Address);
    email.From(sender.Address);
    email.Body(message.Text);
    _smtpService.SendEmail(email);
  }
}
```

```
_notificator.SendEmail(new Recipient("address1@mail.com"),
                       new Sender("address2@mail.com"),
                       new Message("message"));
```

```
public class Notificator
{
  public void SendEmail(Notification notification)
  {
    Email email = new Email();
    email.To(notification.Recipient);
    email.From(notification.Sender);
    email.Body(notification.Message);
    _smtpService.SendEmail(email);
  }
}


_notificator.SendEmail(new Notification()
                           .To("address1@mail.com")
                           .From("address2@mail.com")
                           .WithMessage("message"));
```

```
public class OrderProcessor {

  public void ProcessOrder(Tuple<Order, bool> orderInfo)
  {
    ProcessOrder(orderInfo.Item1);

    if (orderInfo.Item2)
      SendInvoice(orderInfo.Item1);
  }


  public void ProcessOrder(Tuple<Order, bool, int, datetime, string, bool> orderInfo)
  {
    ProcessOrder(orderInfo.Item1);

    if (orderInfo.Item2)
      SendInvoice(orderInfo.Item1);

    NotifyEmployee(orderInfo.Item1,orderInfo.Item3,orderInfo.Item4,orderInfo.Item5);
  }
```

Degree 2

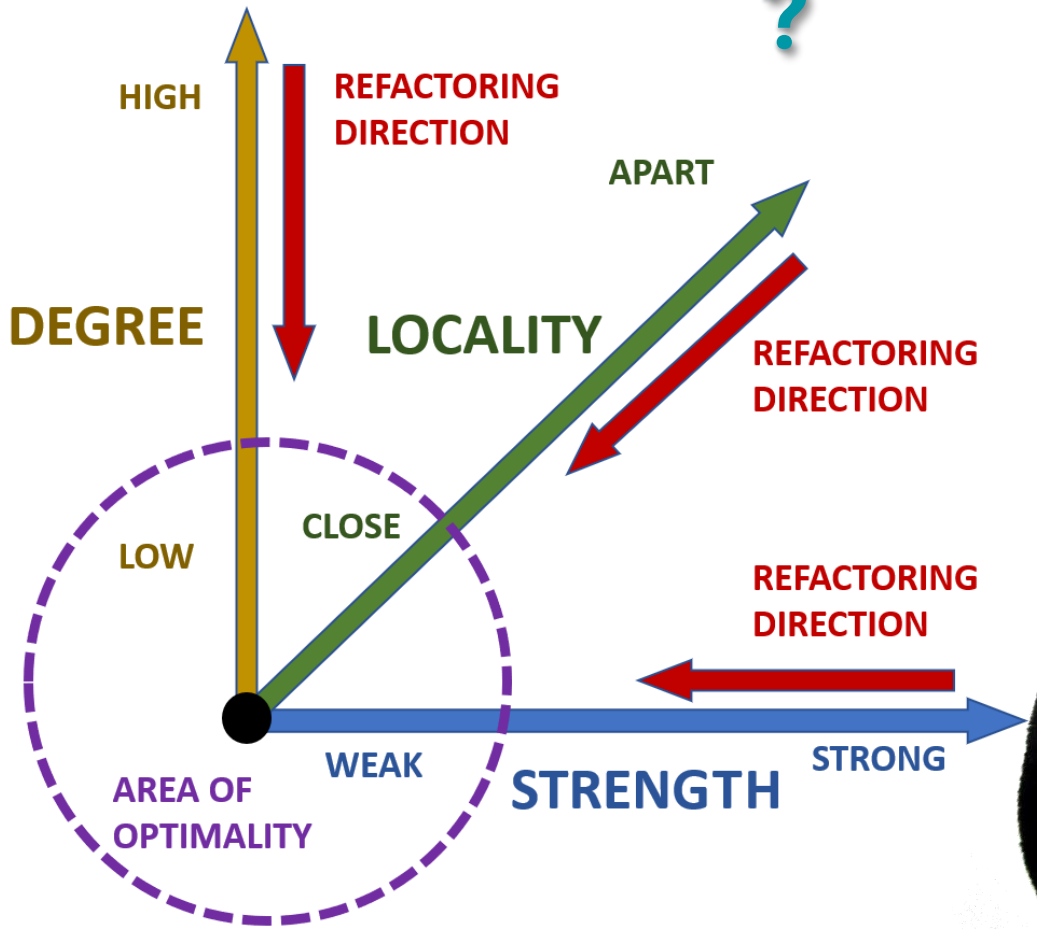
Degree 5

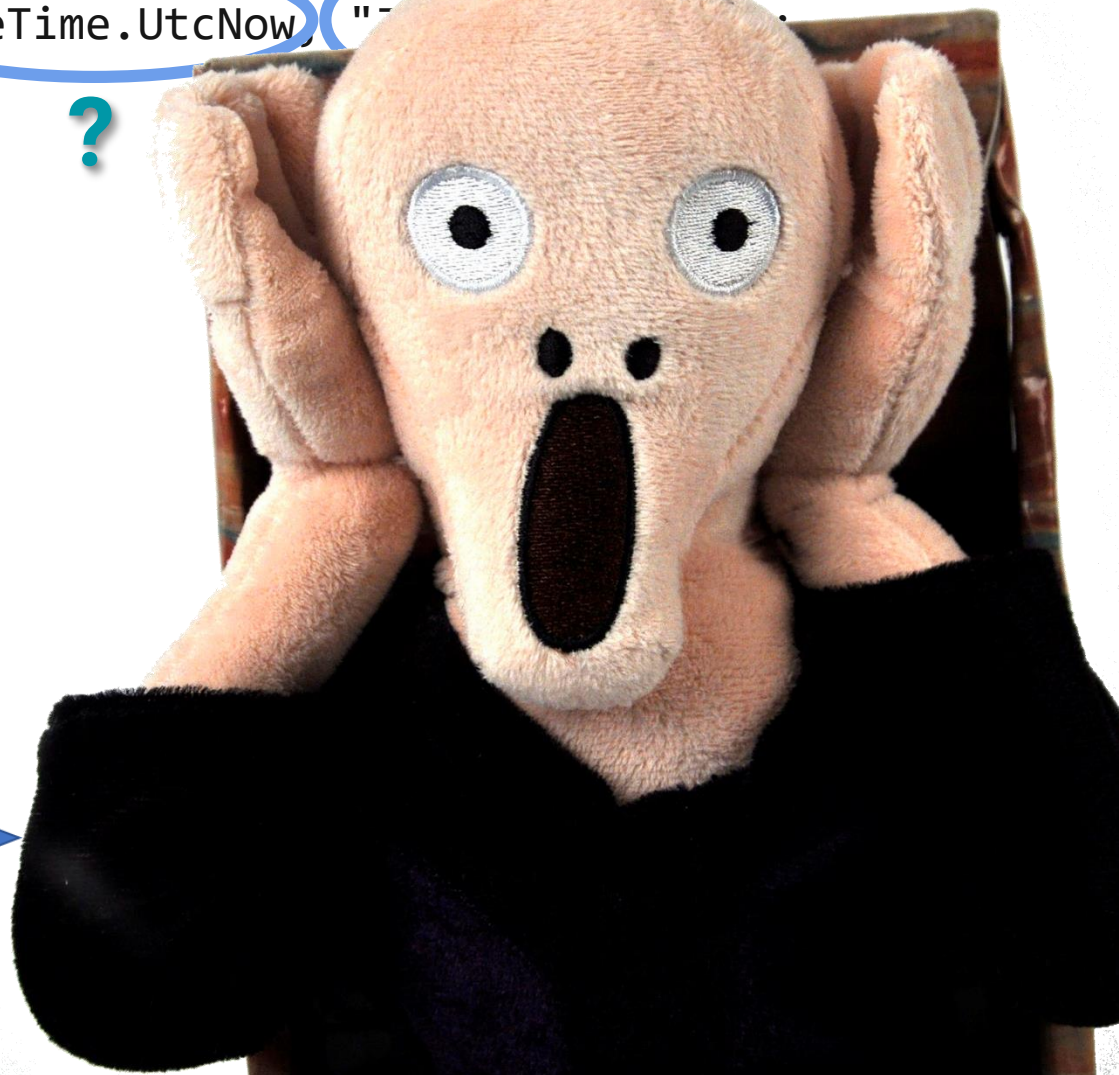
Degree 2

