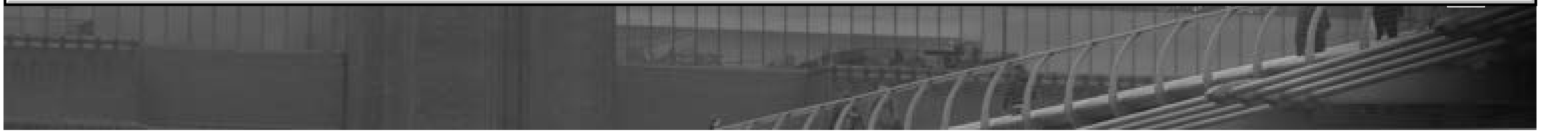

Refactoring with Loops and Collection Pipelines

The loop is the classic way of processing collections, but with the greater adoption of first-class functions in programming languages the collection pipeline is an appealing alternative. In this article I look at refactoring loops to collection pipelines with a series of small examples.

14 July 2015

Martin Fowler

Translations: Chinese
Find **similar articles** to this by looking at these tags: object collaboration design · refactoring

A common task in programming is processing a list of objects. Most programmers naturally do this with a loop, as it's one of the basic control structures we learn with our very first programs. But loops aren't the only way to represent list processing, and in recent years more people are making use of another approach, which I call the **collection pipeline**. This style is often considered to be part of functional programming, but I used it heavily in Smalltalk. As OO languages support lambdas and libraries that make first class functions easier to program with, then collection pipelines become an appealing choice.

Refactoring a simple loop into a pipeline

I'll start with a simple example of a loop and show the basic way I refactor one into a collection pipeline.

Let's imagine we have a list of authors, each of which has the following data structure.

class Author...
```
  public string Name { get; set; }
  public string TwitterHandle { get; set;}
  public string Company { get; set;}
```

This example uses C#

Here is the loop.

class Author...
```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author>
authors, string company) {
    var result = new List<String> ();
    foreach (Author a in authors) {
      if (a.Company == company) {
        var handle = a.TwitterHandle;
        if (handle != null)
          result.Add(handle);
      }
    }
    return result;
  }
```

My first step in refactoring a loop into a collection pipeline is to apply Extract Variable on the loop collection. [1]

class Author...
```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author>
authors, string company) {
    var result = new List<String> ();
    var loopStart = authors;
    foreach (Author a in loopStart) {
      if (a.Company == company) {
        var handle = a.TwitterHandle;
        if (handle != null)
          result.Add(handle);
      }
    }
    return result;
  }
```

This variable gives me a starting point for pipeline operations. I don't have a good name for it right now, so I'll use one that makes sense for the moment, expecting to rename it later.

I then start looking at bits of behavior in the loop. The first thing I see is a conditional check, I can move this to the pipeline with a filter operation.

class Author...
```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author>
authors, string company) {
    var result = new List<String> ();
    var loopStart = authors
      .Where(a => a.Company == company);
    foreach (Author a in loopStart) {
      ~~if (a.Company == company) {~~
        var handle = a.TwitterHandle;
        if (handle != null)
          result.Add(handle);
      ~~}~~
    }
    return result;
  }
```

I see the next part of the loop operates on the twitter handle, rather than the author, so I can use a a map operation [2].

class Author...
```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author>
authors, string company) {
    var result = new List<String> ();
    var loopStart = authors
      .Where(a => a.Company == company)
      .Select(a => a.TwitterHandle);
    foreach (string handle in loopStart) {
      ~~var handle = a.TwitterHandle;~~
      if (handle != null)
        result.Add(handle);
     }
    return result;
  }
```

Next in the loop as another conditional, which again I can move to a filter operation.

class Author...

```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author> 
authors, string company) {
    var result = new List<String> ();
    var loopStart = authors
      .Where(a => a.Company == company)
      .Select(a => a.TwitterHandle)
      .Where (h => h != null);
    foreach (string handle in loopStart) {
      if (handle != null)
        result.Add(handle);
    }
    return result;
  }
```

All the loop now does is add everything in its loop collection into the result collection, so I can remove the loop and just return the pipeline result.

class Author...

```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author> 
authors, string company) {
    var result = new List<String> ();
    return authors
      .Where(a => a.Company == company)
      .Select(a => a.TwitterHandle)
      .Where (h => h != null);
    foreach (string handle in loopStart) {
      result.Add(handle);
    }
    return result;
  }
```

Here's the final state of the code

class Author...

```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author> 
authors, string company) {
    return authors
      .Where(a => a.Company == company)
      .Select(a => a.TwitterHandle)
      .Where (h => h != null);
  }
```

What I like about collection pipelines is that I can see the flow of logic as the elements of the list pass through the pipeline. For me it reads very closely to how I'd define the outcome of the loop "take the authors, choose those who have a company, and get their twitter handles removing any null handles".

Furthermore, this style of code is familiar even in different languages who have different syntaxes and different names for pipeline operators. [3]

Java

```
  public List<String> twitterHandles(List<Author> authors, String company) {
    return authors.stream()
            .filter(a -> a.getCompany().equals(company))
            .map(a -> a.getTwitterHandle())
            .filter(h -> null != h)
            .collect(toList());
  }
```

Ruby

```
  def twitter_handles authors, company
    authors
      .select {|a| company == a.company}
      .map    {|a| a.twitter_handle}
      .reject {|h| h.nil?}
  end
```

while this matches the other examples, I would replace the final `reject` with `compact`

Clojure

```
  (defn twitter-handles [authors company]
    (->> authors
         (filter #(= company (:company %)))
         (map :twitter-handle)
         (remove nil?)))
```

F#

```
  let twitterHandles (authors : seq<Author>, company : string) =
       authors
           |> Seq.filter(fun a -> a.Company = company)
           |> Seq.map(fun a -> a.TwitterHandle)
           |> Seq.choose (fun h -> h)
```

again, if I wasn't concerned about matching the structure of the other examples I would combine the map and choose into a single step

I've found that once I got used to thinking in terms of pipelines I could apply them quickly even in an unfamiliar language. Since the fundamental approach is the same it's relatively easy to translate from even unfamiliar syntax and function names.

Refactoring within the pipeline, and to a comprehension

Once you have some behavior expressed as a pipeline, there are potential refactorings you can do by reordering steps in the pipeline. One such move is that if you have a map followed by a filter, you can usually move the filter before the map like this.

class Author...
```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author>
authors, string company) {
    return authors
      .Where(a => a.Company == company)
      .Where (a => a.TwitterHandle != null)
      .Select(a => a.TwitterHandle);
  }
```

When you have two adjacent filters, you can combine them using a conjunction. [4]

class Author...
```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author>
authors, string company) {
    return authors
      .Where(a => a.Company == company && a.TwitterHandle != null)
      .Select(a => a.TwitterHandle);
  }
```

Once I have a C# collection pipeline in the form of a simple filter and map like this, I can replace it with a Linq expression

class Author...
```
  static public IEnumerable<String> TwitterHandles(IEnumerable<Author>
authors, string company) {
    return from a in authors where a.Company == company &&
a.TwitterHandle != null select a.TwitterHandle;
  }
```

I consider Linq expressions to be a form of list comprehension, and similarly you can do something like this with any language that supports list comprehensions. It's a matter of taste whether you prefer the list comprehension form, or the pipeline form (I prefer pipelines). In general pipelines are more powerful, in that you can't refactor all pipelines into comprehensions.

