Testing, Design, and Refactoring

(Workshop with Lab)

Jim Weirich
Chief Scientist
EdgeCase
@jimweirich

Joe O'Brien
Software Artisan
EdgeCase
@objo

EdgeCase
software artisans

Who We Are

EdgeCase
software artisans

Joe O'Brien is a father, speaker, author and developer. Before helping found EdgeCase, LLC, Joe was a developer with ThoughtWorks and spent much of his time working with large J2EE and .NET systems for Fortune 500 companies.

Jim Weirich has been active in the software development world for over twenty-five years, with experience that ranges from real-time data acquisition for jet engine testing to image processing and web services for the financial industry. He is current the chief scientist for EdgeCase, LLC.

Download

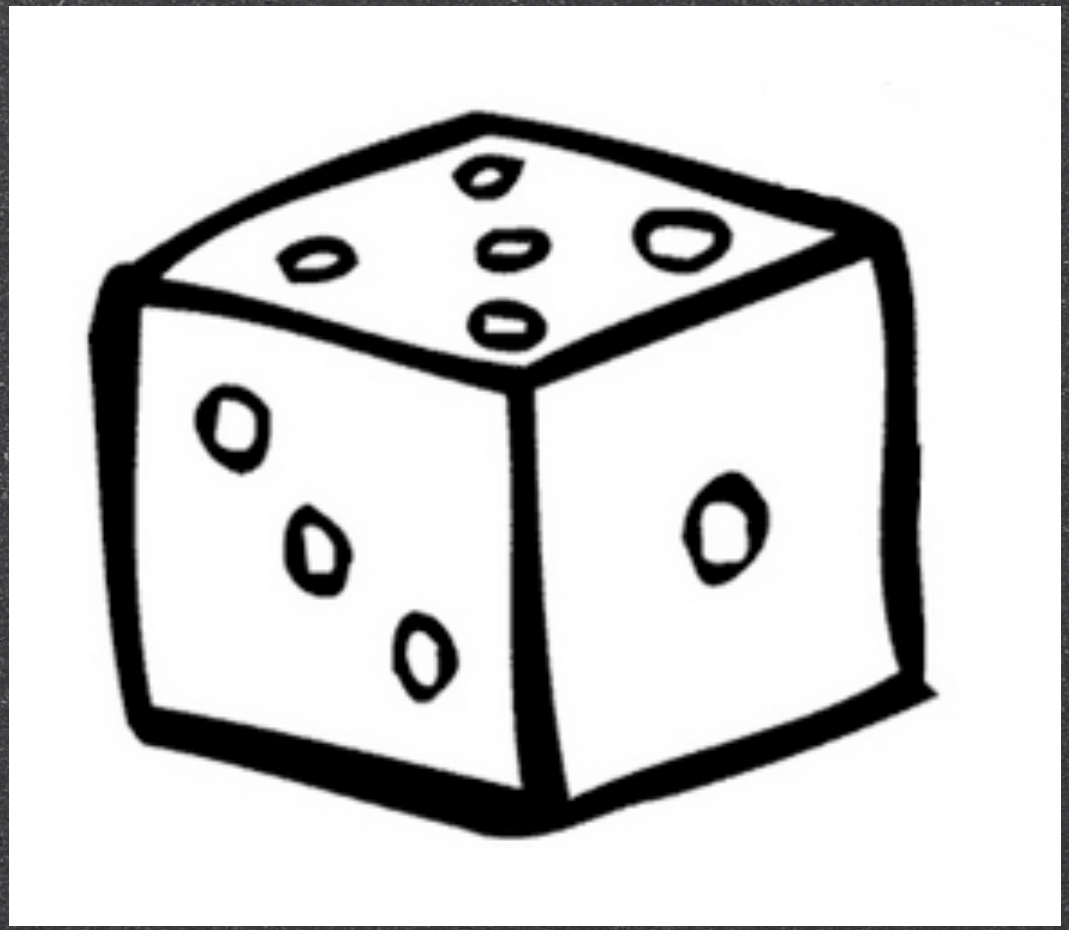
http://onestepback.org/download/refactor.zip

What makes a good design?

What makes a good design?

Object Oriented

What's an Object?



What's an Object?

- Attributes ...
- Behaviors ...

What's an Object?

