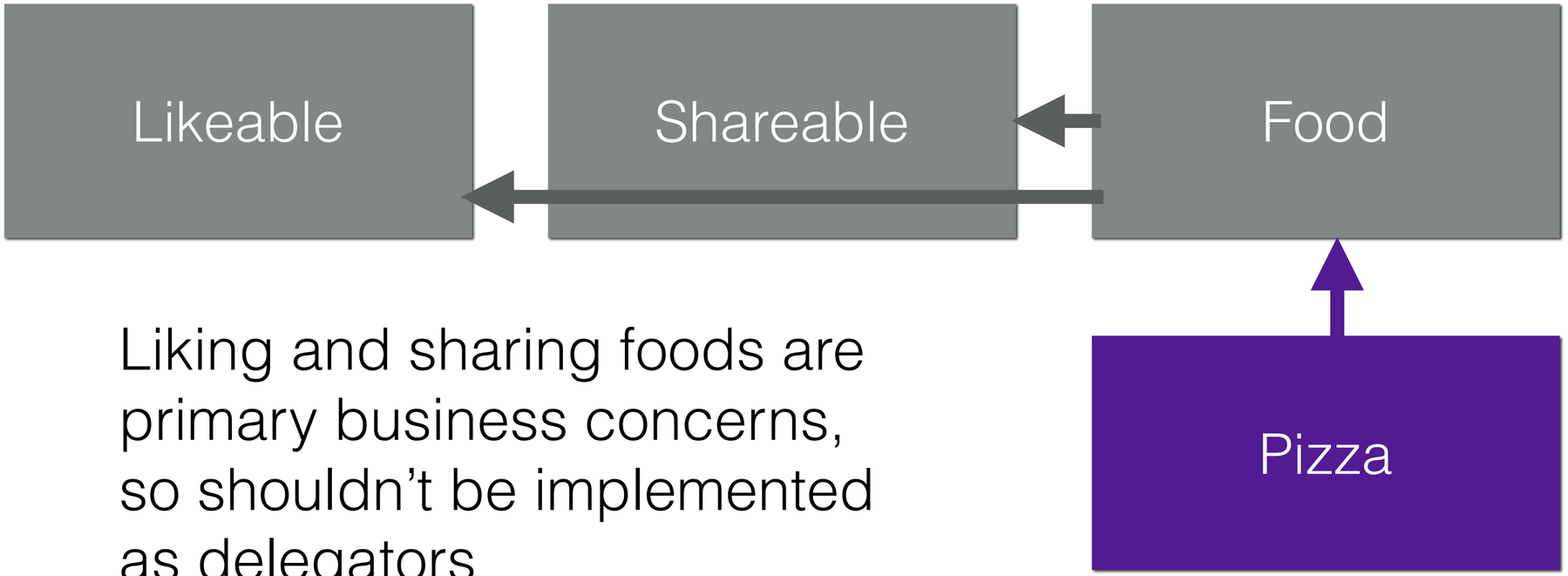
Plug-ins

Designing and Maintaining Software (DAMS)

