Refactoring code that accesses external services

When I write code that deals with external services, I find it valuable to separate that access code into separate objects. Here I show how I would refactor some congealed code into a common pattern of this separation.

17 February 2015

Martin Fowler

Find **similar articles** to this by looking at these tags: object collaboration design · clean code · refactoring · application architecture

One of the characteristics of software systems is that they don't live on their own. In order to do something useful, they usually need to talk to other bits of software, written by different people, people that we don't know and who neither know or care about the software that we're writing.

When we're writing software that does this kind of external collaboration, I think it's particularly useful to apply good modularity and encapsulation. There are common patterns which I see and have found valuable in doing this.

In this article I'll take a simple example, and walk through the refactorings that introduce the kind of modularity I'm looking for.

The starting code

The example code's job is to read some data about videos from a JSON file, enrich it with data from YouTube, calculate some simple further data, and then return the data in JSON.

Here is the starting code.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    client = GoogleAuthorizer.new(
      token_key: 'api-youtube',
      application_name: 'Gateway Youtube Example',
      application_version: '0.1'
      ).api_client
    youtube = client.discovered_api('youtube', 'v3')
    request = {
      api_method: youtube.videos.list,
      parameters: {
        id: ids.join(","),
        part: 'snippet, contentDetails, statistics',
      }
    }
    response = JSON.parse(client.execute!(request).body)
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = response['items'].find{|v| id == v['id']}
      video['views'] = youtube_record['statistics']['viewCount'].to_i
      days_available = Date.today - Date.parse(youtube_record['snippet']
['publishedAt'])
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end
```

The language for this example is Ruby

The first thing for me to say here is that there isn't much code in this example. If the entire codebase is just this script, then you don't have to worry as much about modularity. I need a small example, but any reader's eyes would glaze over if we looked at a real system. So I have to ask you to imagine this code as typical code in a system of tens of thousands of lines.

The access to the YouTube API is mediated through a GoogleAuthorizer object which, for this article's purposes, I'm going to treat as an external API. It handles the messy details of connecting to a Google service (such as YouTube) and in particular handles authorization issues. If you want to understand how it works, take a look at an article I wrote recently about accessing Google APIs.

What's up with this code? You may not understand everything this code is doing, but you should be able to see that it mixes different concerns, which I've suggested by coloring the code example below. In order to make any changes you have to comprehend how to access to YouTube's API, how YouTube structures its data , and some domain logic.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    client = GoogleAuthorizer.new(
      token_key: 'api-youtube',
      application_name: 'Gateway Youtube Example',
      application_version: '0.1'
      ).api_client
    youtube = client.discovered_api('youtube', 'v3')
    request = {
      api_method: youtube.videos.list,
      parameters: {
        id: ids.join(","),
        part: 'snippet, contentDetails, statistics',
      }
    }
    response = JSON.parse(client.execute!(request).body)
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = response['items'].find{|v| id == v['id']}
      video['views'] = youtube_record['statistics']['viewCount'].to_i
      days_available = Date.today - Date.parse(youtube_record['snippet']
['publishedAt'])
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end
```

It's common for software mavens like me to talk about "separation of concerns" - which basically means different topics should be in separate modules. My primary reason for

this is comprehension: in a well-modularized program each module should be about one topic, so I can remain ignorant of anything I don't need to understand. Should YouTube's data formats change, I shouldn't have to understand the domain logic of the application to rearrange the access code. Even if I'm making a change that takes some new data from YouTube and uses it in some domain logic, I should be able to split my task into those parts and deal with each one separately, minimizing how many lines of code I need to keep spinning in my head.

My refactoring mission is to split these concerns out into separate modules. When I'm done the only code in the Video Service should be the uncolored code - the code that coordinates these other responsibilities.

Putting the code under test

The first step in refactoring is always the same. You need to be confident that you aren't going to inadvertently break anything. Refactoring is all about stringing together a large set of small steps, all of which are behavior preserving. By keeping the steps small, we increase the chances that we don't screw up. But I know myself well enough to know I can screw up even the simplest change, so to get the confidence I need I have to have tests to catch my mistakes.

But code like this isn't straightforward to test. It would be nice to write a test that asserts on the calculated monthly views field. After all if anything else goes wrong, this is going to give an incorrect answer. But the trouble is that I'm accessing live YouTube data, and people have a habit of watching videos. The view count field from YouTube will change regularly, causing my tests to go red non-deterministically

So my first task is to remove that piece of flakiness. I can do that by introducing a Test Double, an object that looks like YouTube but instead responds in a deterministic fashion. Unfortunately here I run into The Legacy Code Dilemma.