- Attributes ...
  - Number of faces
  - Values on each face
  - Current top face
- Behaviors ...

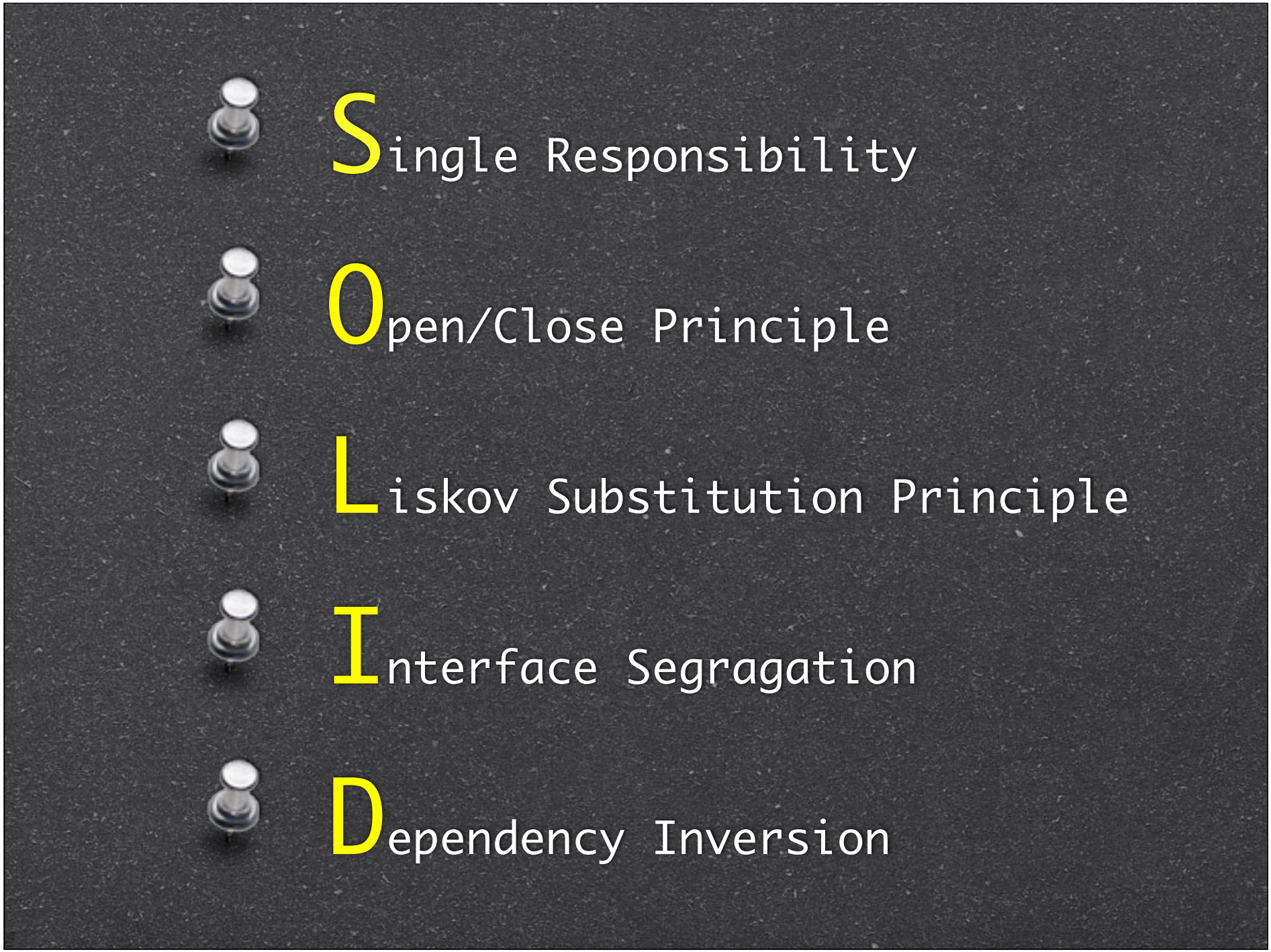
What's an Object?

