Getting loose coupling

Designing and Maintaining Software (DAMS)

Louis Rose

Recap: DIP

“High-level modules should not depend on low-level modules. Both should depend on abstractions.”

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

- Bob Martin

http://www.objectmentor.com/resources/articles/dip.pdf

Tactics

Inject dependencies to avoid coupling with specific implementations

Invert control to avoid coupling with implementations

“Tell, Don’t Ask” to avoid coupling with intermediaries

Introduce gateways to avoid coupling with external services

Inject Dependencies

```
class Pizza
  def initialize(temperature, duration)
    @temperature = temperature
    @duration = duration
  end

  def bake
    Oven.new(@temperature, @duration).cook(self)
  end
end

Pizza.new(200, 30).bake
```

Inject Dependencies

```ruby
class Pizza
  def initialize(temperature, duration)
    @temperature = temperature
    @duration = duration
  end

  def bake
    Oven.new(@temperature, @duration).cook(self)
  end
end

Pizza.new(200, 30).bake
```

Inject Dependencies

```ruby
class Pizza
  def initialize(oven)
    @oven = oven
  end

  def bake
    @oven.cook(self)
  end
end

Pizza.new(Oven.new(200, 30)).bake
Pizza.new(Barbeque.new(:smoky, 30)).bake
```

Invert Control

```
connection = Connection.new(@credentials)
connection.open
connection.send("foo")
connection.send("bar")
connection.flush
connection.close
```

Invert Control

```
connection = Connection.new(@credentials)
connection.open
connection.send(“foo")
connection.send("bar")
connection.flush
connection.close
```

Invert Control

```
Connection.perform(@credentials) do |connection|
  connection.send("foo")
  connection.send("bar")
end
```

Client code no longer specifies when it’s called
Hollywood Principle: “Don’t call us, we’ll call you”

Connection.perform?

```
class Connection
  def self.perform(@credentials)
    connection = Connection.new(@credentials)
    connection.open
    yield
    connection.flush
    connection.close
  end
end
```

Recap: Law of Demeter

"Talk only to your neighbours"

allowance	example
yourself	`self`
your own toys	`formatter`
toys you've been given	`printable`
toys you've made yourself	`out`

```
class Printer
  attr_reader :formatter

  def print(printable)
    out = TempFile.new
    ...
  end
end
```

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.wallet.withdraw(self.cost)
  end
end
```

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.wallet.withdraw(self.cost)
  end
end
```

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.wallet.withdraw(self.cost)
  end
end
```

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.wallet.withdraw(self.cost)
  end
end
```

Courier is now coupled to how a customer stores their money!

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.wallet.withdraw(self.cost)
  end
end
```

Courier is now coupled to how a customer stores their money!
But, really, a courier only cares that they get paid, not how.

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.request_payment(self.cost)
  end
end
```

Courier is now coupled to how a customer stores their money!
But, really, a courier only cares that they get paid, not how.

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.request_payment(self.cost)
  end
end

class Customer
  def request_payment(amount)
    @wallet.withdraw(amount)
  end
end
```

But, really, a courier only cares that they get paid, not how.

Tell, Don’t Ask

```
class Courier
  def deliver_to(customer, parcel)
    drive_to(customer.address)
    customer.give(parcel)
    customer.request_payment(self.cost)
  end
end

class Customer
  def request_payment(amount)
    @wallet.withdraw(amount)
  end
end
```

But, really, a courier only cares that they get paid, not how.

Introduce Gateways

```
class Order
  def bill
    braintree_id = Braintree.find_user(@user.email).braintree_id
    Braintree.charge(braintree_id, @amount)
  end
end

class Refund
  def run
    transaction_id = Braintree.find_transaction(@order.braintree_id)
    Braintree.refund(transaction_id, @amount)
  end
end
```

Order and Refund are coupled to this payment processor (Braintree).
What if we want to use a Braintree2 gem? Or switch to Stripe?

Introduce Gateways

```
class Order
  def bill
    user_id = PaymentGateway.find_user(@user.email).id
    PaymentGateway.charge(user_id, @amount)
  end
end

class Refund
  def run
    transaction_id = PaymentGateway.find_transaction(@order)
    PaymentGateway.refund(transaction_id, @amount)
  end
end
```

Introduce Gateways

```
class Order
  def bill
    PaymentGateway.charge(@user, @amount)
  end
end

class Refund
  def run
    PaymentGateway.refund(@order, @amount)
  end
end
```

Prefer gateways that have interfaces closer to the client code.
Changes to the external service are then isolated in the gateway.

Summary

Dependencies are powerful yet perilous

Depend on abstractions using:
- dependency injection
- inversion of control
- tell, don’t ask
- gateways

Avoid Ordering Dependencies

```
class Oven
  def initialize(temperature, duration)
    @temperature = temperature
    @duration = duration
  end
end

Oven.new(200, 30)
```

Avoid Ordering Dependencies

```ruby
class Oven
  def initialize(temperature:, duration:)
    @temperature = temperature
    @duration = duration
  end
end

Oven.new(temperature: 200, duration: 30)
Oven.new(duration: 30, temperature: 200)
```

Avoid Ordering Dependencies

```ruby
class Oven
  def initialize(temperature: 200, duration:)
    @temperature = temperature
    @duration = duration
  end
end

Oven.new(temperature: 200, duration: 30)
Oven.new(duration: 30, temperature: 200)
Oven.new(duration: 30)
```