```
_orderProcessor
   .ProcessOrder(new Tuple(order, true));

_orderProcessor
   .ProcessOrder(new Tuple(order, true, 12345, DateTime.UtcNow, "
```

? ?

Degree 5

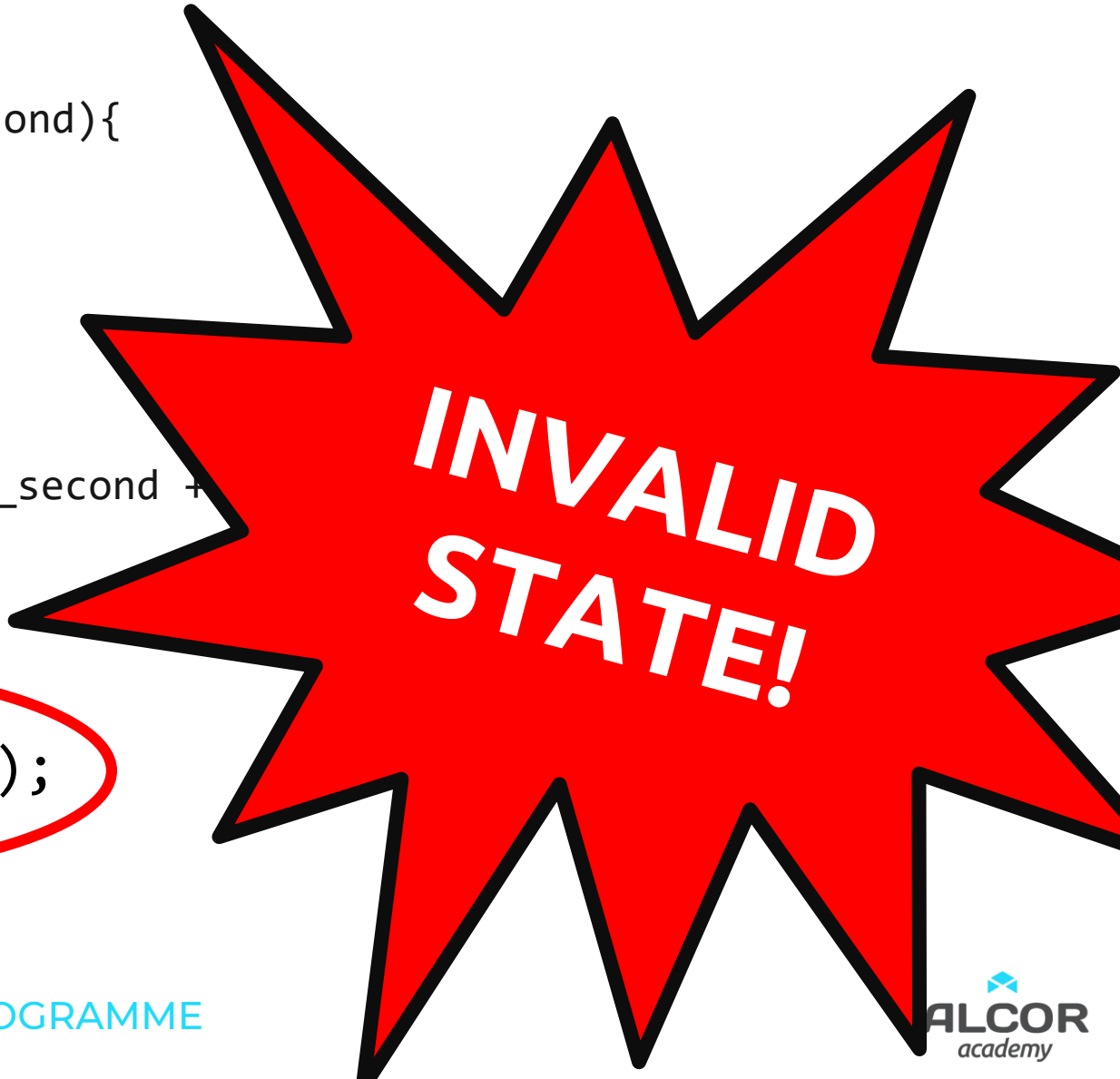




Connascence of Value

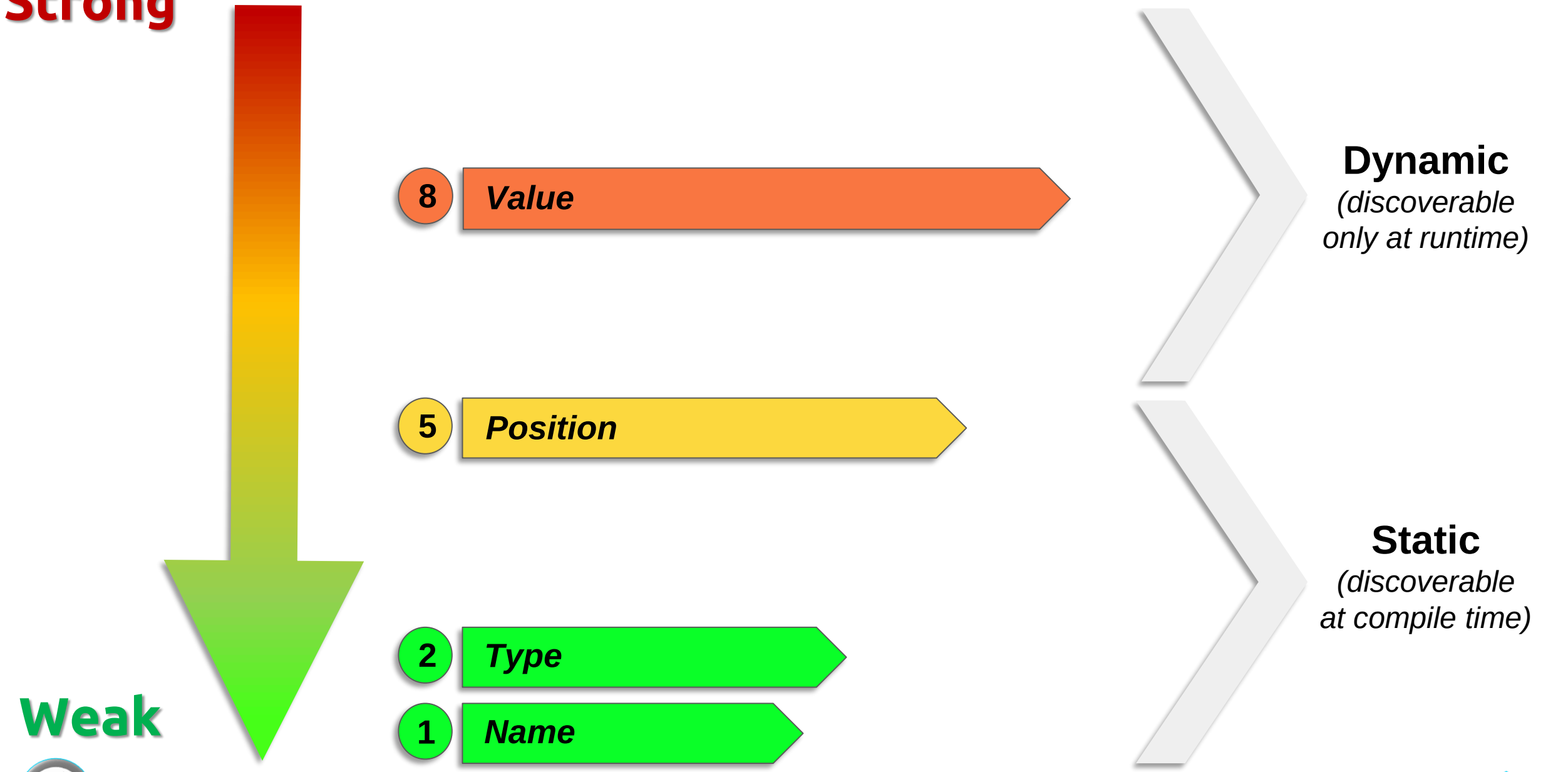
```
public class Time{
        int _hour;
        int _minute;
        int _second;

