Sharing behavior

Well defined objects

==

Collaboration

Coupling

Coupling is the degree to which each program module relies on each one of the other modules.

Coupling

When you change one object you are forced to change the ones it is coupled to.

```ruby
class Car
  def initialize(sound)
    @sound = sound
    @cities = []
  end

  def cities
    @cities
  end
end

first_car = Car.new("Broom")
first_car.cities.push "Gijón"
puts first_car.cities
```

Trainwreck

Chaining a ton of methods can lead to a maintenance hell

```
client.mortgage.payments.next_payment.apply(300.00)
```

```ruby
class Car
  def initialize(sound)
    @sound = sound
    @cities = []
  end

  def cities
    @cities
  end
end

first_car = Car.new("Broom")
first_car.cities.push "Gijón"
puts first_car.cities
```

Cohesion

The degree to which the elements of a module belong together

Types of relationships

- Inheritance
- Mixins
- Composition / Collaboration
- Duck Types

Inheritance

It allows you to create a class that is a refinement or a specialisation of another.

```
class KaraokeSong < Song
```

```ruby
class Programmer

  def program
    consume_caffeine
    do_wonderful_things_with_computers
  end

  def consume_caffeine
    #[...]
  end

  def do_wonderful_things_with_computers
    #[...]
  end
end
```

```ruby
class Designer

  def design_things
    consume_caffeine
    select_typography
    select_colors
    #[...]
  end

  def consume_caffeine
    #[...]
  end

  def select_typography
    #[...]
  end

  def select_colors
    #[...]
  end
end
```

```ruby
class CaffeineConsumer
  def consume_caffeine
    #[...]
  end
end
```

```ruby
class Programmer < CaffeineConsumer

  def program
    consume_caffeine
    do_wonderful_things_with_computers
  end

  def do_wonderful_things_with_computers
    #[...]
  end
end
```

```ruby
class Designer < CaffeineConsumer

  def design_things
    consume_caffeine
    select_typography
    select_colors
    #[...]
  end

  def select_typography
    #[...]
  end

  def select_colors
    #[...]
  end
end
```

IRON HACK

```ruby
class Programmer < CaffeineConsumer


  def program
    consume_caffeine
    do_wonderful_things_with_computers
    receive_salary
  end


  def do_wonderful_things_with_computers
    #[...]
  end


  def receive_salary
    #[...]
  end
end
```

```ruby
class Designer < CaffeineConsumer


  def design_things
    consume_caffeine
    select_typography
    select_colors
    receive_salary
  end


  def select_typography
    #[...]
  end


  def select_colors
    #[...]
  end


  def receive_salary
    #[...]
  end
end
```

```ruby
class Programmer < CaffeineConsumer


  def program
    consume_caffeine
    do_wonderful_things_with_computers
    receive_salary
  end


  def do_wonderful_things_with_computers
    #[...]
  end


  def receive_salary
    #[...]
  end
end
```

```ruby
class Designer < CaffeineConsumer


  def design_things
    consume_caffeine
    select_typography
    select_colors
    receive_salary
  end


  def select_typography
    #[...]
  end


  def select_colors
    #[...]
  end


  def receive_salary
    #[...]
  end
end
```

```ruby
class SalaryReceiver
  def receive_salary
    #[...]
  end
end
```

Multiple inheritance

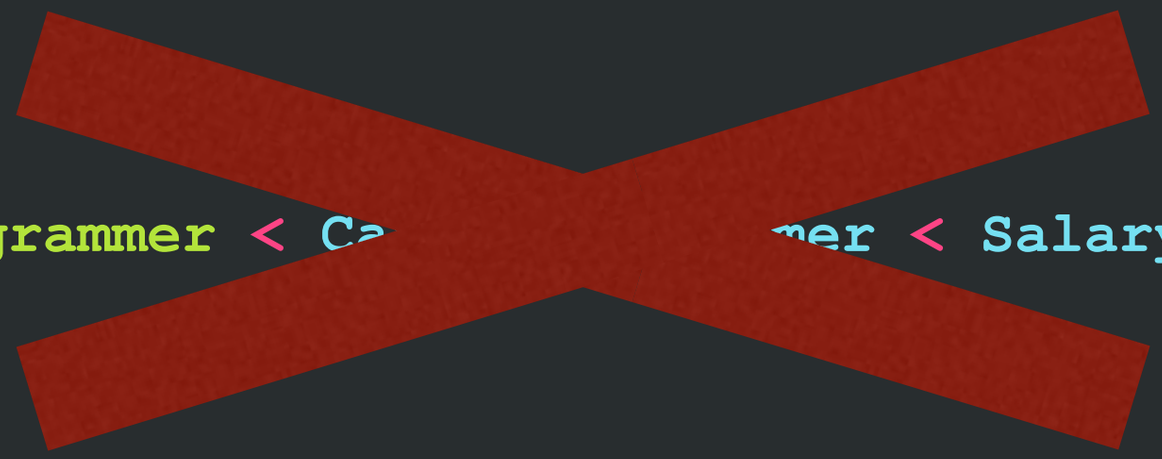
Well what if we just had `Programmer` inherit from both?

```
class Programmer < CaffeineConsumer < SalaryReceiver
```