Nested loop - readers of books

For a second example, I'll refactor a simple, doubly nested loop. I'll assume I have a online system that allows readers to read books. I have a data service that will tell me which books each reader has read during a particular day. This service returns a hash whose keys are identifiers of readers and values are a collection of identifiers of books

interface DataService…
```
  Map<String, Collection<String>> getBooksReadOn(Date date);
```

for this example, I'll switch to Java because I'm sick of method names with a capitalized first letter

Here's the loop

```
public Set<String> getReadersOfBooks(Collection<String> readers,
Collection<String> books, Date date) {
  Set<String> result = new HashSet<>();
  Map<String, Collection<String>> data =
dataService.getBooksReadOn(date);
  for (Map.Entry<String, Collection<String>> e : data.entrySet()) {
    for (String bookId : books) {
      if (e.getValue().contains(bookId) && readers.contains(e.getKey()))
{
          result.add(e.getKey());
      }
    }
  }
  return result;
}
```

I'll use my usual first step, which is to apply Extract Variable to the loop collection

```
public Set<String> getReadersOfBooks(Collection<String> readers,
Collection<String> books, Date date) {
  Set<String> result = new HashSet<>();
  Map<String, Collection<String>> data =
dataService.getBooksReadOn(date);
  final Set<Map.Entry<String, Collection<String>>> entries =
data.entrySet();
  for (Map.Entry<String, Collection<String>> e : entries) {
    for (String bookId : books) {
      if (e.getValue().contains(bookId) && readers.contains(e.getKey()))
{
         result.add(e.getKey());
      }
    }
  }
  return result;
}
```

Moves like this make me so glad that IntelliJ's automated refactoring saves me from typing that gnarly type expression.

Once I've got the initial collection into a variable, I can work on elements of the loop behavior. All the work is going on inside the conditional so I'll begin with the second clause in that conditional and move its logic to a filter.

```
public Set<String> getReadersOfBooks(Collection<String> readers,
Collection<String> books, Date date) {
  Set<String> result = new HashSet<>();
  Map<String, Collection<String>> data =
dataService.getBooksReadOn(date);
  final Set<Map.Entry<String, Collection<String>>> entries =
data.entrySet().stream()
          .filter(e -> readers.contains(e.getKey()))
          .collect(Collectors.toSet());
  for (Map.Entry<String, Collection<String>> e : entries) {
    for (String bookId : books) {
      if (e.getValue().contains(bookId)~~ && ~~
~~readers.contains(e.getKey()))~~) {
        result.add(e.getKey());
      }
    }
  }
  return result;
}
```

With Java's streams library, the pipeline has to end with a terminal (such as a collector)

The other clause is more tricky to move since it refers to the inner loop variable. This clause is testing to see if the value of the map entry contains any book that's also in the list of books in the method parameter. I can do this by using a set intersection. The java core classes don't include a set intersection method. I can get by using one of the common collection oriented add-ins to Java such as Guava or Apache Commons. Since this is a simple pedagogical example I'll write a crude implementation.

class Utils...

```
  public static <T> Set<T> intersection (Collection<T> a, Collection<T>
b) {
    Set<T> result = new HashSet<T>(a);
    result.retainAll(b);
    return result;
  }
```

This works here, but for any substantial project, I'd use a common library.

Now I can move the clause from the loop into the pipeline.

```
public Set<String> getReadersOfBooks(Collection<String> readers,
Collection<String> books, Date date) {
  Set<String> result = new HashSet<>();
  Map<String, Collection<String>> data =
dataService.getBooksReadOn(date);
  final Set<Map.Entry<String, Collection<String>>> entries =
data.entrySet().stream()
          .filter(e -> readers.contains(e.getKey()))
          .filter(e -> !Utils.intersection(e.getValue(),
books).isEmpty())
          .collect(Collectors.toSet());
  for (Map.Entry<String, Collection<String>> e : entries) {
    for (String bookId : books) {
      ~~if (e.getValue().contains(bookId) ) {~~
        result.add(e.getKey());
      ~~}~~
    }
  }

  return result;
}
```

Now all the loop is doing is returning the key of map entry, so I can kill the remainder of the loop by adding a map operation to the pipeline

```
public Set<String> getReadersOfBooks(Collection<String> readers,
Collection<String> books, Date date) {
  Set<String> result = new HashSet<>();
  Map<String, Collection<String>> data =
dataService.getBooksReadOn(date);
  result = data.entrySet().stream()
          .filter(e -> readers.contains(e.getKey()))
          .filter(e -> !Utils.intersection(e.getValue(),
books).isEmpty())
          .map(e -> e.getKey())
          .collect(Collectors.toSet());
  ~~for (Map.Entry<String, Collection<String>> e : entries) {~~
    ~~for (String bookId : books) {~~
      ~~result.add(e.getKey());~~
    ~~}~~
  ~~}~~

  return result;
}
```

Then I can use Inline Temp on `result`.

```
public Set<String> getReadersOfBooks(Collection<String> readers,
Collection<String> books, Date date) {
  ~~Set<String> result = new HashSet<>();~~
  Map<String, Collection<String>> data =
dataService.getBooksReadOn(date);
  return data.entrySet().stream()
          .filter(e -> readers.contains(e.getKey()))
          .filter(e -> !Utils.intersection(e.getValue(),
books).isEmpty())
          .map(e -> e.getKey())
          .collect(Collectors.toSet());
  ~~return result;~~
 }
```

Looking that use of intersection, I find it's rather complicated, I have to figure out what it does when I read it - which means I should extract it. [5]

class Utils...

```
  public static <T> boolean hasIntersection(Collection<T> a,
Collection<T> b) {
    return !intersection(a,b).isEmpty();
  }
```

class Service...

```
  public Set<String> getReadersOfBooks(Collection<String> readers,
Collection<String> books, Date date) {
    Map<String, Collection<String>> data =
dataService.getBooksReadOn(date);
    return data.entrySet().stream()
            .filter(e -> readers.contains(e.getKey()))
            .filter(e -> Utils.hasIntersection(e.getValue(), books))
            .map(e -> e.getKey())
            .collect(Collectors.toSet());
   }
```

The object-oriented approach is at a disadvantage with when you need to do something like this. Switching between static utility methods and normal methods on objects is awkward. With some languages I have a way to make it read like a method on the stream class, but I don't have that option in Java.

Despite this issue, I still find the pipeline version easier to comprehend. I could combine the filters into a single conjunction, but I usually find it easier to understand each filter as a single element.

Equipment Offerings

The next bit example uses some simple criteria to mark preferred equipment for particular regions. To understand what it does, I need to explain the domain model. We have an organization that makes equipment available across different regions. When you request some equipment, you may be able to get the exact thing you want, but often you'll end up being offerred a substitute, which will do almost what you want, but perhaps not quite so well. To take a rather stretched example: you are in Boston and want a snow blower, but if there aren't any in the store, you may get offerred a snow-shovel instead. However if you're in Miami, they don't even offer snow blowers, so all you can get is the snow shovel. The data model captures this through three classes.

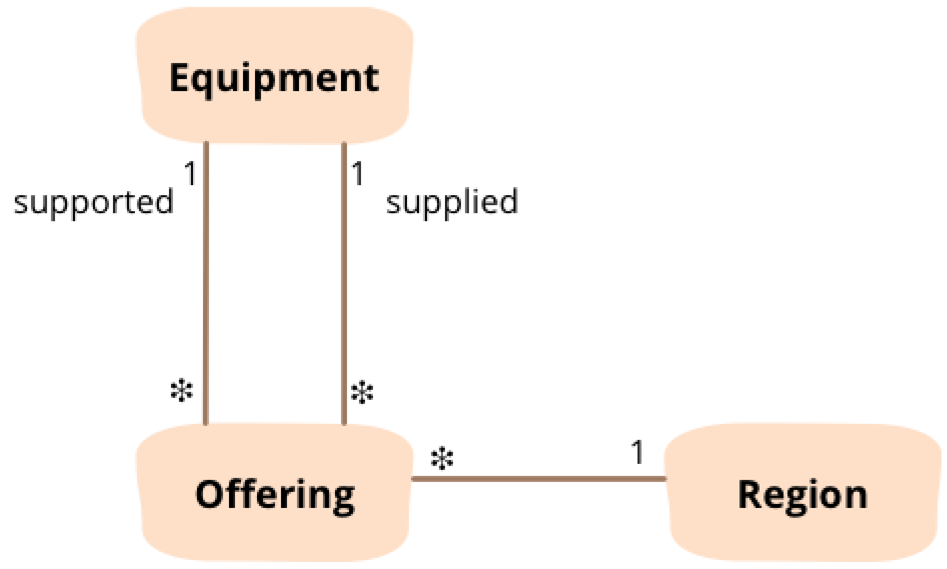

Figure 1: Each instance of offering represents the fact that a particular kind of equipment is offered in a region to support the need for a kind of equipment.

