XML Design-by-Composition

Roger L. Costello
December 2012

Preface

This article shows how to design XML documents by composition (design-by-composition). The benefits of design-by-composition are many: (a) each component within your XML document is a loose coupler, (b) each component can be independently understood and verified, (c) each component can be combined (composed) with other components, thus providing flexibility and power.

Introduction

Design-by-composition invites us to ask, “What properties do we desire of our data?” We shall approach design-by-composition from the perspective of identifying desired properties and then ensuring the properties are maintained as data is composed.

One property that is often desired is “isomorphism.” This paper explains what isomorphism is, why it’s important, and how to design XML documents to retain the isomorphism property as components are composed.

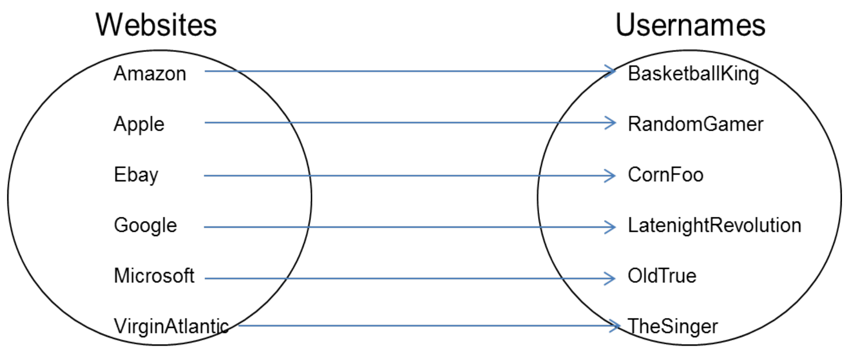
Terminology

This article uses the term *map* and *function* interchangeably.

Isomorphism

Isomorphism means there is a one-to-one correspondence between data items.

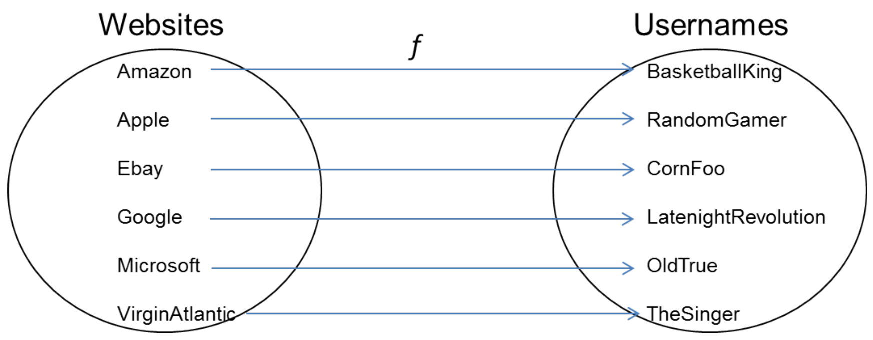
Example: Suppose we have a policy that each website account must have a different username. Examine the following diagram and you will see that the policy is satisfied for this set of websites:



There is a one-to-one correspondence between websites and usernames. That is, this data possesses the property of isomorphism.

By requiring the data possess the isomorphism property we can be certain that our policy is satisfied (i.e., every website has a unique username). Given any website we can determine its username and, vice versa, given any username we can determine its website.

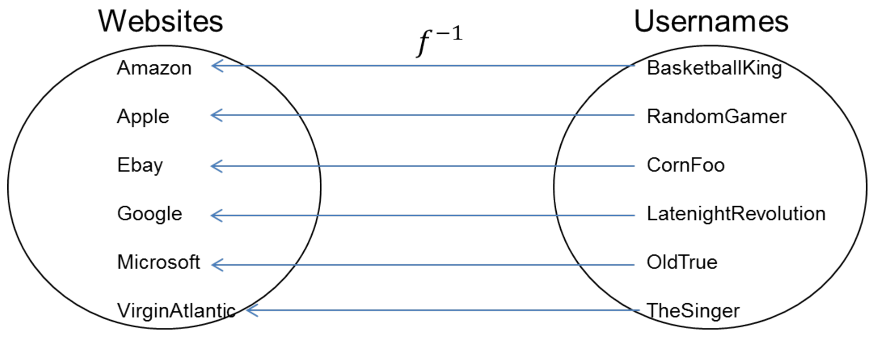
There is a *mapping* from the set of websites to the set of usernames. Let *f* denote the mapping:



