# Refactoring Module Dependencies

*As a program grows in size it's important to split it into modules, so that you don't need to understand all of it to make a small modification. Often these modules can be supplied by different teams and combined dynamically. In this refactoring essay I split a small program using Presentation-Domain-Data layering. I then refactor the dependencies between these modules to introduce the Service Locator and Dependency Injection patterns. These apply in different languages, yet look different, so I show these refactorings in both Java and a classless JavaScript style.*

---

13 October 2015

**Martin Fowler**

Find **similar articles** to this by looking at these tags: refactoring · API design · application architecture

---

As programs go larger than a few hundred lines of code, you need to think about how to split them up into modules. At the very least it's useful to have smaller files to better manage your editing. But more seriously you want to divide up your program so that you don't have to keep it all in your head in order to make changes.

A well designed modular structure should allow you to only understand a small part of a larger program when you need to make a small change to it. Sometimes a small change will cross-cut over the modules, but most of the time you'll just need to understand a single module and its neighbors.

The hardest part of splitting a program into modules is just deciding on what the module boundaries should be. There's no easy guidelines to follow for this, indeed a major theme of my life's work is to try and understand what good module boundaries will look

like. Perhaps the most important part of drawing good module boundaries is paying attention to the changes you make and refactoring your code so that code that changes together is in the same or nearby modules.

On top of this is the mechanics of making the separation of how the various parts relate to each other. In the simplest case you have client modules that call suppliers. But often the configuration of these clients and suppliers can get tangled because you don't always want the client program to know too much about how its suppliers fit together.

I'm going to explore this problem with an example, where I'll take a hunk of code and see how it can be split into pieces. In fact I'm going to do this twice, using two different languages: Java and JavaScript, which despite their similar names are really very different when it comes to the affordances they have for modularity.

## The Starting Point(s)

We begin with a startup that is doing sophisticated data analysis of sales data. They have this valuable indicator, the Gondorff number, that is an extremely useful predictor for sales of products. Their web application takes a company's sales data, feeds it into their sophisticated algorithm, and then prints a simple table of products and their Gondorff numbers.

The code for the initial state is all in a single file, which I'll walk through in sections. First is the code that emits the table in HTML.

*app.js*

```
  function emitGondorff(products) {
    function line(product) {
      return [
        `  <tr>`,
        `    <td>${product}</td>`,
        `    <td>${gondorffNumber(product).toFixed(2)}</td>`,
        `  </tr>`].join('\n');
    }
    return
encodeForHtml(`<table>\n${products.map(line).join('\n')}\n</table>`);
  }
```

> *I don't use multi-line strings as the demands of indentation in the output don't line up with indentation in the source code.*

This isn't the worlds most sophisticated UI, positively pedestrian in a world of single page this and responsive that. The only important thing, for this example, is that the UI needs to call the `gondorffNumber` function at various points.

> **JavaScript Style**
>
> For this example I'm going to use the ES6 standard of JavaScript, which provides a number of valuable advantages over older versions of the language. I'm also going to use a classless style of JavaScript, primarily as this shows a greater contrast to Java.
>
> (This doesn't mean I don't like using classes in JavaScript - I do - but avoiding classes allows me to illustrate how these patterns play out without them.)

Next I'll move over to the calculation of the gondorff number.

*app.js*

```
function gondorffNumber(product) {
  return salesDataFor(product, gondorffEpoch(product), hookerExpiry())
      .find(r => r.date.match(/01$/))
      .quantity * Math.PI
    ;
}
function gondorffEpoch(product) {
  const countingBase = recordCounts(baselineRange(product));
  return deriveEpoch(countingBase);
}


function baselineRange(product){
  // redacted
}
function deriveEpoch(countingBase) {
  // redacted
}
function hookerExpiry() {
  // redacted
}
```

That may not look like a million-dollar algorithm to us, but that's thankfully not the important part of this code. The important part is that this logic that's about calculating the gondorff number requires two functions (`salesDataFor` and `recordCounts`) that simply return basic data from some kind of data source of sales. These data source functions are not particularly sophisticated, they merely filter some data sourced from a

CSV file.

*app.js*

```
  function salesDataFor(product, start, end) {
    return salesData()
      .filter(r =>
        (r.product === product)
        && (new Date(r.date) >= start)
        && (new Date(r.date) < end)
      );
  }
  function recordCounts(start) {
    return salesData()
      .filter(r => new Date(r.date) >= start)
      .length
  }
  function salesData() {
    const data = readFileSync('sales.csv', {encoding: 'utf8'});
    return data
      .split('\n')
      .slice(1)
      .map(makeRecord)
      ;
  }
  function makeRecord(line) {
    const [product,date,quantityString,location] =
line.split(/\s*,\s*/);
    const quantity =  parseInt(quantityString, 10);
    return { product, date, quantity, location };
  }
```

These functions are entirely boring as far as this discussion's concerned - I show them only out of a sense of completeness. The important thing about them is that they take data from some data source, massage it into simple objects, and provide it in two different flavors to the core algorithmic code.

At this point the java version looks very similar, first the HTML generation.

*class App...*

```
  public String emitGondorff(List<String> products) {
    List<String> result = new ArrayList<>();
    result.add("\n<table>");
    for (String p : products)
      result.add(String.format("  <tr><td>%s</td><td>%4.2f</td></tr>",
```

```
p, gondorffNumber(p)));
    result.add("</table>");
    return
HtmlUtils.encode(result.stream().collect(Collectors.joining("\n")));
  }
```

The gondorff algorithm

*class App...*

```
  public double gondorffNumber(String product) {
    return salesDataFor(product, gondorffEpoch(product), hookerExpiry())
            .filter(r -> r.getDate().toString().matches(".*01$"))
            .findFirst()
            .get()
            .getQuantity() * Math.PI
            ;
  }
  private LocalDate gondorffEpoch(String product) {
    final long countingBase = recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }


  private LocalDate baselineRange(String product) {
    //redacted
  }
  private LocalDate deriveEpoch(long base) {
    //redacted
  }
  private LocalDate hookerExpiry() {
    // yup, redacted too
  }
```

Since the body of the data source code isn't that important, I'll just show the method declarations

*class App*

```
  private Stream<SalesRecord> salesDataFor(String product, LocalDate
start, LocalDate end) {
    // unimportant details
  }
  private long recordCounts(LocalDate start) {
    // unimportant details
```

```
}
```

## Presentation-Domain-Data Layering

I said earlier that setting module boundaries was a subtle and nuanced art, but one guideline that many people follow is Presentation-Domain-Data Layering - separating presentation code (UI), business logic, and data access. There are good reasons for following this kind of split. Each of those three categories involve thinking about different concerns, and often use different frameworks to assist in the task. Furthermore there is also a desire for substitution - multiple presentations using the same core business logic, or the business logic using different data sources in different environments.

