STP SOFTWARE TEST PROFESSIONALS CONFERENCE SPRING 2019

Software Design Patterns for Quality Engineers

Peter Kim

Hello

- Peter

- Software Engineer
- Quality Assurance
- Quality Engineer / Automation Specialist
- Engineering Manager
- QE Manager
- 8 Startups
- 5 Fortune 100 Companies

Hello

- Peter

- Co-organizer for DC Agile Software Testing Meetup (dcast.io)
- Network with QE/DEV leadership
  - Reality vs. “best practices”
  - Work closely with Consultants
  - Work closely with HR/Hiring
- Learning to improve my soft skills
- Tinkering with new technologies
- Crash/Reliability (Chaos) Testing
- AI/ML
- Advanced “Near Real Time” Test Reporting Systems that Scale
- GPU Powered Databases

… observations

- Hard working employees that could’ve got more done if they knew a little bit more ...
- Too much work on boiler plate code vs. writing code for the task at hand.
- Scalability pains due to poor design
- Lack of trust from DEV team due to poor coding practices
- Brittle automation …
- Poor Test Reports
- Too much time to analyze post-execution results
- Fear of changing

“un-Learn”
“Learn”

- Hard working employees that could’ve got more done if they knew a little bit more ...
- Too much work on boiler plate code vs. writing code for the task at hand.
- Scalability pains due to poor design
- Lack of trust from DEV team due to poor coding practices
- Brittle automation …
- Poor Test Reports
- Too much time to analyze post-execution results

Change Yourself

- Fear of change → Passion to Improve

  *Accept that “change” is a good thing.

  .. if your so passionate about QA, then wouldn’t you want to learn more ..

- Self Advocate (yourself and the team)

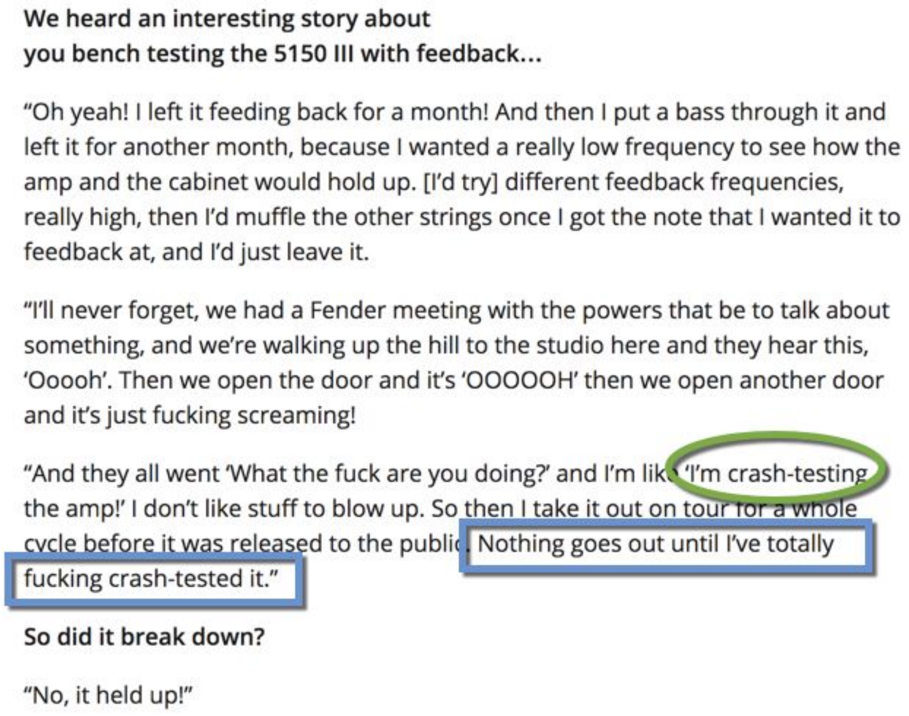
Testing is Fun

MusicRadar

Eddie Van Halen talks building the Frankenstein, honing the 5150 and EVH Gear

Guitar legend goes in-depth on the guitar gear behind his career (53 kB)

- https://www.musicradar.com/news/guitars/eddie-van-halen-talks-building-the-frankenstein-honing-the-5150-and-evh-gear-641564



We heard an interesting story about you bench testing the 5150 III with feedback...

"Oh yeah! I left it feeding back for a month! And then I put a bass through it and left it for another month, because I wanted a really low frequency to see how the amp and the cabinet would hold up. [I'd try] different feedback frequencies, really high, then I'd muffle the other strings once I got the note that I wanted it to feedback at, and I'd just leave it.

"I'll never forget, we had a Fender meeting with the powers that be to talk about something, and we're walking up the hill to the studio here and they hear this, 'Ooooh'. Then we open the door and it's 'OOOOOH' then we open another door and it's just fucking screaming!

"And they all went 'What the fuck are you doing?' and I'm like 'I'm crash-testing the amp!' I don't like stuff to blow up. So then I take it out on tour for a whole cycle before it was released to the public. Nothing goes out until I've totally fucking crash-tested it."

So did it break down?

"No, it held up!"

“Future You” in the next 3 hrs

- OOP
- Inheritance vs. Composition
- Understand the Significance of SWDP
- How SWDP are used
- When to use SWDP
- Key SWDP for QE
- Applying SWDP into practice

Agenda

Intro

OOD

Inheritance vs. Composition
Key Principles

Singleton
Composite
Command
Template
Strategy

re:Intro

Reasons that you are here ..