Louis Rose

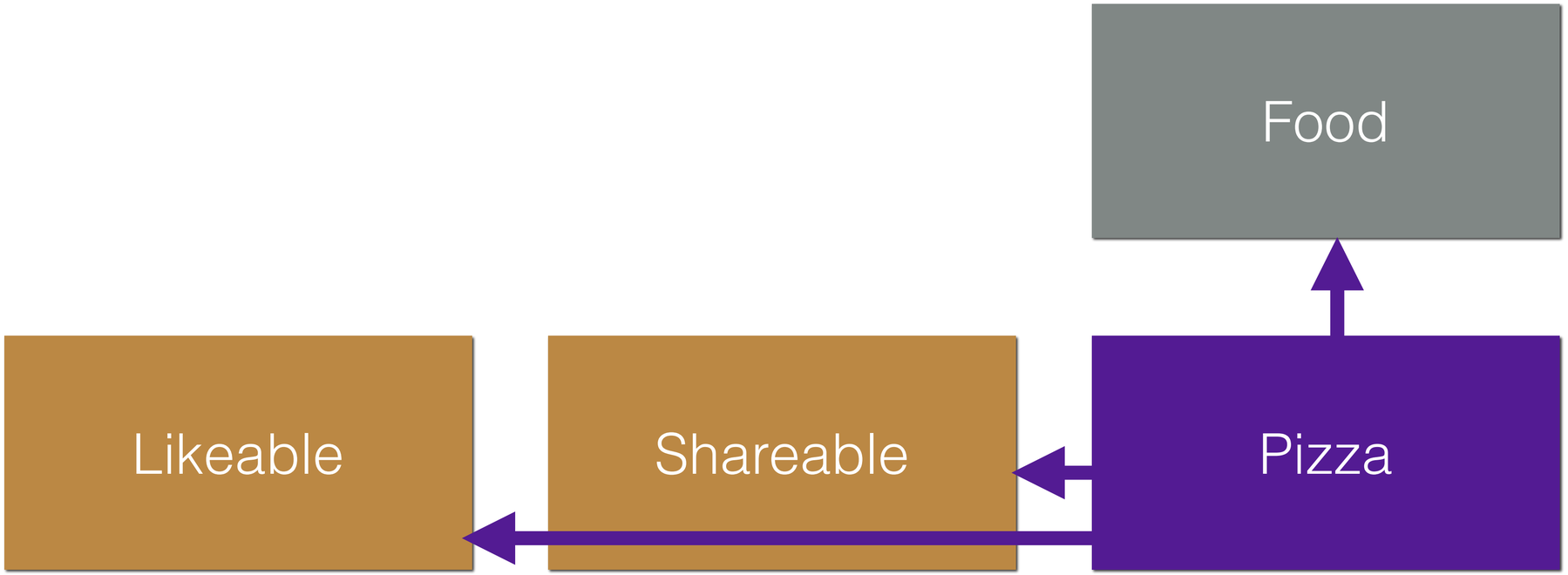
Problem

Current Architecture

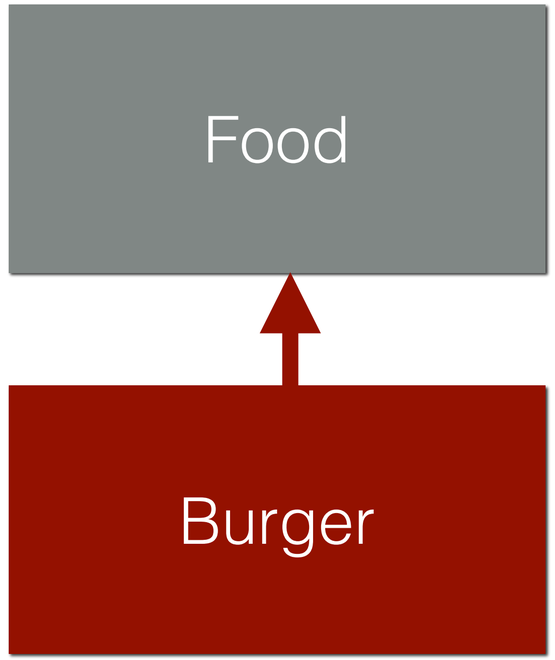


Liking and sharing foods are primary business concerns, so shouldn't be implemented as delegators

Target Architecture



Target Architecture



Recap: Ruby Modules

A module is a group of methods & constants

```
module Likeable
  def likes
    @likes ||= 0
  end

  def like!
    @likes += 1
  end
end
```

Including a module appends the methods & constants

```ruby
module Likeable
  def likes
    @likes ||= 0
  end

  def like!
    @likes += 1
  end
end
```

```ruby
class Pizza
  include Likeable
end

p = Pizza.new
p.likes # returns 0
p.like!
p.likes # returns 1
```

Object#extend includes a module for a specific instance

```ruby
module Likeable
  def likes
    @likes ||= 0
  end

  def like!
    @likes += 1
  end
end
```

```ruby
class Chef
end

c = Chef.new
c.likes # throws NoMethodError
c.extend(Likeable)
c.likes # returns 0

c2 = Chef.new
c.likes # throws NoMethodError
```