So for this example I'm going to follow this common split, and I'll also stress the substitution justification. After all this gondorff number is such a valuable metric that many people will want to make use of it - encouraging me to package it as a unit that can easily be reused by multiple applications. Furthermore not all applications will keep their sales data in a csv file, some will use a database or a remote microservice. We want an application developer to be able to take the gondorff code and plug it into her specific data source, which she may write herself or get from yet another developer.

But before we embark on the refactoring to enable all this, I do need to stress that presentation-domain-data layering does have its limitations. The general rule of modularity is that we want to confine the consequences of change to one module if we can. But separate presentation-domain-data modules often do have to change together. The simple act of adding a data field will usually cause all three to update. As a result I favor using this approach in smaller scopes, but larger applications need high level modules to be developed along different lines. In particular you shouldn't use the presentation-domain-data layers as a basis for team boundaries.

## Performing the split

I'll begin splitting into modules by separating the presentation. For the JavaScript case, this is almost merely cutting and pasting code into a new file.

*gondorff.es6*

```
  export default function gondorffNumber …
```

```
  function gondorffEpoch(product) {…
  function baselineRange(product){…
  function deriveEpoch(countingBase) { …
  function hookerExpiry() { …
  function salesDataFor(product, start, end) { …
  function recordCounts(start) { …
  function salesData() { …
  function makeRecord(line) { …
```

> **ES6 Modules**
>
> I'm using the facilities for modules that are part of ECMAScript 6, which became settled as I was writing this. I found Axel Rauschmayer's book, Exploring ES6, very helpful in understanding how these features worked.

By using `export default` I can import the reference to `gondorffNumber` and I only have to add an import statement.

*app.es6*

```
  import gondorffNumber from './gondorff.es6'
```

On the java side, it's almost as straightforward. Again I copy everything other than `emitGondorff` over to a new class.

*class Gondorff…*

```
  public double gondorffNumber(String product) { …
  private LocalDate gondorffEpoch(String product) { …
  private LocalDate baselineRange(String product) { …
  private LocalDate deriveEpoch(long base) { …
  private LocalDate hookerExpiry() { …

  Stream<SalesRecord> salesDataFor(String product, LocalDate start,
LocalDate end) { …
  long recordCounts(LocalDate start) {…
  Stream<SalesRecord> salesData() { …
  private SalesRecord makeSalesRecord(String line) { …
```

For the original App class I don't need an import unless I put the new class into a new package, but I do need to instantiate the new class.

*class App...*

```
  public String emitGondorff(List<String> products) {
    List<String> result = new ArrayList<>();
    result.add("\n<table>");
    for (String p : products)
      result.add(String.format("  <tr><td>%s</td><td>%4.2f</td></tr>",
p, new Gondorff().gondorffNumber(p)));
    result.add("</table>");
    return
HtmlUtils.encode(result.stream().collect(Collectors.joining("\n")));
  }
```

I now want to do second separation between the calculation logic and the code that offers up the data records.

*dataSource.es6…*

```
  export function salesDataFor(product, start, end) {


  export function recordCounts(start) {


  function salesData() { …
  function makeRecord(line) { …
```

A difference between this move and the earlier one is that gondorff file needs to import two functions rather than just one. It can do that with this import, nothing else needs to change.

*Gondorff.es6…*

```
  import {salesDataFor, recordCounts} from './dataSource.es6'
```

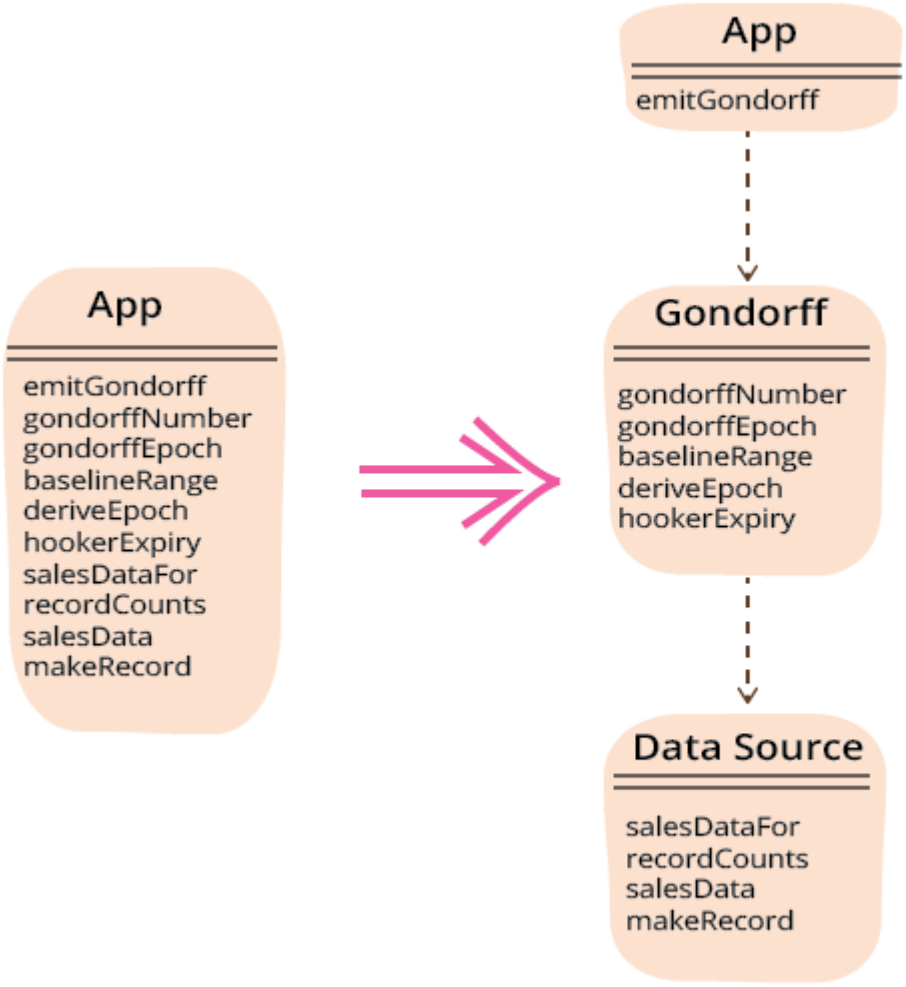
The java version is very similar to the previous case, move into a new class, and instantiate the class for a new object.

