




[newtFire {dhlds}](#)

Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu)  Last modified: Monday, 16-Oct-2017 20:01:37 EDT. [Powered by firebellies](#).

Schematron Exercise 2



Meet Schematroll, the [Schematron](#) mascot! Schematroll is a cross between a [bilby](#) and a [bettong](#).

Preliminaries

To work on this assignment, you will need to find and do the following:

- **Information resources at the ready:** Review [our Schematron tutorial](#), and read more about the XPath functions and syntax we describe below either on the web (see w3Schools' "[XSLT, XPath, and XQuery Functions](#)", Obdurodon's "[The XPath Functions We Use the Most](#)") or through offline searching with the index of the Michael Kay book. You also want to read [our tutorial on validating id attributes](#).
- **XML file to test:** Right-click to save this TEI file locally and open it in <oXygen/>: [Sample for Digital Mitford Site Index](#). You will need to associate your Schematron file with this document **in addition to** the currently associated TEI schema lines.
- Open a new Schematron document in <oXygen/> by going to **File → New** and typing "Schematron" in the "Type filter text" box, or by going to **File → New → New Document → (scroll to Schematron in the alphabetized list) → Schematron**. Once opened, you will keep the default xml line at the top, but you will delete everything from <sch:schema> down. To write Schematron rules for a document in the TEI namespace, you will then replace this with:

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
  xmlns:sqf="http://www.schematron-quickfix.com/validator/process"
  xmlns="http://purl.oclc.org/dsdl/schematron">
  <ns uri="http://www.tei-c.org/ns/1.0" prefix="tei"/>

</schema>
```

- Write your Schematron patterns **inside** the </schema> root element.
- **Important:** You must use the `tei:` prefix before each of your elements since we are working with a document in the TEI namespace; otherwise none of your schema rules involving elements will fire! Remember that we do **not** use that prefix before attributes because attributes are in no namespace.

Analysis of the task

The goal:

The Digital Mitford project is working on a collection of prosopography data, that is, a record of people, places, organizations, published works, and other named entities relevant to British author Mary Russell Mitford's world in the nineteenth century. After some years of collaborative research the collection (which we call our "Site Index") contains thousands of entries, all contributed in batches by members of the editing team in the course of their research. It's common for our editors to make typographical errors as they enter details about historical people in particular, since these entries can be especially complicated! Your task is to write a helpful Schematron file to guide the editors in their process, flag errors if they reverse date ranges like birth and death dates, check for white space errors and other common problems, and check to see that the referencing of `@xml:id` attributes is correct. We hope that learning these things will give you ideas for writing Schematron to guide your own projects.

As you work on the rules below, think about how to group them logically into related `pattern` elements. You can use an `@id` on `pattern` elements to help label them and organize your work. Also, be sure to associate your Schematron file with the XML file you are testing *as soon as you write your first rule* so you can test it to make sure it is working.

A little orientation

Skim through the Digital Mitford project XML you downloaded, and get a sense of how it is organized and the way we have nested information about individuals inside each `person` element. You will see that each `person` has an `@xml:id` whose value is a distinct identity marker. Inside the `person` elements you will see `persName` elements, some of which contain nested `surname`, and `forename` elements. You will also see elements for `birth` and `death` with attributes and contents telling us about when and where a person was born and died. And most `person` elements contain a `biographical note` element with more information. These notes sometimes include references (made with `@ref` attributes) to people, places, books, and more listed elsewhere in the site index.

Rules to write and test

1. We want to close up extra white spaces that our editors inevitably type at the start of their elements. Write a Schematron rule that checks for leading white space inside the `tei:persName` element in particular. (That is, raise a warning when an element *starts with* a white space.) **Hints:**
 - You may want to look up the `starts-with()` function, one of the family related to `contains()`. If you would rather "play with matches()", the `matches()` function can handle this too, as long as you know how to write regex to find the start of a node. (Hint for safely "playing with matches": Remember the regular expression `^` and `$`? In XPath contexts, they refer to the start or end of an XML node, instead of the start or end of a line of text.)
 - One thing you will notice in writing these string-matching functions is that you need to represent the *haystack* (in this case, each XML node you're checking), followed by the *needle* (or the thing you're looking to find inside), and when that needle is a *literal string* as in with `starts-with()`, or a *regex pattern* as in `matches()`, you need to wrap it in quotation marks. But in the context of writing Schematron, your tests are written as the **value** of the attribute `@test`, so they must *already* be inside quotation marks: a NEW set of quotation marks inside is going to throw your computer off so it will not know how to find the end of your attribute value: and your computer will throw a *well-formedness* error if you use the same *kind* of quotation marks. So, we switch over to **single** quotation marks when we need to use quotes inside functions like we do here:

```
<report test="starts-with(., ' ')">
```

This practice is called *nesting* your quotation marks, and we use it in ordinary writing, too! In XML code and in formal editorial practice, we alternate between double and single quotation marks to nest them in layers.

2. Let's work on some Schematron tests for the `tei:person` element. We want to check the way its `@xml:id` is written. In our project when a historical person is given a unique identifier, that `@xml:id` value is supposed to begin with the most distinctive part of the person's name, their *last* name. Since we code the `tei:surname` element as a descendant of `tei:person`, you may write a Schematron rule that tests whether the `@xml:id` *starts with* the contents of the TEI's surname element. **Hint:** You are used to writing `starts-with()` and related functions so that they look for literal strings of text or regex patterns, but you can *also* use these functions to locate the contents of an element and make sure it matches up to what you see in an attribute. To locate *whatever is in an XML node* (element or attribute) instead of a specific string of text, simply do not use the quotation marks that indicate a string.
3. Sometimes our editors don't capitalize proper names! Check that all the `tei:forename`, `tei:surname`, and `tei:placeName` elements, as well as any `tei:persName` elements that hold text and do not wrap around `forename` and `surname` elements start with capital letters. **Hints:**
 - You can do that with one rule, and you can set multiple contexts using the **union operator** or pipe: `|` to join these together. You last used the pipe when writing Relax NG. You can use it in Schematron (and XSLT) contexts here specifically to join together multiple context items in one rule.
 - You actually DO need to "play with matches()" this time, because you need to find a regular expression pattern at the start of each node. The `starts-with()` function looks only for literal strings, not regex patterns. (We'll repeat our Hint for safely "playing with matches" in case you didn't read it on number 1: Remember the regular expression `^` and `$`? In XPath contexts, they refer to the start or end of an XML node, instead of the start or end of a line of text.)

4. Now let's take a look at the dates coded in this file, coded in the `tei:birth` and `tei:death` elements. All death dates need to be later than birth dates, but surprisingly, the TEI does not have a built-in way of checking this. Write a Schematron rule to flag when the dates coded in the `@when` attributes on any `tei:birth` and `tei:death` elements don't make sense. **Hints:**
- We use a few different kinds of dating attributes here: `@notBefore`, `@notAfter`, and `@when`, depending on how certain we are of when a birth or death occurred. For the purposes of this homework, it is fine to **concentrate only on the `@when` attributes** coded on `tei:birth` and `tei:death`.
 - **How to test for this:** Some dates are given as full ISO years (yyyy-mm-dd) and others are only partial and those, alas, will NOT convert to a machine-readable date with `xs:date()`, so we do not want to use that function here. Instead, we recommend that you work with the `tokenize()` function to isolate the year as the piece that we really need to look at, that is, the four-digit year that sits in front of the first hyphen. To reliably capture this piece, write the `tokenize()` function to break the attribute values in pieces around hyphens ("tokenize on the hyphen") and write a position predicate to grab the *first* of the tokens. (Note: `tokenize()` is a wonderfully adaptable function! Even if the date value lacks any hyphens and only contains a year, this will still return that year since the token just won't break off!)
 - Remember, you are testing to see when a *birth year* is later than a *death year*, so you need to write a test that uses comparison operators, like you did in [Schematron Exercise 1](#).
5. For the last required task in this assignment, it is very important for our site index file that `@ref` attributes must begin with a leading hashtag (#), since (as we explain more fully in our guide on "[Coding with Unique Identifiers and Testing Them with Schematron](#)"), the hashtag is reserved for `@ref` attributes that *point* to `@xml:ids`, so they do not duplicate those ids (whose values should only ever turn up once in a project). Write Schematron rule(s) to test and flag those errors on our `@ref` attributes, to help us find where these are missing their required hashtags.
6. **Optional Bonus Challenge:** These last two tasks are challenging, but may be useful to adapt in projects, so if you do not have time to write them now, you may wish to come back to them later on. To work on these, you need to consult our guide on "[Coding with Unique Identifiers and Testing Them with Schematron](#)". Finally, carefully following our guide, adapt the code we provide there to write a test that checks whether the `@ref` and `@resp` attribute values, *following their hashtags*, actually match up to a defined `@xml:id` in this file *or* in the Digital Mitford Site Index at <http://digitalmitford.org/si.xml>. (Note that this rule will also ensure that these values actually begin with a hashtag!) Following our guide, you will learn how to write a `let` statement to define a variable that points to another file's `@xml:ids`, and then *refer* to that variable in your Schematron test. Also, it is perfectly legal in our project for there to be *multiple* values on an `@ref` or `@resp`, separated by white space, just as you see in our guide, so you should follow our lead to adapt our code there.

7. **Optional Bonus Challenge:** We need a more sophisticated way than we used in number 3 to check the way people type out full names in the persName elements. Can we test for errors like these?

Dorothy wordsworth

or

Percy bysshe Shelley

Of course we can, by adapting the `tokenize()` we have been using here to break on white space, and to test **each** token in turn to see if it is capitalized. You can do this by applying the `for $i in (sequence) return ...` (or "**for-loop**" XPath feature) so we can walk through each token in the full sequence. To see how to write the code, consult our [our guide on testing unique identifiers](#): Look at our `let` statement, defining a variable containing a sequence of tokens, and then consider how we processed each one in turn in our `assert @test`. Can you adapt that code to tokenize the parts of a name, and test to see if each part is capitalized? Write your Schematron rule!

Submission

Upload your completed Schematron schema AND the si-Add-MRMsample.xml file **with your Schematron associated** to Courseweb, and follow our [standard filenaming conventions for homework assignments uploaded to Courseweb](#).