- You want to improve your software design skills
- You have experienced disappointment and challenges in designing well thought out programs
- You’re already knowledgeable with object-oriented programming, but it’s just not enough ..
- You’re passionate to improve your skill set
- Your automated tests are brittle and often times require refactoring
- You’ve been tasked to write high quality test scripts
- You’ve been tasked to design and implement a test framework
- You want to understand how to recognize problems that can be solved with design patterns

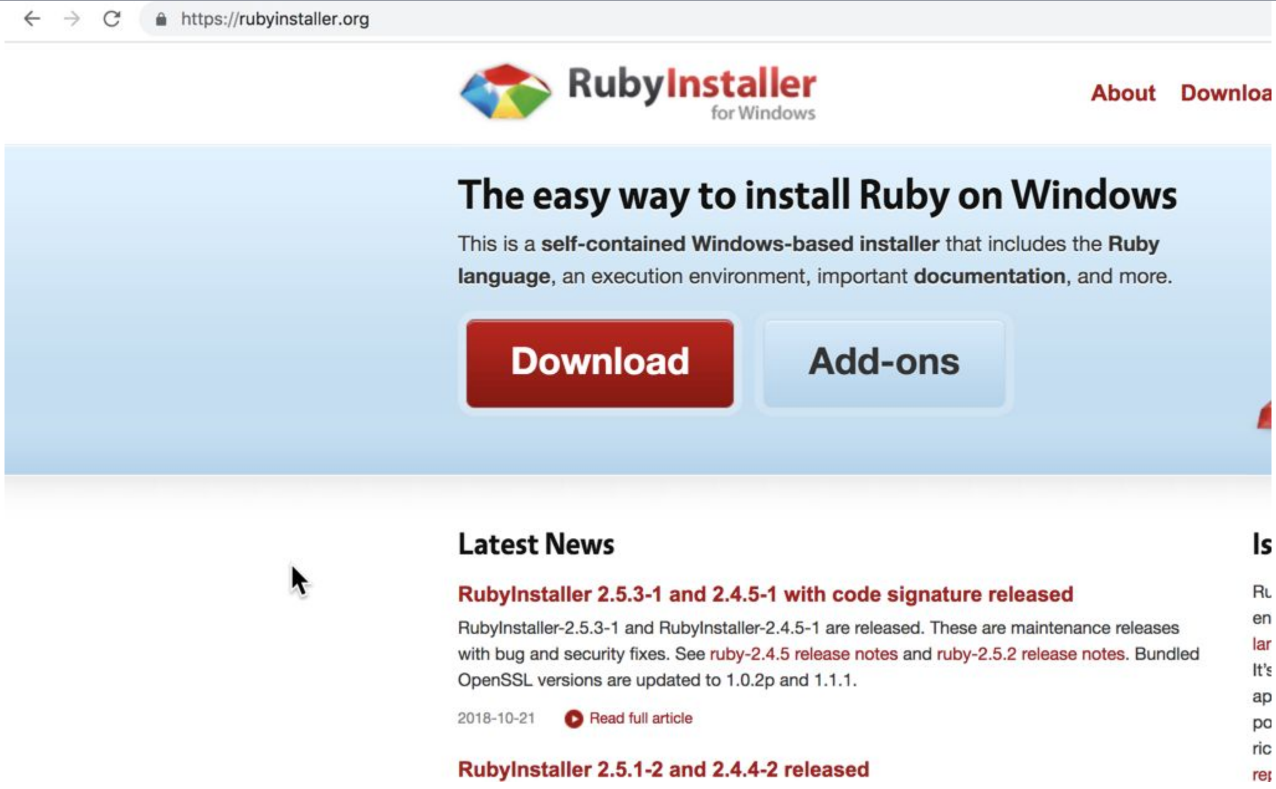
Philosophy

“Absorb what is useful,
Discard what is useless,
And add what is
Specifically your own.”

Code: GIT

https://github.com/h20dragon/qe-design-patterns

Prep: Install Ruby


https://rubyinstaller.org

RubyInstaller for Windows

About Downloa

The easy way to install Ruby on Windows

This is a **self-contained Windows-based installer** that includes the **Ruby language**, an execution environment, important **documentation**, and more.