Multiple inheritance

There is no multiple inheritance in Ruby!

```
class Programmer < Ca                mer < SalaryReceiver
```



Mixin

Provides the ability of mixing methods in classes *

Take the company owner

```
class CompanyOwner < CaffeineConsumer
  def look_busy
    consume_caffeine
  end
end
```

They consume caffeine and look busy

```
class CompanyOwner < CaffeineConsumer
  def look_busy
    consume_caffeine
  end
end
```

They don't receive a salary, though!

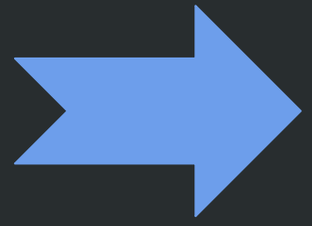
```
class CompanyOwner < CaffeineConsumer
  def look_busy
    consume_caffeine
  end
end
```

If we had a `Payable` module for our mixin

```
module Payable
  def receive_salary
    #[...]
  end
end
```

We can include it in the classes that need it.

```ruby
module Payable
  def receive_salary
    #[...]
  end
end
```

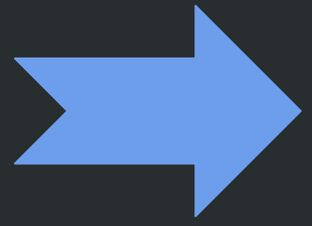


```ruby
class Programmer < CaffeineConsumer
  include Payable
```

```ruby
class Designer < CaffeineConsumer
  include Payable
```

And leave it out of the classes that don't.

```ruby
module Payable
  def receive_salary
    #[...]
  end
end
```



```ruby
class Programmer < CaffeineConsumer
  include Payable
```

✓

```ruby
class Designer < CaffeineConsumer
  include Payable
```

✓

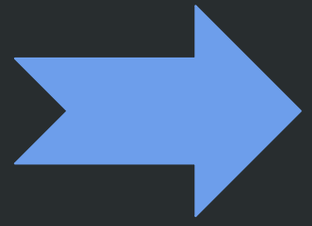
```ruby
class CompanyOwner < CaffeineConsumer
```

✖

IRON HACK

We still have our inheritance too.

```ruby
module Payable
  def receive_salary
    #[...]
  end
end
```



```ruby
class Programmer < CaffeineConsumer
  include Payable
```

✓

```ruby
class Designer < CaffeineConsumer
  include Payable
```

✓

```ruby
class CompanyOwner < CaffeineConsumer
```

✖

You can inherit only once but include many modules.

```
class Programmer < CaffeineConsumer
  include Payable
  include Vacationable
  include Insurable
  include Beerable
```

Exercise

Refactor your payroll classes to use modules.
Choose one:

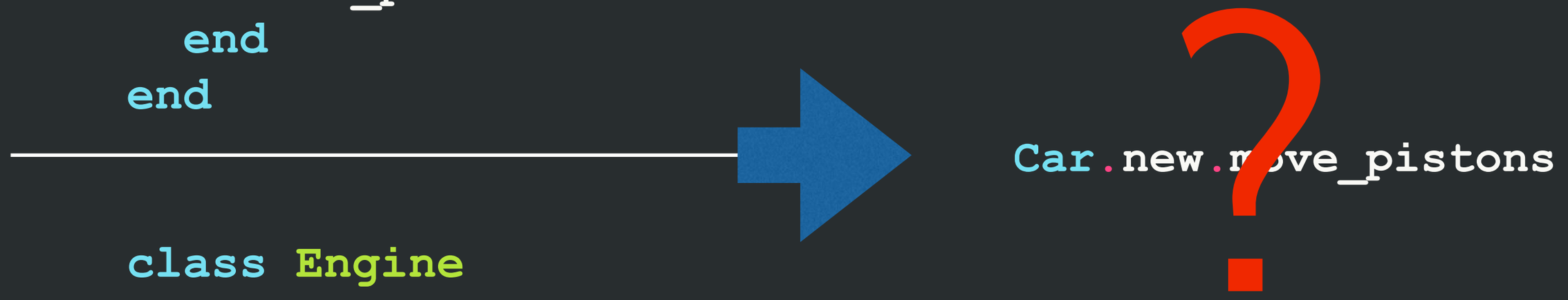
Create a module for hourly pay.
OR create a module for salaried pay.