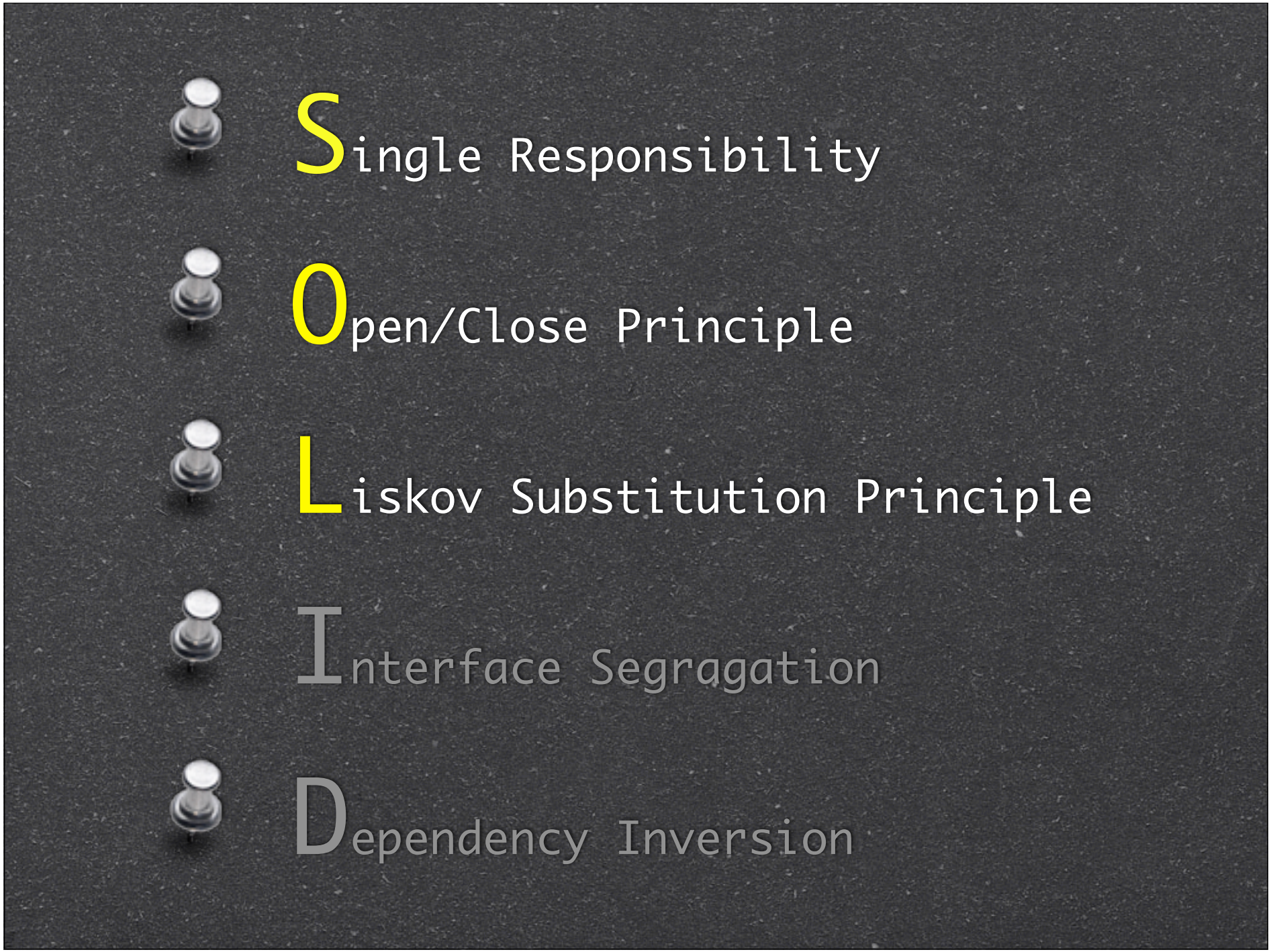
- Attributes ...
  - Number of faces
  - Values on each face
  - Current top face
- Behaviors ...
  - Roll
  - Report top face

What Makes a Good Design?

- Strong Cohesion
- Weak Coupling
- High Clarity

Cohesion


Single Responsibility
Open/Close Principle
Liskov Substitution Principle
Interface Segragation
Dependency Inversion


Single Responsibility
Open/Close Principle
Liskov Substitution Principle
Interface Segragation
Dependency Inversion

Single Responsibility

- Every object should have a single responsibility,
  - All services should be narrowly aligned with that responsibility.