    public Time(int hour, int minute, int second){
        _hour = hour;
        _minute = minute;
        _second = second;
    }

    public string Display(){
        return _hour + “:” + _minute + “:” + _second +
    }
}
```

```
var myTime = new Time (27, 76, 82);
```



Connascence of Value

when two or more components' values are related or have an intrinsic range of validity in their input not expressed by their primitive types.


Strong
8
Value
5
Position
2
Type
1
Name
Weak
Dynamic
(discoverable
only at runtime)
Static
(discoverable
at compile time)

```
public class Time{
        int _hour;
        int _minute;
        int _second;

    public Time(int hour, int minute, int second){
        _hour = hour;
        _minute = minute;
        _second = second;
        Validate();
    }
    ...
    private void Validate(){
      if(_hour < 0 || _hour > 23)
        throw new InvalidHourException();
      if(_minute < 0 || _minute > 59)
        throw new InvalidMinuteException();
      if(_second < 0 || _second > 59)
        throw new InvalidSecondException();
    }
```

The class must participate in the information flow if – and only if – its state is valid.

```
public enum Hour { Midnight = 0, OneAm = 1, TwoAm = 2, ... , ElevenPm = 23}
public enum Minute { ... }
public enum Second { ... }

public class Time{
      Hour _hour;
      Minute _minute;
      Second _second;

    public Time(Hour hour, Minute minute, Second second){
        _hour = hour;
        _minute = minute;
        _second = second;
    }
}
```

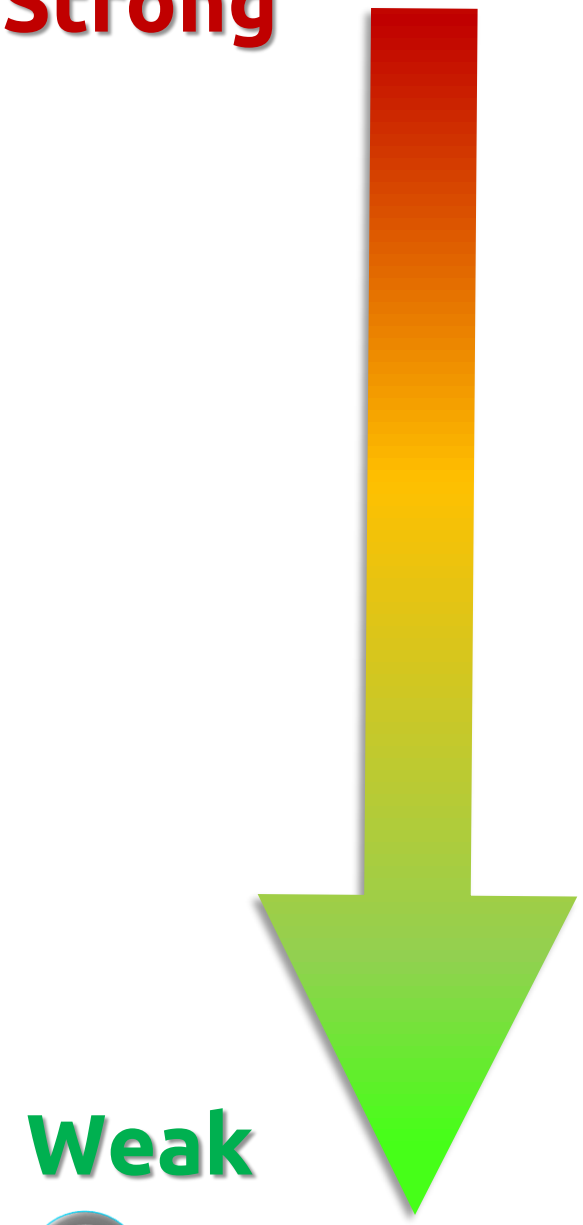
Make invalid states UNREPRESENTABLE!

```
<input type='checkbox' name='transport' value='1' />
I travel by bike <br />
<input type='checkbox' name='transport' value='2' />
I travel by car <br />
<input type='checkbox' name='transport' value='3' />
I travel by train <br />
<input type='checkbox' name='transport' value='4' />
I travel by bus <br />


private SetTransport(string transport){
  switch(transport){
    case "1": AddBike();
    case "2": AddCar();
    case "3": AddTrain();
    case "4": AddBus();
    break;
  }
}
```

Connascence of Meaning (or Convention)

Connascence of Meaning (or Convention)

when two or more components must agree on the meaning of specific values, hence using a convention.



8 ***Value***

5 ***Position***

3 ***Meaning (Convention)***

2 ***Type***

1 ***Name***

Dynamic
(discoverable only at runtime)

Static
(discoverable at compile time)

```
<input type='checkbox' name='transport' value='<%= BIKE %>' />
I travel by bike <br />
<input type='checkbox' name='transport' value='<%= CAR %>' />
I travel by car <br />
<input type='checkbox' name='transport' value='<%= TRAIN %>'
I travel by train <br />
<input type='checkbox' name='transport' value='<%= BUS %>' />
I travel by bus <br />
```


```
private SetTransport(string transport){
  switch(transport){
    case BIKE: AddBike();
    case CAR: AddCar();
    case TRAIN AddTrain();
    case BUS: AddBus();
    break;
  }
}
```