In fact, *f* is a function: provide *f* an argument from Websites and it returns a username, e.g.,

f (Ebay) returns CornFoo

f^{-1} denotes the inverse function, from Usernames to Websites:

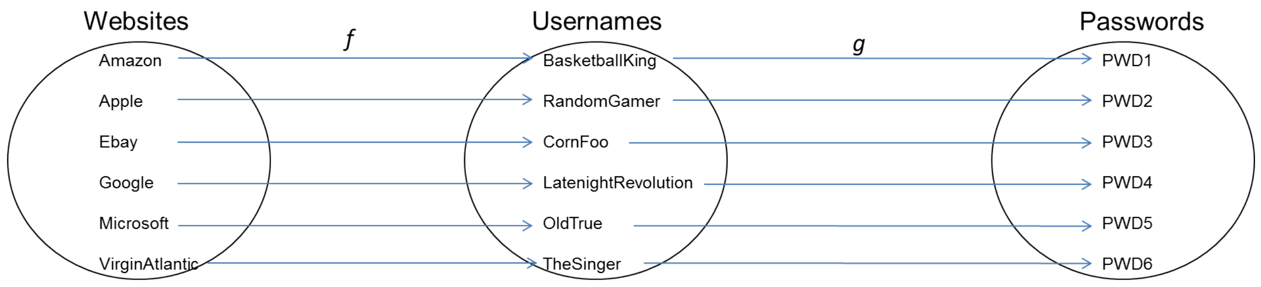


Provide f^{-1} an argument from Usernames and it returns a website, e.g.,

f^{-1}(CornFoo) returns Ebay

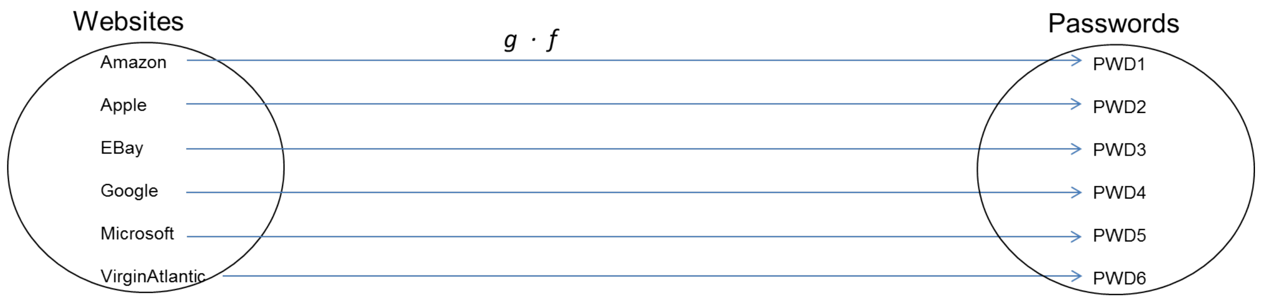
Because the data possesses the property of isomorphism we know that each website has a unique username and each username corresponds to a unique website.

We can compose maps (functions). Suppose we have a second policy: each username must have a different password. This data satisfies the policy:



Note that there are two isomorphisms—there is an isomorphism between websites and usernames, and an isomorphism between usernames and passwords. The composition of isomorphisms is an isomorphism, so there is an isomorphism between websites and passwords.

Composing *f* and *g* yields a new function:



Dot (·) is used to denote composition.