Single Responsibility

- Symptom:
  - Class description contains the words “or” or “and”
  - (or words semantically equivalent)

Open / Closed Principle

- Software entities (classes, modules, functions, etc.)
  - should be open for extension,
  - but closed for modification

Open / Closed Principle

Monkey Patching!

Open / Closed Principle

Monkey Patching!

Open / Closed Principle

```
class Player
  def strategy
    ...
  end
end
```

```
class SmartPlayer < Player
  def strategy
    ...
  end
end
```

Liskov Substitution

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

Liskov Substitution

Subtypes should be usable anywhere the original type was used.

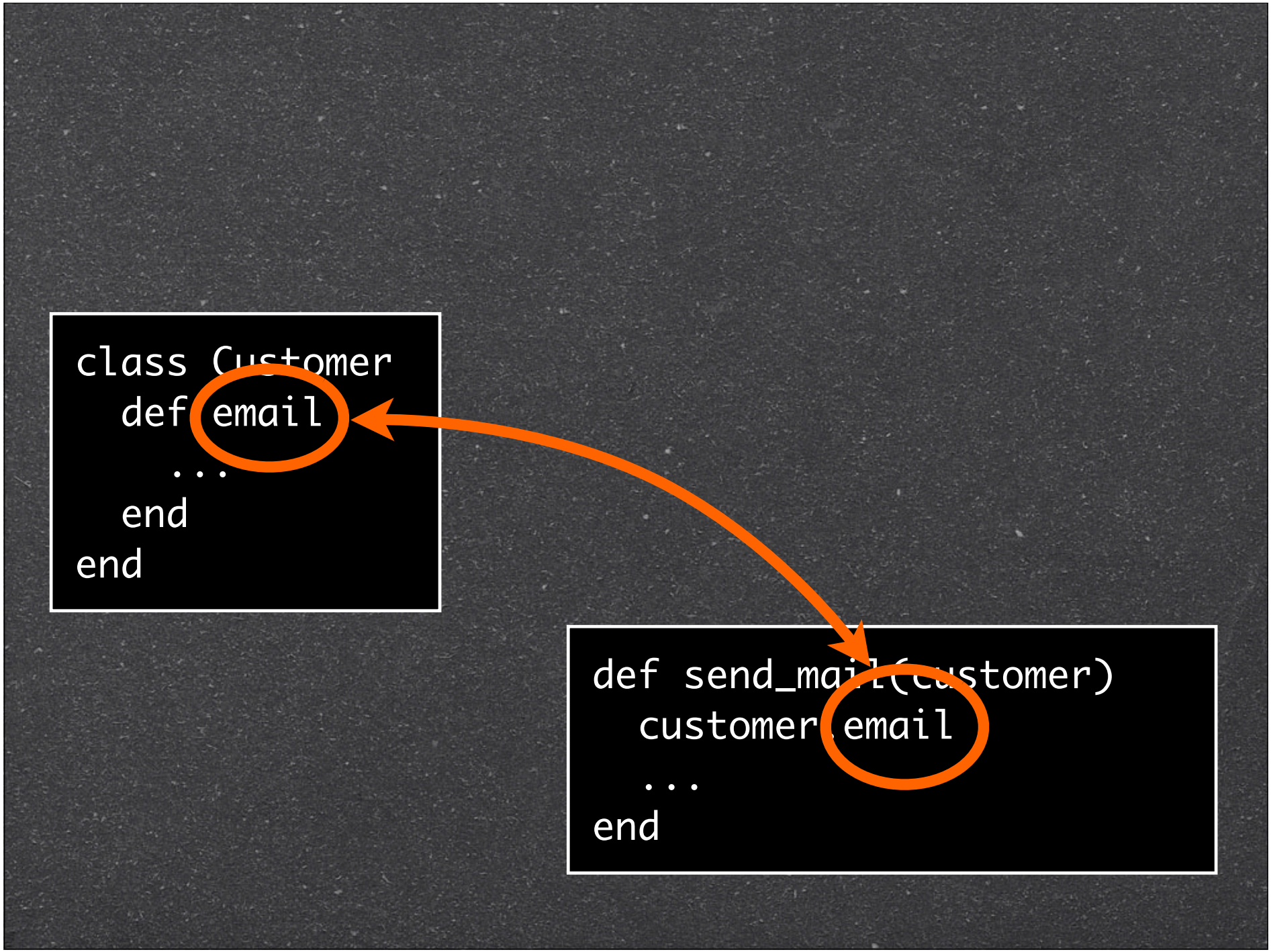
Liskov Substitution

If it quacks like a duck,
then it is a duck

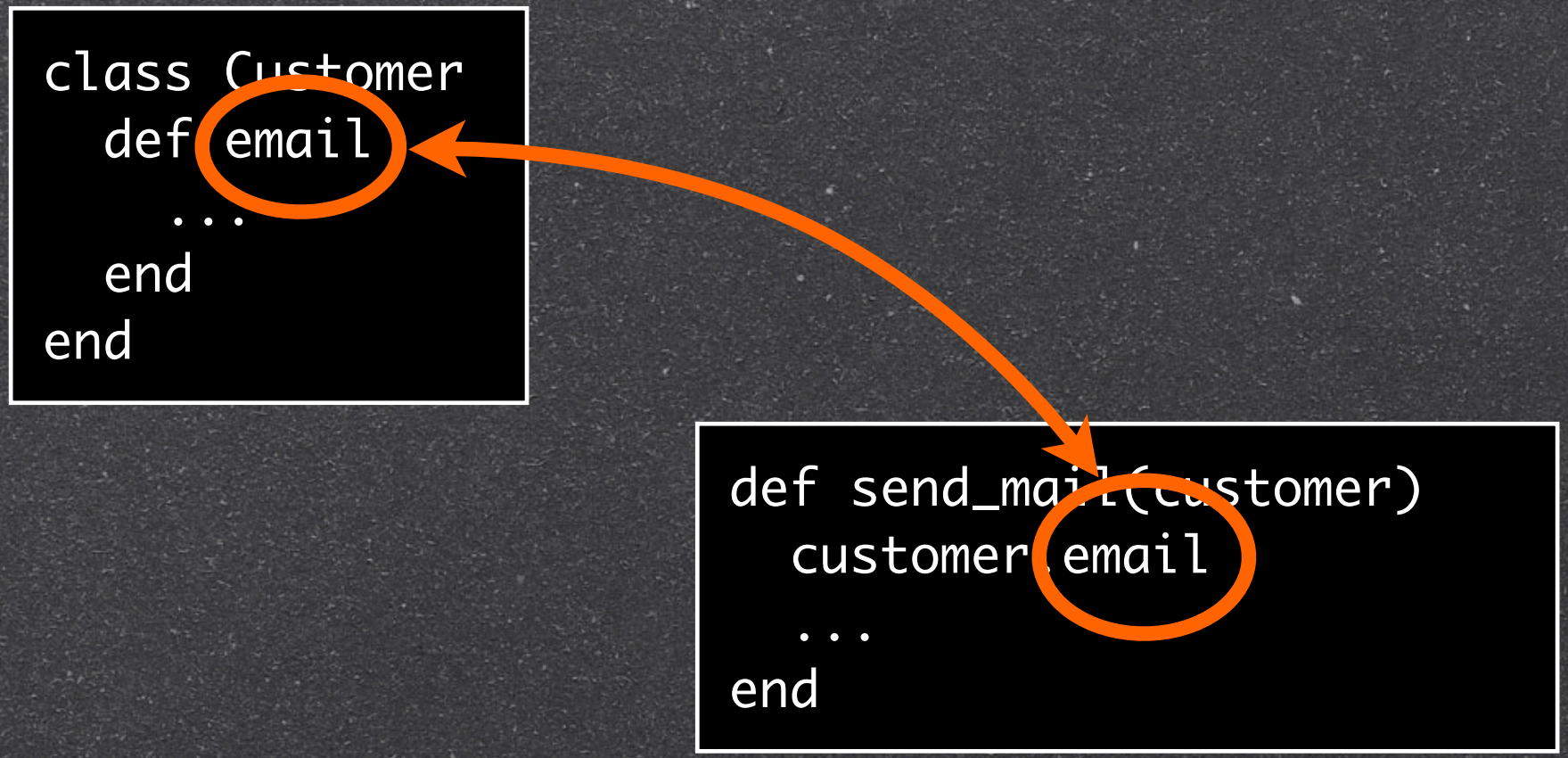
Coupling

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

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


class Customer
def email
...
end
end
def send_mail(customer)
customer.email
...
end