So we might see data like this:

```
products: ['snow-blower', 'snow-shovel']
regions: ['boston', 'miami']
offerings:
  - {region: 'boston', supported: 'snow-blower', supplied: 'snow-
blower'}
  - {region: 'boston', supported: 'snow-blower', supplied: 'snow-
shovel'}
  - {region: 'miami',  supported: 'snow-blower', supplied: 'snow-
shovel'}
```

The code we're going to look at is some code that marks some of these offerings as preferred, meaning that this offering is the preferred offering for some equipment in a region.

class Service...

```
  var checkedRegions = new HashSet<Region>();
  foreach (Offering o1 in equipment.AllOfferings()) {
    Region r = o1.Region;
    if (checkedRegions.Contains(r)) {
      continue;
    }

    Offering possPref = null;
    foreach(var o2 in equipment.AllOfferings(r)) {
      if (o2.isPreferred) {
        possPref = o2;
        break;
      }
      else {
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        }
      }
    }
    possPref.isPreferred = true;
    checkedRegions.Add(r);
  }
```

This example is in C#, because I'm a flip-flopper.

My suspicion is that there's some easily comprehensible logic for what this loop is doing, by refactoring it I can hopefully surface that logic.

My first step is to begin with the outer loop and apply Extract Variable to the initial loop variable.

class Service...

```
  var loopStart = equipment.AllOfferings();
  var checkedRegions = new HashSet<Region>();
  foreach (Offering o1 in loopStart) {
    Region r = o1.Region;
    if (checkedRegions.Contains(r)) {
      continue;
    }

    Offering possPref = null;
    foreach(var o2 in equipment.AllOfferings(r)) {
      if (o2.isPreferred) {
        possPref = o2;
        break;
      }
      else {
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        }
      }
    }
    possPref.isPreferred = true;
    checkedRegions.Add(r);
  }
```

I then look at the first part of the loop. It has a control variable, `checkedRegions` which the loop uses to keep a track of regions it's already processed to avoid processing the same region more than once. That's a smell right there, but it also suggests that the loop variable `o1` is only a stepping stone to the get to the region `r`. I confirm that by highlighting `o1` in my editor. Once I know that, I know I can simplify this with a map.

class Service...

```
  var loopStart = equipment.AllOfferings()
    .Select(o => o.Region);
  var checkedRegions = new HashSet<Region>();
  foreach (Region r in loopStart) {
    ~~Region r = o1.Region;~~
    if (checkedRegions.Contains(r)) {
      continue;
    }

    Offering possPref = null;
    foreach(var o2 in equipment.AllOfferings(r)) {
      if (o2.isPreferred) {
        possPref = o2;
        break;
      }
      else {
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        }
      }
    }
    possPref.isPreferred = true;
    checkedRegions.Add(r);
  }
```

Now I can talk about the `checkedRegions` control variable. The loop uses this to avoid processing a region more than once. It's not clear to me yet whether checking a region is idempotent, if it is I might get away with avoiding this check completely (and measure to see if there is an appreciable impact on performance). Since I'm not sure I decide to retain that logic, especially since avoiding duplicates is trivial with a pipeline.

class Service...

```
  var loopStart = equipment.AllOfferings()
    .Select(o => o.Region)
    .Distinct();
  ~~var checkedRegions = new HashSet<Region>();~~
  foreach (Region r in loopStart) {
    ~~if (checkedRegions.Contains(r)) {~~
      ~~continue;~~
    ~~}~~

    Offering possPref = null;
    foreach(var o2 in equipment.AllOfferings(r)) {
      if (o2.isPreferred) {
        possPref = o2;
        break;
      }
      else {
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        }
      }
    }
    possPref.isPreferred = true;
    ~~checkedRegions.Add(r);~~
  }
```

The next step is about determining the `possPref` variable. I think this will be easier to deal with as its own method, so apply Extract Method

class Service...

```
  var loopStart = equipment.AllOfferings()
    .Select(o => o.Region)
    .Distinct();
  foreach (Region r in loopStart) {
    var possPref = possiblePreference(equipment, r);
    possPref.isPreferred = true;
  }


  static Offering possiblePreference(Equipment equipment, Region region)
{
    Offering possPref = null;
    foreach (var o2 in equipment.AllOfferings(region)) {
      if (o2.isPreferred) {
        possPref = o2;
        break;
      }
      else {
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        }
      }
    }
    return possPref;
  }
```

I extract the loop collection into a variable.

class Service...

```
  static Offering possiblePreference(Equipment equipment, Region region)
{
    Offering possPref = null;
    var allOfferings = equipment.AllOfferings(region);
    foreach (var o2 in allOfferings) {
      if (o2.isPreferred) {
        possPref = o2;
        break;
      }
      else {
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        }
      }
    }
    return possPref;
  }
```

Since the loop is now in its own function, I can use a return rather than a break.

class Service...

```
  static Offering possiblePreference(Equipment equipment, Region region)
{
    Offering possPref = null;
    var allOfferings = equipment.AllOfferings(region);
    foreach (var o2 in allOfferings) {
      if (o2.isPreferred) {
        return o2;
        ~~break;~~
      }
      else {
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        }
      }
    }
    return possPref;
  }
```

The first condition is looking for the first offering, if any, that passes a conditional. That's a job for the detect operation (called `First` in C#.) [6]

class Service...

```
  static Offering possiblePreference(Equipment equipment, Region region)
{
    Offering possPref = null;
    var allOfferings = equipment.AllOfferings(region);
    possPref = allOfferings.FirstOrDefault(o => o.isPreferred);
    if (null != possPref) return possPref;
    foreach (var o2 in allOfferings) {
      ~~if (o2.isPreferred) {~~
        ~~return o2;~~
      ~~}~~
        ~~else {~~
        if (o2.isMatch || possPref == null) {
          possPref = o2;
        ~~}~~
      }
    }
    return possPref;
  }
```

The last conditional is rather tricky. It sets `possPref` to the first offering in the list, but will overwrite that value if any offering passes the `isMatch` test. But the loop doesn't break with that `isMatch` pass, so any later `isMatch` offerings will overwrite that match. So to replicate that behavior, I need to use `LastOrDefault`. [7]

class Service...

```
  static Offering possiblePreference(Equipment equipment, Region region)
{
    Offering possPref = null;
    var allOfferings = equipment.AllOfferings(region);
    possPref = allOfferings.FirstOrDefault(o => o.isPreferred);
    if (null != possPref) return possPref;
    possPref = allOfferings.LastOrDefault(o => o.isMatch);
    if (null != possPref) return possPref;
    foreach (var o2 in allOfferings) {
      if (~~o2.isMatch || ~~possPref == null) {
        possPref = o2;
      }
    }
    return possPref;
  }
```

The last remaining bit of the loop just returns the first item.

class Service...

```
  static Offering possiblePreference(Equipment equipment, Region region)
{
    Offering possPref = null;
    var allOfferings = equipment.AllOfferings(region);
    possPref = allOfferings.FirstOrDefault(o => o.isPreferred);
    if (null != possPref) return possPref;
    possPref = allOfferings.LastOrDefault(o => o.isMatch);
    if (null != possPref) return possPref;
    return allOfferings.First();
    ~~foreach (var o2 in allOfferings) {~~
      ~~if (possPref == null) {~~
        ~~possPref = o2;~~
      ~~}~~
    ~~}~~
    ~~return possPref;~~
  }
```

My personal convention is to use `result` for the name of any variable used for returns in a function, so I rename it.

class Service...

```
  static Offering possiblePreference(Equipment equipment, Region region)
{
    Offering result = null;
    var allOfferings = equipment.AllOfferings(region);
    result = allOfferings.FirstOrDefault(o => o.isPreferred);
    if (null != result) return result;
    result = allOfferings.LastOrDefault(o => o.isMatch);
    if (null != result) return result;
    return allOfferings.First();
  }
```

I'm reasonably happy now with `possiblePreference`, I think it pretty clearly states the logic in a way that makes sense within the domain. I no longer need to figure out what the code is doing in order to understand its intent.