Include your module in the two classes that need it.

Sharing methods using inheritance

```ruby
class Car < Engine
  def start
    move_pistons
  end
end
```

```ruby
class Engine
  def move_pistons
    "Pshhhhh"
  end
end
```



Exercise

```
class Car < Engine
  def start
    move_pistons
  end
end


class Engine
  def move_pistons
    "Pshhhhh"
  end
end
```

Can you loosen the coupling a little bit?

Exercise

```ruby
class Car < Engine
  def start
    move_pistons
  end
end


class Engine
  def move_pistons
    "Pshhhhh"
  end
end
```

Is a `Car` a more specialized `Engine`?

How should `Engine` share its methods?

```ruby
class Car
  def start
    Engine.new.move_pistons
  end
end
```

```ruby
class Engine
  def move_pistons
    "Pshhhhh"
  end
end
```

Composition

When an object achieves its behaviour by containing another object

```ruby
class Car
  def initialize
    @engine = Engine.new
  end

  def start
    @engine.move_pistons
  end
end
```

Dependency injection

An injection is the passing of a dependency (an object) to a dependent object (a client). The object is made part of the client's state.

Dependency injection

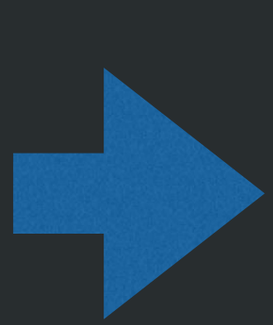
```ruby
class Car
  def initialize(engine)
    @engine = engine
  end

  # ...
end
```

Dependency injection

```ruby
class Car
  def start
    Engine.new.move_pistons
  end
end

class Engine
  def move_pistons
    "Pshhhhh"
  end
end
```



```ruby
class Car
  def initialize(engine)
    @engine = engine
  end

  def start
    @engine.move_pistons
  end
end

class Engine
  def move_pistons
    "Pshhhhh"
  end
end
```

Exercise

Create Car and Engine classes.
Both of them make their own noises.

When Car makes noise the output is the noise of the Car plus the noise of the engine.

Create different types of engines that should work with any car.

Prefer composition over inheritance

Duck Types

Duck type

```
class Duck
  def walk
    "Like a duck"
  end
  def do_quack
    "Quack"
  end
end
```

If it walks like a duck

And does quack

It’s a duck!!!!

Even though the objects are not exactly equal, they share a common implicit interface

```ruby
class Duck
  def walk
    "Like a duck"
  end
  def do_quack
    "Quack"
  end
end
```

```ruby
class DuckCosplayer
  def walk
    "Like a human duck"
  end
  def do_quack
    "Quackity"
  end
end
```

We don’t care so much about classes, but about the capabilities of objects.

```ruby
ducks = [Duck.new, DuckCosplayer.new]

ducks.each do |duck|
  puts duck.do_quack
end
```

Exercise

Create different vehicle classes. A vehicle has a number of wheels and makes some noise.

Create a class that counts the total number of wheels of the vehicles inside of a given array and another class that prints all the different noises that vehicles in an array make.

You can’t use inheritance.

Choosing a relationship

- Use inheritance for is-a relationships
- Use mixins for is-able-to relationships
- Use duck types for behaves-like-a relationships
- Use composition for has-a relationships

*Modules

Namespacing with modules

```ruby
module Accounting
  class Person
    # ...
  end
end

module CRM
  class Person
    # ...
  end
end
```

Include vs Extends

- Include mixes methods into instances of the base object

- Extends mixes methods into the base object itself

```ruby
module Greetable
  def hi
    puts "Hello"
  end
end
```

```ruby
class Included
  include Greetable
end

Included.new.hi
```

```ruby
class Extended
  extend Greetable
end

Extended.hi

my_obj = Object.new
my_obj.extend(Greetable)
my_obj.hi
```