*class DataSource…*

```
  public Stream<SalesRecord> salesDataFor(String product, LocalDate
start, LocalDate end) { …
  public long recordCounts(LocalDate start) {…
  Stream<SalesRecord> salesData() { …
  private SalesRecord makeSalesRecord(String line) { …
```

*class Gondorff...*

```
  public double gondorffNumber(String product) {
    return new DataSource().salesDataFor(product,
gondorffEpoch(product), hookerExpiry())
            .filter(r -> r.getDate().toString().matches(".*01$"))
            .findFirst()
            .get()
            .getQuantity() * Math.PI
            ;
  }
  private LocalDate gondorffEpoch(String product) {
    final long countingBase = new
DataSource().recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```




This separation into files is a mechanical process that's not really that interesting. But it's a necessary first step before we reach the interesting refactorings.

## Linker Substitution

Dividing up the code into several modules is helpful, but the interesting difficulty in all of this is the desire to distribute the Gondorff calculations as a separate component. Currently the Gondorff calculations assume that the sales data comes from a csv file with a particular path. Separating the data source logic gives me some ability to change that, but the current mechanism I have is awkward and there are other options to explore.

So what is the current mechanism? Essentially this is what I'll call Linker Substitution. The term “linker” is something of a throwback to compiled programs like C, where the link stage resolves symbols across separate compilation units. In JavaScript I can achieve the moral equivalent of this by manipulating the lookup path of files for the import command.

Let's imagine I want to install this application in an environment where they don't keep sales records in a CSV file, but instead run a query on a SQL database. To make this work I first need to create a CorporateDatabaseDataSource file with exported functions for `salesDataFor` and `recordCounts` that return the data in the form that the Gondorff file expects it. I then replace the DataSource file with this new one. Then when I run the application it "links" to the replaced DataSource file.

For many dynamic languages that rely on some kind of path lookup mechanism for linking, the Linker Substitution is a pretty nice technique for simple component substitutions. I don't have to do anything with my code to make it work, other than the simple separation into files that I've just done. If I have a build script, I can build the code for different data source environments by simply copying different files into the appropriate points in the path. This illustrates the advantage of keeping a program factored into small pieces - it allows substitution of those pieces, even if the original writer didn't have any substitutions in mind. It enables unforeseen customization.

To do Linker Substitution in Java is essentially the same task. I would need to package `DataSource` in a separate jar file to `Gondorff`, then instruct the user of `Gondorff` to create a class called `DataSource` with the appropriate methods and put it onto the classpath.

However with Java I'd do an additional step, applying Extract Interface on the data source.

```
public interface DataSource {
  Stream<SalesRecord> salesDataFor(String product, LocalDate start,
LocalDate end);
  long recordCounts(LocalDate start);
}

public class CsvDataSource implements DataSource {
```

Using a Required Interface like this is helpful because it makes explicit what functions gondorff is expecting from its data source.

One of the downsides of dynamic languages is they lack this explicitness, which can be a problem when combining components that have been developed separately. JavaScript's module system works well here because it statically defines the module dependencies, so they are explicit and can be checked statically. Static declarations have costs and benefits, one of the nice developments in recent language design is trying a more nuanced approach to static declarations rather than just treating languages as purely static or dynamic.

Linker Substitution has the advantage that it requires little work on the part of the component author, so fits in with unforeseen customization. But it has its downsides. In some environments, such as Java, it can be fiddly to work with. The code doesn't reveal how the substitution works, so there's no mechanism for controlling the substitution in the code base.

An important consequence of this lack of presence in the code is that the substitution cannot occur dynamically - that is once the program has been assembled and run, I can't change the data source. This usually isn't a big deal in production, there are cases where hot-swapping the data source is useful, but they are minority of cases. But the value of dynamic substitution comes with testing. It's very common to want to use Test Doubles to provide canned data for testing, which often means I'll want to throw in different doubles for different test cases.

These demands for greater explicitness in the code base and dynamic substitution for testing, usually lead us to explore other alternatives, ones that allow us to specify how components are wired up explicitly rather than just relying on path lookups.

## Data source as parameter with each call

If we want to support calling gondorff with different data sources, then one obvious way to do it is to pass it as a parameter each time we call it.

Let's look at this might look in the Java version first, beginning with the current state of the Java version, after extracting the DataSource interface

*class App...*

```
  public String emitGondorff(List<String> products) {
    List<String> result = new ArrayList<>();
    result.add("\n<table>");
    for (String p : products)
      result.add(String.format(
              "  <tr><td>%s</td><td>%4.2f</td></tr>",
              p,
              new Gondorff().gondorffNumber(p)
      ));
    result.add("</table>");
    return
HtmlUtils.encode(result.stream().collect(Collectors.joining("\n")));
  }
```

*class Gondorff...*

```
  public double gondorffNumber(String product) {
    return new CsvDataSource().salesDataFor(product,
gondorffEpoch(product), hookerExpiry())
            .filter(r -> r.getDate().toString().matches(".*01$"))
            .findFirst()
            .get()
            .getQuantity() * Math.PI
            ;
  }
  private LocalDate gondorffEpoch(String product) {
    final long countingBase = new
CsvDataSource().recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

To pass in the data source as a parameter, the resulting code looks like this.