However since I'm in C#, I can make it read even better by using the null-coalescing operator (??). This allows me to chain several expressions together and return the first one that isn't a null.

class Service...

```
  static Offering possiblePreference(Equipment equipment, Region region)
{
    var allOfferings = equipment.AllOfferings(region);
    return allOfferings.FirstOrDefault(o => o.isPreferred)
      ?? allOfferings.LastOrDefault(o => o.isMatch)
      ?? allOfferings.First();
  }
```

In less strictly typed languages that treat null as a falsey value, you do the same thing with an "or" operator. Another alternative is to compose first-class functions (but that's a whole other topic).

Now I go back to the outer loop, which currently looks like this.

class Service...

```
  var loopStart = equipment.AllOfferings()
    .Select(o => o.Region)
    .Distinct();
  foreach (Region r in loopStart) {
    var possPref = possiblePreference(equipment, r);
    possPref.isPreferred = true;
  }
```

I can use my `possiblePreference` in the pipeline.

class Service...

```
  var loopStart = equipment.AllOfferings()
    .Select(o => o.Region)
    .Distinct()
    .Select(r => possiblePreference(equipment, r))
    ;
  foreach (Offering o in loopStart) {
    ~~var possPref = possiblePreference(product, r);~~
    o.isPreferred = true;
  }
```

Note the style of putting the semi-colon on its own line. I often do that with longer pipelines as it makes it easier to manipulate the pipeline.

With renaming the initial loop variable, result reads nicely clear.

class Service...

```
  var preferredOfferings = equipment.AllOfferings()
    .Select(o => o.Region)
    .Distinct()
    .Select(r => possiblePreference(equipment, r))
    ;
  foreach (Offering o in preferredOfferings) {
    o.isPreferred = true;
  }
```

I'd be happy leaving it at that, but I can also move the `forEach` behavior into the pipeline like this.

class Service...

```
  equipment.AllOfferings()
    .Select(o => o.Region)
    .Distinct()
    .Select(r => possiblePreference(equipment, r))
    .ToList()
    .ForEach(o => o.isPreferred = true)
    ;
```

This is a more controversial step. Many people don't like using functions with side-effects in a pipeline. That's why I have to use the intermediate `ToList` since `ForEach` isn't available on `IEnumerable`. As well the issue around side-effects, using `ToList` also reminds us that whenever we use side-effects, we'll also lose any laziness in the evaluation of the pipe (which is not an issue here since the whole point of the pipe is to select some objects for modification).

Either way, however, I find this much clearer than the original loop. The earlier loop examples were reasonably clear to follow, but this one took a some thinking to figure out what it was doing. Certainly extracting `possiblePreference` is a big factor in making it clearer, and you could do that and retain a loop, although I'd certainly want to avoid futzing with the logic to ensure I avoided duplicate regions.

Grouping flight records

With this example I'll look at some code that summarizes flight delay information. The code starts with records of ontime flight performance, originally sourced from U.S. Department of Transportation Bureau of Transportation Statistics. After some preliminary massaging, the resulting data looks something like this

```
[
  {
    "origin":"BOS","dest":"LAX","date":"2015-01-12",
    "number":"25","carrier":"AA","delay":0.0,"cancelled":false
  },
  {
    "origin":"BOS","dest":"LAX","date":"2015-01-13",
    "number":"25","carrier":"AA","delay":0.0,"cancelled":false
  },
 …
```

Here's the loop that processes it

```
export function airportData() {
  const data = flightData();
  const count = {};
  const cancellations = {};
  const totalDelay = {};
  for (let row of data) {
    const airport = row.dest;
    if (count[airport] === undefined) {
      count[airport] = 0;
      cancellations[airport] = 0;
      totalDelay[airport] = 0;
    }
    count[airport]++;
    if (row.cancelled) {
      cancellations[airport]++ ;
    }
    else {
      totalDelay[airport] += row.delay;
    }
  }

  const result = {};
  for (let i in count) {
    result[i] = {};
    result[i].meanDelay = totalDelay[i] / (count[i] - cancellations[i]);
    result[i].cancellationRate = cancellations[i] / count[i];
  }
  return result;
}
```

this example uses Javascript (es6 on node), since everything has to be written in Javascript these days.

The loop summarizes the flight data by destination airport (`dest`) and calculates the cancellation rate and mean delay. The central activity here is grouping the flight data by destination, which is well suited to the group operator in a pipeline. So my first move is to make a variable that captures this grouping.

```
import _ from 'underscore';

export function airportData() {
  const data = flightData();
  const working = _.groupBy(data, r => r.dest);
  const count = {};
  const cancellations = {};
  const totalDelay = {};
  for (let row of data) {
    const airport = row.dest;
    if (count[airport] === undefined) {
      count[airport] = 0;
      cancellations[airport] = 0;
      totalDelay[airport] = 0;
    }
    count[airport]++;
    if (row.cancelled) {
      cancellations[airport]++;
    }
    else {
      totalDelay[airport] += row.delay;
    }
  }

  const result = {};
  for (let i in count) {
    result[i] = {};
    result[i].meanDelay = totalDelay[i] / (count[i] - cancellations[i]);
    result[i].cancellationRate = cancellations[i] / count[i];
  }
  return result;
}
```

A couple of things about this step. Firstly I can't think of a good name for it yet, so I just call it `working`. Secondly, while javascript has a good set of collection pipeline operators on `Array`, it lacks a grouping operator. I could write one myself, but instead I'll make use of the underscore library, which has long been a useful tool in Javascript-land.

The count variable captures how many flight records there are for each destination airport. I can easily calculate this in the pipeline with a map operation.

```
export function airportData() {
  const data = flightData();
  const working = _.chain(data)
      .groupBy(r => r.dest)
      .mapObject((val, key) => {return {count: val.length}})
      .value()
    ;
  const count = {};
  const cancellations = {};
  const totalDelay = {};
  for (let row of data) {
    const airport = row.dest;
    if (count[airport] === undefined) {
      count[airport] = 0;
      cancellations[airport] = 0;
      totalDelay[airport] = 0;
    }
    count[airport]++;
    if (row.cancelled) {
      cancellations[airport]++;
    }
    else {
      totalDelay[airport] += row.delay;
    }
  }
  const result = {};
  for (let i in count) {
    result[i] = {};
    result[i].meanDelay = totalDelay[i] / (working[i].count -
cancellations[i]);
    result[i].cancellationRate = cancellations[i] / working[i].count;
  }
  return result;
}
```

To do a multi-step pipeline like this in underscore, I have to start the pipeline with the `chain` function. This ensures that each step in the pipeline is wrapped inside underscore, so I can use a method chain to build the pipeline. The downside is that I have to use `value` at the end to get the underlying array out of it.

The map operations isn't the standard map, since it operates on the contents of a Javascript object, essentially a hash map, so the mapping function acts on a key/value pair. In underscore I do this with the `mapObject` function.

Usually when I move behavior into the pipeline, I like to remove the control variable entirely, but it's also playing a role in keeping track of the needed keys, so I'll leave it for a while until I've dealt with the other calculations.