```
const string BIKE = "1";
const string CAR = "2";
const string TRAIN = "3";
const string BUS = "4";
```

Connascence of Algorithm

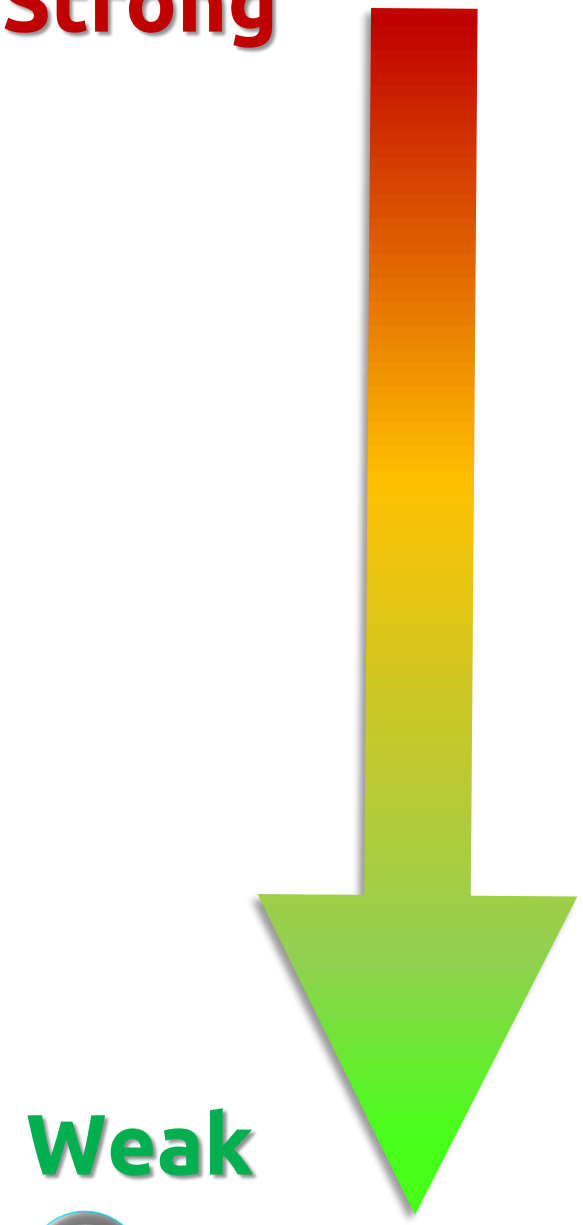
```
AddChecksum("32143") => "321437"
Check("321437") => true
Check("321432") => false
```

```
public string AddChecksum(string inputData){
  var sum = SumCharsOf(inputData);
  var difference = sum % 10;
  return inputData + difference;
}

public bool Check(string inputData){
  var sum = SumCharsOf(inputData);
  return sum % 10 == 0;
}
```

Same algorithm!!!

Connascence of Algorithm

when two or more components must agree on using a particular algorithm.



8 ***Value***

5 ***Position***

4 ***Algorithm***

3 ***Meaning (Convention)***

2 ***Type***

1 ***Name***

Dynamic
(discoverable only at runtime)

Static
(discoverable at compile time)

```
public string AddChecksum(string inputData){
  return inputData + CheckSum(inputData);
}

public bool Check(string inputData){
  return Checksum(inputData) == 0;
}

private int Checksum(string inputData){
  var sum = SumCharsOf(inputData);
  return sum % 10;
}
```

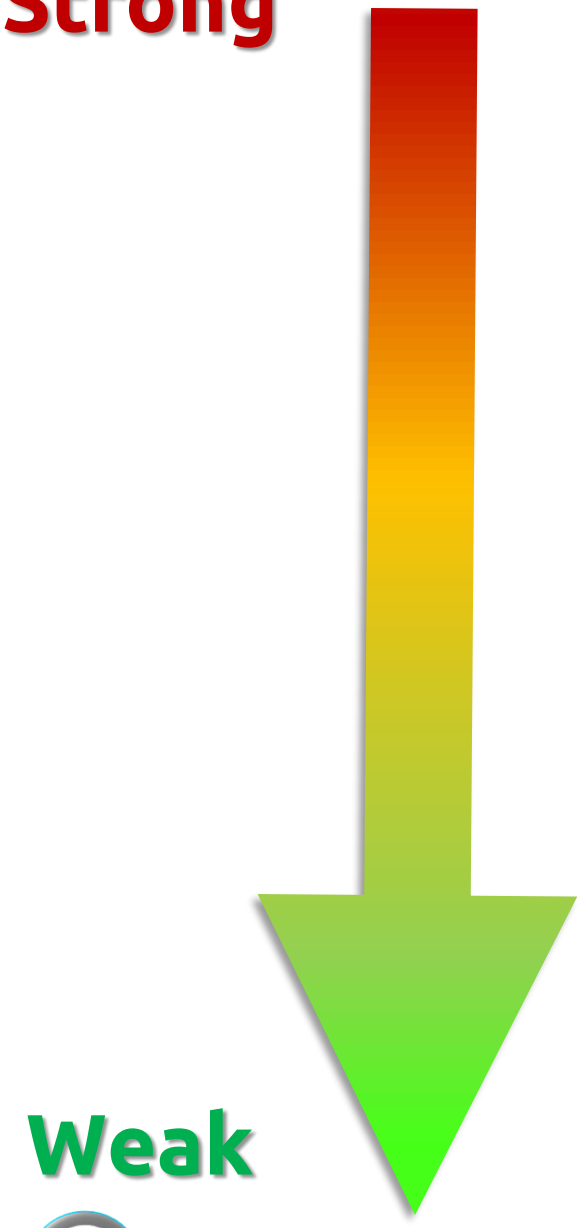
encapsulated in a method

Connascence of Execution Order

```
public void SendReceipts(){
  var receiptSender = new ReceiptSender();
  var receiptId = NextUnsentReceiptId();

  while(receiptId != null){
    receiptSender.SendToCustomer(receiptId);
    receiptSender.Archive(receiptId);

    receiptId = NextUnsentReceiptID();
  }
}
```

Connascence of Execution Order

when the caller of a component must have some unexpressed knowledge about the correct order of the methods to be called.



8 ***Value***

6 ***Execution order***

5 ***Position***

4 ***Algorithm***

3 ***Meaning (Convention)***

2 ***Type***

1 ***Name***

Dynamic
(discoverable only at runtime)

Static
(discoverable at compile time)

Builder pattern

1) setup the builder

2) build the car

```
public interface IBuildCars {

    IBuildCars WithBrand(Brand brand);

    IBuildCars WithEngine(Engine brand)

    IBuildCars WithColor(Color brand);

    Car Build();

}
```

Connascence of Execution Order expressed by convention

What if we want to make it explicit? Can we do better?

Let’s segregate the interface…

```
public interface IBuildCarsWithBrand{
 IBuildCarsWithEngine WithBrand(Brand brand);
}

public interface IBuildCarsWithEngine{
 IBuildCarsWithColor WithEngine(Engine engine);
}

public interface IBuildCarsWithColor{
 IBuildCars WithColor(Color color);
}

public interface IBuildCars{
 Car Build();
}
```

…the builder will become

```
public class CarBuilder : IBuildCarsWithBrand,
                          IBuildCarsWithEngine,
                          IBuildCarsWithColor,
                          IBuildCars
{

  private CarBuilder(){}

  public static IBuildCarsWithBrand New(){
       return new CarBuilder();
  }

  ...
}
```

Factory Method

…code now express the correct execution order without conventions!