Connascence of Name



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
```

X

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

Connascence of Name

Another example?

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

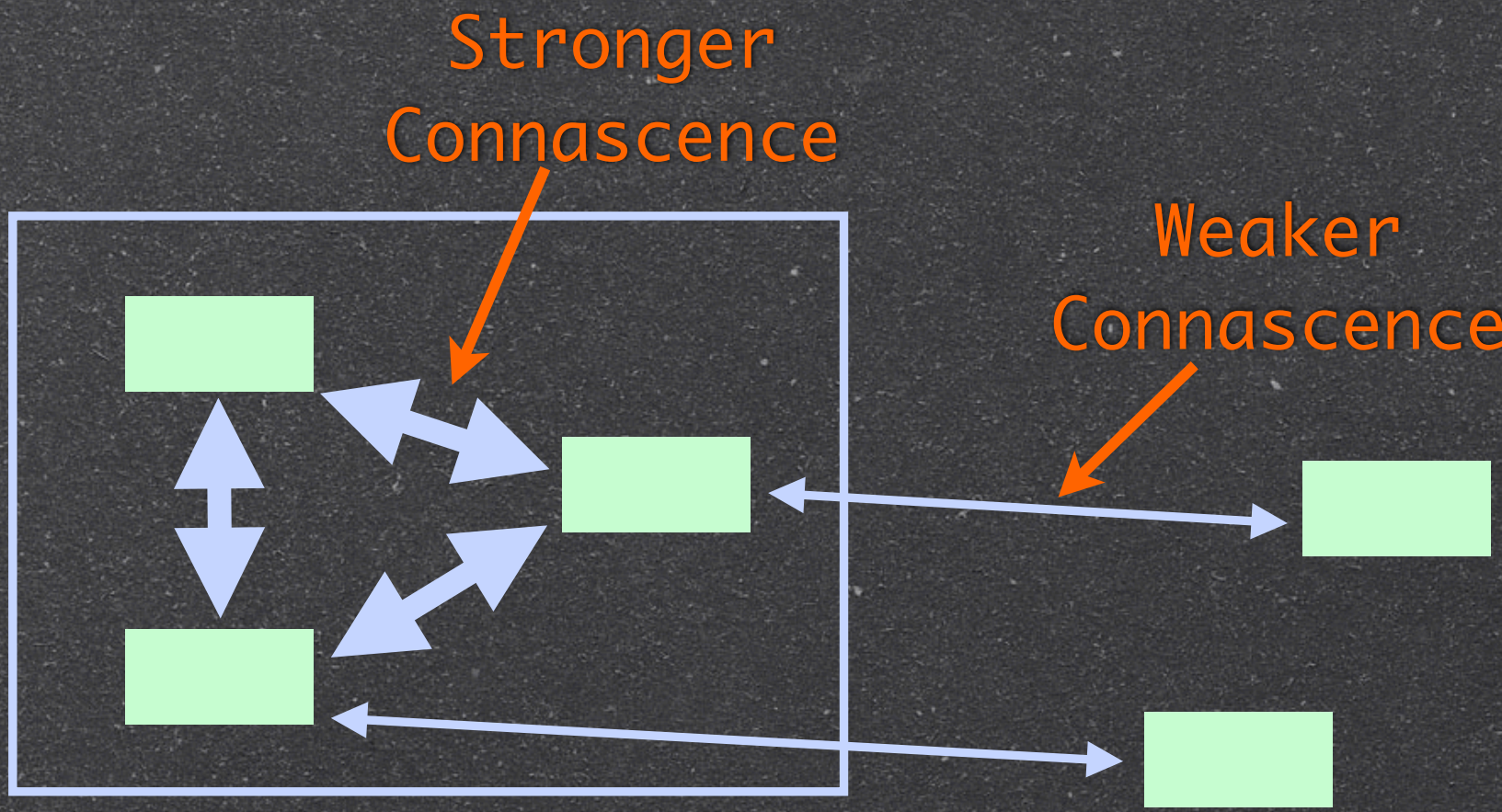
```
def send_mail(customer)
  customer.email
  ...
end
```

Connascence

- **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**

Locality Matters

Rule of Locality



Locality Matters

Rule of Locality

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

Rule of Degree

Convert high degrees of connascence
into
weaker forms of connascence

Clarity

Simplicity

- Correctly runs (and passes) all the tests
- Contains no duplication
- Clearly expresses all the ideas/ intentions we needed to express
- Minimizes the number of classes and methods (has no superfluous parts)

How do you get a good design?

How to Detect Problems in your Design

Code Smells