$g \cdot f$ is the composition of function *g* and function *f* and is read as:

g following *f*

Composing *g* and *f* results in a new function we call *h*:

$h = g \cdot f$

If we provide *h* an argument from Websites, then it will return the password for the website, e.g.,

h(Ebay) returns PWD3

Composition is incredibly powerful.

Implementing Isomorphisms in XML

See Appendix A for the complete example.

The most direct way to implement the Website-Username isomorphism is to simply mirror in XML the above diagrams: create a <Websites> element consisting of each website, a <Usernames> element consisting of each username, and a <map> element that connects (using ID/IDREF) each website to a username. Here is the set of websites:

```
<Websites>
     <Website id="Amazon">http://www.amazon.com</Website>
     <Website id="Apple">http://www.apple.com</Website>
     <Website id="Ebay">http://www.ebay.com</Website>
     <Website id="Google">http://www.google.com</Website>
     <Website id="Microsoft">http://www.microsoft.com</Website>
     <Website id="VirginAtlantic">http://www.virgin-atlantic.com</Website>
```

```
</Websites>
```

It is important to uniquely identify each website with an attribute of type ID.

Each <Website> element may have a complex internal structure such as this:

```
<Websites>
     <Website id="Amazon">
          <URL>http://www.amazon.com</URL>
          <Headquarters>Seattle, WA</Headquarters>
          <Founder>Jeff Bezos</Founder>
     </Website>
     <Website id="Apple">
          <URL>http://www.apple.com</URL>
          <Headquarters>Cupertino, CA</Headquarters>
          <Founder>Steve Jobs</Founder>
     </Website>
     <Website id="Ebay">
          <URL>http://www.ebay.com</URL>
          <Headquarters>San Jose, CA</Headquarters>
          <Founder>Pierre Omidyar</Founder>
     </Website>
     <Website id="Google">
          <URL>http://www.google.com</URL>
          <Headquarters>Mountain View, D.C.</Headquarters>
          <Founder>Sergey Brin</Founder>
     </Website>
     <Website id="Microsoft">
          <URL>http://www.microsoft.com</URL>
          <Headquarters>Redmond, WA</Headquarters>
          <Founder>Bill Gates</Founder>
     </Website>
     <Website id="VirginAtlantic">
          <URL>http://www.virgin-atlantic.com</URL>
          <Headquarters>Crawley, London</Headquarters>
          <Founder>Richard Branson</Founder>
     </Website>
</Websites>
```

We will see below that this design is the result of performing several mapping operations. For maximum flexibility it is best to keep the internal structure as simple as possible.

Here is the set of usernames:

```
<Usernames>
    <Username id="BasketballKing" />
    <Username id="RandomGamer" />
    <Username id="CornFoo" />
    <Username id="LatenightRevolution" />
    <Username id="OldTrue" />
    <Username id="TheSinger" />
```

```
</Usernames>
```

Again notice that each value is uniquely identified

Next, create a <map> element that contains pairs of IDREF attributes: *from* points to a website and *to* points to a username.

```
<map id="website-to-username">
     <singletonMap from="Amazon"           to="BasketballKing" />
     <singletonMap from="Apple"            to="RandomGamer" />
     <singletonMap from="Ebay"             to="CornFoo" />
     <singletonMap from="Google"           to="LatenightRevolution" />
     <singletonMap from="Microsoft"        to="OldTrue" />
     <singletonMap from="VirginAtlantic"   to="TheSinger" />
</map>
```

Read as: The website identified as *Amazon* is mapped to the username identified as *BasketballKing*; the website identified as *Apple* is mapped to the username identified as *RandomGamer*; and so forth.

To validate that the XML is an isomorphism we must check that:

1. Each website is distinct (no duplicates).
2. Each username is distinct (no duplicates).
3. No two websites map to the same username.
4. The number of websites equals the number of usernames.

Each of those checks can be implemented with a simple XPath expression. See Appendix B.