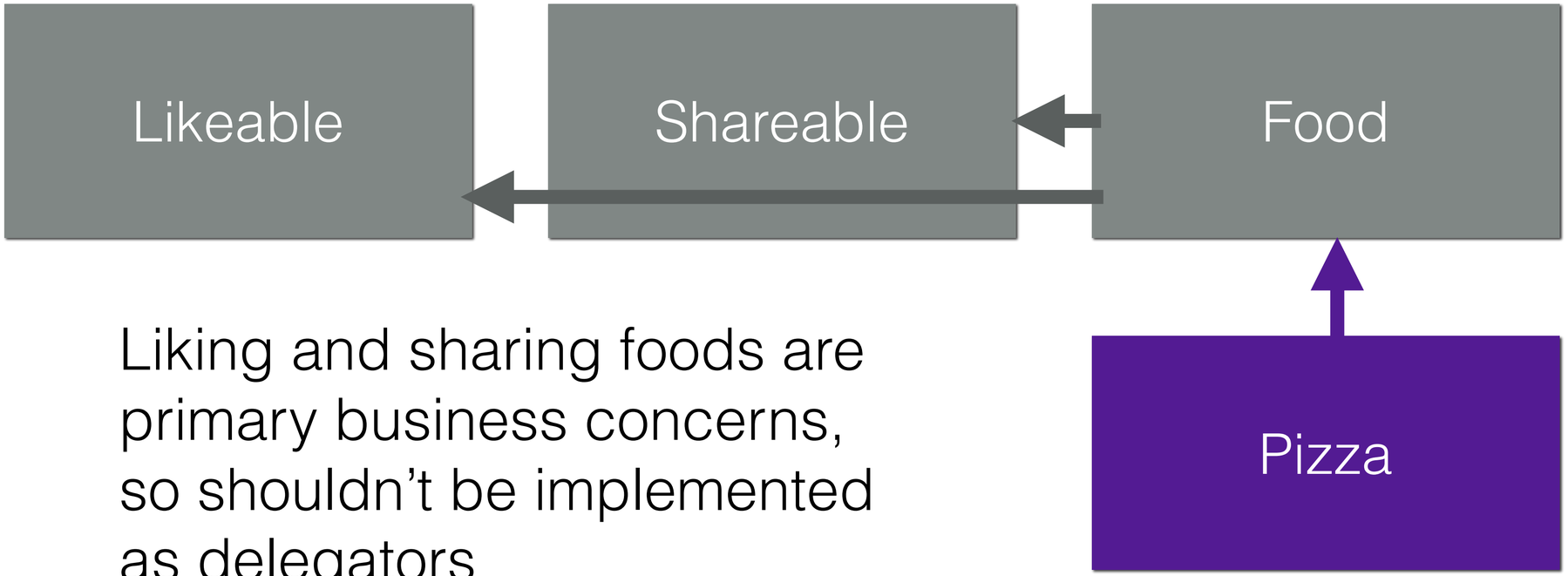
Classes are objects too, so can be extended

```ruby
module Likeable
  def likes
    @likes ||= 0
  end

  def like!
    @likes += 1
  end
end
```

```ruby
class Company
  extend Likeable
end

Company.likes # returns 0
Company.like!
Company.likes # returns 1

c = Company.new
c.likes # throws NoMethodError
```