*class App...*

```
  public String emitGondorff(List<String> products) {
    List<String> result = new ArrayList<>();
    result.add("\n<table>");
```

```
    for (String p : products)
      result.add(String.format(
              "  <tr><td>%s</td><td>%4.2f</td></tr>",
              p,
              new Gondorff().gondorffNumber(p, new CsvDataSource())
      ));
    result.add("</table>");
    return
HtmlUtils.encode(result.stream().collect(Collectors.joining("\n")));
  }
```

*class Gondorff...*

```
  public double gondorffNumber(String product, DataSource dataSource) {
    return dataSource.salesDataFor(product, gondorffEpoch(product,
dataSource), hookerExpiry())
            .filter(r -> r.getDate().toString().matches(".*01$"))
            .findFirst()
            .get()
            .getQuantity() * Math.PI
            ;
  }
  private LocalDate gondorffEpoch(String product, DataSource dataSource)
{
    final long countingBase =
dataSource.recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

I can do this refactoring in a few small steps.

- Use Add Parameter on `gondorffEpoch` to add `dataSource`
- Replace the call to `new CsvDataSource()` to use the just added `dataSource` parameter
- Compile and test
- Repeat for `gondorffNumber`

Now over to the JavaScript version, again here's the current state.

*app.es6...*

```
  import gondorffNumber from './gondorff.es6'


  function emitGondorff(products) {
```

```
    function line(product) {
      return [
        `  <tr>`,
        `    <td>${product}</td>`,
        `    <td>${gondorffNumber(product).toFixed(2)}</td>`,
        `  </tr>`].join('\n');
    }
    return
encodeForHtml(`<table>\n${products.map(line).join('\n')}\n</table>`);
  }
```

*Gondorff.es6...*

```
  import {salesDataFor, recordCounts} from './dataSource.es6'


  export default function gondorffNumber(product) {
    return salesDataFor(product, gondorffEpoch(product), hookerExpiry())
        .find(r => r.date.match(/01$/))
        .quantity * Math.PI
      ;
  }
  function gondorffEpoch(product) {
    const countingBase = recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

In this case I can pass both functions as parameters

*app.es6...*

```
  import gondorffNumber from './gondorff.es6'
  import * as dataSource from './dataSource.es6'


  function emitGondorff(products) {
    function line(product) {
      return [
        `  <tr>`,
        `    <td>${product}</td>`,
        `    <td>${gondorffNumber(product, dataSource.salesDataFor,
dataSource.recordCounts).toFixed(2)}</td>`,
        `  </tr>`].join('\n');
    }
    return
encodeForHtml(`<table>\n${products.map(line).join('\n')}\n</table>`);
```

```
  }
```

*Gondorff.es6...*

```
  import {salesDataFor, recordCounts} from './dataSource.es6'

  export default function gondorffNumber(product, salesDataFor,
recordCounts) {
    return salesDataFor(product, gondorffEpoch(product, recordCounts),
hookerExpiry())
        .find(r => r.date.match(/01$/))
        .quantity * Math.PI
      ;
  }
  function gondorffEpoch(product, recordCounts) {
    const countingBase = recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

As with the java example, I can apply Add Parameter to `gondorffEpoch` first, compile and test, and then do the same to `gondoffNumber` for each function.

In this situation I'd be inclined to put both the `salesDataFor` and `recordCounts` function onto a single data source object and pass that in instead - essentially using Introduce Parameter Object. I won't do this in this article, primarily because it's a better demonstration of manipulating first class functions if I don't. But if gondorff needed to use more functions from the data source I would.

## Paramterizing the data source file name

As a further step I can parameterize the filename for the datasource. For the java version I do this by adding a field for the filename to the datasource and using Add Parameter to its constructor.

*class CsvDataSource...*

```
  private String filename;
  public CsvDataSource(String filename) {
    this.filename = filename;
  }
```

*class App...*

```
  public String emitGondorff(List<String> products) {
```

```
    DataSource dataSource = new CsvDataSource("sales.csv");
    List<String> result = new ArrayList<>();
    result.add("\n<table>");
    for (String p : products)
      result.add(String.format(
              "  <tr><td>%s</td><td>%4.2f</td></tr>",
              p,
              new Gondorff().gondorffNumber(p, dataSource)
      ));
    result.add("</table>");
    return
HtmlUtils.encode(result.stream().collect(Collectors.joining("\n")));
  }
```

For the JavaScript version I need to use Add Parameter on the functions that need it on the data source.

*dataSource.es6...*

```
  export function salesDataFor(product, start, end, filename) {
    return salesData(filename)
      .filter(r =>
      (r.product === product)
      && (new Date(r.date) >= start)
      && (new Date(r.date) < end)
    );
  }
  export function recordCounts(start, filename) {
    return salesData(filename)
      .filter(r => new Date(r.date) >= start)
      .length
  }
```

Left as it is, this would force me to put the filename parameter into the gondorff functions, but really they shouldn't need to know anything about that. I can fix this by creating a simple adapter.

*dataSourceAdapter.es6...*

```
  import * as ds from './dataSource.es6'

  export default function(filename) {
    return {
      salesDataFor(product, start, end) {return ds.salesDataFor(product,
start, end, filename)},
```

```
      recordCounts(start) {return ds.recordCounts(start, filename)}
    }
  }
```

The application code uses this adapter when it passes the data source into the gondorff function.

*app.es6...*

```
  import gondorffNumber from './gondorff.es6'
  import * as dataSource from './dataSource.es6'
  import createDataSource from './dataSourceAdapter.es6'