Okay, we can verify that we have an isomorphism between the set of websites and the set of usernames. So what? What can we do with it? As with any XML document, our goal is to enable applications to get answers to questions from the data in the XML document. One question that an application might want to get answers to is: *For website A, what is its username?* Since we have an isomorphism we can rest assured that there will be a unique answer to that question. That is a powerful property of our XML design.

The <map> element is an XML representation of a function. Let's denote that function by *f*. We want applications to be able to express this:

f (Ebay)   *returns CornFoo*

f can be implemented with a simple XSLT function. See Appendix C.

Suppose f is applied to each website:

f (Amazon)	*returns BasketballKing*
f (Apple)	*returns RandomGamer*
f (Ebay)	*returns CornFoo*
f (Google)	*returns LatenightRevolution*
f (Microsoft)	*returns OldTrue*
f (VirginAtlantic)	*returns TheSinger*

The result is a set of (website, username) pairs. Those pairs could be wrapped in markup and arranged sequentially:

```
<Websites-and-Usernames>
        <Pair>
            <Website id="Amazon">http://www.amazon.com</Website>
            <Username id="BasketballKing" />
        </Pair>
        <Pair>
            <Website id="Apple">http://www.apple.com</Website>
            <Username id="RandomGamer" />
        </Pair>
        <Pair>
            <Website id="Ebay">http://www.ebay.com</Website>
            <Username id="CornFoo" />
        </Pair>
        <Pair>
            <Website id="Google">http://www.google.com</Website>
            <Username id="LatenightRevolution" />
        </Pair>
        <Pair>
            <Website id="Microsoft">http://www.microsoft.com</Website>
            <Username id="OldTrue" />
        </Pair>
        <Pair>
            <Website id="VirginAtlantic">http://www.virgin-atlantic.com</Website>
            <Username id="TheSinger" />
        </Pair>
</Websites-and-Usernames>
```

We have arrived at a second XML design.

There is an important lesson to be learned:

This second design is the result of mapping each website to username.

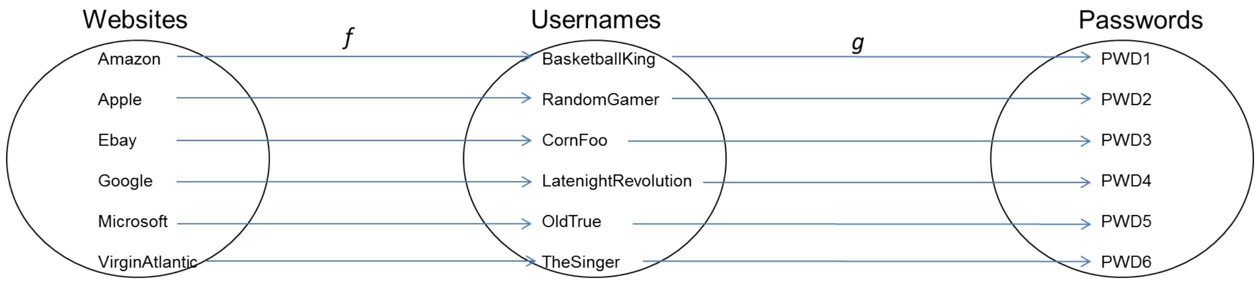
If each website is always to be mapped to a username, then this second design is preferred since answering the question, *For website A, what is its username?* is simply a matter of navigating to the website element and its username is the following sibling. Validating that the second design is an isomorphism requires the same checks as above.

However, it is unlikely that every application will want websites paired with usernames. For example, an application may want to know, *What is the password for this website?* or, *What is the password for this username?* So in general the second design limits flexibility and is not preferred.

The below examples use this Passwords element:

```
<Passwords>
      <Password id="PWD1">K8bNmJm6</Password>
      <Password id="PWD2">onb5Wn9S2bKi</Password>
      <Password id="PWD3">VvpYMbtq</Password>
      <Password id="PWD4">p7vpSKwnLY</Password>
      <Password id="PWD5">FS6Wn9UCG</Password>
      <Password id="PWD6">bugPQg9m4PSC3buR</Password>
</Passwords>
```

The above discussion showed *f* mapping websites to usernames and *g* mapping usernames to passwords:



This is important:

A website's password is the result of composing f and g.