Next I'll deal with the cancellations variable, which this time I can remove.

```
export function airportData() {
  const data = flightData();
  const working = _.chain(data)
      .groupBy(r => r.dest)
      .mapObject((val, key) => {
        return {
          count: val.length,
          cancellations: val.filter(r => r.cancelled).length
        }
      })
      .value()
    ;
  const count = {};
  const cancellations = {};
  const totalDelay = {};
  ~~const cancellations = {};~~
  for (let row of data) {
    const airport = row.dest;
```

```
      if (count[airport] === undefined) {
        count[airport] = 0;
        ~~cancellations[airport] = 0;~~
        totalDelay[airport] = 0;
      }
      count[airport]++;
      if (row.cancelled) {
        ~~cancellations[airport]++;~~
      }
      else {
        totalDelay[airport] += row.delay;
      }
    }

    const result = {};
    for (let i in count) {
      result[i] = {};
      result[i].meanDelay = totalDelay[i] / (working[i].count -
working[i].cancellations);
      result[i].cancellationRate = working[i].cancellations /
working[i].count;
    }
    return result;
  }
```

The mapping function is now getting rather long, so I think it's time to use Extract Method on it.

```
export function airportData() {
 const data = flightData();
  const summarize = function(rows) {
    return {
      count: rows.length,
      cancellations: rows.filter(r => r.cancelled).length
    }
  }
  const working = _.chain(data)
    .groupBy(r => r.dest)
    .mapObject((val, key) => summarize(val))
    .value()
    ;

  const count = {};
  const totalDelay = {}
  for (let row of data) {
    const airport = row.dest;
    if (count[airport] === undefined) {
      count[airport] = 0;
      totalDelay[airport] = 0;
    }
    count[airport]++;
    if (row.cancelled) {
    }
    else {
      totalDelay[airport] += row.delay;
    }
  }

  const result = {};
  for (let i in count) {
    result[i] = {};
    result[i].meanDelay = totalDelay[i] / (working[i].count - 
working[i].cancellations);
    result[i].cancellationRate = working[i].cancellations / 
working[i].count;
  }
  return result;
}
```

Assigning the function to a variable within the overall function is javascript's way of nesting the function definition to limit its scope to the `airportData` function. I could imagine this function being more widely useful, but that's a later refactoring to consider.

Now to handle the total delay calculation.

```
export function airportData() {
 const data = flightData();
  const summarize = function(rows) {
    return {
      count: rows.length,
      cancellations: rows.filter(r => r.cancelled).length,
      totalDelay: rows.filter(r => !r.cancelled).reduce((acc,each) => 
acc + each.delay, 0)
    }
  }
  const working = _.chain(data)
    .groupBy(r => r.dest)
    .mapObject((val, key) => summarize(val))
```

```
      .value()
      ;

  const count = {};
  ~~const totalDelay = {}~~
  for (let row of data) {
    const airport = row.dest;
    if (count[airport] === undefined) {
      count[airport] = 0;
      ~~totalDelay[airport] = 0;~~
    }
    count[airport]++;
    ~~if (row.cancelled) {~~
    ~~}~~
    ~~else {~~
      ~~totalDelay[airport] += row.delay;~~
    ~~}~~
  }

  const result = {};
  for (let i in count) {
    result[i] = {};
    result[i].meanDelay = working[i].totalDelay / (working[i].count -
working[i].cancellations);
    result[i].cancellationRate = working[i].cancellations /
working[i].count;
  }
  return result;
}
```

The expression in the lambda for the total delay mirrors the original formulation, using a reduce operation to calculate a sum. I often find it reads better to use a map first.

```
export function airportData() {
  const data = flightData();
  const summarize = function(rows) {
    return {
      count: rows.length,
      cancellations: rows.filter(r => r.cancelled).length,
      totalDelay: rows.filter(r => !r.cancelled).map(r =>
r.delay).reduce((a,b) => a + b)
    }
  }
  const working = _.chain(data)
```

```
    .groupBy(r => r.dest)
    .mapObject((val, key) => summarize(val))
    .value()
    ;

  const count = {};
  for (let row of data) {
    const airport = row.dest;
    if (count[airport] === undefined) {
      count[airport] = 0;
    }
    count[airport]++;
  }

  const result = {};
  for (let i in count) {
    result[i] = {};
    result[i].meanDelay = working[i].totalDelay / (working[i].count -
working[i].cancellations);
    result[i].cancellationRate = working[i].cancellations /
working[i].count;
  }
  return result;
}
```

That reformulation isn't a big deal, but I increasingly prefer it. That lambda is also a bit long, but it's not quite at the point where I feel the need to extract it.

I also took the opportunity to replace the lambda to invoke `summarize` with just naming the function.

```
export function airportData() {
  const data = flightData();
  const summarize = function(rows) {
    return {
      count: rows.length,
      cancellations: rows.filter(r => r.cancelled).length,
      totalDelay: rows.filter(r => !r.cancelled).map(r =>
r.delay).reduce((a,b) => a + b)
    }
  }
  const working = _.chain(data)
    .groupBy(r => r.dest)
    .mapObject(summarize)
```

```
      .value()
      ;

  const count = {};
  for (let row of data) {
    const airport = row.dest;
    if (count[airport] === undefined) {
      count[airport] = 0;
    }
    count[airport]++;
  }

  const result = {};
  for (let i in count) {
    result[i] = {};
    result[i].meanDelay = working[i].totalDelay / (working[i].count - 
working[i].cancellations);
    result[i].cancellationRate = working[i].cancellations / 
working[i].count;
  }
  return result;
}
```

Now with all the dependent data removed, I'm ready to remove `count`.

```
export function airportData() {
  const data = flightData();
  const summarize = function(rows) {
    return {
      count: rows.length,
      cancellations: rows.filter(r => r.cancelled).length,
      totalDelay: rows.filter(r => !r.cancelled).map(r => 
r.delay).reduce((a,b) => a + b)
    }
  }
  const working = _.chain(data)
    .groupBy(r => r.dest)
    .mapObject(summarize)
    .value()
    ;

  ~~const count = {};~~
  ~~for (let row of data) {~~
```

```
    ~~const airport = row.dest;~~
    ~~if (count[airport] === undefined) {~~
      ~~count[airport] = 0;~~
    ~~}~~
    ~~count[airport]++;~~
  ~~}~~

  const result = {};
  for (let i in working) {
    result[i] = {};
    result[i].meanDelay = working[i].totalDelay / (working[i].count -
working[i].cancellations);
    result[i].cancellationRate = working[i].cancellations /
working[i].count;
  }
  return result;
}
```

Now I turn my attention to the second loop, which is essentially doing a map to calculate its two values.

```
export function airportData() {
  const data = flightData();
  const summarize = function(rows) {
    return {
      count: rows.length,
      cancellations: rows.filter(r => r.cancelled).length,
      totalDelay: rows.filter(r => !r.cancelled).map(r =>
r.delay).reduce((a,b) => a + b)
    }
  }
  const formResult = function(row) {
    return {
      meanDelay: row.totalDelay / (row.count - row.cancellations),
      cancellationRate: row.cancellations / row.count
    }
  }
  let working = _.chain(data)
    .groupBy(r => r.dest)
    .mapObject(summarize)
    .mapObject(formResult)
    .value()
    ;
  return working;
  ~~let result = {};~~
  ~~for (let i in working) {~~
    ~~result[i] = {};~~
    ~~result[i].meanDelay = working[i].totalDelay / (working[i].count - 
working[i].cancellations);~~
    ~~result[i].cancellationRate = working[i].cancellations / 
working[i].count;~~
  ~~}~~
  ~~return result;~~
}
```

With all that done, I can use Inline Temp on `working` and do a little more renaming and tidying.

```
export function airportData() {
  const data = flightData();
  const summarize = function(flights) {
    return {
      numFlights:       flights.length,
      numCancellations: flights.filter(f => f.cancelled).length,
      totalDelay:       flights.filter(f => !f.cancelled).map(f => 
f.delay).reduce((a,b) => a + b)
    }
  }
  const formResult = function(airport) {
    return {
      meanDelay:        airport.totalDelay / (airport.numFlights - 
airport.numCancellations),
      cancellationRate: airport.numCancellations / airport.numFlights
    }
  }
  return _.chain(data)
    .groupBy(r => r.dest)
    .mapObject(summarize)
    .mapObject(formResult)
    .value()
    ;
}
```

As is often the case, a good bit of the greater readability of the final function comes from extracting functions. But I find the grouping operator does much to clarify the purpose of the function and helps set up the extractions.

