

attr_extras

Takes some boilerplate out of Ruby, lowering the barrier to extracting small focused classes, without the downsides of using `Struct`.

Provides lower-level methods like `attr_private` and `attr_value` that nicely complement Ruby's built-in `attr_accessor`, `attr_reader` and `attr_writer`.

Also higher-level ones like `pattr_initialize` (or `attr_private_initialize`) and `method_object` to really cut down on the boilerplate.

Instead of

```
1 class InvoicePolicy
2   def initialize(invoice, company:)
3     @invoice = invoice
4     @company = company
5   end
6
7   def payable?
8     some_logic(invoice, company)
9   end
10
11  private
12
13  attr_reader :invoice, :company
14 end
```

you can just do

```
1 class InvoiceBuilder
2   pattr_initialize :invoice, [:company!]
3
4   def payable?
5     some_logic(invoice, company)
6   end
7 end
```

And instead of

```
1 class PayInvoice
2   def self.call(invoice, amount)
3     new(invoice, amount).call
4   end
5
```

```
6  def initialize(invoice, amount)
7    @invoice = invoice
8    @amount = amount
9  end
10
11  def call
12    PaymentGateway.charge(invoice.id, amount_in_cents)
13  end
14
15  private
16
17  def amount_in_cents
18    amount * 100
19  end
20
21  attr_reader :invoice, :amount
22 end
```

you can just do

```
1  class PayInvoice
2    method_object :invoice, :amount
3
4    def call
5      PaymentGateway.charge(invoice.id, amount_in_cents)
6    end
7
8    private
9
10   def amount_in_cents
11     amount * 100
12   end
13 end
```

Supports positional arguments as well as optional and required keyword arguments.

Also provides conveniences for creating value objects, method objects, query methods and abstract methods.

Usage

- `attr_initialize`
- `attr_private`
- `attr_value`
- `pattr_initialize/attr_private_initialize`
- `vattr_initialize/attr_value_initialize`
- `rattr_initialize/attr_reader_initialize`

-
- `attr_initialize/attr_accessor_initialize`
 - `static_facade`
 - `method_object`
 - `attr_implement`
 - `cattr_implement`
 - `attr_query`
 - `attr_id_query`

attr_initialize

`attr_initialize :foo, :bar` defines an initializer that takes two arguments and assigns `@foo` and `@bar`.

`attr_initialize :foo, [:bar, :baz!]` defines an initializer that takes one regular argument, assigning `@foo`, and two keyword arguments, assigning `@bar` (optional) and `@baz` (required).

`attr_initialize [:bar, :baz!]` defines an initializer that takes two keyword arguments, assigning `@bar` (optional) and `@baz` (required).

If you pass unknown keyword arguments, you will get an `ArgumentError`. If you don't pass required arguments and don't define default value for them, you will get a `KeyError`.

`attr_initialize` can also accept a block which will be invoked after initialization. This is useful for e.g. initializing private data as necessary.

Default values Keyword arguments can have default values:

`attr_initialize [:bar, baz: "default value"]` defines an initializer that takes two keyword arguments, assigning `@bar` (optional) and `@baz` (optional with default value `"default value"`).

Note that default values are evaluated *when the class is loaded* and not on every instantiation. So `attr_initialize [time: Time.now]` might not do what you expect.

You can always use regular Ruby methods to achieve this:

```
1 class Foo
2   attr_initialize [:time]
3
4   private
5
6   def time
7     @time ||= Time.now
```

```
8   end
9   end
```

Or just use a regular initializer with default values.

attr_private

`attr_private :foo, :bar` defines private readers for `@foo` and `@bar`.

attr_value

`attr_value :foo, :bar` defines public readers for `@foo` and `@bar` and also defines object equality: two value objects of the same class with the same values will be considered equal (with `==` and `eq?`, in `Sets`, as `Hash` keys etc).

It does not define writers, because value objects are typically immutable.

attr_initialize

attr_private_initialize

`attr_initialize :foo, :bar` defines both initializer and private readers. Shortcut for:

```
1 attr_initialize :foo, :bar
2 attr_private :foo, :bar
```

`attr_initialize` is aliased as `attr_private_initialize` if you prefer a longer but clearer name.