Thus the password for Ebay is:

(g · f)(Ebay) *returns PWD3*

Let's define a function, *password*, as the composition of *g* and *f*:

password = (g · f)

Then the password for Ebay is:

password(Ebay) *returns PWD3*

Appendix D shows a simple XSLT function for composing functions.

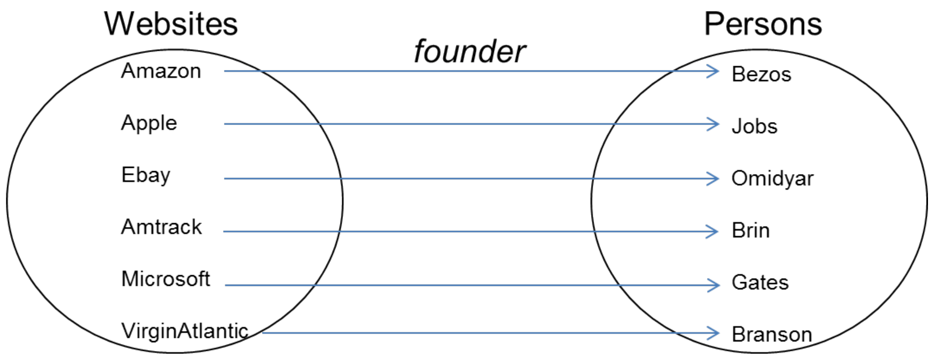
Of course a username's password is found by applying *g* to a username. Thus, the password for CornFoo is:

g(CornFoo) *returns PWD3*

Let's look at one more example to see the flexibility of design-by-composition. Suppose an application wants to determine the founder of a website. Given a set of persons:

```
<Persons>
     <Person id="Bezos">Jeffrey Bezos</Person>
     <Person id="Jobs">Steve Jobs</Person>
     <Person id="Omidyar">Pierre Omidyar</Person>
     <Person id="Brin">Sergey Brin</Person>
     <Person id="Gates">Bill Gates</Person>
     <Person id="Branson">Richard Branson</Person>
</Persons>
```

We can create a map from *Websites* to *Persons*:



Thus, the founder for Ebay is:

founder(Ebay) *returns Pierre Omidyar*

Through the process of composition we can construct designs of any sophistication. The sets (Websites, Usernames, Passwords, Persons, etc.) are the building blocks and the maps and composition are the assembly machinery. This is XML design-by-composition.

Summary

"Design by composition" means that we create independent sets of information with maps between them. Each item in a set has an ID attribute. Each map uses pairs of IDREF attributes, one IDREF pointing to an item in one set, the second IDREF pointing to an item in the other set. This paper showed an example of an XML document containing a Websites set and a Usernames set and a map between websites and usernames. Applications can use this XML document to obtain answers to questions such as, *What is this website's username*? This paper showed how to compose the mappings from Websites-to-Usernames and Usernames-to-Passwords. With the composite mapping an application can obtain answers to questions such as, *What is this website's password*?

The advantages of design by composition are:

1. Each set can be independently developed and maintained.
2. Each set is easily understood.
3. Each set can be combined (composed) with other sets.
4. Each set is a loose coupler.
5. Composing maps yields increasingly rich information.

All of the information presented in this paper is rooted in a branch of mathematics called Category Theory. Therefore, if you design your XML using design-by-composition, then your designs will have the power of mathematics supporting them. For example, in the paper I made this statement: *The composition of isomorphisms is itself an isomorphism*. That is a result from Category Theory. You don't have to think about it or prove it, you can just use it and rest assured that it always holds. Using design-by-composition you can take advantage of that powerful result and many others.

Acknowledgements

I wish to express my thanks to Dr. Mike Brenner for editing this article and for encouraging me to look into Category Theory and apply it to real-world data design. Without him, this article would never have been written.