  function emitGondorff(products) {
    function line(product) {
      const dataSource = createDataSource('sales.csv');
      return [
        `  <tr>`,
        `    <td>${product}</td>`,
        `    <td>${gondorffNumber(product, dataSource.salesDataFor,
dataSource.recordCounts).toFixed(2)}</td>`,
        `  </tr>`].join('\n');
    }
    return
encodeForHtml(`<table>\n${products.map(line).join('\n')}\n</table>`);
  }
```

## Trade offs to parameterizing

Passing in the data source with each call to gondorff gives me the dynamic substitution that I'm looking for. As an application developer I can use any data source I like, I can also easily test by passing in stub data sources whenever I need to.

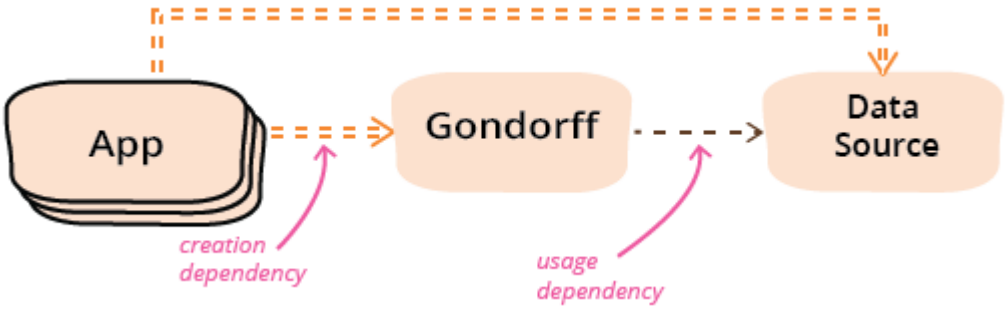
But there are also downsides to using a parameter with each call like this. Firstly I have to pass the data source (or its functions) as a parameter to every function in gondorff that either needs it, or calls another function that needs it. This can result in the data source being a piece of tramp data that wanders around everywhere.

The more serious problem is that now every time I have an application module that uses gondorff I have to ensure I can create and configure the data source too. This can easily get messy if I have a more complicated configuration, with a generic component that needs several required components, each of which have their own set of required

components. Every time I use gondorff I have to embed the knowledge there as to how I configure the gondorff object. That's a duplication that complicates the code making it harder to understand and use.

I can visualize this by looking at the dependencies. Before introducing the data source as a parameter the dependencies look like this:


When I pass the data source as a parameter it looks like this.




In these diagrams, I'm distinguishing between a usage dependency and a creation dependency. Usage dependency means that the client module calls functions defined on the supplier. There will always be a usage dependency between gondorff and data source. The creation dependency is a much more intimate dependency, since you usually need to know more about a supplier module in order to configure and create it. (A creation dependency implies a usage dependency.) Using a parameter with each call reduces the dependency from gondorff from creation to usage, but introduces a creation dependency from any applications.

As well as the creation depenendency problem, there's also another issue since I don't actually want to vary the data source in the production code. Passing the parameter with each call to gondorff implies that I'm varying the parameter between calls, but here whenever I call `gondorffNumber` I'm always passing in exactly the same data source. That dissonance is apt to confuse me in six months time.

If I have the same configuration for the data source all the time, it makes sense to set it up once and refer to it each time I use it. But if I do that, I might as well set gondorff up once, and use a fully configured gondorff every time I want to use it.

So having explored what using a parameter each times looks like, I'll make use of my version control system and do a hard reset to where I was at the beginning of this section so I can explore another path.

## Singular Services

An important property of both gondorff and dataSource is that they both can act as singular service objects. A service object is part of the Evans Classification, referring to an object that's oriented around an activity as opposed to entities or values that are focused around data. Often I refer to service objects as "services", but they are different to services in SOA as they aren't network accessible components. In a functional world, services are often just functions, but sometimes you do find situations where you want to treat a set of functions as a single thing. We see this with data source, where we have two functions, that I can think of as part of a single data source.

I also said “singular”, by this I mean it makes conceptual sense to only have one of these for a whole execution context. Since services are usually stateless, it makes sense to only have one around. If something is singular in an execution context, it means that we may refer to it globally within our program. We may even want to force it to be a singleton, perhaps because it's expensive to set up or there are concurrency constraints on resources it's manipulating. There may be only one of them in the entire process we're running in, or there may be more, such as one per thread using thread-specific storage. But either way, as far as our code's concerned, there's only one of them.

If we choose to make our gondorff calculator and data source be singular services, then it makes sense to configure them once, during the startup of the application, and then refer to them later on when using them.

This introduces a separation in the way services are handled: a separation of configuration and use. There are a couple of ways I can refactor this code to do this separation: introducing either the Service Locator pattern or the Dependency Injection pattern. I'll start with Service Locator.

## Introducing Service Locator

The idea behind the Service Locator pattern is to have a singular point for components to locate services. The locator is a Registry of services. In use, a client uses global lookup for the registry, then asks the registry for a particular service. Configuration sets up the locator with all the services that are needed.

The first step in the refactoring to introduce it is to create the locator. It's a pretty simple structure, little more than a global record, so my JavaScript version is just a few variables and a simple initializer.

*serviceLocator.es6...*

```
export let salesDataFor;
export let recordCounts;
export let gondorffNumber;

export function initialize(arg) {
  salesDataFor: arg.salesDataFor;
  recordCounts: arg.recordCounts;
  gondorffNumber = arg.gondorffNumber;
}
```

> `export let` exports a variable to other modules as a read-only view. [1]

The Java one is, of course, a bit more long-winded.

*class ServiceLocator...*

```
private static ServiceLocator soleInstance;
private DataSource dataSource;
private Gondorff gondorff;

public static DataSource dataSource() {
   return soleInstance.dataSource;
}

public static Gondorff gondorff() {
  return soleInstance.gondorff;
}

public static void initialize(ServiceLocator arg) {
  soleInstance = arg;
}

public ServiceLocator(DataSource dataSource, Gondorff gondorff) {
  this.dataSource = dataSource;
  this.gondorff = gondorff;
```

```
}
```

> *My preference in this situation is to provide an interface of static methods, so that clients of the locator don't need to know about where the data is stored. But I like to use a singleton instance for the data, as that makes it easier to substitution for testing.*

In both cases, the service locator is a set of attributes.

## Refactoring the JavaScript to use the locator

With the locator defined, the next step is to start moving services over to it, I begin with gondorff. To configure the service locator, I'll write a small module to configure the service locator.

*configureServices.es6...*

```
import * as locator from './serviceLocator.es6';
import gondorffImpl from './gondorff.es6';

export default function() {
  locator.initialize({gondorffNumber: gondorffImpl});
}
```

I need to ensure this function is imported and called at application start up.

*some startup file...*

```
import initializeServices from './configureServices.es6';


initializeServices();
```

To refresh our memories, here's the current application code (after the earlier revert).

*app.es6...*

```
import gondorffNumber from './gondorff.es6'


function emitGondorff(products) {
  function line(product) {
    return [
      `  <tr>`,
      `    <td>${product}</td>`,
      `    <td>${gondorffNumber(product).toFixed(2)}</td>`,
      `  </tr>`].join('\n');
```

```
    }
    return
encodeForHtml(`<table>\n${products.map(line).join('\n')}\n</table>`);
  }
```

To use the service locator instead, all I need to do is adjust the import statement.

*app.es6...*

```
  import gondorffNumber from './gondorff.es6'
  import {gondorffNumber} from './serviceLocator.es6';
```

I can run tests with just this change to ensure I didn't mess it up (that sounds better than "to find how I messed that up"). With that change down I do a similar change for the data source.

*configureServices.es6...*