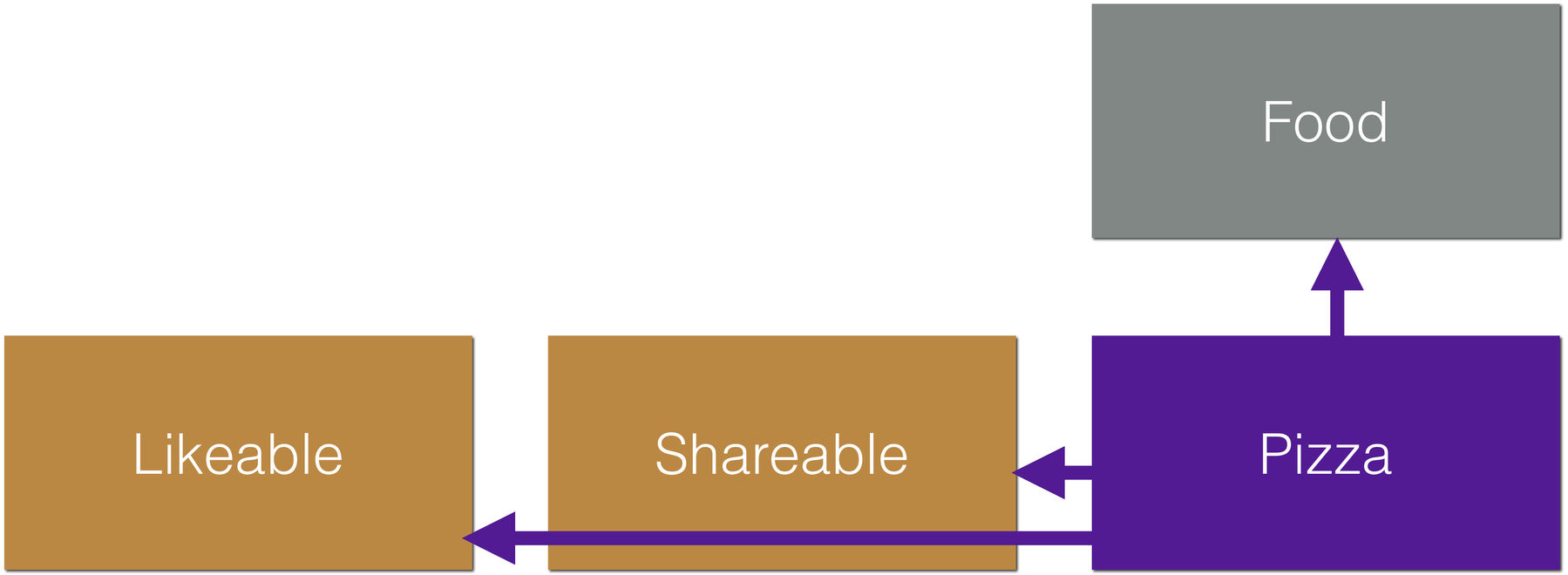
Initial solutions

Current Architecture



Liking and sharing foods are primary business concerns, so shouldn't be implemented as delegators

Target Architecture



Ship as separate gems

```ruby
module Likeable
  def likes
    @likes ||= 0
  end

  def like!
    @likes += 1
  end
end
```

```ruby
require "foodstore"
require "foodstore-social"

class Pizza < Food
  include Likeable
end

class Salad < Food
end
```

Problem: how to configure an optional module?

```
module Shareable
  def share
    self.class.networks.each do |m|
      puts "Shared via #{m}"
    end
  end
end
```

```
require "foodstore"
require "foodstore-social"

class Pizza < Food
  include Shareable
end
```

Solution! But, pizza is now coupled to implementation of Shareable :-(

```
module Shareable
  def share
    self.class.networks.each do |m|
      puts "Shared via #{m}"
    end
  end
end
```

```
require "foodstore"
require "foodstore-social"

class Pizza < Food
  include Shareable
  def self.networks
    %w(twitter facebook)
  end
end
```

We want this, but it's not valid Ruby

```ruby
module Shareable
  def share
    self.class.networks.each do |m|
      puts "Shared via #{m}"
    end
  end
end
```

```ruby
require "foodstore"
require "foodstore-social"

class Pizza < Food
  # not valid Ruby
  include Shareable(%w(twitter facebook))
end
```

Object#extend
to the rescue!

Use extend to add a config method to the class

```ruby
module Shareable
  attr_reader :social_networks

  def shareable_via(social_networks)
    @social_networks = social_networks
    include InstanceMethods
  end

  module InstanceMethods
    def share
      self.class.social_networks.each do |m|
        puts "Shared via #{m}"
      end
    end
  end
end
```

```ruby
require "foodstore"
require "foodstore-shareable"

class Pizza < Food
  extend Shareable
  shareable_via %w(twitter facebook)
end
```

Problem: forget to call the config method?

```
require "foodstore"
require "foodstore-shareable"

class Pizza < Food
  extend Shareable
end

Pizza.new.share
# NoMethodError: undefined method `share'
```

Problem: plug-ins now have an inconsistent API

```
require "foodstore"
require "foodstore-shareable"

class Pizza < Food
  include Likeable
  extend Shareable
  shareable_via %w(twitter facebook)
end
```