```
Car myNewCar = CarBuilder.New()
    .WithBrand("Alfa")
    .WithEngine("2.0")
    .WithColor("red")
    .Build();
```

IBuildCarsWithBrand

IBuildCarsWithEngine

IBuildCarsWithColor

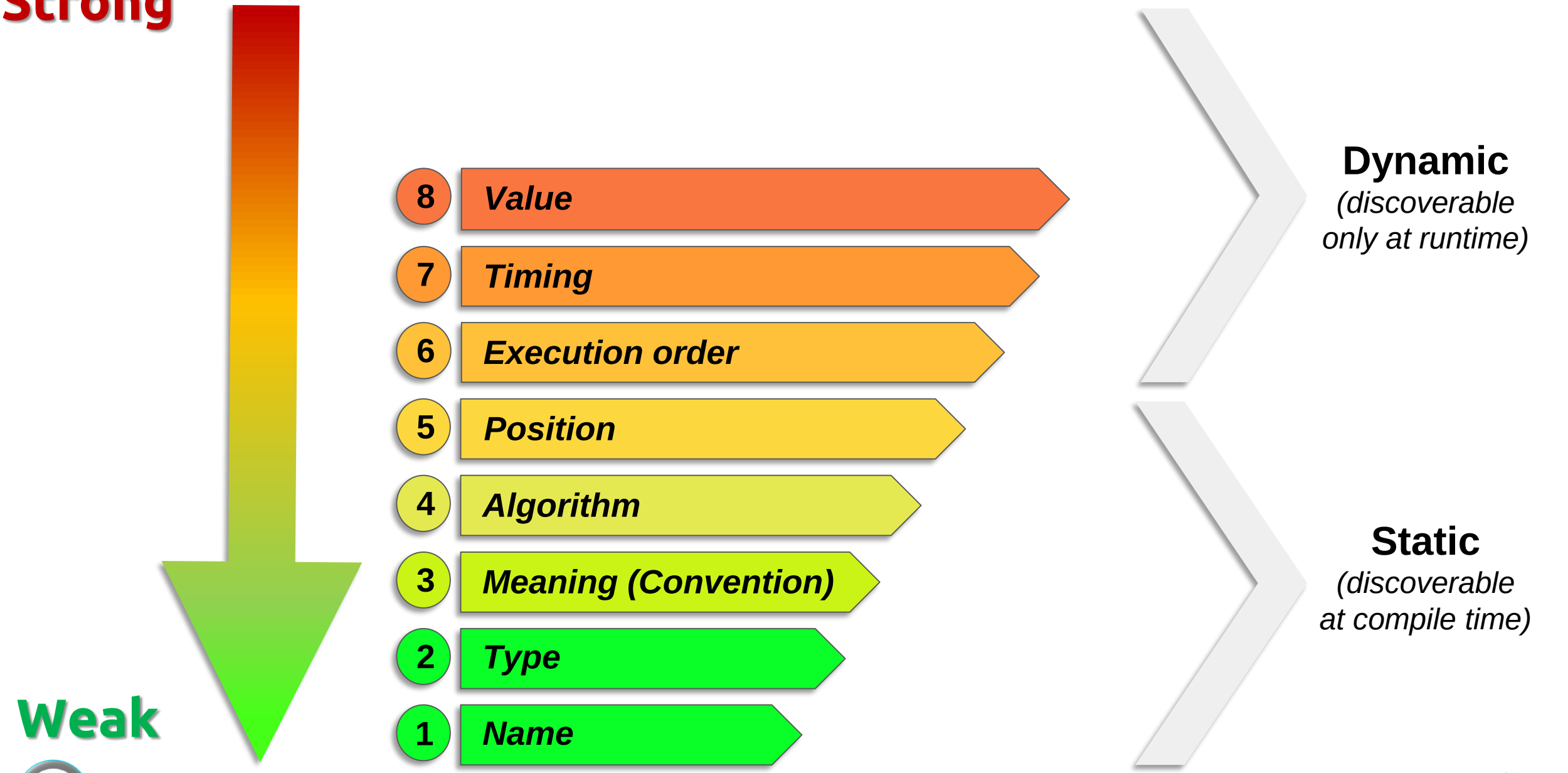
IBuildCars

Connascence of Timing

```
[TestFixture]
public class ServiceBusMessageHandler_Should{
  var messageHandler = new MessageHandler();

  [Test]
  public void ReceiveMessages(){
    var expectedMessage = new TestMessage();
    SendMessageToIntegrationTestBus(expectedMessage);
    Thread.Sleep(1000);

    var receivedMessage = messageHandler.ReadLastMessage();
    Assert.That(receivedMessage, Is.Equal(expectedMessage))
  }
}
```

Connascence of Timing

when the success of two or more calls depends on the timing of when they occur.


Strong
8 Value
7 Timing
6 Execution order
5 Position
4 Algorithm
3 Meaning (Convention)
2 Type
1 Name
Weak
Dynamic
(discoverable only at runtime)
Static
(discoverable at compile time)

```
[TestFixture]
public class ServiceBusMessageHandler_Should{
  var awaiter = new AutoResetEvent(false);
  var messageHandler = new MessageHandler();
  messageHandler.OnNewMessage += ()=>awaiter.Set();

  [Test]
  public void ReceiveMessages(){
    var expectedMessage = new TestMessage();
    SendMessageToIntegrationTestBus(expectedMessage);
    awaiter.WaitOne(1000);

    var receivedMessage = messageHandler.Read();

    Assert.That(receivedMessage, Is.Equal(message))
  }
}
```

Connascence of Time removed on the happy path using an event

Connascence of Time still there on timeout

Connascence of Identity

```
public class GlobalCounter{
  int count = 0;
  public void Increment(){
    count++;
  }
  public int CurrentCount(){
    return count;
  }
}
```

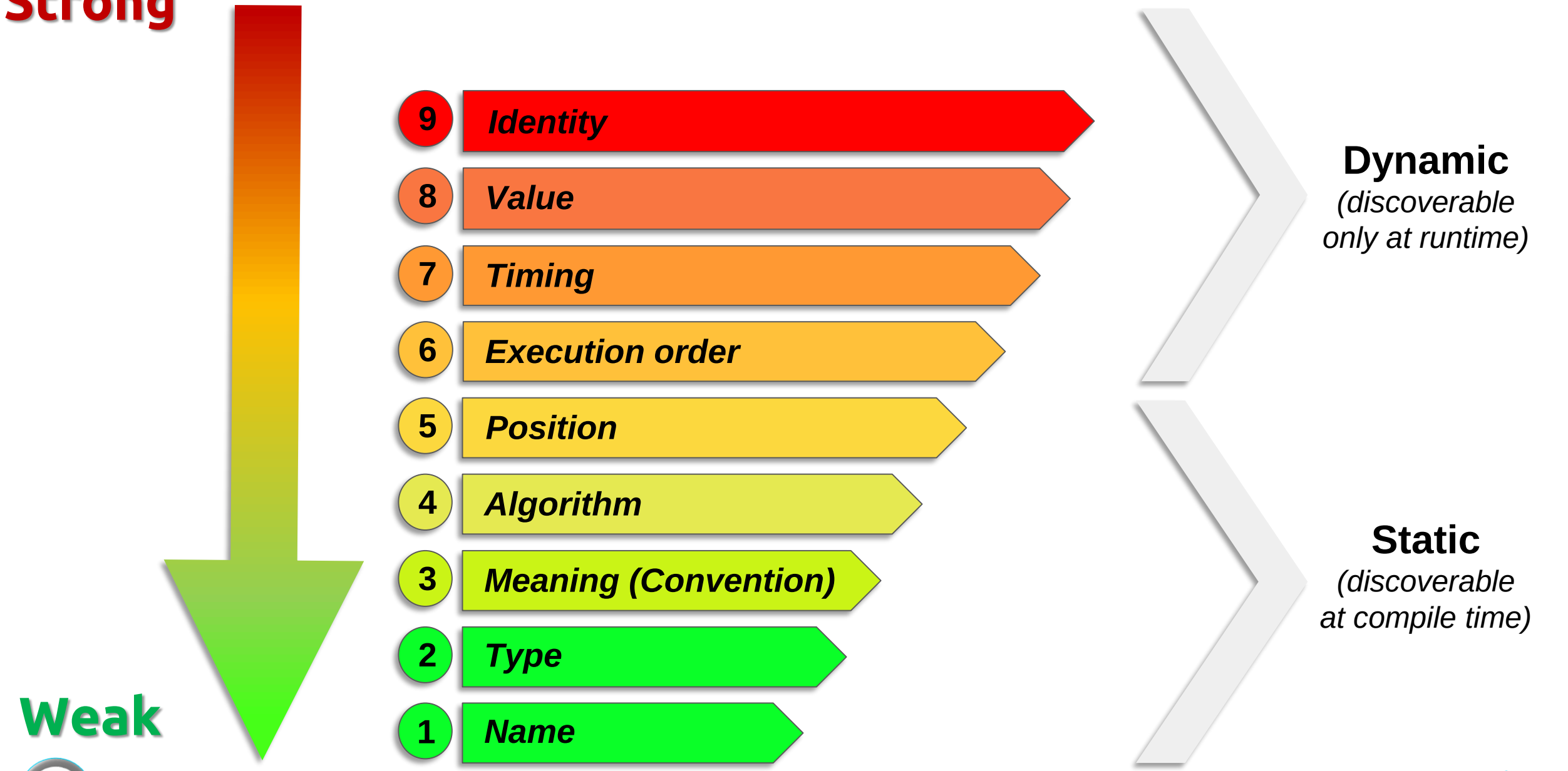
Singleton (anti)pattern