> *The Legacy Code Dilemma: When we change code, we should have tests in place. To put tests in place, we often have to change code.*
>
> *-- Michael Feathers*

Given that I have to make this change without tests, I need to make the smallest and simplest changes I can think of that will get the interaction with YouTube behind a seam where I can introduce a test double. So my first step is to use Extract Method to get the

interaction with YouTube separated from the rest of the routine by into its own method.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    response = call_youtube ids
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = response['items'].find{|v| id == v['id']}
      video['views'] = youtube_record['statistics']['viewCount'].to_i
      days_available = Date.today - Date.parse(youtube_record['snippet']
['publishedAt'])
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end

  def call_youtube ids
    client = GoogleAuthorizer.new(
      token_key: 'api-youtube',
      application_name: 'Gateway Youtube Example',
      application_version: '0.1'
      ).api_client
    youtube = client.discovered_api('youtube', 'v3')
    request = {
      api_method: youtube.videos.list,
      parameters: {
        id: ids.join(","),
        part: 'snippet, contentDetails, statistics',
      }
    }
    return JSON.parse(client.execute!(request).body)
  end
```

Doing this achieves two things. Firstly it nicely pulls out the google API manipulation code into its own method (mostly) isolating it from any other kind of code. This on its own is worthwhile. Secondly, and more urgently, it sets up a seam which I can use to substitute test behavior. Ruby's built in minitest library allows me to easily stub individual methods on an object.

```
class VideoServiceTester < Minitest::Test
  def setup
```

```
    vs = VideoService.new
    vs.stub(:call_youtube, stub_call_youtube) do
      @videos = JSON.parse(vs.video_list)
      @µS  =  @videos.detect{|v| 'wgdBVIX9ifA' == v['youtubeID']}
      @evo =  @videos.detect{|v| 'ZIsgHs0w44Y' == v['youtubeID']}
    end
  end
  def stub_call_youtube
    JSON.parse(File.read('test/data/youtube-video-list.json'))
  end
  def test_microservices_monthly_json
    assert_in_delta 5880, @µS ['monthlyViews'], 1
    assert_in_delta   20, @evo['monthlyViews'], 1
  end
  # further tests as needed…
```

By separating out the YouTube call, and stubbing it, I can make this test behave deterministically. Well at least for today, for it to work tomorrow I need to do the same thing with the call to `Date.today`.

class VideoServiceTester…

```
  def setup
    Date.stub(:today, Date.new(2015, 2, 2)) do
      vs = VideoService.new
      vs.stub(:call_youtube, stub_call_youtube) do
        @videos = JSON.parse(vs.video_list)
        @µS  =  @videos.detect{|v| 'wgdBVIX9ifA' == v['youtubeID']}
        @evo =  @videos.detect{|v| 'ZIsgHs0w44Y' == v['youtubeID']}
      end
    end
  end
```

Separating the remote call into a connection object

Separating concerns by putting code into different functions is a first level of separation. But when the concerns are as different as domain logic and dealing with an external data provider, I prefer to increase the level of separation into different classes.

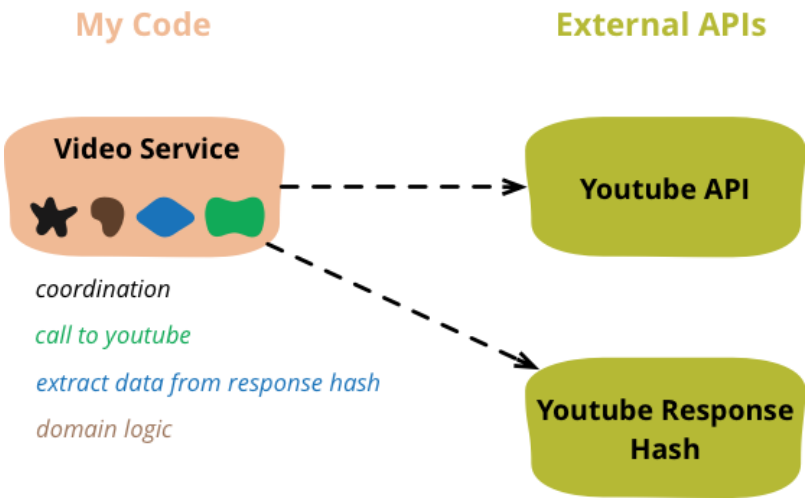


Figure 1: At the beginning, the video service class contains four responsibilities

My first step is therefore to create a new class and use Move Method.

class VideoService...

```
  def call_youtube ids
    YoutubeConnection.new.list_videos ids
  end
```

class YoutubeConnection...