```
def take_turn
  history = []
  turn_score = 0
  roller.roll(5)
  loop do
    if roller.points == 0
      turn_score = 0
      bust(history, roller, turn_score)
      break
    end
    turn_score += roller.points
    if ! roll_again?
      hold(history, roller, turn_score)
      break
    end
    again(history, roller, turn_score)
    dice_count = (roller.unused == 0) ? 5 : roller.unused
    roller.roll(dice_count)
  end
  turns << Turn.new(:rolls => history)
  save
  turns.last
end
```

Long Method

- Extract Method
- Replace Temp with Query
- Introduce Parameter Object
- Preserve Whole Object
- Replace Method with Method Object

Large Class

- Extract Class
- Extract Subclass

```
class AutoPlayer
  def self.players
    [Randy, Connie, Aggie]
  end
end
```

Lazy Class

- Collapse Hierarchy
- Inline Class

```
def roll_dice(dice_count=5)
  roller.roll(dice_count)
  accumulated_score = roller.points
  accumulated_score += turns.last.rolls.last.accumulated_score unless
    turns.last.rolls.empty?
  roll = Roll.new(
    :faces => roller.faces.map { |n| Face.new(:value => n) },
    :score => roller.points,
    :unused => roller.unused,
    :accumulated_score => accumulated_score)
  turns.last.rolls << roll
  if roller.points == 0
    turns.last.rolls.last.action = :bust
  end
end
```