There is another potential benefit from this refactoring if the data comes from a relational database and I'm experiencing performance problems. By refactoring from a loop to a collection pipeline I'm representing the transformation in a form that's much more similar to SQL. In the case of this task I might be pulling a lot of data from the database, but the refactored code makes it easier to consider moving the grouping and first level summarization logic into the SQL, which would reduce how much data I need to ship over the wire. Usually I prefer to keep logic in application code rather than SQL, so I would treat such a move as a performance optimization and only do this if I can measure a significant performance improvement. But this reinforces the point that it's much easier to do optimizations when you have clear code to work with, which is why all the performance wizards I know stress the importance of clarity-first as a foundation for performant code.

Identifiers

For our next example, I'll take a look at some code that checks that a person has a set of required identifiers. It's common for systems to rely on identifying people through some kind of hopefully-unique id, such as a customer id. In many domains, you have to deal with many different identification schemes, and a person should have identifiers with multiple schemes. So a town government might expect a person to have a town id, a state id, and a national id.

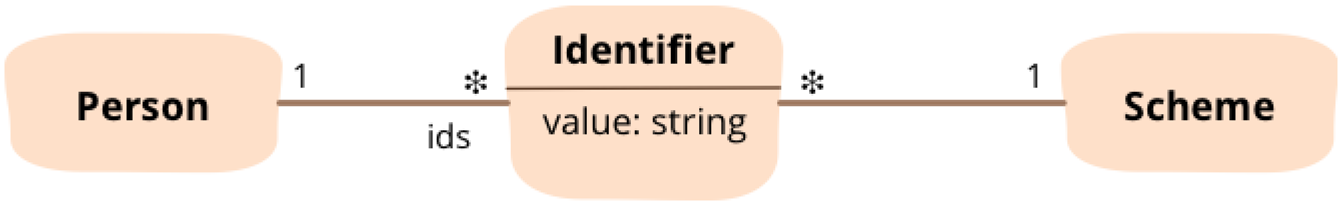

Figure 2: A data model for a person having identifiers with different schemes

The data structure for this situation is pretty simple. The person class has a collection of identifier objects. Identifiers have a field for the scheme and some value. But usually there are further constraints that can't be enforced solely by the data model, such constraints are checked by a validation function such as this.

class Person...
```
  def check_valid_ids required_schemes, note: nil
    note ||= Notification.new
    note.add_error "has no ids"  if @ids.size < 1
    used = []
    found_required = []
    dups = []

    for id in @ids
      next if id.void?
      if used.include?(id.scheme)
        dups << id.scheme
      else
        for req in required_schemes
          if id.scheme == req
            found_required << req
            required_schemes.delete req
            next
          end
        end
      end
      used << id.scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

This example is in Ruby, because I like programming in Ruby

There's a couple of other objects that support this loop. The Identifier class knows its scheme, value, and whether it is void - meaning it's been logically deleted (but is still kept in the database). There is also a notification to keep track of any errors.

```
class Identifier
  attr_reader :scheme, :value
  def void?
    …

class Notification
  def add_error e
    …
```

The biggest smell for me in this routine is that the loop is doing two things at once. It is both finding duplicate identifiers (collected in `dups`) and finding required schemes that are missing (in `required_schemes`). It's quite common that programmers, faced with two things to do with the same collection of objects, decide to do both things in the same loop. One reason is the code required to set up the loop, it seems a shame to write it twice. Modern loop constructs and pipelines remove that burden. The more pernicious reason is concerns over performance. Certainly many performance hotspots involve a loop, and there are cases where loops can be fused to improve matters. But these are only a tiny fraction of all the loops we write, so we should follow the usual principle of programming. Focus on clarity over performance, unless you have a measured, significant performance problem. If you have such a problem, then fixing it takes priority over clarity, but such cases are rare.

> *Focus on clarity over performance, unless you have a measured, significant performance problem.*

So faced with a loop doing two things, I have no hesitation in duplicating the loop to improve clarity. It's extremely rare that performance analysis will cause me to reverse that refactoring.

So my first move is use a refactoring I'll call Split Loop. When I do this I start by getting the loop and the code that's connecting it into a coherent block of code, and apply Extract Method to it.

class Person...

```
  def check_valid_ids required_schemes, note: nil
    note ||= Notification.new
    note.add_error "has no ids"  if @ids.size < 1
    return inner_check_valid_ids required_schemes, note: note
  end
  def inner_check_valid_ids required_schemes, note: nil
    used = []
    found_required = []
    dups = []

    for id in @ids
      next if id.void?
      if used.include?(id.scheme)
        dups << id.scheme
      else
        for req in required_schemes
          if id.scheme == req
            found_required << req
            required_schemes.delete req
            next
          end
        end
      end
      used << id.scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

This extracted method is doing two things, so now I want to duplicate it to form the scaffolding for the two separate methods, each of which will do just one thing. If I duplicate it and call each one, then my accumulating notification will get twice the errors, I can avoid this by removing the irrelevent update from each of the duplicates.

class Person...

```
  def check_valid_ids required_schemes, note: nil
    note ||= Notification.new
    note.add_error "has no ids"  if @ids.size < 1
    check_no_duplicate_ids required_schemes, note: note
    check_all_required_schemes required_schemes, note: note
  end
  def check_no_duplicate_ids required_schemes, note: nil
    used = []
    found_required = []
    dups = []

    for id in @ids
      next if id.void?
      if used.include?(id.scheme)
        dups << id.scheme
      else
        for req in required_schemes
          if id.scheme == req
            found_required << req
            required_schemes.delete req
            next
          end
        end
      end
      used << id.scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      ~~note.add_error "missing schemes: " + missing_names~~
    end

    return note
  end

  def check_all_required_schemes required_schemes, note: nil
    used = []
    found_required = []
    dups = []

    for id in @ids
```

```
        next if id.void?
        if used.include?(id.scheme)
          dups << id.scheme
        else
          for req in required_schemes
            if id.scheme == req
              found_required << req
              required_schemes.delete req
              next
            end
          end
        end
        used << id.scheme
      end

      if dups.size > 0
        ~~note.add_error "duplicate schemes: " + dups.join(", ")~~
      end

      if required_schemes.size > 0
        missing_names = ""
        for req in required_schemes
          missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
        end
        note.add_error "missing schemes: " + missing_names
      end

      return note
    end
```

It's important to remove the double-update, that way my tests all keep passing while I'm refactoring.

The result is rather ugly, but now I can work on each method independently, removing anything that doesn't involve the purpose of each method.

Refactoring the no-duplicates check

I'll start with the no-duplicates case, I can cut out hunks of code in several steps, testing after each one to ensure I don't make a mistake. I start by removing the code that uses `required_schemes` at the end

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    used = []
    found_required = []
    dups = []

    for id in @ids
      next if id.void?
      if used.include?(id.scheme)
        dups << id.scheme
      else
        for req in required_schemes
          if id.scheme == req
            found_required << req
            required_schemes.delete req
            next
          end
        end
      end
      used << id.scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    ~~if required_schemes.size > 0~~
      ~~missing_names = ""~~
      ~~for req in required_schemes~~
        ~~missing_names += (missing_names.size > 0) ? ", " + req.to_s : req.to_s~~
      ~~end~~
    ~~end~~

    return note
  end
```

I then take out the unneeded branch of the conditional

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    used = []
    ~~found_required = []~~
    dups = []
    for id in @ids
      next if id.void?
      if used.include?(id.scheme)
        dups << id.scheme
      ~~else~~
        ~~for req in required_schemes~~
          ~~if id.scheme == req~~
            ~~found_required << req~~
            ~~required_schemes.delete req~~
            ~~next~~
          ~~end~~
        ~~end~~
      end
      used << id.scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    return note
  end
```

At this point I could, and perhaps should, have removed the now unneeded `required_schemes` parameter. I didn't, and you'll see it gets sorted out in the end.