Latest News

RubyInstaller 2.5.3-1 and 2.4.5-1 with code signature released

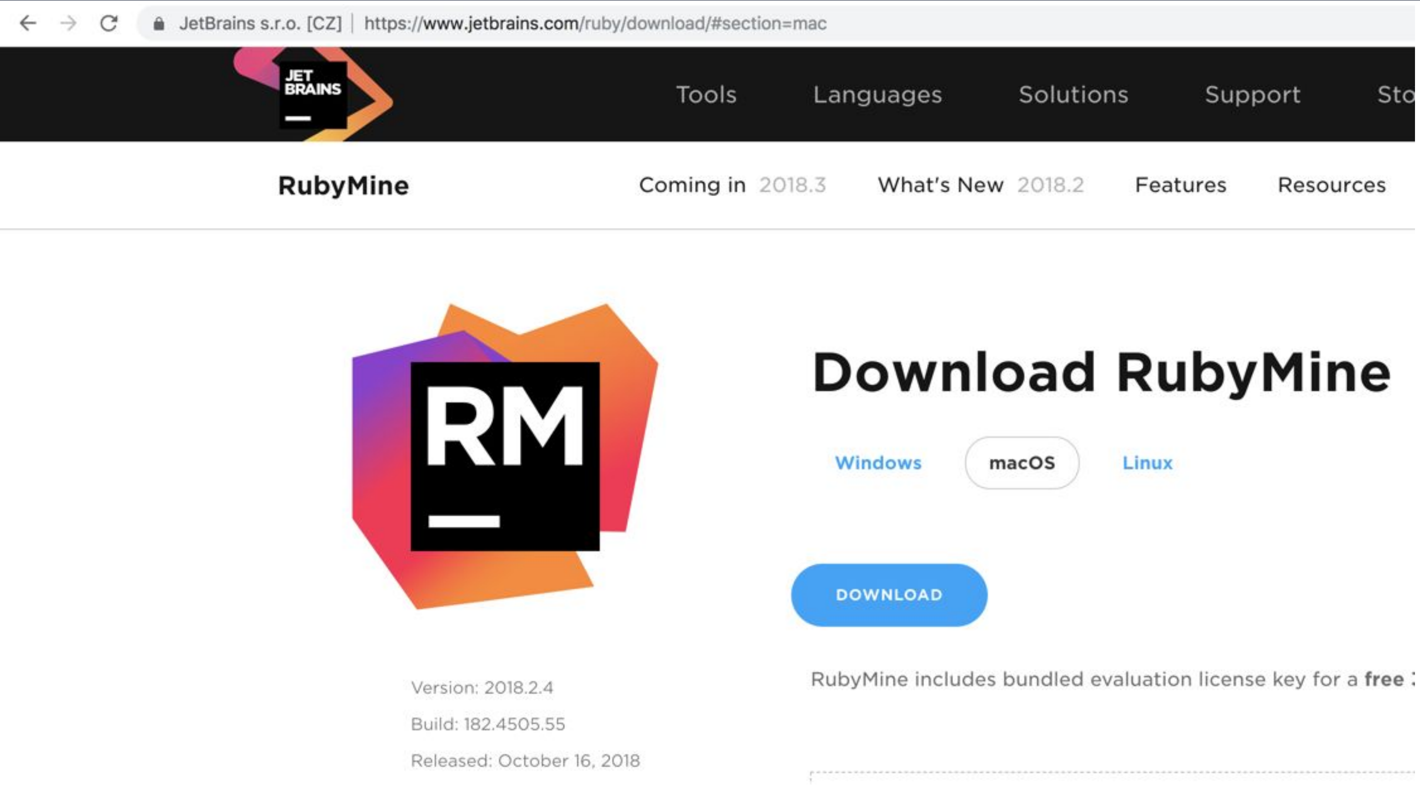
RubyInstaller-2.5.3-1 and RubyInstaller-2.4.5-1 are released. These are maintenance releases with bug and security fixes. See ruby-2.4.5 release notes and ruby-2.5.2 release notes. Bundled OpenSSL versions are updated to 1.0.2p and 1.1.1.

2018-10-21 Read full article

RubyInstaller 2.5.1-2 and 2.4.4-2 released

- Download with DevKit (Link)
- Ensure checkbox for PATH update is enabled.

Prep: Install Rubymine (Optional) Otherwise, any editor



Download RubyMine

RubyMine includes bundled evaluation license key for a **free**

Version: 2018.2.4

Build: 182.4505.55

Released: October 16, 2018

FYI: Running Examples

```
ruby  -I <lib path>   <ruby program>
```

Examples:

```
ruby -I ./    ex.rb

ruby -I ./lib   ex.rb
```

Basics for OO

- Program to an Interface (not the Implementation)

Basics for OO - Program to an Interface

- Program to an Interface (not the Implementation)

    - **Scalability**
    - Less brittleness
    - Loosely Coupled Systems

    *Less dependencies on external systems

Basics for OO - Program to an Interface

- Programming to an Interface (not the Implementation)
    - Demo

      /qe-design-patterns/Program-to-Interface/bash-example

Basics for OO - Program to an Interface (ex1)

- Programming to an Interface (not the Implementation)

/qe-design-patterns/Program-to-Interface/bash-example/ex1

  - We know: Multiple automated tests, however they have different test reports

  - We need: a solution to provide uniform reports regardless of prog. language, OS, framework, ...

Basics for OO – Program to an Interface (ex1)

- Programming to an Interface (not the Implementation)

  /qe-design-patterns/Program-to-Interface/bash-example/ex2

  - **genReport()**
    - **BASH - Leverage Exit Code**
    - **Python - Pytest outputs results into JUNIT**
    - **Any JUNIT (Surefire XML format) - qe-reporter.py**

  - **Ability to change test report in one place**

    ./test-regression.sh

Basics for OO - Program to an Interface (ex2)

- Programming to an Interface (not the Implementation)

  /qe-design-patterns/Program-to-Interface/bash-example/ex2

  - We know: Multiple automated tests, however they all report to an API (genReport) .. passing args.

  - We have: Uniform test reports.
    - Test Output
    - Exit codes

Basics for OO - Program to an Interface (ex2)

- Programming to an Interface (not the Implementation)

  /qe-design-patterns/Program-to-Interface/bash-example/ex3

  - Easy to make updates with minimal impact!