```
  import * as locator from './serviceLocator.es6';
  import gondorffImpl from './gondorff.es6';
  import * as dataSource from './dataSource.es6' ;


  export default function() {
    locator.initialize({
      salesDataFor: dataSource.salesDataFor,
      recordCounts: dataSource.recordCounts,
      gondorffNumber: gondorffImpl
    });
  }
```

*Gondorff.es6...*

```
  import {salesDataFor, recordCounts} from './serviceLocator.es6'
```

I can use the same refactoring as earlier to parameterize the file name, this time the change only affects the service configuration function.

*configureServices.es6...*

```
  import * as locator from './serviceLocator.es6';
  import gondorffImpl from './gondorff.es6';
  import * as dataSource from './dataSource.es6' ;
  import createDataSource from './dataSourceAdapter.es6'
```

```
  export default function() {
    const dataSource = createDataSource('sales.csv');
    locator.initialize({
      salesDataFor: dataSource.salesDataFor,
      recordCounts: dataSource.recordCounts,
      gondorffNumber: gondorffImpl
    });
  }
```

*dataSourceAdapter.es6...*

```
  import * as ds from './dataSource.es6'

  export default function(filename) {
    return {
      salesDataFor(product, start, end) {return ds.salesDataFor(product,
start, end, filename)},
      recordCounts(start) {return ds.recordCounts(start, filename)}
    }
  }
```

## Java

The java case looks much the same. I create a configuration class to populate the service locator.

*class ServiceConfigurator…*

```
  public class ServiceConfigurator {
    public static void run() {
      ServiceLocator locator = new ServiceLocator(null, new Gondorff());
      ServiceLocator.initialize(locator);
    }
  }
```

And ensure I have a call to this somewhere in application startup.

The current application code looks like this:

*class App…*

```
  public String emitGondorff(List<String> products) {
    List<String> result = new ArrayList<>();
    result.add("\n<table>");
```

```
    for (String p : products)
      result.add(String.format(
              "  <tr><td>%s</td><td>%4.2f</td></tr>",
              p,
              new Gondorff().gondorffNumber(p)
      ));
    result.add("</table>");
    return
HtmlUtils.encode(result.stream().collect(Collectors.joining("\n")));
  }
```

I now use the locator to get the gondorff object.

*class App...*

```
  public String emitGondorff(List<String> products) {
    List<String> result = new ArrayList<>();
    result.add("\n<table>");
    for (String p : products)
      result.add(String.format(
              "  <tr><td>%s</td><td>%4.2f</td></tr>",
              p,
              ServiceLocator.gondorff().gondorffNumber(p)
      ));
    result.add("</table>");
    return
HtmlUtils.encode(result.stream().collect(Collectors.joining("\n")));
  }
```

To add the data source object into the mix, I start by adding it to the locator.

*class ServiceConfigurator...*

```
  public class ServiceConfigurator {
    public static void run() {
      ServiceLocator locator = new ServiceLocator(new CsvDataSource(),
new Gondorff());
      ServiceLocator.initialize(locator);
    }
  }
```

Currently the gondorff object looks like this:

*class Gondorff...*

```
  public double gondorffNumber(String product) {
    return new CsvDataSource().salesDataFor(product,
gondorffEpoch(product), hookerExpiry())
            .filter(r -> r.getDate().toString().matches(".*01$"))
            .findFirst()
            .get()
            .getQuantity() * Math.PI
            ;
  }
  private LocalDate gondorffEpoch(String product) {
    final long countingBase = new
CsvDataSource().recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

Using the service locator changes it thus

*class Gondorff…*

```
  public double gondorffNumber(String product) {
    return ServiceLocator.dataSource().salesDataFor(product,
gondorffEpoch(product), hookerExpiry())
            .filter(r -> r.getDate().toString().matches(".*01$"))
            .findFirst()
            .get()
            .getQuantity() * Math.PI
            ;
  }
  private LocalDate gondorffEpoch(String product) {
    final long countingBase =
ServiceLocator.dataSource().recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

As with the JavaScript case, parameterizing the filename just alters the service configuration code.

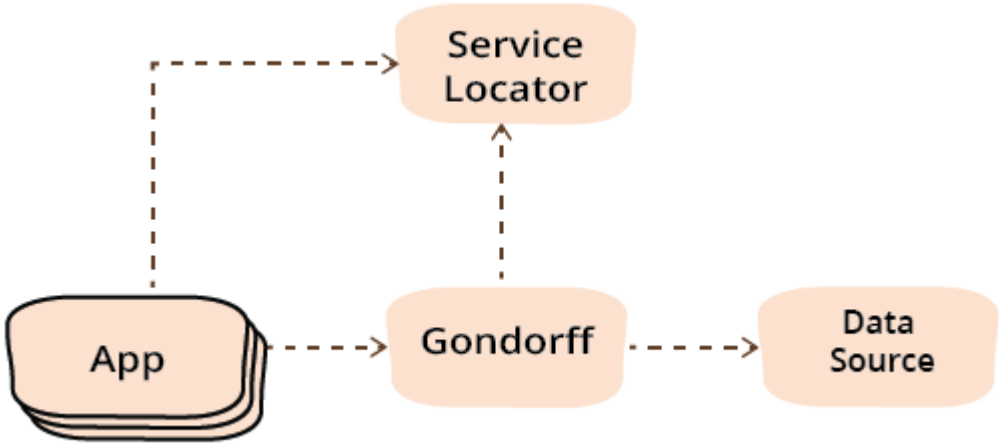
*class ServiceConfigurator…*