I do my usual Extract Variable

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    used = []
    dups = []

    input = @ids
    for id in input
      next if id.void?
      if used.include?(id.scheme)
        dups << id.scheme
      end
      used << id.scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    return note
  end
```

I can then add a filter to the input variable and remove the line that skips over void identifiers.

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    used = []
    dups = []

    input = @ids.reject{|id| id.void?}
    for id in input
      ~~next if id.void?~~
      if used.include?(id.scheme)
        dups << id.scheme
      end
      used << id.scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    return note
  end
```

Looking further in the loop I can see that it uses the scheme rather than the id, so I can add the pipeline step to map ids to schemes.

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    used = []
    dups = []

    input = @ids
      .reject{|id| id.void?}
      .map {|id| id.scheme}
    for scheme in input
      if used.include?(scheme)
        dups << scheme
      end
      used << scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    return note
  end
```

At this point, I've reduced the loop body to the simple removing duplicates behavior. There is a pipeline way to find duplicates, this is to group the schemes by themselves and filter those that appear more than once.

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    used = []
    dups = []

    input = @ids
      .reject{|id| id.void?}
      .map {|id| id.scheme}
      .group_by {|s| s}
      .select {|k,v| v.size > 1}
      .keys
    for scheme in input
      ~~if used.include?(scheme)~~
        dups << scheme
      ~~end~~
      used << scheme
    end

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    return note
  end
```

Now the output of the pipeline is the duplicates, so I can remove the input variable and assign the pipeline to the variable (and remove the now unneeded `used` variable).

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    ~~used = []~~
    ~~dups = []~~

    dups = @ids
      .reject{|id| id.void?}
      .map {|id| id.scheme}
      .group_by {|s| s}
      .select {|k,v| v.size > 1}
      .keys
    ~~for scheme in input~~
      ~~dups << scheme~~
      ~~used << scheme~~
    ~~end~~

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    return note
  end
```

That gives us a nice pipeline, but there is a troubling element to it. The last three steps in the pipeline are there to remove duplicates, but that knowledge is in my head, not in the code. I need to move it into the code by using Extract Method.

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    dups = @ids
      .reject{|id| id.void?}
      .map {|id| id.scheme}
      .duplicates

    if dups.size > 0
      note.add_error "duplicate schemes: " + dups.join(", ")
    end

    return note
  end
```

class Array…

```
  def duplicates
```

```
    self
      .group_by {|s| s}
      .select {|k,v| v.size > 1}
      .keys
  end
```

Here I've used Ruby's ability to add a method to an existing class (known as monkey-patching). I could also use Ruby's refinement feature in the latest Ruby versions. However many OO languages don't support monkey-patching, in which case I have to use a locally defined function, along the lines of this

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    schemes = @ids
      .reject{|id| id.void?}
      .map {|id| id.scheme}

    if duplicates(schemes).size > 0
      note.add_error "duplicate schemes: " + duplicates(schemes).join(", 
")
    end

    return note
  end
  def duplicates anArray
    anArray
      .group_by {|s| s}
      .select {|k,v| v.size > 1}
      .keys
  end
```

Defining a method on person doesn't work so well for the pipeline as putting it on the array. But often we can't put a method on the array because our language doesn't involve monkey patching, or project standards don't make it easy, or it's a method that isn't generic enough to sit on a generic list class. This is a case where the object-oriented approach gets in the way and a functional approach, that doesn't bind methods to objects, works better.

Whenever I have a local variable like this, I always consider using Replace Temp with Query to turn the variable into a method - resulting in something like this.

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    if duplicate_identity_schemes.size > 0
      note.add_error "duplicate schemes: " + 
duplicate_identity_schemes.join(", ")
    end

    return note
  end
  def duplicate_identity_schemes
     @ids
      .reject{|id| id.void?}
      .map {|id| id.scheme}
      .duplicates
  end
```

I base this decision on whether I think the `duplicate_identity_schemes` behavior is likely to be useful to other methods in the person class. But despite the fact that I prefer to err in making the query method, for this case I prefer to keep it as a local variable.

Refactoring the check for all required schemes

Now I've cleaned up the check for no duplicates I can work on the check that we have all the required schemes. Here's the current method.

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    used = []
    found_required = []
    dups = []

    for id in @ids
      next if id.void?
      if used.include?(id.scheme)
        dups << id.scheme
      else
        for req in required_schemes
          if id.scheme == req
            found_required << req
            required_schemes.delete req
            next
          end
        end
      end
      used << id.scheme
    end

    if dups.size > 0
    end

    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

As with the earlier method, my first steps are to remove anything that's to do with checking for duplicates.

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    ~~used = []~~
    found_required = []
    ~~dups = []~~

    for id in @ids
      next if id.void?
      ~~if used.include?(id.scheme)~~
        ~~dups << id.scheme~~
      ~~else~~
        for req in required_schemes
          if id.scheme == req
            found_required << req
            required_schemes.delete req
            next
          end
        end
      ~~end~~
      ~~used << id.scheme~~
    end
    ~~if dups.size > 0~~
    ~~end~~

    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

To get into the meat of this function, I start by looking at the `found_required` variable. As with the checking for duplicates case, it's primarily interested in the schemes for which we have non-void identifiers, so I'm inclined to start by capturing the schemes into a variable and using those rather than the ids.

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    found_required = []
    schemes = @ids
      .reject{|i| i.void?}
      .map {|i| i.scheme}

    for s in schemes
      ~~next if id.void?~~
      for req in required_schemes
        if s == req
          found_required << req
          required_schemes.delete req
          next
        end
      end
    end


    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
```

```
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

The purpose of `found_required` is to capture schemes that are both in the `required_schemes` list and among the schemes from our ids. That sounds like a set intersection to me, and that's a function I should expect on any self-respecting collection. So I should be able to determine found_required with that.

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    ~~found_required = []~~
    schemes = @ids
      .reject{|i| i.void?}
      .map {|i| i.scheme}

    ~~for s in schemes~~
      ~~for req in required_schemes~~
        ~~if s == req~~
          ~~found_required << req~~
          ~~required_schemes.delete req~~
          ~~next~~
        ~~end~~
      ~~end~~
    ~~end~~
    found_required = schemes & required_schemes


    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

Sadly that move failed the tests. Now I look harder at the code and realize that `found_required` isn't used at all by the code later on, it's a zombie variable that was probably used for something once, but that use was abandoned later and the variable was never removed from the code. So I back out the change I just made and remove it.

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    ~~found_required = []~~
    schemes = @ids
      .reject{|i| i.void?}
      .map {|i| i.scheme}

    for s in schemes
      for req in required_schemes
        if s == req
          ~~found_required << req~~
          required_schemes.delete req
          next
        end
      end
    end


    if required_schemes.size > 0
      missing_names = ""
      for req in required_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

Now I see that the loop is removing elements from the parameter `required_schemes`. Modifying a parameter like this is a strict no-no for me, unless it's a collecting parameter (such as the note). So I immediately apply Remove Assignments to Parameters

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    missing_schemes = required_schemes.dup
    schemes = @ids
```

```
      .reject{|i| i.void?}
      .map {|i| i.scheme}

    for s in schemes
      for req in required_schemes
        if s == req
          missing_schemes.delete req
          next
        end
      end
    end


    if missing_schemes.size > 0
      missing_names = ""
      for req in missing_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s : 
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

Doing this also revealed that the loop was deleting items from the list it's enumerating - an even worse thing than modifying the parameter.

Now that this is clarified, I can see that a set operation is appropriate, but what I need to do is remove the schemes we have from the required list - using a set difference operation.