```
  def list_videos ids
    client = GoogleAuthorizer.new(
      token_key: 'api-youtube',
      application_name: 'Gateway Youtube Example',
      application_version: '0.1'
      ).api_client
    youtube = client.discovered_api('youtube', 'v3')
    request = {
      api_method: youtube.videos.list,
      parameters: {
        id: ids.join(","),
        part: 'snippet, contentDetails, statistics',
      }
    }
    return JSON.parse(client.execute!(request).body)
  end
```

With that I can also change the stub so it returns a test double rather than simply stubbing the method.

class VideoServiceTester...

```
  def setup
    Date.stub(:today, Date.new(2015, 2, 2)) do
      YoutubeConnection.stub(:new, YoutubeConnectionStub.new) do
        @videos = JSON.parse(VideoService.new.video_list)
        @µS  =  @videos.detect{|v| 'wgdBVIX9ifA' == v['youtubeID']}
        @evo =  @videos.detect{|v| 'ZIsgHs0w44Y' == v['youtubeID']}
      end
    end
  end
```

class YoutubeConnectionStub...

```
  def list_videos ids
    JSON.parse(File.read('test/data/youtube-video-list.json'))
  end
```

When doing this refactoring, I have to be wary that my shiny new tests won't catch any mistakes I make behind the stub, so I have to manually ensure that the production code still works. (And yes, since you asked, I did make a mistake while doing this (leaving off the argument to list-videos). There's a reason I need to test so much.)

The greater separation of concerns you get with a separate class also gives you a better seam for testing - I can wrap everything that needs to be stubbed into a single object creation, which is particularly handy if we need to make multiple calls to the same service object during the test.

With the call to YouTube moved to the connection object, the method on the video service isn't worth having any more so I subject it to Inline Method.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    response = YoutubeConnection.new.list_videos ids
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = response['items'].find{|v| id == v['id']}
      video['views'] = youtube_record['statistics']['viewCount'].to_i
      days_available = Date.today - Date.parse(youtube_record['snippet']
['publishedAt'])
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
```

```
    return JSON.dump(@video_list)
  end

  def call_youtube_ids
    YoutubeConnection.new.list_videos_ids
  end
```

I don't like that my stub has to parse the json string. On the whole I like to keep connection objects as Humble Objects, because any behavior they do isn't tested. So I prefer to pull the parsing out into the callers.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    response = JSON.parse(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = response['items'].find{|v| id == v['id']}
      video['views'] = youtube_record['statistics']['viewCount'].to_i
      days_available = Date.today - Date.parse(youtube_record['snippet']
['publishedAt'])
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end
```

class YoutubeConnection...

```
  def list_videos ids
    client = GoogleAuthorizer.new(
      token_key: 'api-youtube',
      application_name: 'Gateway Youtube Example',
      application_version: '0.1'
      ).api_client
    youtube = client.discovered_api('youtube', 'v3')
    request = {
      api_method: youtube.videos.list,
      parameters: {
        id: ids.join(","),
        part: 'snippet, contentDetails, statistics',
      }
    }
```

```
    return JSON.parse(client.execute!(request).body)
  end
```

class YoutubeConnectionStub...