Object Oriented Programming (OOP)

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Object Oriented Programming (OOP) Abstraction (1)

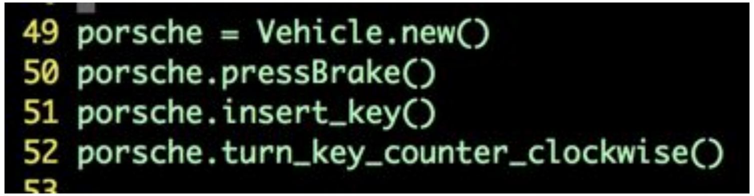
- Conceptualizing a model of the problem
- Breaking down, with modularity, the problem, into simple and clean manageable parts.
- Hide the unnecessary complexity of the details

Object Oriented Programming (OOP) Abstraction (2)

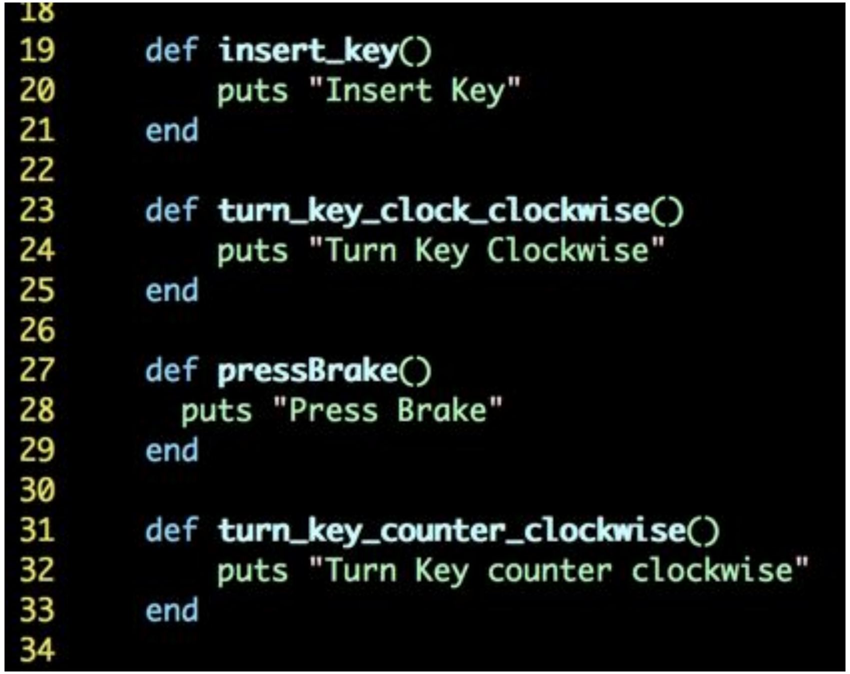
- Java programmers may get confused between “Abstract class” with OOP Abstraction
  - In Java, “Abstract class” is a class that can’t be instantiated.

  - Regarding OOP, it’s a mindset in understanding and breaking down the problem - how things should work and be put together

Object Oriented Programming (OOP) Abstraction - /oop/abstraction/ex1



```
porsche = Vehicle.new()
porsche.pressBrake()
porsche.insert_key()
porsche.turn_key_counter_clockwise()
```

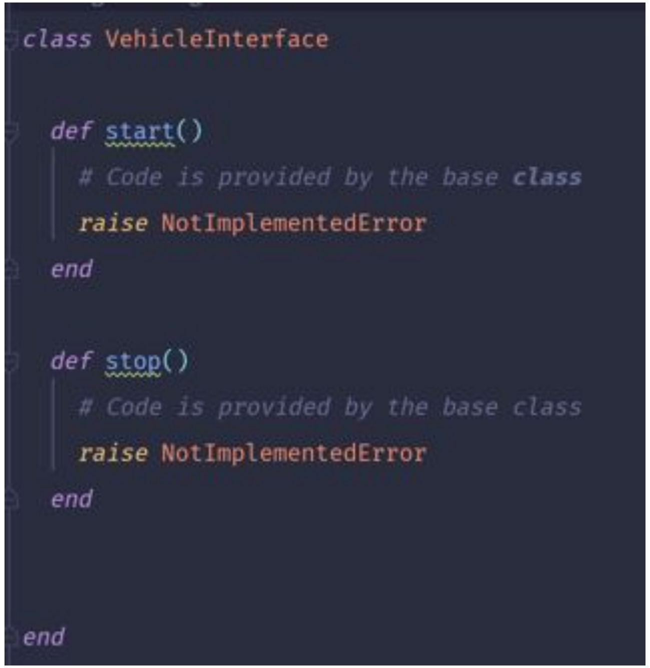


```

    def insert_key()
        puts "Insert Key"
    end

    def turn_key_clock_clockwise()
        puts "Turn Key Clockwise"
    end

    def pressBrake()
      puts "Press Brake"
    end

    def turn_key_counter_clockwise()
        puts "Turn Key counter clockwise"
    end

```

Object Oriented Programming (OOP) Abstraction – /oop/abstraction/ex2

Abstraction

Ahh .. programming to an interface



```
class VehicleInterface

  def start()
    # Code is provided by the base class
    raise NotImplementedError
  end

  def stop()
    # Code is provided by the base class
    raise NotImplementedError
  end


end
```

Object Oriented Programming (OOP) Abstraction – /oop/abstraction/ex2

DEMO

Object Oriented Programming (OOP) Encapsulation

- Hiding awareness and preventing access to a object’s assets that the user doesn’t need to know about ..

… this helps to prevent unwanted side-effects where data is accidently updated.

Object Oriented Programming (OOP) Encapsulation - /oop/encapsulation

DEMO

Object Oriented Programming::Inheritance

- Inheriting methods and properties (attributes) from another class(es)

Object Oriented Programming::Inheritance What most programmers think ..