The beautiful code in the appendices is largely due to the following people:

- David Carlisle
- Ken Holman
- Michael Kay
- Dimitre Novatchev
- Liam Quin

Appendix A: Sample XML Document that uses Design-by-Composition

Here is an XML document that uses design-by-composition. It contains three sets: Websites, Usernames, and Passwords. It has two maps: websites-to-usernames and usernames-to-passwords.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Document>

    <Websites>
        <Website id="Amazon">http://www.amazon.com</Website>
        <Website id="Apple">http://www.apple.com</Website>
        <Website id="Ebay">http://www.ebay.com</Website>
        <Website id="Google">http://www.google.com</Website>
        <Website id="Microsoft">http://www.microsoft.com</Website>
        <Website id="VirginAtlantic">http://www.virgin-atlantic.com</Website>
    </Websites>

    <Usernames>
        <Username id="BasketballKing" />
        <Username id="RandomGamer" />
        <Username id="CornFoo" />
        <Username id="LatenightRevolution" />
        <Username id="OldTrue" />
        <Username id="TheSinger" />
    </Usernames>

    <Passwords>
        <Password id="PWD1">K8bNmJm6</Password>
        <Password id="PWD2">onb5Wn9S2bKi</Password>
        <Password id="PWD3">VvpYMbtq</Password>
        <Password id="PWD4">p7vpSKwnLY</Password>
        <Password id="PWD5">FS6Wn9UCG</Password>
        <Password id="PWD6">bugPQg9m4PSC3buR</Password>
    </Passwords>

    <map id="website-to-username">
        <singletonMap from="Amazon"           to="BasketballKing" />
        <singletonMap from="Apple"            to="RandomGamer" />
        <singletonMap from="Ebay"             to="CornFoo" />
        <singletonMap from="Google"           to="LatenightRevolution" />
        <singletonMap from="Microsoft"        to="OldTrue" />
        <singletonMap from="VirginAtlantic"   to="TheSinger" />
    </map>

    <map id="username-to-password">
        <singletonMap from="BasketballKing"         to="PWD1" />
        <singletonMap from="RandomGamer"            to="PWD2" />
        <singletonMap from="CornFoo"                to="PWD3" />
        <singletonMap from="LatenightRevolution"    to="PWD4" />
        <singletonMap from="OldTrue"                to="PWD5" />
        <singletonMap from="TheSinger"              to="PWD6" />
    </map>
</Document>
```

Appendix B: Validating that the Websites-to-Usernames Mapping is an Isomorphism

Here's the XPath to check that each website is distinct:

count(Websites/) = count(distinct-values(Websites/*))*

The id attributes are of type ID so each username is guaranteed to be distinct.

Here's the XPath to check that no two websites map to the same username:

count(map//@to) = count(distinct-values(map/*/@to))*

And here's the XPath to check that the number of websites equals the number of usernames:

count(Websites/) = count(Usernames/*)*

Those XPath expressions are placed in a file:

isomorphism.xml

```
<Isomorphism>
    <Assert>count(Websites/*) = count(distinct-values(Websites/*))</Assert>
    <Assert>count(map/*/@to)  = count(distinct-values(map/*/@to))</Assert>
    <Assert>count(Websites/*) = count(Usernames/*)</Assert>
</Isomorphism>
```

and instances documents are validated against the file using the <xsl:evaluate> element in XSLT 3.0:

validate.xsl

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                version="3.0">

    <xsl:variable name="constraints" select="doc('Isomorphism.xml')" />

    <xsl:template match="Document">
        <xsl:evaluate xpath="string-join($constraints//Assert,',')"
                      as="xs:boolean*"/>
    </xsl:template>

</xsl:stylesheet>
```

Appendix C: “Here is a Website, what is its Username?”

Creating a function that returns the username of a website is readily implemented in XSLT.