```
  def list_videos ids
    JSON.parse(File.read('test/data/youtube-video-list.json'))
  end
```

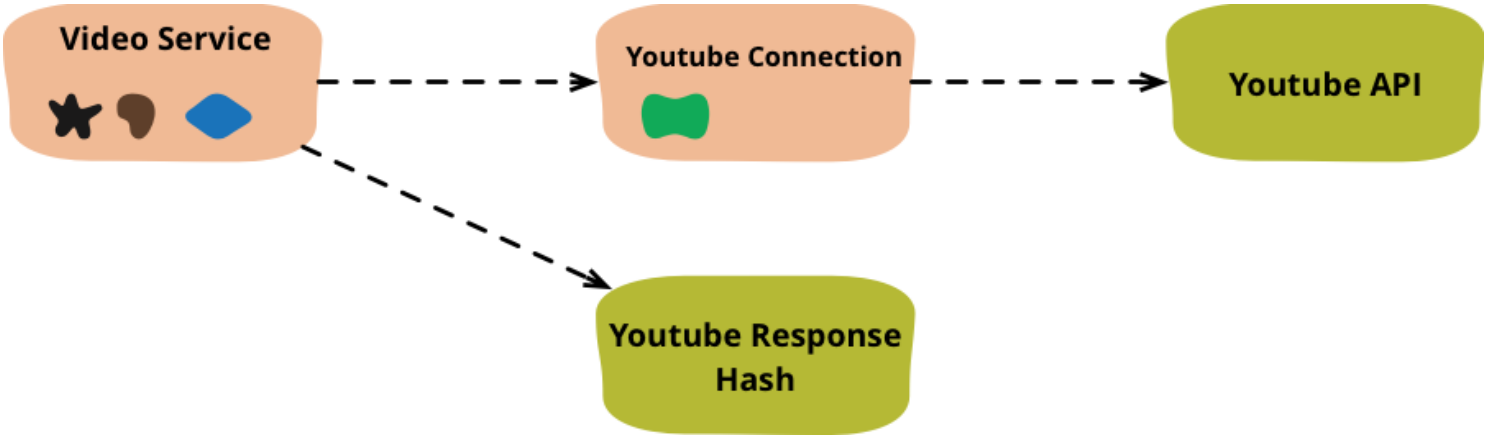


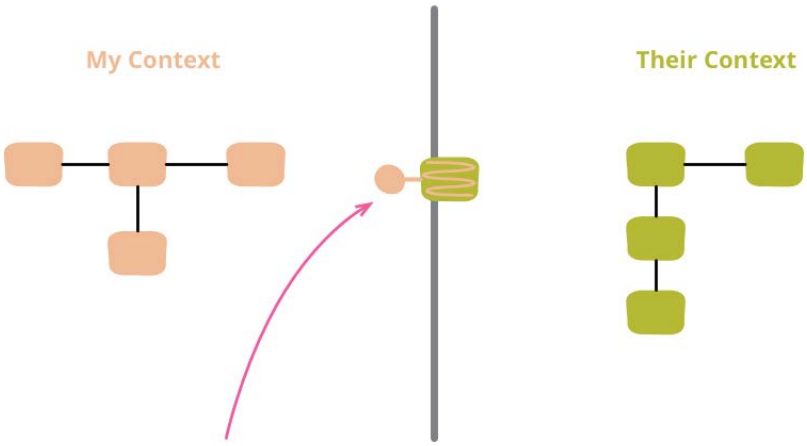
*Figure 2: The first major step separates the youtube connection code into a **connection** object.*

Separating the youtube data structure into a Gateway

Now that I have the basic connection to YouTube separated and stubbable, I can work on the code that delves through the YouTube data structures. The problem here is that a bunch of code needs to know that to get the view count data, you have to look into the "statistics" part of the result, but to get the published date you need to delve into "snippet" section instead. Such delving is common with data from remote sources, it's organized the way that makes sense for them, not for me. This is entirely reasonable behavior, they don't have insights into my needs, I have enough trouble doing that on my own.

I find that a good way to think about this is Eric Evans's notion of Bounded Context. YouTube organizes its data according to its context, while I want to organize mine according to a different one. Code that combines two bounded contexts gets convoluted because it's mixing two separate vocabularies together. I need to separate them with what Eric calls an *anti-corruption layer*, a clear boundary between them. His illustration of an anti-corruption layer is of the Great Wall of China, and as with any wall like this, we need gateways that allow some things to pass between them. In software terms a gateway allows me to reach through the wall to get the data I need from the YouTube

bounded context. But the gateway should be expressed in a way that makes sense within my context rather than theirs.



*The **interface** to the gateway object is written in terms of **my** context. The **body** of the gateway **translates** from their context to mine.*

In this, simple, example, that means a gateway object that can give me the published date and view counts without the client needing to know how that's stored in the YouTube data structure. The gateway object translates from YouTube's context into mine.

I begin by creating a gateway object that I initialize with response I got from the connection.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = youtube.record(id)
      video['views'] = youtube_record['statistics']['viewCount'].to_i
      days_available = Date.today - Date.parse(youtube_record['snippet']
['publishedAt'])
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end
```

class YoutubeGateway...

```
  def initialize responseJson
    @data = JSON.parse(responseJson)
  end
  def record id
    @data['items'].find{|v| id == v['id']}
  end
```

I created the simplest behavior I could at this point, even though I don't intend to use the gateway's record method eventually, indeed unless I stop for a cup of tea I don't think it will last for half-an-hour.

Now I'll move the delving logic for the views from the service into the gateway, creating a separate gateway item class to represent each video record.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = youtube.record(id)
      video['views'] = youtube.item(id)['views']
      days_available = Date.today - Date.parse(youtube_record['snippet']
['publishedAt'])
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end
```

class YoutubeGateway...