- Equating inheritance as OOP.
- Typically “inheritance” is the big take-a-way per OOP
- “Inheritance” is like getting stuff for free, where you gain the advantages of accessing methods and behaviors for your purposes

Object Oriented Programming::Inheritance

- If it’s too good to be true, then … beware.
- Free things comes with hidden costs.
  - Comes with strings attached
  - Unintentional marriage between the superclass and the subclass

- Heavily leveraging inheritance is typically associated as OOP.
- Typically “inheritance” is the big take-away per OOP
- “Inheritance” is like getting stuff for free, where you gain the advantages of accessing methods and behaviors for your purposes

Object Oriented Programming::Inheritance

- <u>Unintentional</u> marriage between superclass and subclass
  - Once a class is subclassed, you now have two classes that are “bound” or “<u>tightly coupled</u>”.

    A bad marriage where you simply can’t escape your new family and their entire ancestry.

    - Any ill behavior from the superclass, whether existing or new, is passed to the new subclass
    - Any secrets that are not properly hidden in the superclass, are now available for abuse by the new subclass.
    - A single change, made to the superclass, can be detrimental, or fatal, to the innocent subclass
    - If the superclass has multitudes of subclasses, you could be setting yourself up for a big catastrophe .. refactoring.

Object Oriented Programming::Inheritance

We should probably rely less on inheritance.

Object Oriented Programming::Inheritance Demo - /oop/inheritance/

DEMO

qe-design-patterns/oop/inheritance

Object Oriented Programming::Inheritance Demo – /oop/inheritance/

DEMO

qe-design-patterns/oop/inheritance

qe-design-patterns/oop/inheritance/ex2
***(inheritance / polymorphism)**

Object Oriented Programming::Polymorphism

- Creating objects that can take on forms of multiple objects (static and at runtime).

Object Oriented Programming::Polymorphism Java example

```
// homepage.java
class BasePage {

    public WebDriver driver;


    public void click() {
        ….
    }


}
```

```
// homepage.java
class HomePage extends BasePage {

    // Home page code
    public void loadPage() { .. }
}
```

```
// searchpage.java
class SearchPage extends BasePage {

   // Search page code
   public void loadPage() { … }

   public void search(text) { … }
}
```

Object Oriented Programming::Polymorphism Java example

```
...
pg = new BasePage();
homePg = new HomePage();
searchPg = new SearchPage();


// variable ‘pg’ can be used to for any subclass of BasePage.

pg = homePg
pg.loadPage();

Pg = searchPg;
pg.search(“Elvis”);
```

Object Oriented Programming::Polymorphism Example

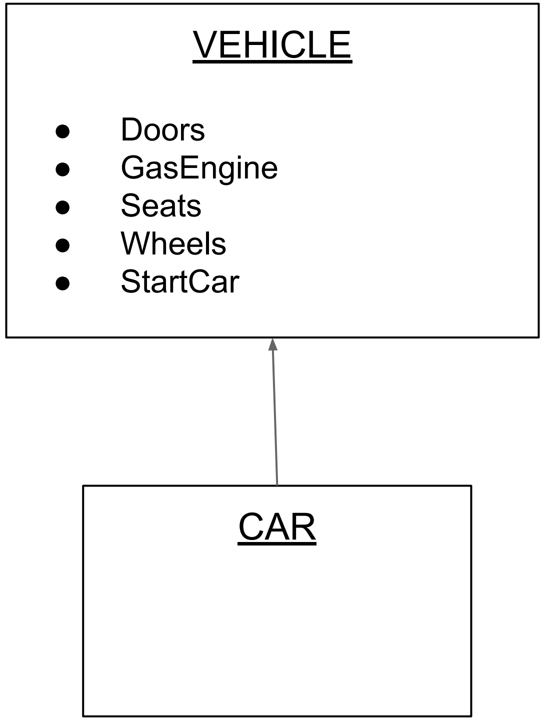
DEMO

qe-design-patterns/oop/polymorphism

Composition: “is a ..” vs. “has a ..”

