PABLO TORRECILLA

Tech Lead at Rakuten

Twitter

PAU@NOSOLOPAU.COM

GitHub

PABLO TORRECILLA

ALMOST SOLID

WHAT IS A GOOD DESIGN?

WHAT IS A GOOD DESIGN?

A good design make code changes as side-effect free as possible.

WHAT IS A GOOD DESIGN?

Robert C. Martin proposed 5 rules: the SOLID Principles.

WHAT IS SOLID?

Single Responsibility Principle

Open/closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

SOLD

SINGLE RESPONSIBILITY PRINCIPLE

A class should have only one responsibility and should fully execute it.

A class should have only one reason to change.

SINGLE RESPONSIBILITY PRINCIPLE

```
class Product
  attr_accessor :id, :price

  def initialize(id, price)
    @id = id
    @price = price
  end
end
```

NEW

REQUIREMENT

SINGLE RESPONSIBILITY PRINCIPLE

```
class Product
  attr_accessor :id, :price

  def initialize(id, price)
    @id = id
    @price = price
  end
end
```

SINGLE RESPONSIBILITY PRINCIPLE

```ruby
require 'rest-client'

class Product
  attr_accessor :id, :price

  def initialize(id, price)
    @id = id
    @price = price
  end

  def update_price
    response = RestClient.get("http://service/prices/" + @id)
    @price = response.to_i
  end
end
```


BIRD HOUSE
INSTRUCTIONS
EASY TO
BUILD!
EASY BIRDHOUSE
BUILD IN MINUTES!!!

SINGLE RESPONSIBILITY PRINCIPLE

```ruby
require 'rest-client'

class Product
  attr_accessor :id, :price

  def initialize(id, price)
    @id = id
    @price = price
  end

  def update_price
    response = RestClient.get("http://service/prices/" + @id)
    @price = response.to_i
  end
end
```

SINGLE RESPONSIBILITY PRINCIPLE

```
require 'rest-client'

class ProductPriceService
  attr_accessor :product

  def initialize(product)
    @product = product
  end

  def get_price
    RestClient.get("http://service/prices" + @product.id).to_i
  end
end
```

SINGLE RESPONSIBILITY PRINCIPLE

```ruby
class Product
  attr_accessor :name, :price

  def initialize(id, price)
    @id = id
    @price = price
  end

  def update_price
    @price = ProductPriceService.new(self).get_price
  end
end
```

SINGLE RESPONSIBILITY PRINCIPLE

If you describe the purpose of a module using AND or OR, you are doing it wrong.

SOLD

OPEN/CLOSED PRINCIPLE

A class should be open for extension and closed for modification.

When you add a new behavior to a system, **create new classes** which inherit from the class you wish to extend, **instead of modifying** the original class.

OPEN/CLOSED PRINCIPLE

```
class ProductParser
  def initialize(file)
    @file = file
  end

  def parse
    # Parse CSV
  end
end
```

NEW

REQUIREMENT

OPEN/CLOSED PRINCIPLE

```
class ProductParser
  def initialize(file)
    @file = file
  end

  def parse
    # Parse CSV
  end
end
```

OPEN/CLOSED PRINCIPLE

```ruby
class ProductParser
  def initialize(file, format)
    @file = file
    @format = format
  end

  def parse
    case @format
      when :xml
        parse_xml
      when :csv
        parse_csv
    end
  end

  private

  def parse_xml
    # Parse XML
  end

  def parse_csv
    # Parse CSV
  end
end
```

OPEN/CLOSED PRINCIPLE

```
class XMLParser
  def parse
    # Parse XML
  end
end

class CSVParser
  def parse
    # Parse CSV
  end
end
```

OPEN/CLOSED PRINCIPLE

```
class ProductParser
  def initialize(parser)
    @parser = parser
  end

  def parse
    @parser.parse
  end
end

ProductParser.new(CSVParser.new).parse
```

OPEN/CLOSED PRINCIPLE

Prefer extension over monkey patching.

SOLD

LISKOV SUBSTITUTION PRINCIPLE

Subclasses must be substitutable for their base classes.

Objects that are meant to be treated as subtypes of a base type should not break the contracts of the base type.

THE MOUNTAIN

SUBSTITUTION PRINCIPLE

LISKOV SUBSTITUTION PRINCIPLE

```
class Parser
  def initialize(file)
    @file = file
  end

  def read_file
    File.open(@file, "rb").read
  end
end
```

LISKOV SUBSTITUTION PRINCIPLE

```ruby
class XMLParser < Parser
  def parse
    @content = read_file

    # Do stuff with @content
  end
end
```

```ruby
class CSVParser < Parser
  def parse
    @content = read_file

    # Do stuff with @content
  end
end
```

LISKOV SUBSTITUTION PRINCIPLE

```ruby
class XMLParser < Parser
  def parse
    @content = read_file

    # Do stuff with @content
  end
end
```

```ruby
class CSVParser < Parser
  def parse
    @content = read_file

    # Do stuff with @content
  end

  def separator
    ','
  end
end
```


STAIRWAY TO FAIL

LISKOV SUBSTITUTION PRINCIPLE

```
parsers = [
  CSVParser.new('file.csv'),
  XMLParser.new('file.xml')
]

parsers.each do |parser|
  puts parser.separator
end
```

LISKOV SUBSTITUTION PRINCIPLE

```
class CSVParser < Parser
  def parse
    @content = read_file

    # Do stuff with @content
  end

  private

  def separator
    ','
  end
end
```

LISKOV SUBSTITUTION PRINCIPLE

```ruby
class Parser
  def initialize(file)
    @file = file
  end

  def read_file
    File.open(@file, "rb").read
  end

  def separator
    raise 'This method needs to be implemented!'
  end
end
```

LISKOV SUBSTITUTION PRINCIPLE

```
class Parser
  def initialize(file)
    @file = file
  end

  def read_file
    File.open(@file, "rb").read
  end

  def separator
    ''
  end
end
```

LISKOV SUBSTITUTION PRINCIPLE

```
class CSVParser
  include HasSeparator

  def parse
    @content = read_file

    # Do stuff with @content
  end
end


module HasSeparator
  def separator
    ','
  end
end
```

LISKOV SUBSTITUTION PRINCIPLE

Require no more, promise no less.

SOLD

DEPENDENCY INVERSION PRINCIPLE

Abstractions should not depend on details.
Details should depend on abstractions.

DEPENDENCY INVERSION PRINCIPLE

```
class ProductReport
  def content
    # Report
  end

  def print
    HTMLFormatter.new.format(content)
  end
end


class HTMLFormatter
  def format(content)
    %{<html><body>#{content}</body></html>}
  end
end
```


YOUR SHIPMENT OF FAIL
HAS BEEN DELIVERED

DEPENDENCY INVERSION PRINCIPLE

```
class ProductReport
  def content
    # Report
  end

  def print(formatter)
    formatter.format(content)
  end
end

ProductReport.print(HTMLFormatter.new)
```

DEPENDENCY INVERSION PRINCIPLE

Decouple high-level functionality from low-level implementation.

&I

INTERFACE SEGREGATION PRINCIPLE

When a client depends on a class that contains interfaces that the client does not use, but that other clients do use, then that client will be affected by the changes that those other clients force upon the class.

INTERFACE SEGREGATION PRINCIPLE

Make interfaces that are client specific.

THANKS!

Twitter

PAU@NOSOLOPAU.COM

GitHub