```
public class Controller{
  GlobalCounter _counter;
  public Controller(GlobalCounter counter){
    _counter = counter;
  }

  public ActionResult Home(){
    _counter.Increment();
  }
}
```

Works correctly if – and only if – **the same instance** is passed to all controllers

Connascence of Identity

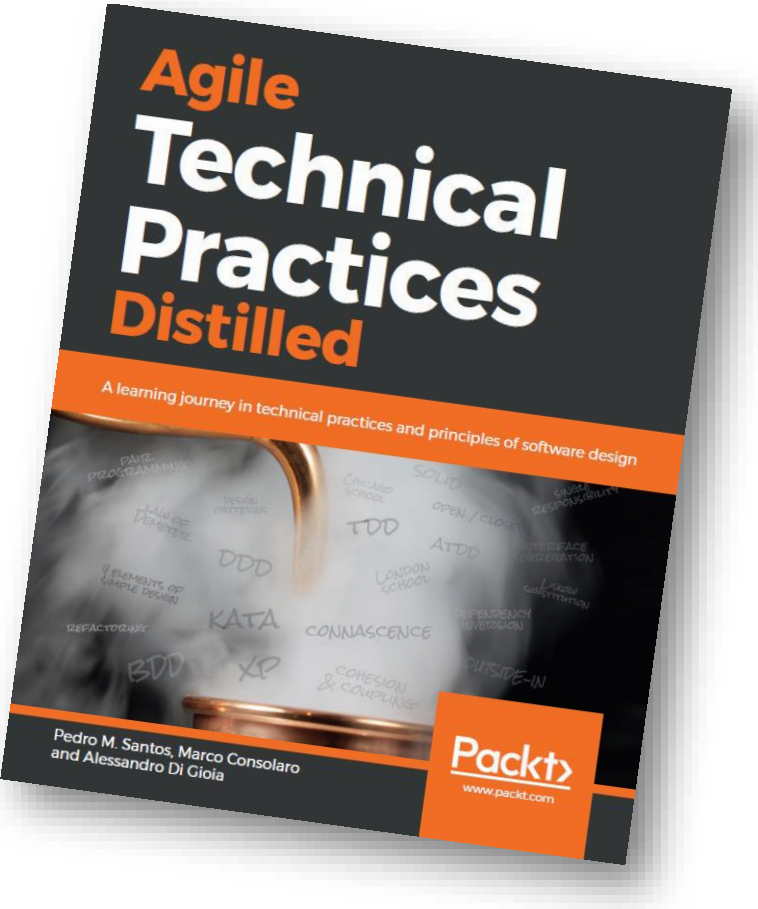
when one or more components must reference exactly one particular instance of another entity to work.


Strong
9 Identity
8 Value
7 Timing
6 Execution order
5 Position
4 Algorithm
3 Meaning (Convention)
2 Type
1 Name
Weak
Dynamic
(discoverable
only at runtime)
Static
(discoverable
at compile time)

Connascence of Manual Task

when expanding the functionality requires some tasks to be executed outside the codebase (manual).


Strong
10
Manual task
9
Identity
8
Value
7
Timing
6
Execution order
5
Position
4
Algorithm
3
Meaning (Convention)
2
Type
1
Name
Weak
Dynamic
(discoverable
only at runtime)
Static
(discoverable
at compile time)



"Add enough manual tasks and teams will spend more time trying to make things work, fidgeting with scripts and configuration, than actually focusing on writing well-designed code."

Marco Consolaro

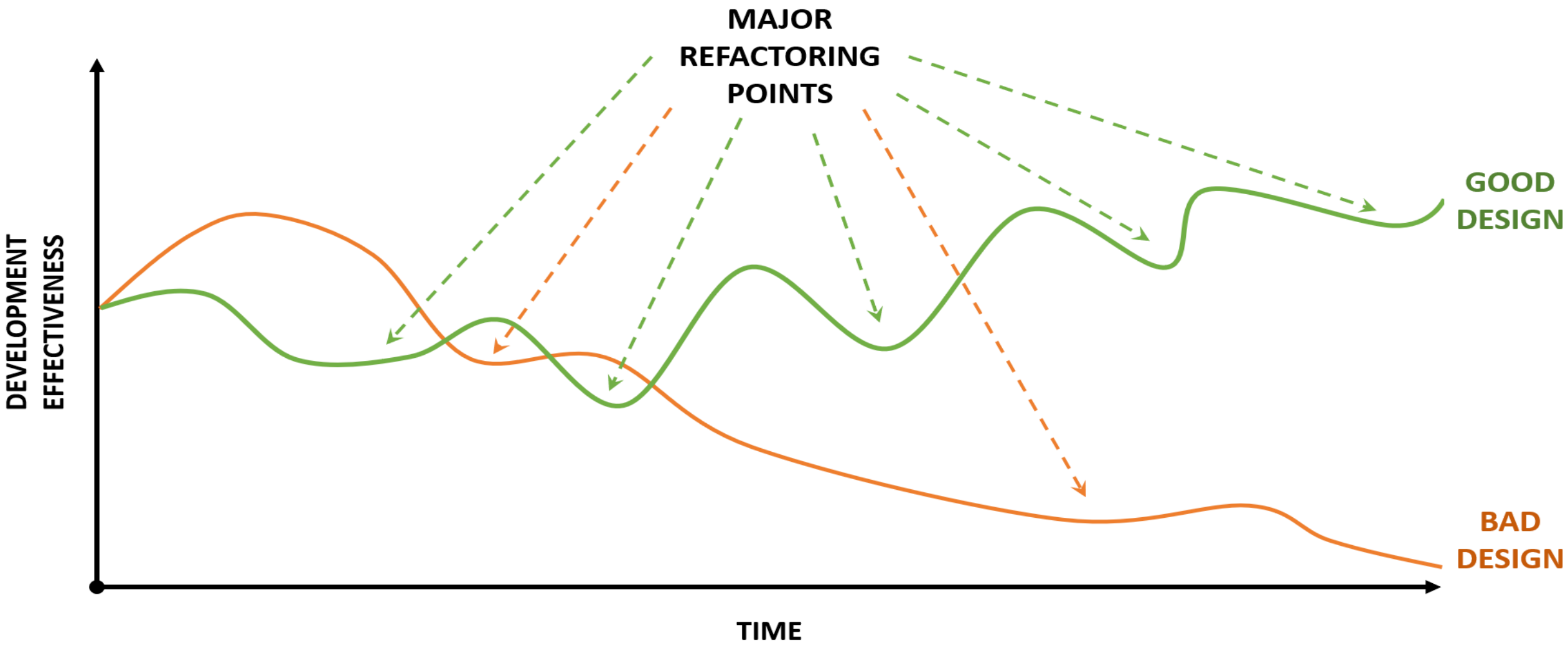
CONCLUSION

"It's all about the design..."

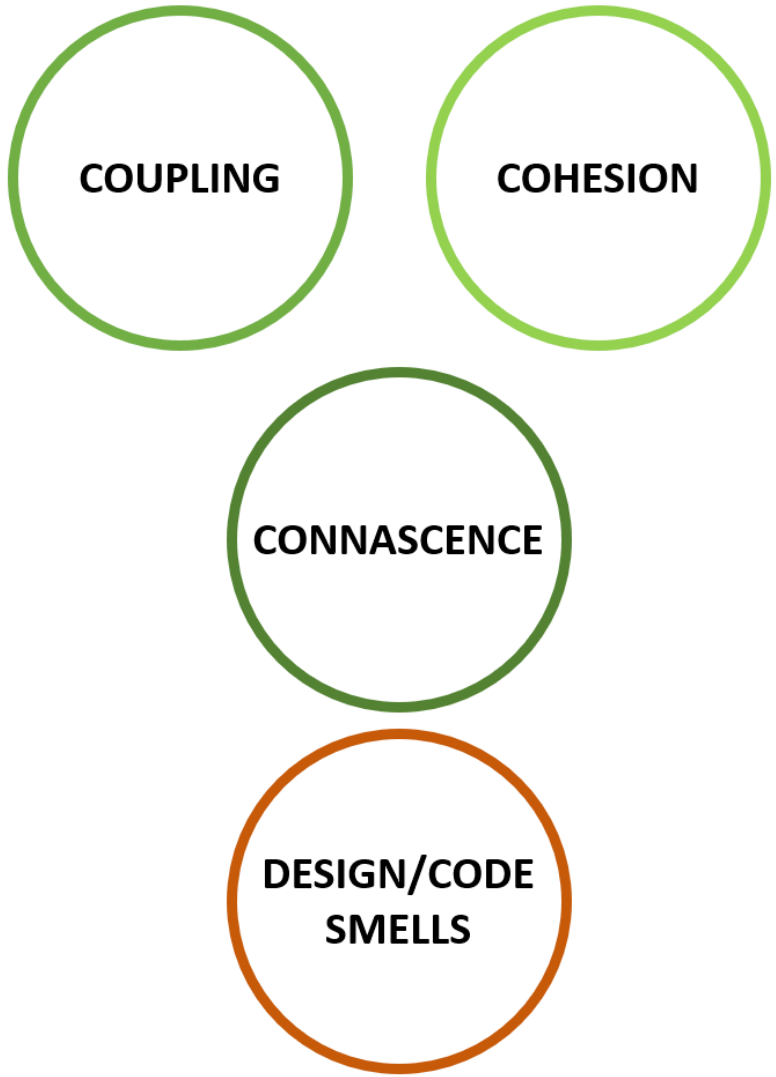
Pedro Santos

"We know the past but cannot control it,
we control the future but cannot know it."

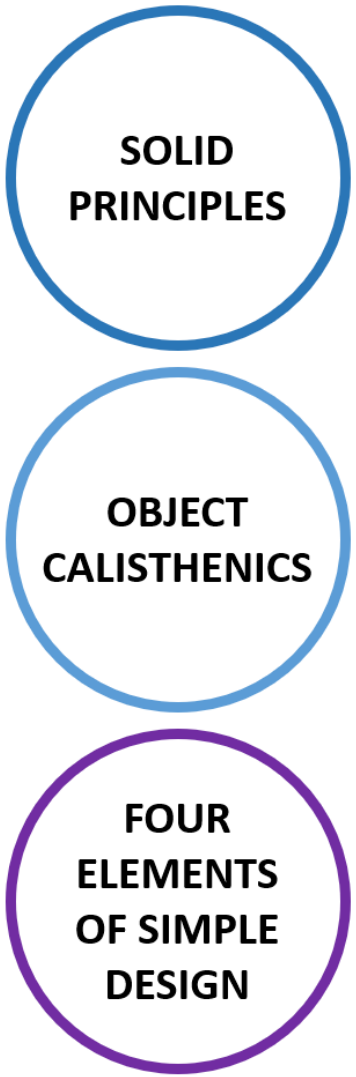
Claude Shannon



For the **PAST** we need **METRICS** and **ANALITYCS**
(we know it and can change it)



For the **FUTURE** we need **GUIDANCE** via **PRINCIPLES**
(we don't know it, but have control)