```
  def item id
    {
      'views' => record(id)['statistics']['viewCount'].to_i
    }
  end
```

I do the same for the published date

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_record = youtube.record(id)
      video['views'] = youtube.item(id)['views']
      days_available = Date.today - youtube.item(id)['published']
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end
```

class YoutubeGateway...

```
  def item id
    {
      'views'     => record(id)['statistics']['viewCount'].to_i,
      'published' => Date.parse(record(id)['snippet']['publishedAt'])
    }
  end
```

Since I'm using the records in the gateway looked up by key, I'd like to reflect that usage better in the internal data structure, which I can do by replacing the list with a hash

class YoutubeGateway...

```
  def initialize responseJson
    @data = JSON.parse(responseJson)['items']
      .map{|i| [ i['id'], i ] }
      .to_h
  end
  def item id
    {
      'views' =>  @data[id]['statistics']['viewCount'].to_i,
      'published' => Date.parse(@data[id]['snippet']['publishedAt'])
    }
  end
  def record id
    @data['items'].find{|v| id == v['id']}
  end
```

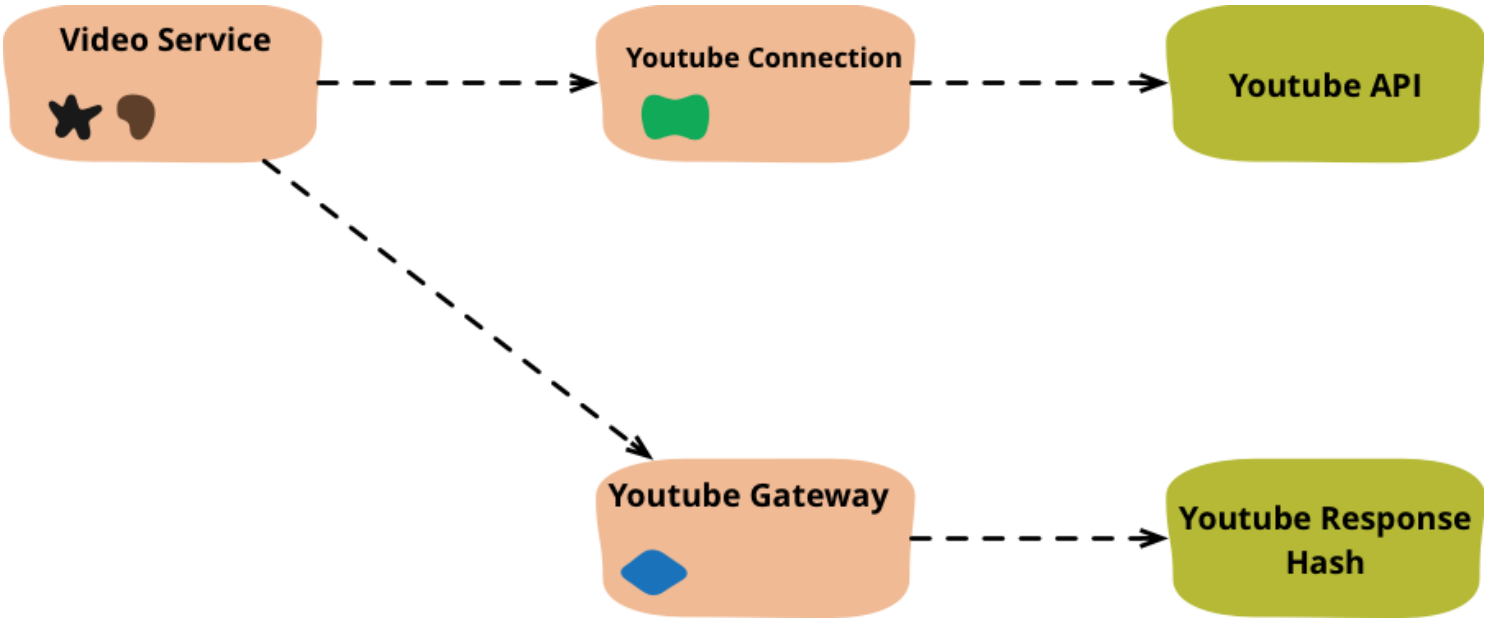


*Figure 4: Separating data handling into a **gateway** object*

With that done, I've done the key separation that I wanted to do. The YouTube connection object handles the calls to YouTube, returning a data structure that it gives to the YouTube gateway object. The service code is now all about how I want to see the data rather than how it's stored in different services.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json'))
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      video['views'] = youtube.item(id)['views']
      days_available = Date.today - youtube.item(id)['published']
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list)
  end
```

Separating the domain logic into a Domain Object

Although all the interaction of YouTube is now parceled out to separate objects, the video service still mixes its domain logic (how to calculate monthly views) from

choreographing the relationship between the data stored locally and the data in the service. If I introduce a domain object for the video, I can separate that out.

My first step is to simply wrap the hash of video data in an object.

class Video...