Example:

```
1 class Item
2   attr_initialize :name, :price
3
4   def price_with_vat
5     price * 1.25
6   end
7 end
8
9 Item.new("Pug", 100).price_with_vat # => 125.0
```

The `attr_initialize` notation for keyword arguments is also supported: `attr_initialize :foo, [:bar, :baz!]`

vattr_initialize

attr_value_initialize

`vattr_initialize :foo, :bar` defines initializer, public readers and value object identity. Shortcut for:

```
1 attr_initialize :foo, :bar
2 attr_value :foo, :bar
```

`vattr_initialize` is aliased as `attr_value_initialize` if you prefer a longer but clearer name.

Example:

```
1 class Country
2   vattr_initialize :code
3 end
4
5 Country.new("SE") == Country.new("SE") # => true
6 Country.new("SE").code # => "SE"
```

The `attr_initialize` notation for keyword arguments is also supported: `vattr_initialize :foo, [:bar, :baz!]`

rattr_initialize

attr_reader_initialize

`rattr_initialize :foo, :bar` defines both initializer and public readers. Shortcut for:

```
1 attr_initialize :foo, :bar
2 attr_reader :foo, :bar
```

`rattr_initialize` is aliased as `attr_reader_initialize` if you prefer a longer but clearer name.

Example:

```
1 class PublishBook
2   rattr_initialize :book_name, :publisher_backend
3
4   def call
5     publisher_backend.publish book_name
6   end
7 end
8
```

```
9 service = PublishBook.new("A Novel", publisher)
10 service.book_name # => "A Novel"
```

The `attr_initialize` notation for keyword arguments is also supported: `rattr_initialize :foo, [:bar, :baz!]`

attr_initialize

attr_accessor_initialize

`attr_initialize :foo, :bar` defines an initializer, public readers, and public writers. It's a shortcut for:

```
1 attr_initialize :foo, :bar
2 attr_accessor :foo, :bar
```

`attr_initialize` is aliased as `attr_accessor_initialize`, if you prefer a longer but clearer name.

Example:

```
1 class Client
2   attr_initialize :username, :access_token
3 end
4
5 client = Client.new("barsoom", "SECRET")
6 client.username # => "barsoom"
7
8 client.access_token = "NEW_SECRET"
9 client.access_token # => "NEW_SECRET"
```

The `attr_initialize` notation for keyword arguments and blocks is also supported.

static_facade

`static_facade :allow?, :user` defines an `.allow?` class method that delegates to an instance method by the same name, having first provided `user` as a private reader.

This is handy when a class-method API makes sense but you still want the refactorability of instance methods.

Example:

```
1 class PublishingPolicy
2   static_facade :allow?, :user
```

```
3
4  def allow?
5    user.admin? && complicated_extracted_method
6  end
7
8  private
9
10  def complicated_extracted_method
11    # ...
12  end
13 end
14
15 PublishingPolicy.allow?(user)
```

`static_facade :allow?, :user` is a shortcut for

```
1 attr_initialize :user
2
3 def self.allow?(user)
4   new(user).allow?
5 end
```

The `attr_initialize` notation for keyword arguments is also supported: `static_facade :allow?, :user, [:user_agent, :ip!]`

You don't have to specify arguments/readers if you don't want them: just `static_facade :tuesday?` is also valid.

You can specify multiple method names as long as they can share the same initializer arguments: `static_facade [:allow?, :deny?], :user, [:user_agent, :ip!]`

Any block given to the class method will be passed on to the instance method.

“Static façade” is the least bad name for this pattern we’ve come up with. Suggestions are welcome.

method_object

NOTE: v4.0.0 made a breaking change! `static_facade` does exactly what `method_object` used to do; the new `method_object` no longer accepts a method name argument.

`method_object :foo` defines a `.call` class method that delegates to an instance method by the same name, having first provided `foo` as a private reader.

This is a special case of `static_facade` for when you want a Method Object, and the class name itself will communicate the action it performs.