A plug-in API

What responsibilities does Shareable have?

```
module Shareable
  attr_reader :social_networks

  def shareable_via(social_networks)
    @social_networks = social_networks
    include InstanceMethods
  end

  module InstanceMethods
    def share
      self.class.social_networks.each do |m|
        puts "Shared via #{m}"
      end
    end
  end
end
```

What responsibilities does Shareable have?

```
module Shareable
  attr_reader :social_networks

  def shareable_via(social_networks)
    @social_networks = social_networks
    include InstanceMethods
  end

  module InstanceMethods
    def share
      self.class.social_networks.each do |m|
        puts "Shared via #{m}"
      end
    end
  end
end
```

Extract the plug-in configuration logic

```ruby
module Shareable
  def self.configure(pluggable, networks)
    pluggable.social_networks = networks
  end

  module ClassMethods
    attr_accessor :social_networks
  end

  module InstanceMethods
    def share
      self.class.social_networks.each do |m|
        puts "Shared via #{m}"
      end
    end
  end
end
```

```ruby
module Pluggable
  def plugin(mod, options)
    extend mod::ClassMethods
    include mod::InstanceMethods
    mod.configure(self, options)
  end
end
```

```ruby
require "foodstore"
require "foodstore-shareable"

class Pizza < Food
  extend Pluggable
  plugin Likeable
  plugin Shareable, %w(twitter facebook)
end
```

A final flourish

Avoid tight coupling…

```
require "foodstore"
require "foodstore-shareable"

class Pizza < Food
  extend Pluggable
  plugin Likeable
  plugin Shareable, %w(twitter facebook)
end
```

… by referencing plug-ins by name rather than by constant

```
require "foodstore"
require "foodstore-shareable"

class Pizza < Food
  extend Pluggable
  plugin :likeable
  plugin :shareable, %w(twitter facebook)
end
```

Adjust plug-in setup to locate by name

```
module Pluggable
  def plugin(mod_name, *options)
    mod = Plugins.locate(mod_name)
    extend mod::ClassMethods if defined?(mod::ClassMethods)
    include mod::InstanceMethods if defined?(mod::InstanceMethods)
    mod.configure(self, *options)
  end
end
```

Create Plugins module

```ruby
module Pluggable
  def plugin(mod_name, *options)
    mod = Plugins.locate(mod_name)
    extend mod::ClassMethods if defined?(mod::ClassMethods)
    include mod::InstanceMethods if defined?(mod::InstanceMethods)
    mod.configure(self, *options)
  end
end
```

```ruby
module Plugins
  def self.locate(name)
    registry.fetch(name)
  end

  def self.registry
    @registry ||= {}
  end
end
```

How to register a plug-in?