```
  public class ServiceConfigurator {
    public static void run() {
      ServiceLocator locator = new ServiceLocator(new
CsvDataSource("sales.csv"), new Gondorff());
      ServiceLocator.initialize(locator);
    }
```

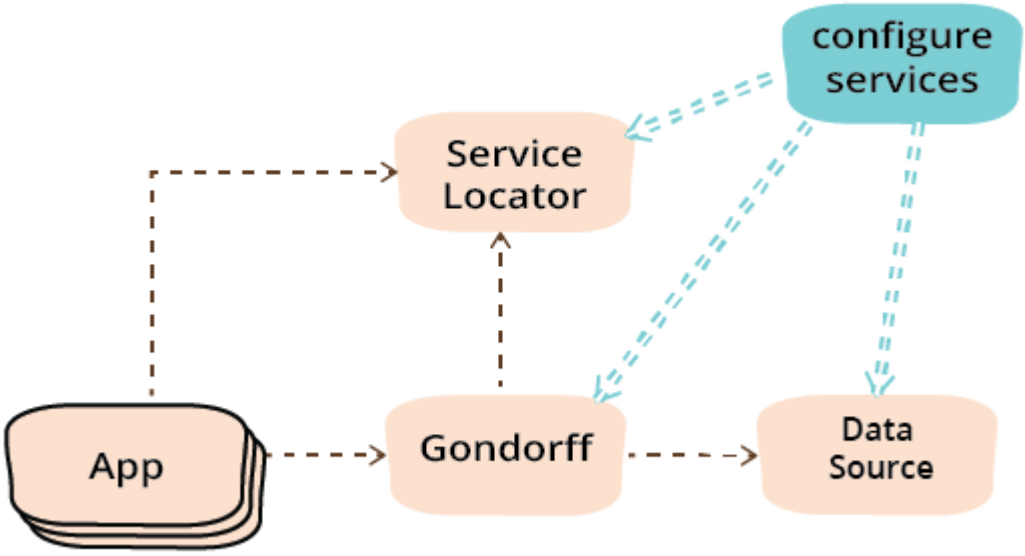
```
}
```

## Consequences of using a service locator

The immediate effect of using a service locator is altering the dependences between our three components. After the simple division of components, we can see the dependencies look like this.




Introducing the service locator removes all the creation dependencies between the primary modules. [2]. Of course this is ignoring the configure services module, which has all the creation dependencies.




I'm sure some of you might have noticed that the application customization is being

done by the service configuration function, which implies any customization is being done by the same linker substitution mechanism that I earlier said we need to get away from. That's true to some extent, but the fact that the service configuration module is clearly independent gives me a lot more flexibility. A library provider can supply a range of data source implementations, and clients can write a service configuration module that will select one at runtime based on configuration parameters such as a configuration file, environment variables, or a command line variable. There's a potential refactoring here to introduce parameters from a configuration file, but I'll leave that for another day.

But a particular result of using a Service Locator is that I can now easily substitute services for testing. I can put a test stub in for gondorff's data source like this:

*test...*

```
  it('can stub a data source', function() {
    const data = [
      {product: "p", date: "2015-07-01", quantity: 175}
    ];
    const newLocator = {
      recordCounts: () => 500,
      salesDataFor: () => data,
      gondorffNumber: serviceLocator.gondorffNumber
    };
    serviceLocator.initialize(newLocator);
    assert.closeTo(549.7787, serviceLocator.gondorffNumber("p"), 0.001);
  });
```

*class Tester...*

```
  @Test
  public void can_stub_data_source() throws Exception {
    ServiceLocator.initialize(new ServiceLocator(new DataSourceStub(),
ServiceLocator.gondorff()));
    assertEquals(549.7787,
ServiceLocator.gondorff().gondorffNumber("p"), 0.001);
  }
  private class DataSourceStub implements DataSource {
    @Override
    public Stream<SalesRecord> salesDataFor(String product, LocalDate
start, LocalDate end) {
      return Collections.singletonList(new SalesRecord("p",
LocalDate.of(2015, 7, 1), 175)).stream();
    }
    @Override
```

```
    public long recordCounts(LocalDate start) {
      return 500;
    }
  }
```

## Split Phase

While I was working on this article, I visited Kent Beck. After being fed his home made cheese, our conversation turned to refactoring topics and he told me about an important refactoring that he'd recognized a decade ago, but never got into a decent written form. This refactoring involved taking a complex computation and splitting it into two phases with the first phase passing it's result to the second phase with some intermediate results data structure. A large scale example of this pattern is that used by compilers, which split their work into many phases: tokenizing, parsing, code generation, with data structures such as token streams and parse trees acting as intermediate result.

When I got back home and started on this article again, I quickly recognized that introducing a Service Locator like this is an example of the Split Phase refactoring. I've extracted the configuration of the service objects into its own phase using the Service Locator as the intermediate results to pass the result of the configure-services phase to the rest of the program.

Splitting computation like this into separate phases is a useful refactoring because it allows us to think separately about the different needs in each phase, there is a clear indication of the results of each phase (in the intermediate results), and each phase can be tested independently by checking or supplying the intermediate results. This refactoring works especially well when we treat the intermediate results as an immutable data structure, giving us the benefits of working with the later phase code without having to reason about mutation behavior on any data generated by the earlier phase.

As I write this, it's barely a month since that conversation with Kent, but I feel that the notion of Split Phase is a powerful one to use for refactoring. Like many great patterns it has that notion of obviousness - I feel like it's just putting a name to something that I've been doing for decades. But such a name isn't a small thing, once you name an oft-used technique like this, it makes it easier to talk to other people about and alters my own thinking: giving it a more central role and more deliberate usage than comes when it's done unconsciously.

# Dependency Injection

Using a service locator has the downside that the component objects needs to know how the service locator works. This isn't a problem if the gondorff calculator is only used in the context of well-understood range of applications that use the same service locator machinery, but should I want to sell it to make my fortune that coupling is a problem. Even if all my eager buyers use service locators, it's unlikely that they will all use the same API. What I need is a way to configure gondorff with a data source in such a way that doesn't require any machinery other than what's built into the language itself.

This is the need that led to a different form of configuration that's called dependency injection. Dependency injection is trumpeted a lot, particularly in the Java world, with all sorts of frameworks to implement it. While these frameworks can be useful, the basic idea is really very simple and I'll illustrate it with refactoring this example to a simple implementation.

## Java example

The heart of the idea is that you should be able to write components like the gondorff object without needing to know about any special conventions or tools for configuring dependent components. The natural way to do this is in Java is for the gondorff object to have a field that holds the data source. That field can be populated by service configuration in the usual ways you populate any field - either with a setter or during construction. Since the gondorff object needs a datasource to do anything useful, my usual approach is to put it into the constructor.

*class Gondorff...*

```
  private DataSource dataSource;

  public Gondorff(DataSource dataSource) {
    this.dataSource = dataSource;
  }
  private DataSource getDataSource() {
    return (dataSource != null) ? dataSource :
ServiceLocator.dataSource();
  }
  public double gondorffNumber(String product) {
    return getDataSource().salesDataFor(product, gondorffEpoch(product),
hookerExpiry())
```