Example:

```
1 class CalculatePrice
2   method_object :order
3
4   def call
5     total * factor
6   end
7
8   private
9
10  def total
11    order.items.map(&:price).inject(:+)
12  end
13
14  def factor
15    1 + rand
16  end
17 end
18
19 class Order
20   def price
21     CalculatePrice.call(self)
22   end
23 end
```

You could even do `CalculatePrice.(self)` if you like, since we're using the `call` convention.

`method_object :foo` is a shortcut for

```
1 static_facade :call, :foo
```

which is a shortcut for

```
1 attr_initialize :foo
2
3 def self.call(foo)
4   new(foo).call
5 end
```

The `attr_initialize` notation for keyword arguments is also supported: `method_object :foo, [:bar, :baz!]`

You don't have to specify arguments/readers if you don't want them: just `method_object` is also valid.

Any block given to the class method will be passed on to the instance method.

attr_implement

`attr_implement :foo, :bar` defines nullary (0-argument) methods `foo` and `bar` that raise e.g. `"Implement a 'foo()' method"`.

`attr_implement :foo, [:name, :age]` will define a binary (2-argument) method `foo` that raises `"Implement a 'foo(name, age)' method"`.

This is suitable for abstract methods in base classes, e.g. when using the template method pattern.

It's more or less a shortcut for

```
1 def my_method
2   raise "Implement me in a subclass!"
3 end
```

though it is shorter, more declarative, gives you a clear message and handles edge cases you might not have thought about (see tests).

Note that you can also use this with modules, to effectively mix in interfaces:

```
1 module Bookable
2   attr_implement :book, [:bookable]
3   attr_implement :booked?
4 end
5
6 class Invoice
7   include Bookable
8 end
9
10 class Payment
11   include Bookable
12 end
```

cattr_implement

Like `attr_implement` but for class methods.

Example:

```
1 class TransportOrder
2   cattr_implement :must_be_tracked?
3 end
```

attr_query

`attr_query :foo?, :bar?` defines query methods like `foo?`, which is true if (and only if) `foo` is truthy.

attr_id_query

`attr_id_query :foo?, :bar?` defines query methods like `foo?`, which is true if (and only if) `foo_id` is truthy. Goes well with Active Record.

Explicit mode

By default, `attr_extras` will add methods to every class and module.

This is not ideal if you're using `attr_extras` in a library: those who depend on your library will get those methods as well.

It's also not obvious where the methods come from. You can be more explicit about it, and restrict where the methods are added, like this:

```
1 require "attr_extras/explicit"
2
3 class MyLib
4   extend AttrExtras.mixin
5
6   attr_initialize :now_this_class_can_use_attr_extras
7 end
```

Crucially, you need to `require "attr_extras/explicit"` instead of `require "attr_extras"`. Some frameworks, like Ruby on Rails, may automatically require everything in your `Gemfile`. You can avoid that with `gem "attr_extras", require: "attr_extras/explicit"`.

In explicit mode, you need to call `extend AttrExtras.mixin` in every class or module that wants the `attr_extras` methods.

Philosophy

Findability is a core value. Hence the long name `attr_initialize`, so you see it when scanning for the initializer; and the enforced questionmarks with `attr_id_query :foo?`, so you can search for that method.

Q & A

Why not use Struct instead of pattr_initialize?

See: “Struct inheritance is overused”

Why not use private; attr_reader :foo instead of attr_private :foo?

Other than being more to type, declaring `attr_reader` after `private` will actually give you a warning (deserved or not) if you run Ruby with warnings turned on.

If you don’t want the dependency on `attr_extras`, you can get rid of the warnings with `attr_reader :foo; private :foo`. Or just define a regular private method.

Can I use attr_extras in BasicObjects?

No, sorry. It depends on various methods that `BasicObjects` don’t have. Use a regular `Object` or make do without `attr_extras`.

Installation

Add this line to your application’s `Gemfile`:

```
1 gem "attr_extras"
```

And then execute:

```
1 bundle
```

Or install it yourself as:

```
1 gem install attr_extras
```

Running the tests

Run them with:

```
1 rake
```

Or to see warnings (try not to have any):

```
1 RUBYOPT=-w rake
```

You can run an individual test using the `m` gem:

```
1 m spec/attr_extras/attr_initialize_spec.rb:48
```

The tests are intentionally split into two test suites for reasons described in [Rakefile](#).

License

MIT