```
public class OrderFlow
{
  public void Execute(int customerId, int categoryId, int[] itemIds)
  {
    var orderId = GenerateOrderId();
    orderProcessor.ProcessOrder(orderId, customerId, categoryId, itemIds);
    invoiceProcessor.ProcessInvoice(orderId, customerId, categoryId, itemIds);
    ...
  }
}
```

Long parameter list

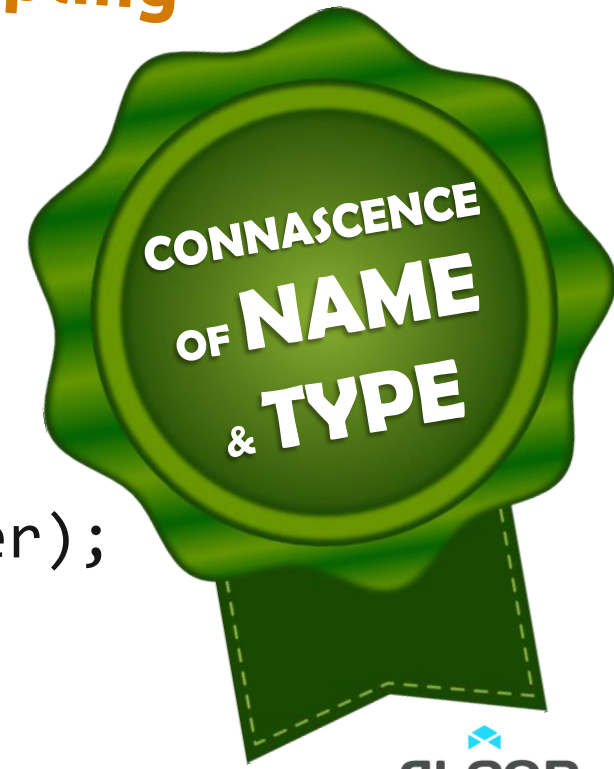
Connascence of Position

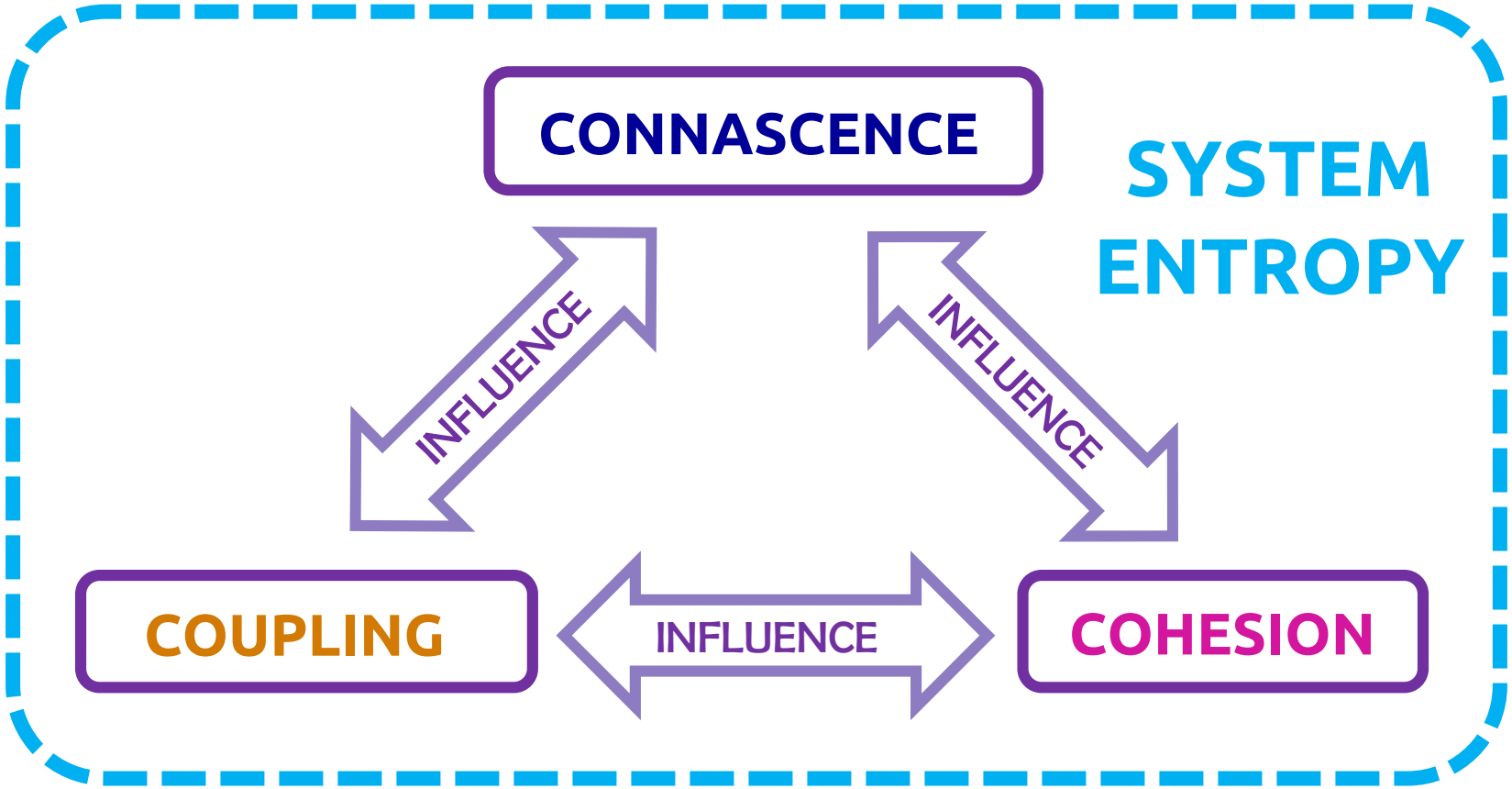
No single responsibility → low cohesion

Primitive Obsession

Data Coupling