```
  def initialize aHash
    @data = aHash
  end
  def [] key
    @data[key]
  end
  def []= key, value
    @data[key] = value
  end
  def to_h
    @data
  end
```

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
Video.new(h)}
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      video['views'] = youtube.item(id)['views']
      days_available = Date.today - youtube.item(id)['published']
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list.map{|v| v.to_h})
  end
```

To move the calculation logic into the new video object, I first need to get it into the right shape for the move - which I can do by parcelling it all into a single method on video service with the video domain object and the YouTube gateway item as arguments. The first step to that is to use Extract Variable on the gateway item.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
```

```
Video.new(h)}
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_item = youtube.item(id)
      video['views'] = youtube_item['views']
      days_available = Date.today - youtube_item['published']
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
    end
    return JSON.dump(@video_list.map{|v| v.to_h})
  end
```

With that done I can easily extract the calculation logic into its own method.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
Video.new(h)}
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_item = youtube.item(id)
      enrich_video video, youtube_item
    end
    return JSON.dump(@video_list.map{|v| v.to_h})
  end

  def enrich_video video, youtube_item
      video['views'] = youtube_item['views']
      days_available = Date.today - youtube_item['published']
      video['monthlyViews'] = video['views'] * 365.0 / days_available /
12
  end
```

And then it's easy to apply Move Method to move it into the video.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
Video.new(h)}
```

```
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_item = youtube.item(id)
      video.enrich_with_youtube youtube_item
    end
    return JSON.dump(@video_list.map{|v| v.to_h})
  end
```

class Video...

```
  def enrich_with_youtube youtube_item
    @data['views'] = youtube_item['views']
    days_available = Date.today - youtube_item['published']
    @data['monthlyViews'] = @data['views'] * 365.0 / days_available / 12
  end
```

With that done, I can remove the updates to video's hash.

class Video...

```
  def []= key, value
    @data[key] = value
  end
```

Now I have proper objects, I can simplify the choreography with ids in the service method. I start by using Inline Temp on `youtube_item` and then replace the reference to the enumeration index with a method call on the video object.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
Video.new(h)}
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    ids.each do |id|
      video = @video_list.find{|v| id == v['youtubeID']}
      youtube_item = youtube.item(id)
      video.enrich_with_youtube(youtube.item(video.youtube_id))
    end
    return JSON.dump(@video_list.map{|v| v.to_h})
```

```
  end
```

class Video...

```
  def youtube_id
    @data['youtubeID']
  end
```

That allows me to just use the objects directly for the enumeration.

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
Video.new(h)}
    ids = @video_list.map{|v| v['youtubeID']}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    @video_list.each {|v|
v.enrich_with_youtube(youtube.item(v.youtube_id))}
    return JSON.dump(@video_list.map{|v| v.to_h})
  end
```

And remove the accessor for the hash in the video

class Video...

```
  def [] key
    @data[key]
  end
```

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
Video.new(h)}
    ids = @video_list.map{|v| v.youtube_id}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    @video_list.each {|v|
v.enrich_with_youtube(youtube.item(v.youtube_id))}
    return JSON.dump(@video_list.map{|v| v.to_h})
  end
```

I could replace the video object's internal hash with fields, but I don't think it's worth it as it's primarily loaded with a hash and its final output is a jsonified hash. An Embedded

Document is a perfectly reasonable form of domain object.