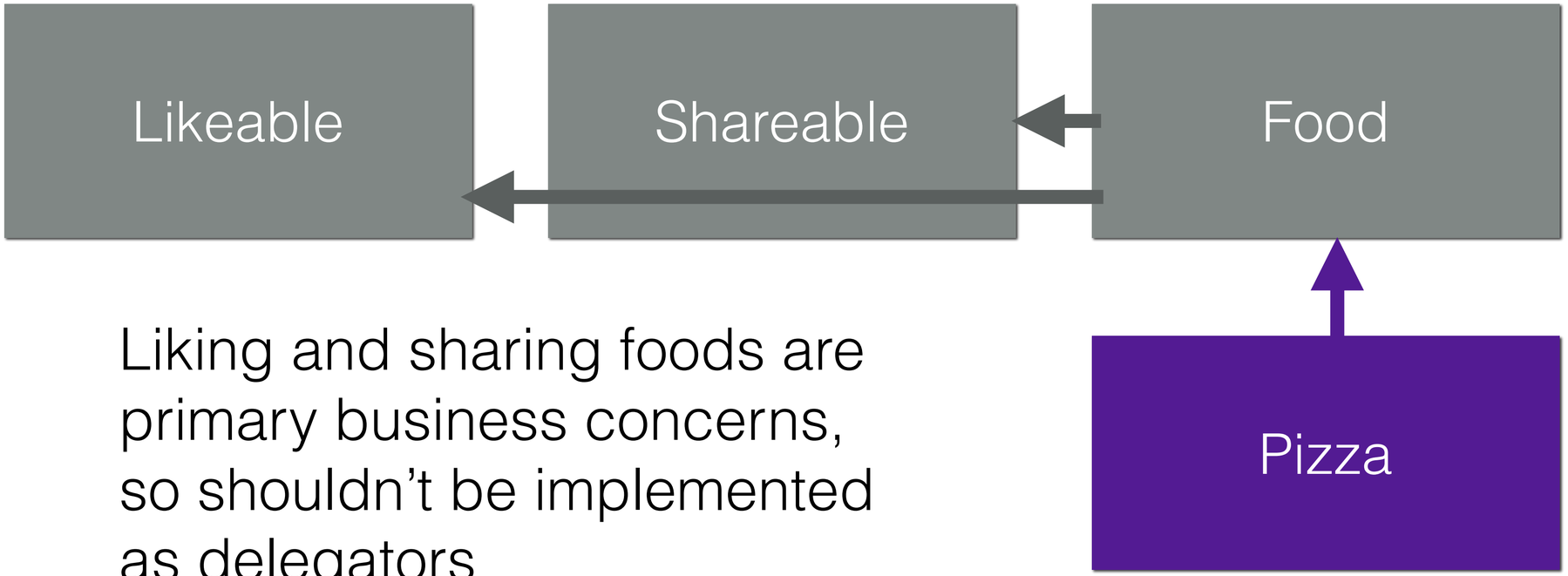
```
module Plugins
  def self.locate(name)
    registry.fetch(name)
  end

  def self.registry
    @registry ||= {}
  end

  def self.register_plugin(name, mod)
    registry[name] = mod
  end
end
```

A finished plug-in

```
module Shareable
  Plugins.register_plugin :shareable, Shareable

  def self.configure(pluggable, *methods)
    pluggable.social_networks = methods
  end

  module ClassMethods
    attr_accessor :social_networks
  end

  module InstanceMethods
    def share
      self.class.sharing_methods.each do |m|
        puts "Shared via #{m}"
      end
    end
  end
end
```

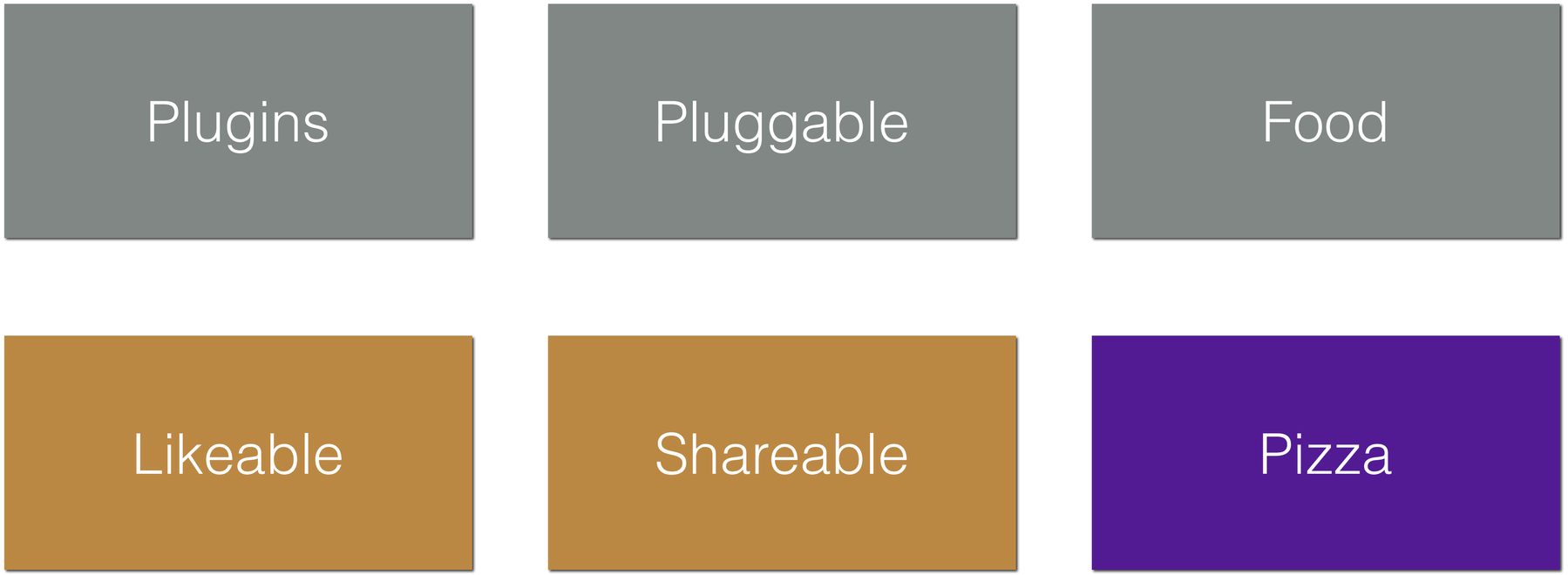
What did we achieve?