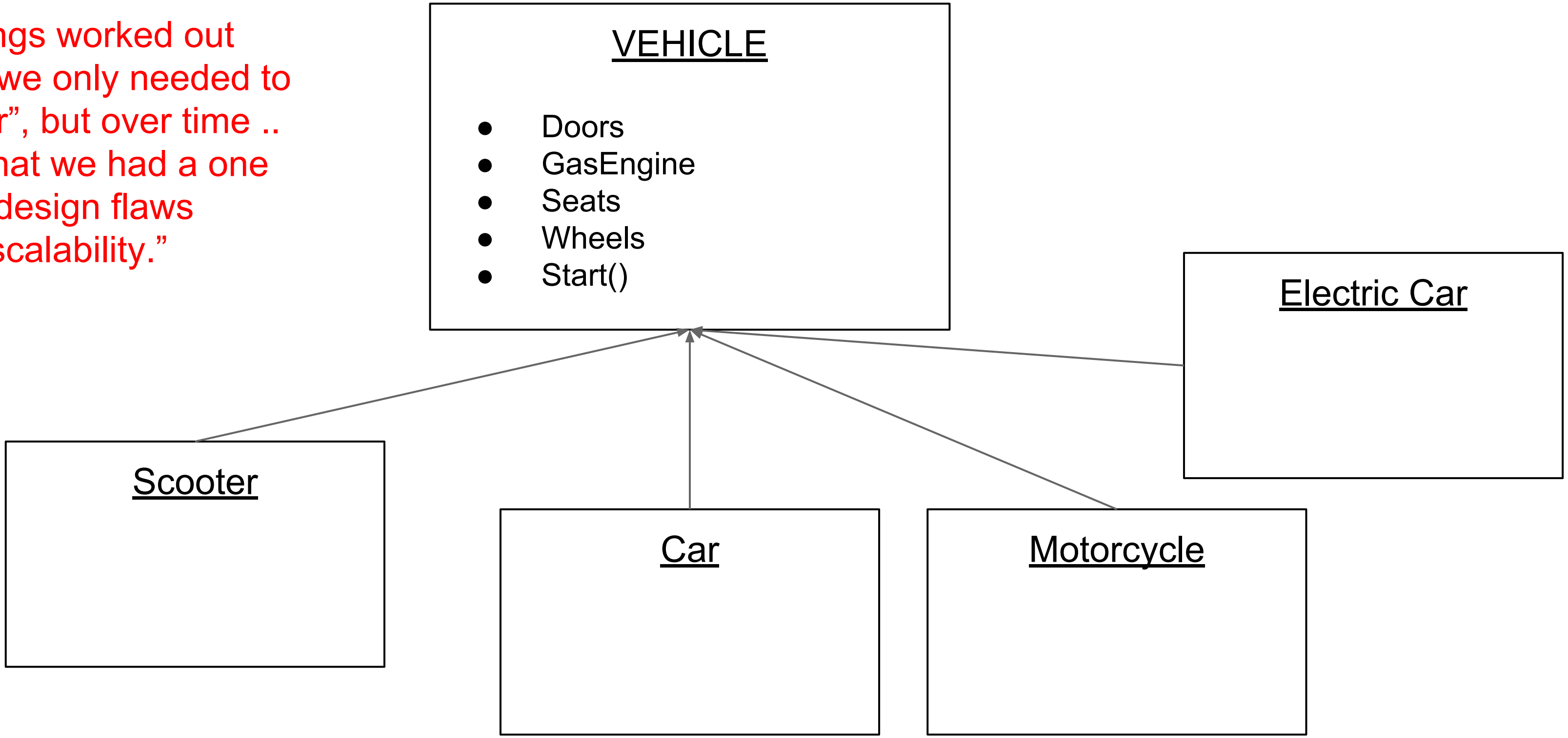
- Design classes that are assembled based on “here’s what I need”.
- Building classes from the “bottom-up”
- Minimize overhead and side-effects (e.g. avoid dependencies on a superclass)
- Scalable design
    - Easier to add/remove behaviors without affecting other classes