Musings on the final objects

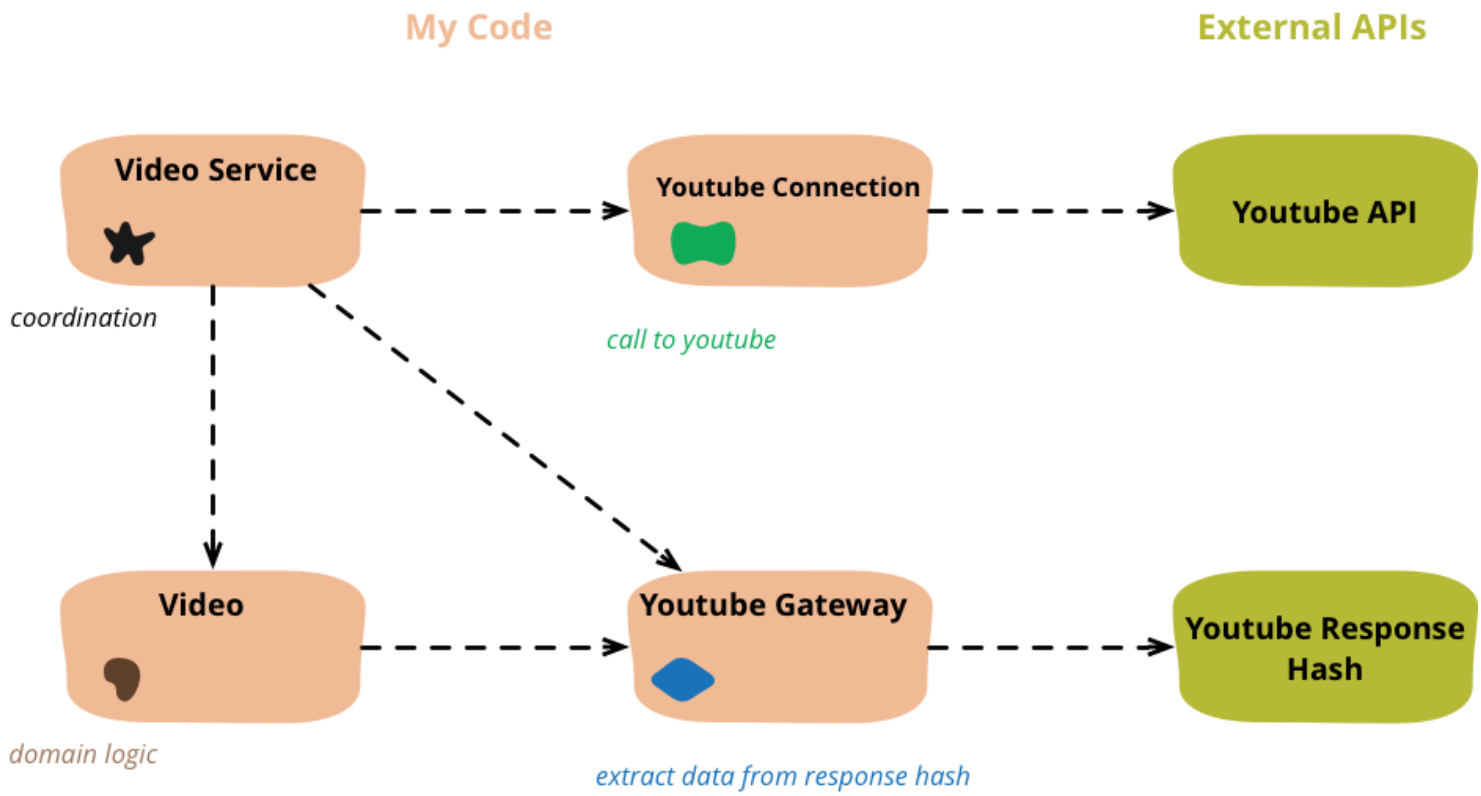


Figure 5: The objects created through this refactoring and their dependencies

class VideoService...

```
  def video_list
    @video_list = JSON.parse(File.read('videos.json')).map{|h|
Video.new(h)}
    ids = @video_list.map{|v| v.youtube_id}
    youtube = YoutubeGateway.new(YoutubeConnection.new.list_videos(ids))
    @video_list.each {|v|
v.enrich_with_youtube(youtube.item(v.youtube_id))}
    return JSON.dump(@video_list.map{|v| v.to_h})
  end
```

```
class YoutubeConnection
  def list_videos ids
    client = GoogleAuthorizer.new(
      token_key: 'api-youtube',
      application_name: 'Gateway Youtube Example',
      application_version: '0.1'
      ).api_client
```

```
    youtube = client.discovered_api('youtube', 'v3')
    request = {
      api_method: youtube.videos.list,
      parameters: {
        id: ids.join(","),
        part: 'snippet, contentDetails, statistics',
      }
    }
    return client.execute!(request).body
  end
end

class YoutubeGateway
  def initialize responseJson
    @data = JSON.parse(responseJson)['items']
      .map{|i| [ i['id'], i ] }
      .to_h
  end
  def item id
    {
      'views' =>  @data[id]['statistics']['viewCount'].to_i,
      'published' => Date.parse(@data[id]['snippet']['publishedAt'])
    }
  end
end

class Video
  def initialize aHash
    @data = aHash
  end
  def to_h
    @data
  end
  def youtube_id
    @data['youtubeID']
  end
  def enrich_with_youtube youtube_item
    @data['views'] = youtube_item['views']
    days_available = Date.today - youtube_item['published']
    @data['monthlyViews'] = @data['views'] * 365.0 / days_available / 12
  end
end
```

So what have I achieved? Refactoring often reduces code size, but in this case it's almost doubled from 26 to 54 lines. All else being equal, less code is better. But here I think the better modularity you get by separating the concerns is usually worth the size increase. This is also where the size of a pedagogical (i.e. toy) example can obscure the point. 26 lines of code isn't much to comprehend, but if we are talking about 2600 lines written in this style, then the modularity becomes well worth any code size increase. And usually any such increase is much smaller when you do this kind of thing with a larger code base since you uncover more opportunities to reduce code size by eliminating duplication.