Create a function, *f*. Pass to it a `<Website id= "..." />` element and it returns the matching `<Username id= "..." />` element. For example,

```
$f(Websites/Website[@id eq 'Ebay'])
```

returns:

```
<Username id="CornFoo"/>
```

The following XSLT program takes advantage of the new anonymous (no name) functions in XPath 3.0. The value of the variable *f* is an anonymous function. Thus, *$f* is a function. The function takes one argument, a website, and it uses the website-to-username map to locate the username. Here is the XSLT program:

get-websites-username.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:map="http://www.example.org"
                version="3.0">

    <xsl:output method="xml"/>

    <xsl:key name="ids" match="*[@id]" use="@id"/>

    <xsl:variable name="document" select="/Document" />

    <xsl:variable name="f" select="
        function(
                  $website as element(Website)
                )
                as element(Username)
        {$document/key('ids', $document/map[@id eq
            'website-to-username']/singletonMap[@from eq $website/@id]/@to)}" />

    <xsl:template match="Document">
        <xsl:variable name="Ebay" select="Websites/Website[@id eq 'Ebay']" />
        <Results>
            <Apply-f>
                <website>
                    <xsl:value-of select="$Ebay" />
                </website>
                <websites-username>
                    <xsl:value-of select="$f($Ebay)/@id" />
                </websites-username>
            </Apply-f
        </Results>
    </xsl:template>
```

```
</xsl:stylesheet>
```

Appendix D: Function Composition

Composing two functions is readily implemented using the new capabilities in XPath 3.0. For a detailed explanation of how function composition is implemented see Pearls of XSLT and XPath Design.

The following XSLT program creates two functions, *f* and *g*: *f* maps websites to usernames and *g* maps usernames to passwords. The two functions are composed to create a new function, *h*. Then a <Website id= "..." /> element is passed to *$h* which returns the <Password id= "..." /> element. For example,

```
$h(Websites/Website[@id eq 'Ebay'])
```

returns:

```
<Password id="PWD3">VvpYMbtq</Password>
```

Here is the XSLT program:

get-websites-password.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:map="http://www.example.org"
                exclude-result-prefixes="#all"
                version="3.0">

    <xsl:output method="xml"/>

    <xsl:key name="ids" match="*[@id]" use="@id"/>

    <xsl:variable name="document" select="/Document" />

    <xsl:variable name="f" select="
        function(
                   $website as element(Website)
                )
                as element(Username)
                {$document/key('ids', $document/map[@id eq
                 'website-to-username']/singletonMap[@from eq $website/@id]/@to)}" />

    <xsl:variable name="g" select="
        function(
                   $username as element(Username)
                )
                as element(Password)
```

```
            {$document/key('ids', $document/map[@id eq
             'username-to-password']/singletonMap[@from eq $username/@id]/@to)}" />

    <xsl:variable name="compose" select="
        function(
                   $a as function(item()*) as item()*,
                   $b as function(item()*) as item()*
                )
                as function(item()*) as item()*
            {function($c as item()*) as item()* {$b($a($c))}}" />

    <xsl:template match="Document">
        <Results>
            <xsl:variable name="Ebay" select="Websites/Website[@id eq 'Ebay']" />

            <Apply-f>
                <website>
                    <xsl:value-of select="$Ebay" />
                </website>
                <websites-username>
                    <xsl:value-of select="$f($Ebay)/@id" />
                </websites-username>
            </Apply-f>

            <xsl:variable name="CornFoo"
                          select="Usernames/Username[@id eq 'CornFoo']" />

            <Apply-g>
                <username>
                    <xsl:value-of select="$CornFoo/@id" />
                </username>
                <usernames-password>
                    <xsl:value-of select="$g($CornFoo)" />
                </usernames-password>
            </Apply-g>

            <Compose-f-and-g>
                <website>
                    <xsl:value-of select="$Ebay" />
                </website>
                <websites-password>
                    <xsl:variable name="h" select="$compose($f, $g)" />
                    <xsl:value-of select="$h($Ebay)" />
                </websites-password>
            </Compose-f-and-g>

        </Results>
    </xsl:template>

</xsl:stylesheet>
```