```
            .filter(r -> r.getDate().toString().matches(".*01$"))
            .findFirst()
            .get()
            .getQuantity() * Math.PI
            ;
  }
  private LocalDate gondorffEpoch(String product) {
    final long countingBase =
getDataSource().recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

*class ServiceConfigurator...*

```
  public class ServiceConfigurator {
    public static void run() {
      ServiceLocator locator = new ServiceLocator(new
CsvDataSource("sales.csv"), new Gondorff(null));
      ServiceLocator.initialize(locator);
    }
  }
```

By putting in the accessor `getDataSource` I can do the refactoring in smaller steps. This code works fine with the configuration done with service locator, I can gradually replace tests that set up the locator with tests that use this new dependency injection mechanism. The first refactoring just adds the field and applies Add Parameter. Callers can use the constructor with a null argument initially and I can work them one at a time to supply a data source, testing after each change. (Of course, since we do all service configuration in the configuration phase, there are usually not many callers. Where we get more callers is the stubbing in tests.)

*class ServiceConfigurator...*

```
  public class ServiceConfigurator {
    public static void run() {
      DataSource dataSource = new CsvDataSource("sales.csv");
      ServiceLocator locator = new ServiceLocator(dataSource, new
Gondorff(dataSource));
      ServiceLocator.initialize(locator);
    }
  }
```

Once I've done them all, I can remove all references to the service locator from the

gondorff object.

*class Gondorff...*

```
  private DataSource getDataSource() {
    return (dataSource != null) ? dataSource :
ServiceLocator.dataSource();
    return dataSource;
  }
```

*I could also inline `getDataSource` if I felt so inclined*

## JavaScript example

Since I'm eschewing classes in the JavaScript example, a way to ensure the gondorff calculator gets the data source functions without extra frameworks is to pass them as parameters with each call.

*Gondorff.es6...*

```
  import {recordCounts} from './serviceLocator.es6'

  export default function gondorffNumber(product, salesDataFor,
recordCounts) {
    return salesDataFor(product, gondorffEpoch(product, recordCounts),
hookerExpiry())
        .find(r => r.date.match(/01$/))
        .quantity * Math.PI
    ;
  }
  function gondorffEpoch(product, recordCounts) {
    const countingBase = recordCounts(baselineRange(product));
    return deriveEpoch(countingBase);
  }
```

I did this approach before of course, but this time need to ensure that clients don't need to do any set up with each call. I can do this by providing a partially applied gondorff function to clients.

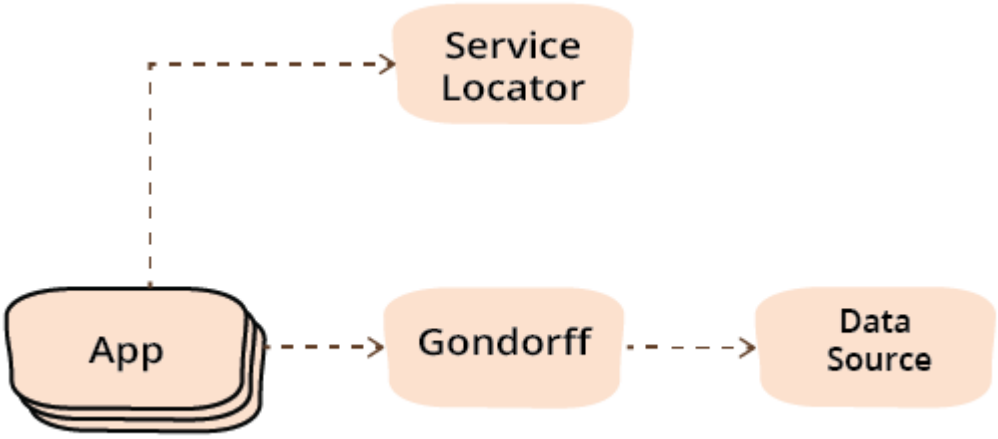
*configureServices.es6...*

```
  import * as locator from './serviceLocator.es6';
  import gondorffImpl from './gondorff.es6';
  import createDataSource from './dataSourceAdapter.es6'
```

```
  export default function() {
    const dataSource = createDataSource('sales.csv');
    locator.initialize({
      salesDataFor: dataSource.salesDataFor,
      recordCounts: dataSource.recordCounts,
      gondorffNumber: (product) => gondorffImpl(product,
dataSource.salesDataFor, dataSource.recordCounts)
    });
  }
```

## Consequences

If we look at the dependencies during the usage phase, the diagram looks like this.




The only difference between this and the earlier use of service locator is that there's no longer any dependency between gondorff and the service locator, which is the whole point of using dependency injection. (The configuration phase dependencies are the same set of creation dependencies.)

Once I've removed the dependency from gondorff to service locator, I can also remove the data source field from the service locator entirely if there aren't any other classes that need to get a data source from the service locator. I could also use dependency injection to provide the gondorff object to application classes, although there's much less value in doing that since the application classes aren't shared and thus aren't disadvantaged by using a locator. It's common to see the service locator and dependency injection patterns used together like this, with a service locator used to get an initial service whose further configuration has been done through dependency injection. Dependency injection containers are often used as service locators by providing a mechanism to look up a service.

## Final Thoughts

The key message of this refactoring episode is that of splitting the phase of service configuration from the use of the services. Exactly how you use service locators and dependency injection to perform this is less of an issue, and depends on the specific circumstances you're in. These circumstances may well lead you to a packaged framework to manage these dependencies, or if your case is simple it may be fine to roll your own.

---



## Footnotes

**1:** I developed these examples using Babel. At the moment Babel has a bug allowing you to reassign exported variables. The specification for ES6 says exported variables are exported as a read-only view.

**2:** One might argue that the java version of the service locator has dependencies on gondorff and data source due to them being mentioned in the type signatures. I'm discounting that here, since the locator doesn't actually invoke any methods on those classes. I could also remove those static type dependencies with some type gymnastics, although I suspect the cure would be worse than the disease.

## Acknowledgements

Pete Hodgson and Axel Rauschmayer gave me valuable help in improving my JavaScript. Ben Wu (伍斌) suggested a useful illustration. Jean-Noël Rouvignac corrected several typos.

## Significant Revisions

*13 October 2015:* First published