Tightly Coupled Systems

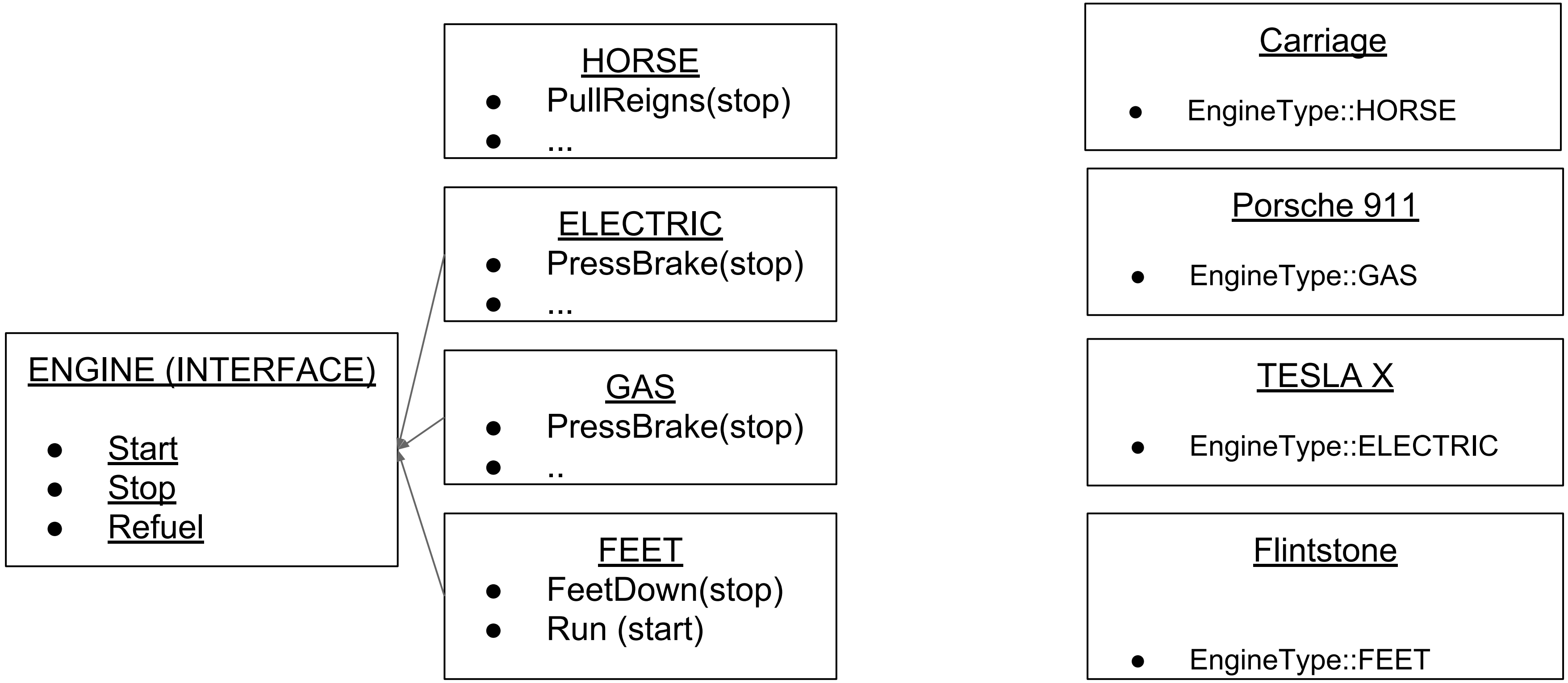


Tightly Coupled Systems

“Initially, things worked out great when we only needed to support “Car”, but over time .. we realize that we had a one trick pony - design flaws prevented scalability.”



Loosely Coupled Systems - Simplicity with Scalability (Abstraction / Composition)



Loosely Coupled Systems - Simplicity with Scalability (Abstraction / Composition)

DEMO

/qe-design-patterns/principles/composition/ex1.rb

qe-design-patterns/Composite/composition_ex.rb

Good Programming is Gangsta

- GoF (Gang of Four) - 1994
    - 23 “Classic” Design Patterns
    - Goal is to build clean, well-designed object-oriented programs.

- Design Patterns are used everywhere
    - Real-time firmware (microcode)
    - Large scale real-time systems
    - Enterprise software
    - Video games

Programming Principles

- Separate out the things that change from those that stay the same.

  Things unfortunately change .. so does our user stories, requirements, and unexpected defects. These are better mitigated with a design that isolates things that don't change from things that change.

  Objective is to minimize any negative impacts where those areas that do change have little harm to those that don't change.

Programming Principles

- Separate out the things that change from those that stay the same.
- YAGNI

Programming Principles

- Separate out the things that change from those that stay the same.
- **YAGNI**

  You Ain't Gonna Need It.

  Why add code when it's most likely that it won't be used?

  Why implement features that have an inflexible design?

  Leverage a design that focuses on what's needed now, while building "in" the flexibility that you'll need in the future.

Programming Principles

- Separate out the things that change from those that stay the same.
- YAGNI
- **Program to an Interface, not an Implementation**

Programming Principles

- Separate out the things that change from those that stay the same.
- YAGNI

- **“Program to an Interface, not an Implementation”**

  Loosely coupled systems.

  The “interface” tells others what they can “do”.

Programming Principles

- Separate out the things that change from those that stay the same.
- YAGNI
- **“Program to an Interface, not an Implementation”**
- **Composition over Inheritance**

Programming Principles

- Separate out the things that change from those that stay the same.
- YAGNI
- **“Program to an Interface, not an Implementation”**
- Composition over inheritance
- **Delegation**

  Pass the along the responsibility to the one who’s really accountable (responsible)

Programming Principles

- Separate out the things that change from those that stay the same.
- YAGNI
- **“Program to an Interface, not an Implementation”**
- Composition over inheritance
- **Delegation**
- **Memoization**

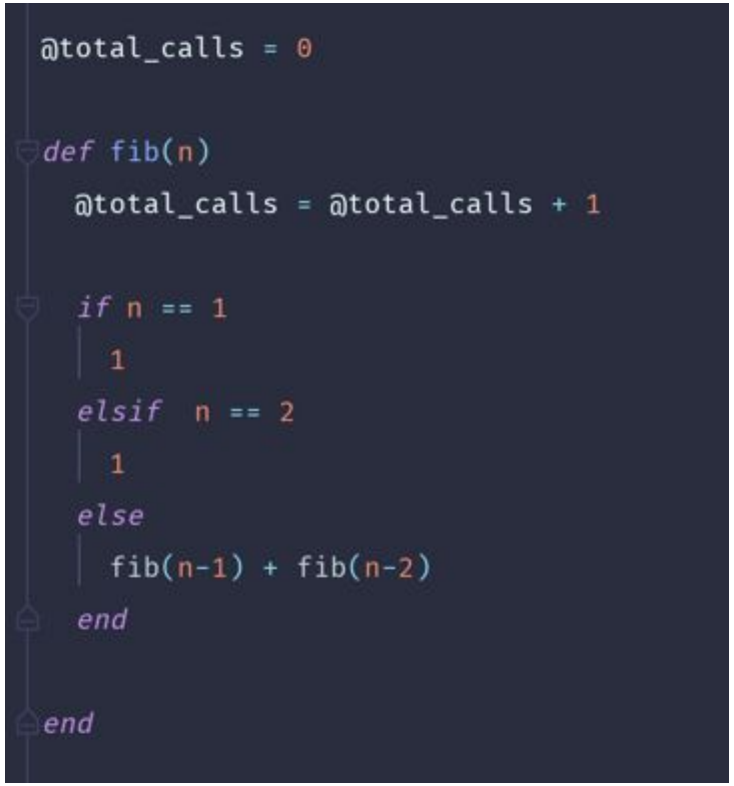
  Remember previous results for performant processing.