```
def roll_dice(dice_count=5)
  roller.roll(dice_count)
  accumulated_score = roller.points
  accumulated_score += turns.last.rolls.last.accumulated_score unless
    turns.last.rolls.empty?
  roll = Roll.new(
    :faces => roller.faces.map { |n| Face.new(:value => n) },
    :score => roller.points,
    :unused => roller.unused,
    :accumulated_score => accumulated_score)
  turns.last.rolls << roll
  if roller.points == 0
    turns.last.rolls.last.action = :bust
  end
end
```

Feature Envy

- Move Method
- Extract Method, then Move Method

Inappropriate Intimacy

- Move Method
- Move Field
- Change Bidirectional Association to Unidirectional Association
- Extract Class
- Hide Delegate
- Replace Inheritance with Delegation

5.minutes.ago

Primitive Obsession

- Replace Data Value with Object
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Extract Class

Common Smells

- Duplicated Code
- Message Chains
- Divergent Change
- Data Class
- Data Clumps
- Speculative Generality

Code Smell Catalog

http://wiki.java.net/bin/view/People/SmellsToRefactorings

http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm

Greed

Greed

- Greed is a dice game for two or more players using 5 six-sided dice
- Each player takes a turn consisting of several rolls
- The player decides whether to roll again, or not
- Each turn is scored and the points are added to the player's total score.

Scoring Rolls

- A triplet of any number is worth 100 * that number
- Except a triplet of ones, which are worth 1000 points
- Single “ones” are worth 100 points
- Single “fives” are worth 50 points
- All other values do not contribute points to the roll.

Rolling Again

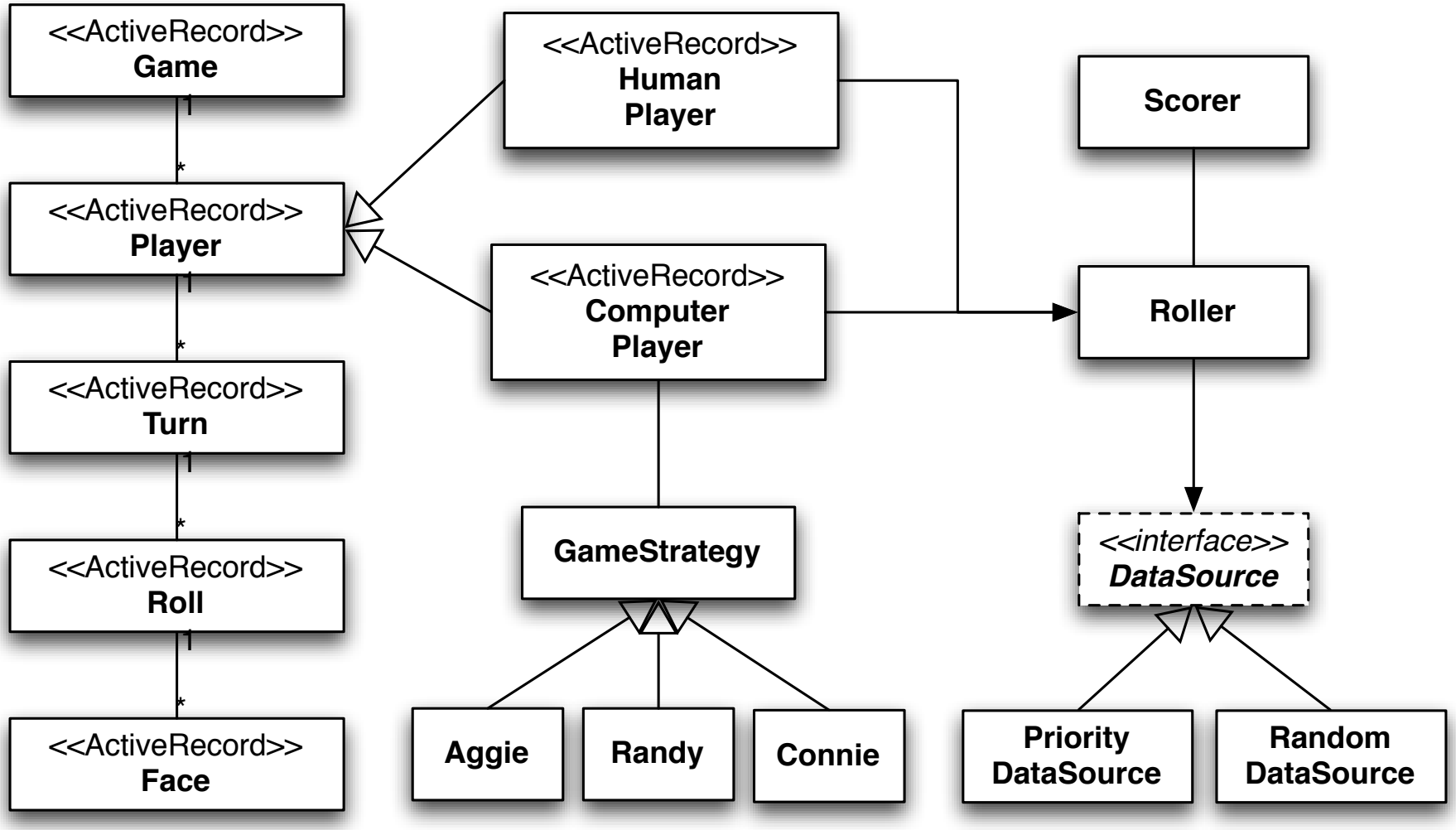
- If a roll scored points, those points are added to the turns points, and the player may roll again with any non-scoring dice.
- If all dice have scored, then the player may roll again with all five of the dice.

Rolling Again

- If a roll does not score any points, then the player loses his turn and loses any points from that turn. This is called going “bust”.
- If a player elects to stop rolling before going bust, the points accumulated in this turn are added to his total score.

Winning

- First player to 3000 points wins


<<ActiveRecord>>
Game
1
*
<<ActiveRecord>>
Player
1
*
<<ActiveRecord>>
Turn
1
*
<<ActiveRecord>>
Roll
1
*
<<ActiveRecord>>
Face
<<ActiveRecord>>
Human
Player
<<ActiveRecord>>
Computer
Player
Scorer
Roller
GameStrategy
<<interface>>
DataSource
Aggie
Randy
Connie
Priority
DataSource
Random
DataSource

Extra Credit #1

- Get in the game
  - A player must roll at least 300 in a single turn before he is allowed to start accumulating points.

Extra Credit #2

- Allow Multiple Computer Players

Extra Credit #2

- Allow configurable
  - Winning Score
  - Number of Dice

Illustrations from:

Frits Ahlefeldt

HikingArtist.com