Old Architecture

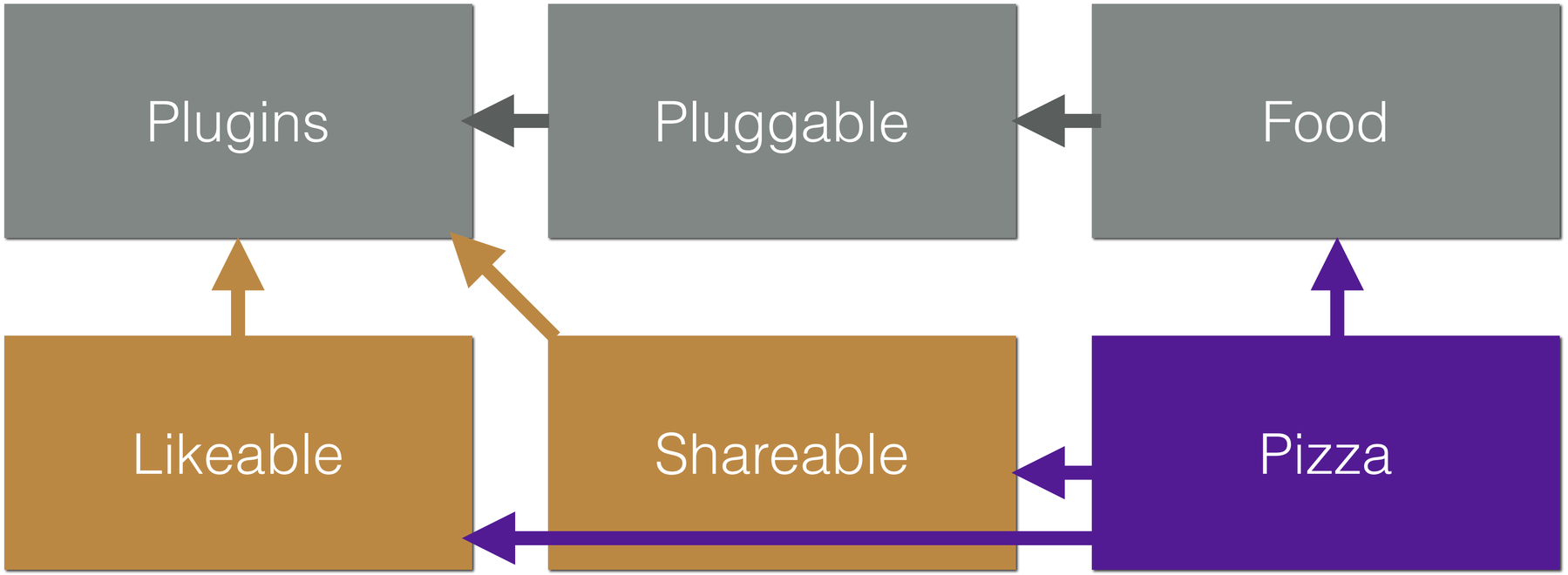


Liking and sharing foods are primary business concerns, so shouldn't be implemented as delegators

New Architecture



New Architecture



Summary

A plug-in architecture allows (possibly as-yet-unforeseen) extensions to be made without redeploying core components.

Ruby modules provide various avenues for implementing plug-in architectures, though some fairly complicated Ruby code is needed for sophisticated solutions.