You'll notice I've finished with four kinds of object here: coordinator, domain object, gateway, and connection. This is a common arrangement of responsibilities, although different cases see reasonable variations in how the dependencies are laid out. The best arrangement of responsibilities and dependencies differs due to particular needs. Code that needs to change frequently should be separated from code that changes rarely, or simply changes for different reasons. Code that is widely reused should not depend on code that is used only for a particular case. These drivers differ from circumstance to circumstance, and dicate the dependency patterns.

One common change is to reverse the dependency between domain object and gateway - turning the gateway into a Mapper. This allows the domain object to be independent of how it's populated, at the cost of the mapper knowing about the domain object and getting some access to its guts. If the domain object is used in many contexts, then this can be a valuable arrangement.

Another change might be to shift the code for calling the connection from the coordinator to the gateway. This simplifies the coordinator but makes the gateway a bit more complex. Whether this is a good idea depends on whether the coordinator is getting too complex, or if many coordinators use the same gateway leading to duplicate code in setting up the connection.

I also think it's likely I'd move some of the behavior of the connection out to the caller, particularly if the caller is a gateway object. The gateway knows what data it needs, so should supply the list of parts in the parameters of the call. But that's really only an issue once we have other clients calling `list_videos`, so I'd be inclined to wait until that day.

One thing that I feel is important, whatever the details of your case, is to have a consistent naming policy for the roles of the objects involved. I sometimes hear people say that you shouldn't put pattern names into your code, but I don't agree. Often pattern names help to communicate roles different elements play, so it's silly to spurn the

opportunity. Certainly within a team your code will show common patterns and the naming should reflect that. I use "gateway" following my coining of the Gateway pattern in P of EAA. I've used "connection" here to show the raw link to an external system, and intend to use that convention in my future writing. This naming convention isn't universal and while my pride would be gratifyingly inflated by you using my naming conventions, the important point isn't which naming convention you should use but that you should pick some convention.

When I break a method into a group of objects like this, there's a natural question about the consequences for testing. I had a unit test for the original method in the video service, should I now write tests for the three new classes? My inclination is that, providing the existing tests cover the behavior sufficiently, there isn't a need to add any more right away. As we add more behavior, then we should add more tests, and if this behavior is added to the new objects then the new tests will focus on them. In time that may mean that some of tests currently targeting the video service will look out of place and should move. But all of this in the future and should be dealt with in the future.

A particular concern I'd be watching for in the tests is the use of the stubs I put in over the YouTube connection. It's very easy for stubs like this to get out of hand, then they can actually slow down changes because a simple production code change leads to updating many tests. The essence here is to pay attention to duplication in the test code and address it as seriously as you do with duplication in production code.

Such thinking about organizing test doubles naturally leads into the broader question of assembling service objects. Now that I have split a behavior from a single service object into three service objects and a domain entity (using the Evans Classification) there's a natural question about how the service objects should be instantiated, configured, and assembled. Currently the video service does this directly for its dependents, but this can easily get out of hand with larger systems. To handle this complexity it's common to use techniques such as Service Locators and Dependency Injection. I'm not going to talk about that right now, but that may be the topic for a follow-on article.

This example uses objects, in large part because I'm far more familiar with object-oriented style than functional styles. But I would expect the fundamental division of responsibilities to be the same, but with boundaries set by function (or perhaps namespace) rather than classes and methods. Some other details would change, the video object would be a data structure and enriching it would create new data structures rather than modifying in place. Looking at this in a functional style would be an interesting article.

Finally I want to re-stress an important general point about refactoring. Refactoring isn't

a term you should use to any restructuring of a code base. It specifically means the approach that applies a series of very small behavior-preserving changes. We saw a couple of examples here where I deliberately introduced code that I knew I was going to remove shortly afterwards, just so I can take small steps that preserve behavior. This is the essence of refactoring, as I recently tweeted:

> *Refactoring is a specific way to change code by a series of tiny behavior-preserving transformations. It is not just moving code around.-- Martin Fowler*

The point here is that by taking small steps, you end up going faster because you don't break anything and thus avoid debugging. Most people find this counter-intuitive, I certainly did when Kent Beck first showed me how he refactored. But I quickly discovered how effective it is.

Significant Revisions

17 February 2015: First published