```
public class OrderFlow
{
  public void Execute(Order order)
  {
    orderProcessor.Process(order);
    invoiceProcessor.ProcessInvoiceFor(order);
    ...
  }
}
```



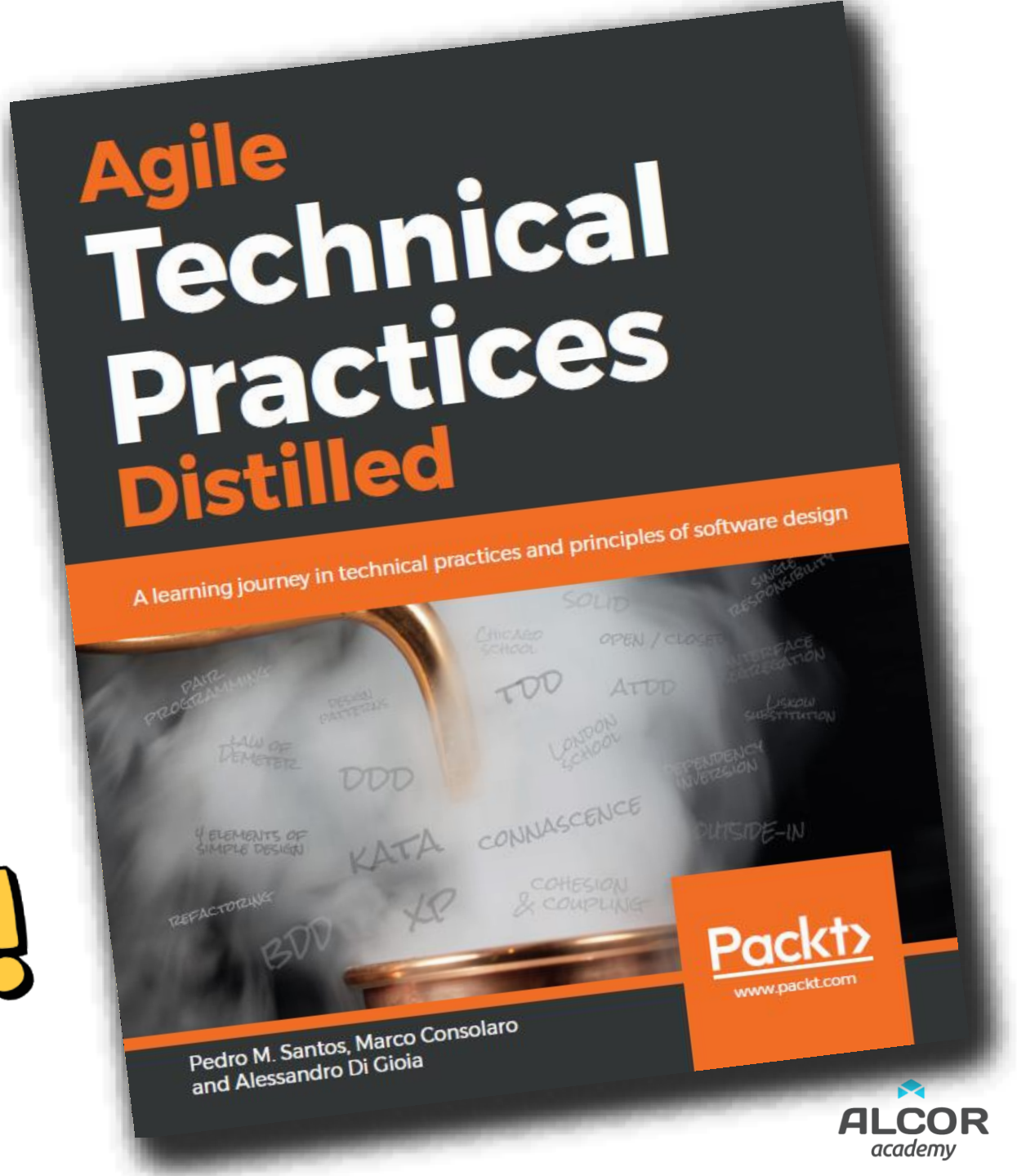


ENTROPY as the degree of disorder in a system

"Learn the rules
like a PRO,
so you can break
them like an
ARTIST."

Pablo Picasso

THANK YOU!



ALCOR
academy

Book a call with us

We are based in London, UK.

We serve customers worldwide.

alcor.academy/booking

info@alcor.academy

ALESSANDRO DI GIOIA

alex@alcor.academy

parajao

MARCO CONSOLARO

marco@alcor.academy

consolondon

ALCOR academy