Memoisation

- Improved performance
- Example with fibonacc

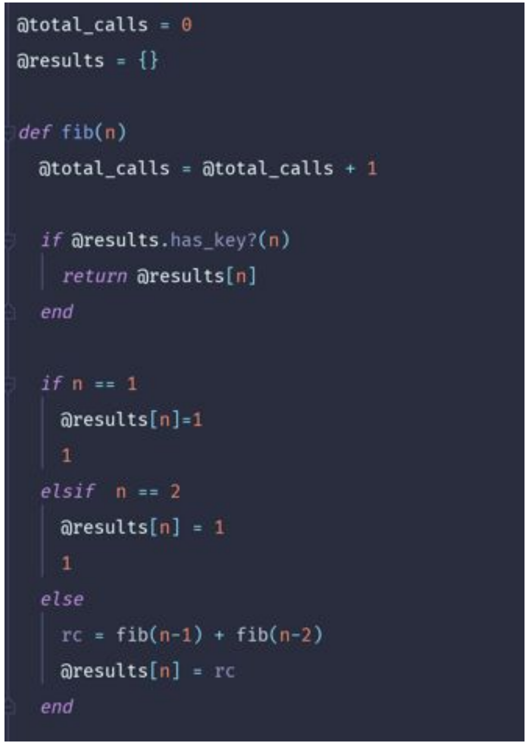
./principles/memoisation/

Memoisation::/principles/memoisation/fib1.rb



```ruby
@total_calls = 0

def fib(n)
  @total_calls = @total_calls + 1

  if n == 1
    1
  elsif  n == 2
    1
  else
    fib(n-1) + fib(n-2)
  end

end
```

Memoisation::/principles/memoisation/fib2.rb



```ruby
@total_calls = 0
@results = {}

def fib(n)
  @total_calls = @total_calls + 1

  if @results.has_key?(n)
    return @results[n]
  end

  if n == 1
    @results[n]=1
    1
  elsif  n == 2
    @results[n] = 1
    1
  else
    rc = fib(n-1) + fib(n-2)
    @results[n] = rc
  end
```

Memoisation::/principles/memoisation/fib2.rb

DEMO

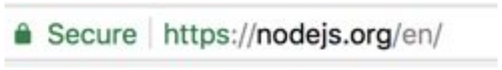
What are Design Patterns?

- Proven solutions to common design problems
- GoF - thanks!

Why do you need to know them?

- Leverage a proven solution

How are they used?



Secure | https://nodejs.org/en/

Singleton – There can only be one.

Problem:

You need access to data/methods “everywhere” in your codebase - almost like a global variable. This means side effects due to global scoping and challenges to passing around a global object.

Solution:

Simple and clean design to manage global access to only one object.

Singleton - There can only be one.

Example: You need to manage multiple browsers.

Solution:

Design a single “browser manager” that manages any creation of browsers and accessing them.

Singleton – There can only be one.

DEMO

/qe-design-patterns/Singleton

Composite

Problem:

Managing tasks and/or objects that are built on other tasks and/or objects.

Solution:

GoF - “... the sum acts like one of the parts”.

Composite

Example:

Managing components (pageObjects) to ensure scalability.

Solution:

Pages are composed of other pages and/or components.

Composite

DEMO

Template

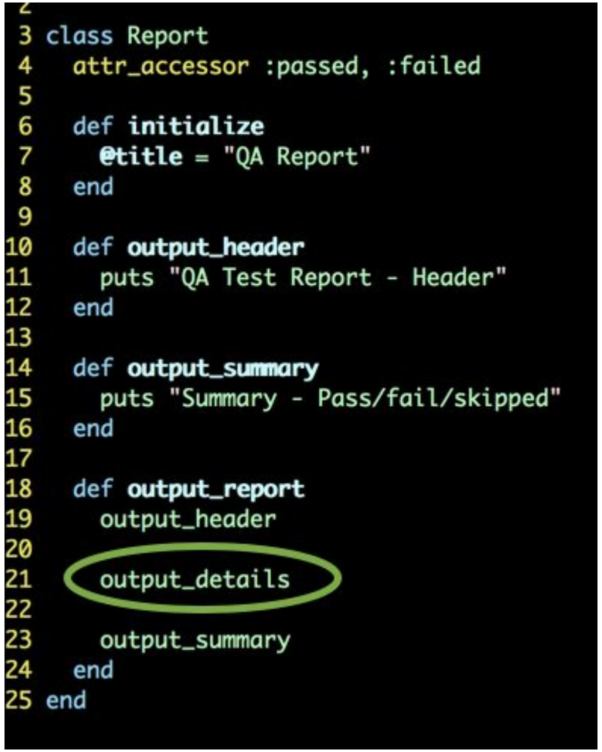
Problem:

You have a series of steps, however you need to vary one of those steps.

Template

DEMO

/qe-design-patterns/Template/



```
class Report
  attr_accessor :passed, :failed

  def initialize
    @title = "QA Report"
  end

  def output_header
    puts "QA Test Report - Header"
  end

  def output_summary
    puts "Summary - Pass/fail/skipped"
  end

  def output_report
    output_header

    output_details

    output_summary
  end
end
```

Strategy

Problem: The algorithm needs to change during runtime.

Solution: Separate out those algorithms into their own class.

Strategy

DEMO

/qe-design-patterns/Strategy/ruby -I ./ **strategy_drv.rb**

Command

Problem: The complexity involved with managing “actions” is getting out of control.

Solution: Manage “actions” as objects, where they can be created, customized, and executed with a simple interface.

Command

DEMO

/qe-design-patterns/command/ex1
/qe-design-patterns/command/ex2

STP SOFTWARE TEST PROFESSIONALS CONFERENCE SPRING 2019

Peter Kim
LinkedIn: peterkim777
Twitter: peter_kim777
Email: h20dragon@outlook.com

dcast.io