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    ~~missing_schemes = required_schemes.dup~~
    schemes = @ids
      .reject{|i| i.void?}
      .map {|i| i.scheme}

    missing_schemes = required_schemes - schemes

    ~~for s in schemes~~
      ~~for req in required_schemes~~
        ~~if s == req~~
```

```
          ~~missing_schemes.delete req~~
          ~~next~~
        ~~end~~
      ~~end~~
    ~~end~~



    if missing_schemes.size > 0
      missing_names = ""
      for req in missing_schemes
        missing_names += (missing_names.size > 0) ? ", " + req.to_s :
req.to_s
      end
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

Now I look at the second loop, forming the error message. This just converts the schemes to string and joins them with commas - which is the work of the string join operation.

class Person...

```
  def check_all_required_schemes required_schemes, note: nil
    schemes = @ids
      .reject{|i| i.void?}
      .map {|i| i.scheme}
    missing_schemes = required_schemes - schemes

    if missing_schemes.size > 0
      missing_names = missing_schemes.join(", ")
      ~~for req in missing_schemes~~
        ~~missing_names += (missing_names.size > 0) ? ", " + req.to_s :~~
~~req.to_s~~
      ~~end~~
      note.add_error "missing schemes: " + missing_names
    end

    return note
  end
```

Consolidating the two methods

Both methods are cleaned up now, they do just one thing and are clear in what they are doing. Both of them need the list of schemes for non-void identifiers, so I'm inclined to use Extract Method

class Person...

```
def check_no_duplicate_ids required_schemes, note: nil
  ~~dups = @ids~~
    ~~.reject{|id| id.void?}~~
    ~~.map {|id| id.scheme}~~
    ~~.duplicates~~
  dups = identity_schemes.duplicates
  if dups.size > 0
    note.add_error "duplicate schemes: " + dups.join(", ")
  end
  return note
end


def check_all_required_schemes required_schemes, note: nil
  ~~schemes = @ids~~
    ~~.reject{|i| i.void?}~~
    ~~.map {|i| i.scheme}~~
  missing_schemes = required_schemes - identity_schemes
  if missing_schemes.size > 0
    missing_names = missing_schemes.join(", ")
    note.add_error "missing schemes: " + missing_names
  end
  return note
end

def identity_schemes
  @ids
    .reject{|i| i.void?}
    .map {|i| i.scheme}
end
```

I then fancy a couple of small clean-ups. Firstly I'm testing to see if a collection is empty by checking its size. I always prefer a more intention-revealing empty method.

class Person...

```
def check_no_duplicate_ids required_schemes, note: nil
  dups = identity_schemes.duplicates
```

```
    unless dups.empty?
      note.add_error "duplicate schemes: " + dups.join(", ")
    end
    return note
  end

  def check_all_required_schemes required_schemes, note: nil
    missing_schemes = required_schemes - identity_schemes
    unless missing_schemes.empty?
      missing_names = missing_schemes.join(", ")
      note.add_error "missing schemes: " + missing_names
    end
    return note
  end
```

I don't have a name for this refactoring, it should be something like "replace implementation-revealing method with intention-revealing method".

The `missing_names` variable isn't pulling its weight, so I'll use Inline Temp on it.

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    dups = identity_schemes.duplicates
    unless dups.empty?
      note.add_error "duplicate schemes: " + dups.join(", ")
    end
    return note
  end

  def check_all_required_schemes required_schemes, note: nil
    missing_schemes = required_schemes - identity_schemes
    unless missing_schemes.empty?
      ~~missing_names = missing_schemes.join(", ")~~
      note.add_error "missing schemes: " + missing_schemes.join(", ")
    end
    return note
  end
```

I also fancy converting both of these to use the single line conditional syntax

class Person...

```
  def check_no_duplicate_ids required_schemes, note: nil
    dups = identity_schemes.duplicates
    ~~unless dups.empty?~~
    note.add_error "duplicate schemes: " + dups.join(", ") unless
dups.empty?
    ~~end~~
    return note
  end

  def check_all_required_schemes required_schemes, note: nil
    missing_schemes = required_schemes - identity_schemes
    ~~unless missing_schemes.empty?~~
    note.add_error "missing schemes: " + missing_schemes.join(", ")
unless missing_schemes.empty?
    ~~end~~
    return note
  end
```

Again there's no defined refactoring for that, and it would be very much a ruby-specific one.

With that, I don't think the methods are worthwhile any more, so I inline them, and the extracted `identity_schemes` method, back into caller

class Person...

```
  def check_valid_ids required_schemes, note: nil
    note ||= Notification.new
    note.add_error "has no ids"  if @ids.size < 1
    identity_schemes = @ids.reject{|i| i.void?}.map {|i| i.scheme}
    dups = identity_schemes.duplicates
    note.add_error("duplicate schemes: " + dups.join(", ")) unless
dups.empty?
    missing_schemes = required_schemes - identity_schemes
    note.add_error "missing schemes: " + missing_schemes.join(", ")
unless missing_schemes.empty?
    return note
  end
```

The final method is a bit longer than I usually go for, but I like its cohesiveness. If it grew much bigger I'd want to split it up, perhaps using Replace Method with Method Object Even as long as it is, I find it much clearer in communicating what errors the validation is checking for.

Final Thoughts

That concludes this set of refactoring examples. I hope it's given you a good sense of how collection pipelines can clarify the logic of code that manipulates a collection, and how it's often quite straightforward to refactor a loop into a collection pipeline.

As with any refactoring, there is a similar inverse refactoring to turn a collection pipeline into a loop, but I hardly ever do that.

These days most modern languages offer first class functions and a collection library that includes the necessary operations for collection pipelines. If you're unused to collection pipelines, it is a good exercise to take loops that you run into and refactor them like this. If you find the final pipeline isn't clearer than the original loop, you can always revert the refactoring when you're done. Even if you do revert the refactoring, the exercise can teach you a lot about this technique. I've been using this programming pattern for a long time and find it a valuable way of helping me read my own code. As such I think it's worth the effort of exploring, to see if your team comes to a similar conclusion.

Footnotes

1: Well, actually, my first move is to consider applying Extract Method on the loop, as it's often easier to manipulate a loop if its isolated into its own function.

2: For me, it's odd to see the map operator called "Select". The reason is that the pipeline methods in C# come from Linq, whose main aim is abstracting database access, so the method names were chosen to be similar to SQL. "Select" is the projection operator in SQL, which makes sense when you think of it as selecting columns, but is an odd name if you think of it of using a function mapping.

3: Of course this is not a comprehensive list of all the languages that can do collection pipelines, so I expect the usual rush of complaints that I'm not showing Javascript, Scala, or Whatever++. I don't want a huge list of languages here, just a small set that's varied enough to convey the notion that collection pipelines are easy to follow in an unfamiliar language.

4: in some cases it will need to be a short-circuiting one, although that's not the case here

5: I often find this with negative booleans. This is because the negation (`!`) is at the start of the expression, while the predicate (`isEmpty`) is at the end. With any substantive expression between the two, the result is hard to parse. (At least for me.)

6: I haven't put this in the operator catalog yet.

7: If I'm using a language that doesn't have an operation that detects the last item that passes a predicate, I can reverse it first and then detect the first one.

Acknowledgements

Kit Eason helped make the F# example a bit more idiomatic. Les Ramer helped me improve my C#. Richard Warburton corrected some loose wording in my Java. Daniel Sandbecker spotted an error in the ruby example. Andrew Kiellor, Bruno Trecenti, David Johnston, Duncan Cragg, Karel Alfonso, Korny Sietsma, Matteo Vaccari, Pete Hodgson, Piyush Srivastava, Scott Robinson, Steven Lowe, and Vijay Aravamudhan discussed drafts of this article on the ThoughtWorks mailing list.

Significant Revisions

14 July 2015: Added the identifiers example and final thoughts

07 July 2015: Added the grouping flight data example

30 June 2015: Added the equipment offering example.

25 June 2015: Added the nested loop installment

23 June 2015: Published first installment